COMMUNICATION FACILITY IN MANYCORE-BASED CLUSTER SYSTEM
メニーコア並列計算機における通信機構の設計

by

Min Si
思敏

A Master Thesis
修士論文

Submitted to
the Graduate School of the University of Tokyo
on August 23, 2012
in Partial Fulfillment of the Requirements
for the Degree of Master of Information Science and Technology
in Computer Science

Thesis Supervisor: Yutaka Ishikawa　石川裕
Professor of Computer Science

ABSTRACT

This paper designs a direct communication facility, called DCFA, for manycore-based cluster systems. Manycore-based cluster is a kind of the accelerator-based cluster, in which a node is formed by a host server with multiple floating-point accelerators connected via PCI Express. The floating-point accelerator can be GPGPU or many-core units. Because a many-core unit is a device in the PCI Express, it is not capable of configuring and initializing the communication device also in the PCI Express, according to the PCI Express specification. This means that, in normal way, the host has to control the communication device and assist the data transfer from/to the many-core units, and inevitably extra communication overhead occurs. However, a many-core unit can send commands to the communication device if the PCI Express device address is given by the host. In DCFA, the host configures and initializes the communication device, and then internal structures of the device are distributed to both the host memory and the many-core memory, and subsequently after receiving the device information from the host, the many-core unit may send commands to the device to transfer data directly without host assists. The implementation of DCFA is based on the Mellanox InfiniBand Host Channel Adapter (IB HCA) and Intel's Knights Ferry (KNF), a predecessor of Knights Corner. The evaluation results show that, the latency and throughput of DCFA delivers the same performance as that of host to host data transfer for messages larger than 2Kbytes, and at most 2x worse for very small messages.

Moreover, this paper also implements an MPI library, called DCFA-MPI, to provide the parallel programming interface over DCFA. The MPI initialization, finalization, and other heavy functions such as collective communication are offloaded to host CPU, the floating-point computing and point-to-point communication are directly performed on many-core. DCFA-MPI uses a RDMA read plus RDMA write zero-copy design for rendezvous protocol. It finally delivers the same performance as that of host MPI communication for large messages. It has also been compared with the Intel MPI on Xeon + Offload to KNF mode between two KNF co-processors. The evaluation results show that, it achieves 5x speedup for smaller messages and still keeps at least 1.12x speedup for very large messages. Roughly speaking, the communication performance of DCFA-MPI(MPI on KNF) is always superior to the performance of Intel MPI on Xeon + Offload to KNF mode.

論文要旨

本論文では、メニーコア並列計算機上の DCFA（Direct Communication Facility for manycore-based Accelerators）と呼ばれる直接通信機構を設計する。アクセラレータ型並列計算機は、一台のホストサーバーに複数台の浮動小数点演算アクセラレータを PCI Express で繋げるノードを組合わせた並列計算機である。アクセラレータがメニーコアユニットで組合わせた並列計算機は、メニーコア並列計算機と呼ばれる。PCI Express 規格により、メニーコアユニットは PCI Express デバイスとして、PCI Express 上の通信デバイスを設定及び初期化することはできない。従って、通常の通信方法として、メニーコアユニットのデータ受送信にはホスト CPU の協力が必要となり、ホスト CPU とメニーコアユニット間の通信オーバーヘッドが発生される。しかしながら、もし通信デバイスの PCI Express デバイスアドレスが前もってホストから受ければ、メニーコアユニットは通信デバイスにコマンドを出せるようになる。DCFA では、ホストが通信デバイスを設定・初期化して必要情報をメニーコアに送り、デバイスの内部構造体をホストメモリ及びメニーコアメモリにそれぞれ置く。従って、メニーコアユニットはこれらの情報によって通信デバイスへコマンドを出して、データをホスト転送せずに送受信することができる。本論文では、DCFA が Mellanox 社の InfiniBand Host Channel Adapter (IB HCA) 及び Intel 社の Knights Ferry (KNF) に基づき実現される。評価結果により、DCFA を用いたメニーコア対メニーコア通信は、ホスト対ホスト通信と同レベルの性能が出せるということがわかる。

また、DCFA 上の並列プログラミングインターフェースを提供するために、本論文は DCFA-MPI と呼ばれる MPI 通信ライブラリを設計する。`MPI_Init`、`MPI_Finalize` 及び集団通信などの重い関数をホストにオフロードし、メニーコア自体は浮動小数点計算とメニーコア間の直接 point-to-point 通信を担当する。DCFA-MPI は、RDMA write と RDMA read を用いたゼロコピーの Rendezvous プロトコルを用いた。KNF 上で行った評価の結果により、DCFA-MPI(MPI on KNF) は大きいサイズのメッセージに対し、ホスト MPI 通信と同様な通信性能が出せる。また、Intel 社が発表した Intel MPI on Xeon + Offload to KNF モードと比較した結果により、DCFA-MPI(MPI on KNF) はいつも Intel MPI on Xeon + Offload to KNF モードより 1.12 倍〜5 倍高い通信性能を発揮する。

Acknowledgements

I would like to thank my adviser, Prof. Yutaka Ishikawa for guiding me patiently throughout the duration of my master study. In his teachings, I have learned a lot of high-performance computing knowledge during the last two years. I would also like to thank him for giving me the chance to have a summer internship in NEC Central Research Laboratories, in which I started my master research.

I'm thankful to Dr. Taku Shimosawa, who designed and implemented the basic abstraction layer over the many-core architectures. Without his contribution, I cannot continue the communication research on many-core.

I'm thankful to Dr. Masamichi Takagi for reading the draft of this thesis and give me valuable suggestions.

Grateful acknowledgement is made to Intel for providing the hardware, software and technical support associated with the Intel MIC architecture.

Contents

Chapter 1

Introduction

1.1 Research Background

In modern society, thanks to the wide-spread of computers, the complex computational problems people face in everyday life can be easily solved. However, many large-scale scientific computation problems such as weather prediction, molecular dynamics, astronomy simulation, cannot be handled by traditional computers because of their poor computing capabilities. To resolve these issues, the high-performance parallel computing technology has been attracting attention from the 60s of last century [20] [39]. Supercomputers using high-performance parallel computing technology has been considered one of the best approaches to solve the computational problems.

In the period from early 70s to 90s, the vector processors which parallelize computing as Single Instruction Multiple Data (SIMD) style dominated the supercomputers field. From early 90s, the scalar processors which only execute a single instruction during a clock cycle became popular in the supercomputers field. A large number of scalar processors are integrated with a dedicated inter-node network to achieve high throughput. This kind of system is called PC cluster, and has been widely used from late 90s.

As for inter-node communication network, InfiniBand [11] is an industry-standard architecture to define a low latency, high bandwidth System Area Network (SAN) for interconnecting multiple independent computing nodes and I/O nodes. InfiniBand provides the RDMA capabilities for remote data transfer with low CPU overhead. It has become a popular interconnect for high-performance computing, in the recent years it has also been increasingly adopted in the enterprise data centers.

As for parallel programming models, various parallel programming modes are developed with the growth of supercomputers. Message Passing Interface (MPI) is the de-facto standard for parallel programming for cluster environment. It's a library specification for message-passing model, the initial version is available as MPI1 [29] and an extension is defined as the MPI2 [29]. In this model, subtasks run on different processors and exchange data through passing messages.

MPI provides an application programming interface for the Fortran, C and C++ languages. Two major communication mode, point-to-point and collective, are used for various application communication requirement. There are already many efficient and widely used MPI implementations available as open-source, such as MPICH [41], MPICH2 [41], MVAPICH [6] and OpenMPI [8].

At the beginning of the 2000s, GPGPU started being used to accelerate a range of scientific applications. GPGPU computing has quickly developed, and the first GPGPU based supercomputer, TSUBAME supercomputer, was built

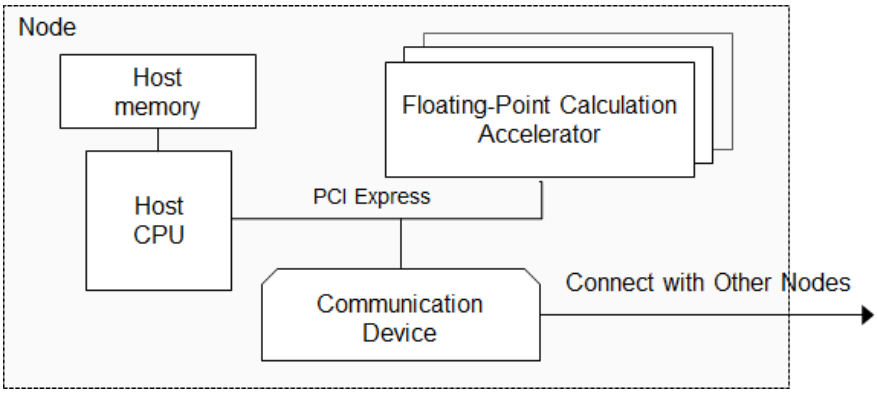


Figure 1.1: A node in Accelerator-based Cluster

by The Tokyo Institute of Technology in 2008. Moreover, three of the peta-scale supercomputers ranked in the November 2011 top500 [9] are GPGPU-based clusters. This computing mode offloads compute-intensive portions of the application to GPGPUs, while other code still runs on CPU [10].

The GPGPUs are formed as floating-point calculation accelerators. The clusters, in which a node is formed by a PC server with this kind of floating-point calculation accelerators, are called as accelerator-based cluster (Figure 1.1). There are two kinds of parallel execution models possible in such an accelerator-based cluster system: the host-assisted parallel execution model and the standalone parallel execution model.

In the host-assisted parallel execution model, a program running in a host CPU dispatches number crunching computations to an accelerator by transferring them from host memory to accelerator memory. Communication between compute nodes is handled by the host CPU. In this execution model, data are moved between the accelerator and the host, and between the host and the remote host. These extra data movements result in communication overhead.

In the other parallel execution model, the standalone parallel execution model in an accelerator-based cluster system, not only number crunching computation but also communication handling is executed in the accelerator unit. If this execution model were implemented, low overhead communication, i.e., low latency, would be achieved. Thus, this model would provide strong scalability, i.e., scalability for a fixed problem size.

Many-core architecture is another attractive candidate for achieving exaflop performance. Intel announced an x86-based many-core architecture called Intel® Many Integrated Core (Intel® MIC) architecture at the ISC 2010 conference [3]. Intel also announced that the first commercial product based on Intel MIC architecture codenamed Knights Corner, achieved 1 Tflops of double precision floating-point performance on first silicon, and connected to the host via the PCI Express bus, at the SC11 conference [1]. Knights Corner will have greater than 50 cores and will be based on Intel's 22nm process technology. Intel has also provided a design and development platform called Knights Ferry (KNF), which is a software development predecessor of Knights Corner. This is enabling software and hardware developers to prepare for the upcoming products based on the MIC architecture. An important advantage of the MIC architecture is that, because of it is based on the general purpose x86 processor, it can write commands to other PCI Express devices. Moreover the existing host libraries and applications can be moved to it with little or no modification.

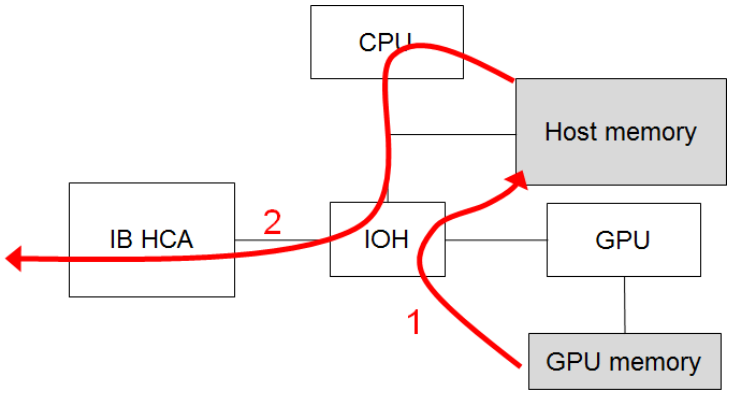


Figure 1.2: Data Transfer in GPUDirect

1.2 Problem Statement

The accelerator-based cluster is an attractive candidate for implementing post peta-scale supercomputer. For the accelerator-based clusters, which could be formed by either GPGPU or many-core, since the computation tasks are offloaded to the accelerators, the communication between accelerators is required. GPGPU-based cluster systems follow the host-assisted parallel execution model, in which extra data movements between accelerator and host are necessary for accelerator communication. Because GPGPU also needs a CPU core to issue commands for controlling communication devices, although its memory might be possible be accessed from other PCI Express devices such as InfiniBand HCA. By now, NVIDIA has designed a technology called GPUDirect to reduce extra data copies but still needs one copy between host and GPU (Figure1.2).

Unlike GPGPUs, a many-core based accelerator can write commands to a communication device if the PCI Express device address is given by the host. This means that, the many-core based accelerator is capable of performing communication directly though communication devices without the extra data movements. However, according to the PCI Express standard, the accelerator, a device in PCI Express, cannot configure devices and cannot receive interrupts from devices. Thus, it is not a straightforward task to implement the standalone parallel execution model in many-core based clusters. This paper focuses on the research of direct communication between many-core accelerators.

1.3 Contribution

This paper designs a direct communication facility for many-core based accelerators, called DCFA, to implement the standalone parallel execution model. DCFA provides the direct communication between two many-core units, which are inserted in different PC servers, without host assists (Figure 1.3). The implementation of DCFA is based on the Mellanox InfiniBand HCA and Intel's Knights Ferry (KNF), a software development predecessor of Knights Corner. Internal structures of the HCA are carefully distributed to memory areas of both the host and the many-core unit so that the many-core unit transfers its memory area directly to/from either remote host or remote many-core unit. The host CPU configures and initializes the InfiniBand HCA. The evaluation results show that, the latency and throughput of DCFA delivers the same performance as that of host to host data transfer for large messages.

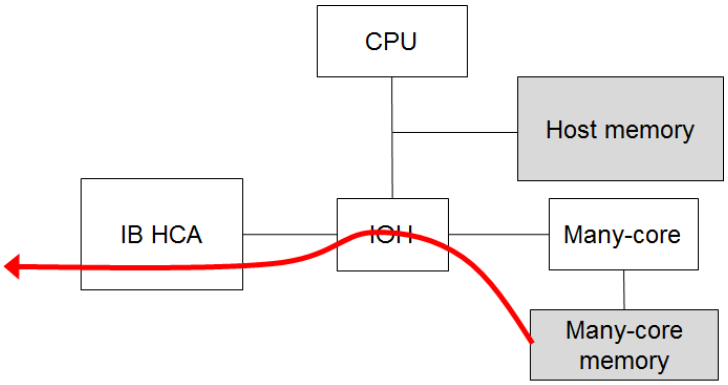


Figure 1.3: Data Transfer in DCFA

Moreover, a light-weight MPI library, called DCFA-MPI, is also implemented to provide the parallel programming interface over DCFA. It is based on the Yet Another MPI Implementation (YAMPII) [19], which is a light-weight MPI implementation for host communication. The MPI initialization, finalization are offloaded to host CPU, the floating-point computing and point-to-point communication are directly performed on many-core units. DCFA-MPI uses a RDMA read plus RDMA write zero-copy design for rendezvous protocol. After tuning several parameter values, it delivers the same performance as that of host MPI communication for the messages larger than 64Kbytes. The DCFA-MPI has also been compared with the Intel MPI on Xeon + Offload to KNF mode between two KNF co-processors. The evaluation results show that, it achieves the best result, 5x speedup for 32bytes messages, and still keeps at least 1.12x speedup for very large messages.

This research is aimed at perfecting the communication over post peta-scale supercomputer with close to a million processors. Although DCFA-MPI has only been implemented over two many-cores so far, the scalability issues will be discussed in the implementation section, and is planned to be solved as future work.

1.4 Thesis Overview

The research is presented over the next several chapters. Chapter 2 introduces the thesis background including an Accelerator Abstraction Layer, the InfiniBand communication and a host MPI Implementation. Then the detailed design and implementation are presented in Chapter 3. This chapter presents an offload mechanism from many-core to host, and the detailed design and implementation of DCFA and DCFA-MPI. In Chapter 4, the experimental environment, evaluation scenarios and results are presented. DCFA has been compared with the host InfiniBand communication. DCFA-MPI has been optimized and compared with the Intel MPI on Xeon + Offload to KNF mode. Chapter 5 presents the related works around the thesis research. And finally, this thesis will be concluded with a brief summary and a discussion of future work in Chapter 6.

Chapter 2

Background

This chapter presents the background of thesis research. DCFA is designed based on an Accelerator Abstraction Layer which provides a programming interface for many-cores, and the InfiniBand software implementation is modified to implement the direct InfiniBand communication between many-core units. Moreover, the MPI implementation DCFA-MPI is implemented based on a light-weight host MPI implementation, called YAMPII. Following sections will present an overview of them.

2.1 Overview of AAL

An Accelerator Abstraction Layer (AAL) [35] is designed to hide hardware-specific functions and provide kernel programming interfaces to operating system developers (Figure 2.1). AAL resides in both host and many-core units. AAL in the host is implemented as a Linux device driver, AAL in the many-core provides low-level kernel programming interfaces to operating system developers, AAL IKC is the inter-kernel communication layer that performs the data transfer and signal notification between host and many-core CPUs.

A micro kernel for many-core units is to be implemented on top of the AAL. It provides the process and thread management, file I/O interface, memory mapped I/O with the host CPU, and a low-level low-latency communication facility. The Linux kernel runs in the host CPUs to perform rich OS functions such as file systems, whose footprints are large enough to pollute the cache memory of many-cores if such a function were to be executed in many-core units.

In DCFA implementation, the memory mapping and communication between

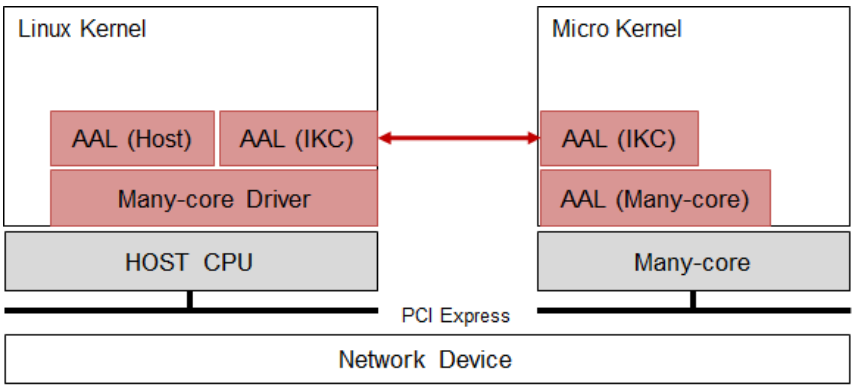


Figure 2.1: AAL Structure

Table 2.1: AAL Functions

AAL (Host)	
`aal_device_map_memory(dev, physaddr, size)`	Map from many-core memory
`aal_device_unmap_memory(dev, physaddr, size)`	Unmap from many-core memory
`aal_device_map_virtual(dev, physaddr, size, *virtual, flag)`	Map to virtual address
`aal_device_unmap_virtual(dev, physaddr, size)`	Unmap from virtual address
AAL (Many-core)	
`aal_mc_map_memory(*os, physaddr, size)`	Map from host memory
`aal_mc_unmap_memory(*os, physaddr, size)`	Unmap from host memory
`aal_mc_map_virtual(physaddr, npages, attr)`	Map to virtual address
`aal_mc_unmap_virtual(*virtual, npages)`	Unmap from virtual address
AAL (IKC)	
`aal_ikc_listen_port(os, *param)`	Listen a IKC port
`aal_ikc_connect(*os, *param)`	Connect to remote IKC port
`aal_ikc_send(*channel, *packet, option)`	Send a packet
`aal_ikc_recv(*channel, *packet, option)`	Receive a packet

host and many-core are required. AAL in the host implements the memory mapping from many-core memory to host memory, and AAL in the many-core implements the mapping from host memory to many-core memory. AAL IKC provides the communication between host and many-core, the same programming interface is defined in both host and many-core. Table 2.1 shows a list of corresponding functions.

2.2 Overview of InfiniBand

The InfiniBand Architecture (IBA) [11] is an industry-standard architecture to define a low latency, high bandwidth System Area Network (SAN) for interconnecting multiple independent computing nodes and I/O nodes. Two channel adapters are defined in the IBA devices, the Host Channel Adapter (HCA) which provides an software interface for the computing nodes, and the Target Channel Adapter (TCA) which provides the interface for I/O nodes.

2.2.1 Queue-based Model

The communication stack for the IBA consists of multiple layers. The interface presented by the InfiniBand HCA to user-level software belongs to the transport layer. A queue-based model (Figure 2.2) is used in this interface. The following queue structures are defined.

- Queue Pair (QP)
  A QP consists of a pair made up of a Send Work Queue (SQ) and a Receive Work Queue (RQ). The SQ holds the elements needed to transmit data, and the RQ holds elements about where to place received data. Communication operations such as send, RDMA read and RDMA write are described in Send Work Queue Request (Send WQR), and the receive operation is described in the Receive WQR. Once one of these WQRs has been submitted to a QP, a Work Queue Element (WQE) is placed on the appropriate work queue. The InfiniBand HCA executes WQEs in the order that they were placed on the work queue.

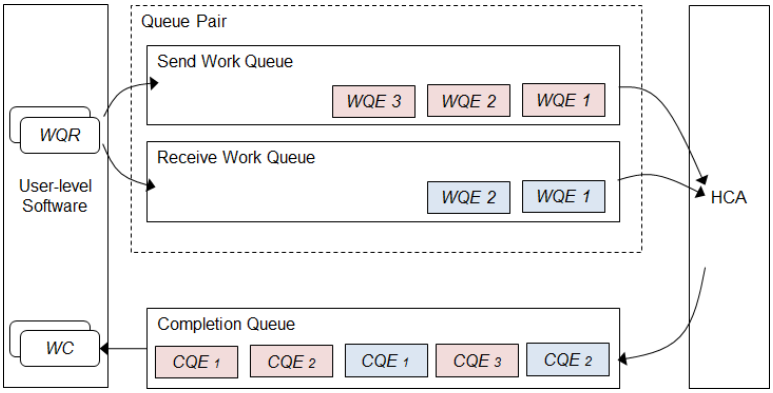


Figure 2.2: Quepe-based Model in InfiniBand
The origin of this figure is shown in [11]

Table 2.2: Comparison of IBA Transport Service Types

	RC	RD	UC	UD
Scalability (Number of QPs)	$O(p^2)$	$O(p)$	$O(p^2)$	$O(p)$
RDMA Operations	yes	yes	no	only RDMA write

The origin of this table is shown in [11]

- Completion Queue (CQ)
  Each work queue is associated with an individual or shared Completion Queue (CQ). When the InfiniBand HCA completes a WQE, a Completion Queue Element (CQE) is placed on the associated CQ. User-level software can poll the CQ to see if any WQR has been finished. If a new CQE is arrived, the polling operation will return a Work Completion (WC) to user-level software. Since a CQ may be associated with several work queues, the order of the CQEs generated for different work queues is not deterministic, except that the CQEs for the same work queue are normally posted in the same order as the corresponding WQEs were posted to the work queue.

2.2.2 Transport Services

InfiniBand supports following four IBA transport services types, Reliable Connection (RC), Reliable Datagram (RD), Unreliable Connection (UC), and Unreliable Datagram (UD). As compared in Table 2.2, the RC and UC are connection oriented, they require a local QP can be only connected to one remote QP, thus for p processes and all processes connected, a $O(p^2)$ number of QPs are required. On the other hand, the RD and UD are connection-less, a local QP can be connected to one or more remote QPs, thus the number of QPs can be reduced to only $O(p)$.

2.2.3 Communication Mode

IBA defines the send, RDMA read and RDMA write operations which are supported for SQ, the receive operation which is supported for RQ, following two communication modes can be used in user-level software.

- Send/Receive mode
  Both the sender and the receiver need explicitly post WQR to their QP.

The Send WQR includes a gather list of several local buffer segments that are sent to receiver side. The Receive WQR includes a scatter list of several local buffer segments, the incoming data is written to these buffer segments in the order specified. It is to be noted that the Receive WQR needs to be posted before the sender posts its Send WQR.

- RDMA mode
  The RDMA operations are one-side communication, i.e., the other side does not have to post any WQR for the data transfer. The RDMA write operation writes the data in a gather list of local buffer segments to a remote buffer, and the RDMA read operation reads the data from remote buffer to local memory buffer. However, one restriction is that the address information of the remote buffer needs to be known by the other side before data transfer.

2.2.4 Memory Management

In the memory management mechanism specified by IBA, the following two structures are defined.

- Memory Region (MR)
  An MR describes a set of memory locations and their access rights. A memory registration operation will pin down a user input buffer to physical memory and notify to the HCA, and then produces an MR which contains the virtual address and the size of the registered set of memory locations, a Local Key (LKey) and a Remote Key (RKey). The registration is usually a high-latency operation because it locks the memory pages to preserve the virtual to physical memory mapping. Moreover, all the memory locations containing data buffers must be registered before the InfiniBand HCA can access them. The information held by an MR is required when creating a WQR.

- Protection Domain (PD)
  Since a node might communicate with many different destinations, the IBA provides PDs to control whether a QP can access the registered MRs or not. QPs are allocated to, a memory locations are registered to a PD. A QP can only access the MRs in the same PD.

2.3 YAMPII

The Yet Another MPI Implementation (YAMPII) [19] was developed by Prof. Yutaka Ishikawa of University of Tokyo since December of 2001. It has been used as the MPI core of GridMPI developed in the NaReGI project in Japan.

It consists of the MPI Core layer which implements group, communicator and topology, the Remote Process Invocation Mechanism (RPIM) interface, the Request layer which handles the send/receive requests from MPI Core layer and forwards to the architecture specified point-to-point (P2P) layer, and the P2P layer which performs architecture or device specific communication (Figure 2.3). The SSH and RSH have been implemented under the RPIM interface, and some implementations of architecture or device specific P2P communication such as TCP and OpenIB (InfiniBand) are provided.

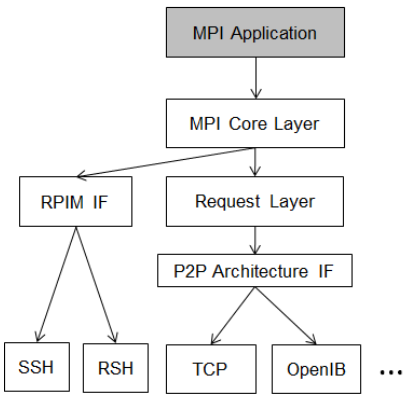


Figure 2.3: Software Structure in YAMPII

Chapter 3

Design and Implementation

3.1 Overview

This chapter describes the detailed design of DCFA and DCFA-MPI (Figure 3.1). DCFA is a direct communication facility for many-cores, it is based on the AAL and provides direct InfiniBand data transfer between many-cores. The initialization, connection and destruction tasks have been offloaded to host CPU, then many-core performs direct data transfer by itself. DCFA-MPI is an MPI implementation over DCFA for providing the direct point-to-point MPI communication between many-cores. The `MPI_Init` and `MPI_Finalize` functions have also been offloaded to host CPU, then the point-to-point communication functions such as `MPI_Rend` and `MPI_Recv` could be handled by many-core.

Since both DCFA and DCFA-MPI needs to offload tasks from many-core to host, a general command module is designed at the beginning of this chapter, to provide the task offloading flow as issuing commands from many-core to host. Then the design and implementation of DCFA is presented in Section 3.3, the DCFA-MPI is presented in Section 3.4.

3.2 Command Module

A general command module has been designed for issuing commands from the user application in many-core kernel to its assist daemon in host user space. Since it's designed for the task offloading in DCFA and DCFA-MPI, it's named DCFA command (DCFA CMD) in this thesis. A command client is designed on

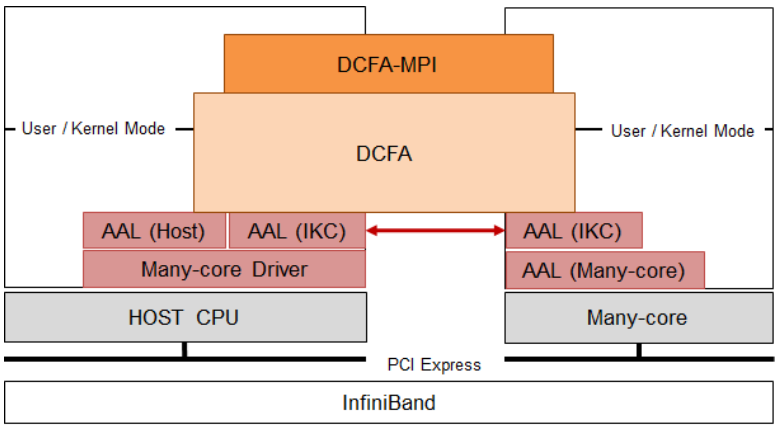


Figure 3.1: A system with DCFA and DCFA-MPI

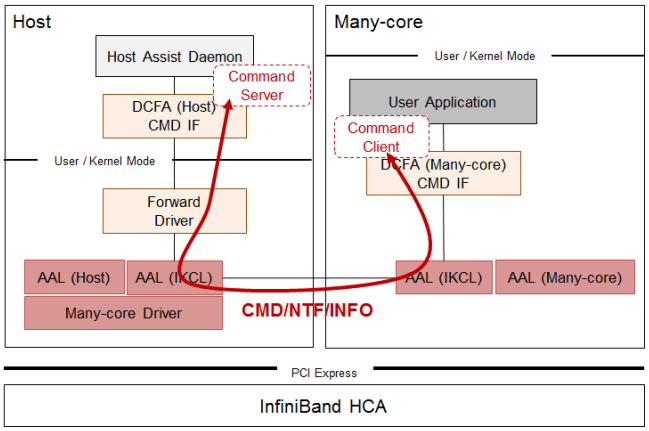


Figure 3.2: Implementation Components of DCFA CMD

the many-core side, and a command server is designed on the host side. Both sides are connected via a command channel. After the command channel established, the command client could issue commands to the command server, the command server performs the appropriate operation and returns the operating results. Three packet types are defined, command (CMD), notification (NTF) and information (INFO). The command holds the request from command client, the notification holds the operating result from command server, and the information is used to package extra parameters of a request or a result.

Every many-core application process creates its own command client, and connected to a dedicated host assist daemon including a command server and the corresponding operations for each command. User application should define the commands for both sides, and implement the corresponding operations in host assist daemon.

As shown in Figure 3.2, following three components are defined to implement the connection and communication between the command server and the command client.

- DCFA (Many-core) Command Interface
  This interface provides the functions to maintain a command client. A command client will wait the connection from command server using the `aal_ikc_listen_port` function after it is established. After the command channel established, it could send the commands to server side. For the commands that require parameters, the information packet will be sent together. If the command requires a result, the sending function will busy wait until the result from server side is returned, otherwise it will return immediately.

- Forward Driver
  The Forward Driver provides two functions, the command forwarding and kernel-level operating assistance. Because the AAL IKC only transfers messages in kernel mode, a driver that forwards messages to/from user space command server is required. It's inserted into host running kernel after the many-core kernel booted, and then it tries to connect to a command client on many-core using the `aal_ikc_connect` function. It also receives the registration from host command server using the Netlink Socket. When a pair of the connection to command server and the one to command client is prepared, a command channel is established. Then, forwarding starts. It

receives the commands from command client and forwards them to command server, and also forwards the notifications from command server to command client. If the received command needs kernel-level operating, the driver will first execute the corresponding kernel operations, and then forward the command and the operating result as a command parameter to user space command server.

- DCFA (Host) Command Interface
  This interface provides the functions to maintain a command server. Because the communication between command server and command client has to across the Forward Driver, the server will register to the Forward Driver using Netlink Socket, and then wait for the forwarding commands. When it receives a command, it will perform the appropriate operations, and then send the completion notification to the Forward Driver. For the notifications that require parameters, the information packet will be sent together. When it receives the exit command, it will unregister from Forward Driver and then exit immediately.

3.3 DCFA

3.3.1 Internal Implementation in InfiniBand

3.3.1.1 Internal Structure

Section 2.2.1 has introduced the queue-based communication model used in InfiniBand. To understand how the queues actually work, we studied the internal implementation of the Mellanox InfiniBand driver by reading its source code [4]. As shown in Figure 3.3(a), every queue consists of a queue buffer and a doorbell record. The details are described below.

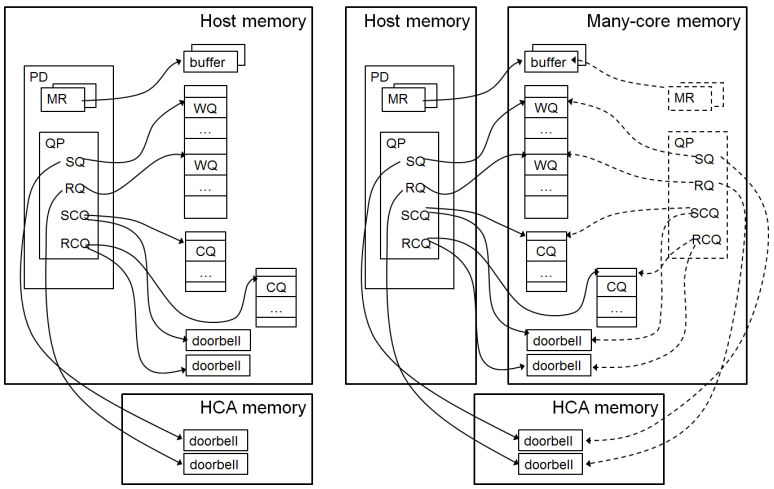
In a QP, the queue buffer is a virtually-contiguous memory buffer allocated in the QP creation process, containing the SQ and RQ, adjacently. Both the SQ and RQ are organized as circular buffers accessible by user-level software and the InfiniBand HCA; WQEs are placed here. The doorbell record is used to notify the InfiniBand HCA that a new WQE has been posted, and has been allocated in the PCI address space that is mapped for direct access to the HCA memory area from the CPU.

In a CQ, the queue buffer is a virtually-contiguous circular buffer accessible by user-level software and the InfiniBand HCA; CQEs are placed here. The doorbell record is a mechanism to implement the circular structure. Consider the InfiniBand HCA as a producer of CQEs and the user-level software as a consumer who consumes CQEs, the consumer should ring the doorbell to notify the producer after it has consumed a new CQE (It's not necessary to ring again if the consumed CQE is the same as the previous one.). The doorbell record is allocated in host memory.

3.3.1.2 Processing Sequence

A typical InfiniBand communication processing between two nodes has the following stages (Figure 3.4(a)).

1. Device configuration
   Configure the InfiniBand HCA by writing configuration commands to the PCI controller.



(a) in Original InfiniBand (b) in DCFA

Figure 3.3: Allocation of Internal Structures

2. InfiniBand configuration
   Set parameters for the configurations that will be used in the next stages (e.g., the maximum capacity of CQ, SQ, and RQ).

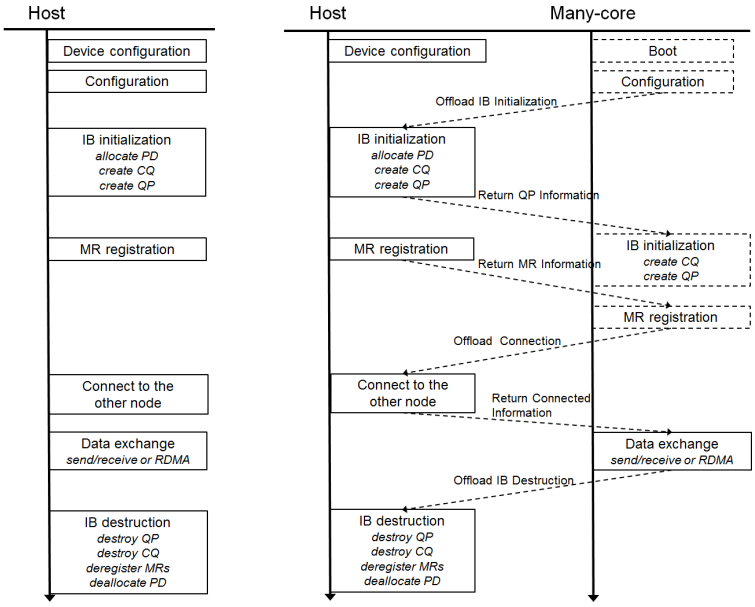
3. InfiniBand initialization
   Initialize a PD, a CQ, and a QP.

4. MR registration
   Register MRs for all the data buffers that will be used to send or receive data. For RDMA communication, a MR with RDMA capability (RDMA-able MR) is prepared.

5. Connection
   After all resources have been prepared, connect to the other node and exchange the relevant QP information. Then, the QP must be changed to the "Ready To Send" state before communication can be started. For RDMA communication, it's also required to exchange the information held by the RDMA-able MR, and give permission for the QP's RDMA operation when modifying the QP state.

6. Data exchange
   Data can be exchanged in either Send/Receive mode or RDMA mode.

   - Send/Receive mode
     Several previously prepared Receive WQRs are usually pre-submitted to the QP on both the receiver and sender sides before the connection stage. After the connection has been made, the receiver waits for a receive completion by polling the CQ, the sender posts a Send WQR to the QP and waits for its completion. Data is copied from the MR associated with the Send WQR to the one in the Receive WQR.
   - RDMA mode
     A host posts a Send WQR that describes an RDMA read/write operation, and then data is read from or written directly to the reserved RDMA-able MR on the remote host. After this operation, the host, issuing the RDMA request, can confirm its completion by polling its



(a) in Original InfiniBand (b) in DCFA

Figure 3.4: Processing Sequence

CQ, but there is no notification sent to the remote host. The remote host can monitor the last bit of the buffer in the RDMA-able MR to confirm the completion of data transfer.

7. InfiniBand destruction
   Finally, it's necessary to call destroy and deregister functions to release all resources.

3.3.2 Design for Many-core Architectures

DCFA is targeted to implement direct data transfer for many-core architectures. To be specific, the InfiniBand HCA must access the data buffers in many-core memory directly, moreover, the many-core must issue commands directly to the InfiniBand HCA.

3.3.2.1 Internal Structure

Based on the study of the internal implementation of the Mellanox InfiniBand driver, it's clear that if a many-core can access the queue buffers and the door-bell records directly, it can issue commands to an InfiniBand HCA. Since AAL, introduced in Section 2.1, has implemented both memory mapping from many-core memory to host memory and vice versa, it's possible to move these memory areas to the many-core side, and provide the mapped memory areas to the InfiniBand HCA, and in the same manner the data buffers allocated in the many-core memory also become accessible to the InfiniBand HCA. The software interface of DCFA has been designed and implemented on the basis of the above analysis. It consists of a modified Mellanox InfiniBand driver placed on the host system, and an InfiniBand communication interface on the many-core side. To keep a similar view of communication processing in many-core programs, the queue structures

are also defined in the many-core side interface. Figure 3.3(b) depicts the modified allocation of internal structures.

- QP
  The queue buffer is allocated in the memory area that has been pre-mapped to the many-core memory. The doorbell record allocated in the PCI address space mapped to HCA memory area is provided to the many-core side after mapping its memory location to a many-core memory location. The SQ, RQ structures, and the doorbell record are also defined in the QP structure in the many-core side interface, thus many-core programs can perform the same QP operations as on the host system.

- CQ
  Since the doorbell record is allocated as a host memory location, both the queue buffer and the doorbell record can be moved to the memory area that has been pre-mapped to many-core memory. In the many-core side interface, the CQ structure also holds pointers to them, and consequently, the CQ operations can be executed by many-core programs as usual.

The memory management mechanism specified by IBA (Section 2.2.4) is not designed for many-cores. The MR structure defined in the many-core side interface is only for keeping the same communication processing and holding the address information received from host.

3.3.2.2 Processing Sequence

Besides the internal structures, a communication scenario for many-core architectures has also been designed and implemented. The stages of an InfiniBand communication processing that were described in Section 3.3.1.2, can be divided into four parts: resource initialization, which contains the InfiniBand initialization stage and the MR registration stage; connection; data exchange; and resource destruction. The initialization and connection parts have to be offloaded to the host side due to their heavy overhead, and consequently, the destruction part is also offloaded. Thus, the only part to be executed on the many-core side is the data exchange part. Although the InfiniBand initialization and MR registration stages are also defined in the processing on the many-core side, they only create similar internal structures, no command is issued to the InfiniBand HCA. The communication scenario for many-core architectures can be described as consisting of the following stages (Figure 3.4(b)).

1. Device configuration
   The host configures the InfiniBand HCA by writing configuration commands to the PCI controller.

2. InfiniBand configuration
   The many-core sets the configurations such as max send WQ and max CQE attributes for InfiniBand structure initialization, and then offloads the InfiniBand initialization task with the configurations to the host.

3. InfiniBand initialization
   After receiving the offloading request from the many-core, the host executes this stage. The memory allocation is performed as described in Section 3.3.2.1. The QP initialization information which will be used in the data exchange stage is returned to the many-core when completed. After

receiving the QP information from the host, the many-core initializes its structures.

4. MR registration
   The host registers all the MRs while the many-core is executing initialization, then the MR registration information which will be used in data exchange stage is returned to the many-core. After receiving the information from the host, the many-core creates its copy of MRs.

5. Connection
   When the many-core decides to connect to the other node, it offloads the connection task to the host, and waits for the connected information.

6. Data exchange
   After receiving the connected information, the many-core can start to exchange data. Both the Send/Receive mode and RDMA mode are supported. Since all the queue structures have also been prepared in the many-core side interface, the processing in both modes is exactly the same as processing in the host.

7. InfiniBand destruction
   The many-core offloads the destruction task to the host before the program exits. The host is responsible for releasing all the InfiniBand resources.

3.3.3 Implementation

In the design of DCFA processing sequence described in Section 3.3.2.2, a task offloading mechanism is required for the initialization, connection and destruction parts. Moreover, as described in Section 3.3.2.1, DCFA requires the memory mapping from many-core memory to host memory in order to move InfiniBand internal structures and data buffers to many-core side, and the mapping from host memory to many-core memory for the doorbells which are allocated in the PCI address space mapped to HCA memory area. Since the task offloading mechanism can be implemented using the DCFA CMD module, and the memory mapping functions have already been provided by the AAL, a DCFA can easily implement these requirements. Following two sections describes their detailed usage, and then Section 3.3.3.3 presents the components defined in DCFA.

3.3.3.1 Task Offloading

DCFA implements the task offloading from many-core to host following the flow provided by DCFA CMD module (Section 3.2). Every process of the many-core InfiniBand communication application has a dedicated host assist daemon. The InfiniBand initialization, connection and destruction tasks are offloaded to the dedicated host assist daemon. The corresponding commands for these offloaded tasks are defined in both many-core application and dedicated host assist daemon, and the operations corresponding for each command is defined in dedicated host assist daemon.

A command server and a command client are created on the host side and the many-core side respectively. After the command channel established, a task can be offloaded to host assist daemon by issuing a corresponding command, the host operating result is returned as a notification. If the offloading command or result needs extra parameters, the parameters will be packaged into a information packet and be sent out together.

3.3.3.2 Memory Map between Host and Many-core

On the host side, AAL(Host) provides the `aal_device_map_memory` function and `aal_device_unmap_memory` function to map/unmap a many-core physical address to a host physical address in kernel mode, then user can access this memory area in user mode by the `mmap` function (Listing 3.1 and 3.1). Many-core initializes a shared memory area which is the area mapped from many-core memory to host memory following the task offloading flow. The mapping operation in host user mode is the corresponding handling, the operation in host kernel is defined as a kernel-level operating assistance for this command.

Listing 3.1: Mapping many-core physical memory area to host physical memory area

```
unsigned long long      host_phys_addr =
    aal_device_map_memory(dev, manycore_phys_addr, size);
```

Listing 3.2: Mapping host physical memory area to host virtual memory area

```
int fd = open("/dev/mem", O_RDWR);
void *v_addr = mmap(NULL, size, PROT_READ | PROT_WRITE,
    MAP_SHARED, fd, host_phys_addr);
```

On the many-core side, AAL(Many-core) also provides the `aal_mc_map_memory` function and `aal_mc_unmap_memory` function to map/unmap a host physical address to a many-core physical address, then user can convert it to virtual address using `aal_mc_map_virtual` function (Listing 3.3).

Listing 3.3: Mapping host physical memory area to many-core virtual memory area

```
unsigned long long manycore_phys_addr = aal_mc_map_memory(
    NULL, host_phys_addr, PAGE_SIZE);
void *v_addr = aal_mc_map_virtual(manycore_phys_addr, 1,
    PTATTR_UNCACHABLE);
```

The corresponding components for memory mapping will be introduced in Section 3.3.3.3.

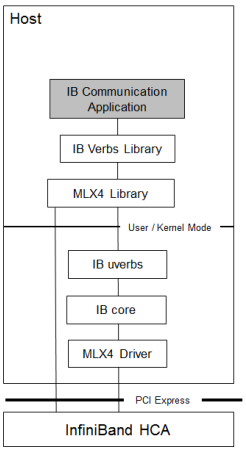
3.3.3.3 Components in DCFA

In original InfiniBand Verbs (IBV), user application calls the functions provided by IBV Library, then go into the vendor specified implementation though the operation pointers which are registered while opening device (Figure 3.5(a)). The resource management operations such as structure creation and memory registration, which should be performed in kernel mode, are transmitted to kernel modules though the InfiniBand uverbs module. The IO operations such as posting send / receive or polling completion can be directly transmitted to the InfiniBand HCA from user mode.

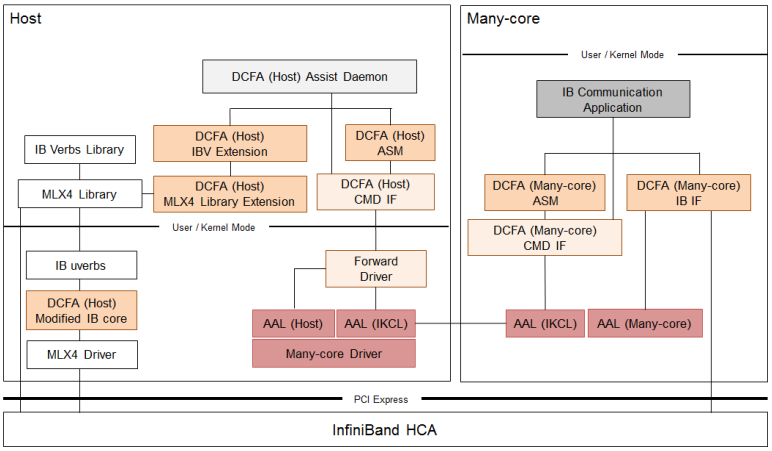
In order to implement the internal structure modification described in Section 3.3.2.1, several components are defined and distributed on both host side and many-core side (Figure 3.5(b)). Following components are defined on host side.

- DCFA (Host) Assist Daemon

  The Assist Daemon handles the resource initialization, connection and resource destruction for many-core direct communication as described in Section 3.3.2.2. When it starts, it will create a command server, and then



(a) in Original InfiniBand



(b) in DCFA

Figure 3.5: Implementation Components of DCFA

wait for the offloaded tasks. When it receives an offloading command, it will perform the appropriate operations, then return the operating result. When it receives the exit command, it will destroy the command server and then exit immediately.

- DCFA (Host) AAL Shared Memory

  The DCFA (Host) AAL Shared Memory (ASM) component is responsible for the memory sharing between many-core and host. Since the command server is established after the assist daemon started, the ASM will be initialized while receiving a memory sharing offloading command. In the host initialization processing, the shared memory area is mapped from the received host physical address, and a structure to manage its allocation/freeing is initialized. Thus the buffers can be easily allocated/freed from the shared memory in later phases.

- DCFA(Host) InfiniBand Library Extensions

  When the host daemon receives the initialize command, it will create a PD, CQ and QP. In order to make these structures also controllable by many-

core, their buffers have to be allocated from the memory area mapped from many-core memory (shared memory). To define a unified interface to make the buffer allocation and destruction can be specified by user application, following extended creation functions have been defined (Listing 3.4). The parameter `pm_ops` defines the buffer allocation operation and freeing operation. The host daemon specifies the allocation operation to allocate buffers from the shared memory, and freeing operation to release the buffer which may be allocated from shared memory or common memory using its appropriate method. The call stack of the extended functions also follows original Verbs, user only touches the IBV layer, actual implementation is defined in vendor library. Therefore, DCFA (Host) IBV Extension provides the function interfaces in IBV layer; DCFA (Host) MLX4 Library Extension implements them according to the original creation functions.

Listing 3.4: Extended functions in IBV

```
struct pm_buf_ops {
        void* (*alloc_buf)(int size);
        void (*free_buf)(void *buf, int size);
};
...
/*The extension for creating CQ with specified buffer
    allocation/freeing operations*/
extern struct ibv_cq *ibv_pm_create_cq(struct
    ibv_context *context, int cqe, void *cq_context,
    struct ibv_comp_channel *channel, int comp_vector,
    struct pm_buf_ops pm_ops);

/*The extension for creating QP with specified buffer
    allocation/freeing operations*/
extern struct ibv_qp *ibv_pm_create_qp(struct ibv_pd *pd
    , struct ibv_qp_init_attr *qp_init_attr, struct
    pm_buf_ops pm_ops);
```

- DCFA(Host) Modified InfiniBand core

The original IB core module uses the get_user_pages function to make user space buffers, such as the queue buffer of QP or CP, be accessible in kernel mode. However, this function cannot handle the memory area in PCI devices such as the shared memory area mapped from many-core memory. Thus it's required to distinguish whether a virtual address of a buffer points to host-only memory or shared many-core memory. The modification is shown as following codes.

Listing 3.5: Modification in IB core

```
/* in file drivers/infiniband/core/umem.c */
struct ib_umem *ib_umem_get(struct ib_ucontext *context,
    unsigned long addr,
                            size_t size, int access, int
                                dmasync)
{
  ...
  /*get the vma structure of the buffer*/
  struct vm_area_struct * vma = find_vma(current->mm,
     cur_base);
```

```
  /*check if it's allocated from device memory*/
  int _is_io_vma = vma!=NULL && vma->vm_flags & (
     VM_PFNMAP|VM_IO);

  if(_is_io_vma){
    /*get the physical address of each page using vma->
       vm_pgoff*/
    ...
  }else{
    get_user_pages(...);
    ...
  }
}
```

On many-core side, following components are defined.

- DCFA (Many-core) AAL Shared Memory

  The DCFA (Many-core) AAL Shared Memory (ASM) component is responsible for the memory sharing between many-core and host. The memory range mapped to host is defined here. It sends the memory sharing offloading command with the memory range information to command server on the host side, then wait for the mapping success. The same structure used to manage the memory allocation/freeing is created, thus it ensures the consistency of the buffer allocated memory areas on both sides.

- DCFA(Many-core) InfiniBand Interface

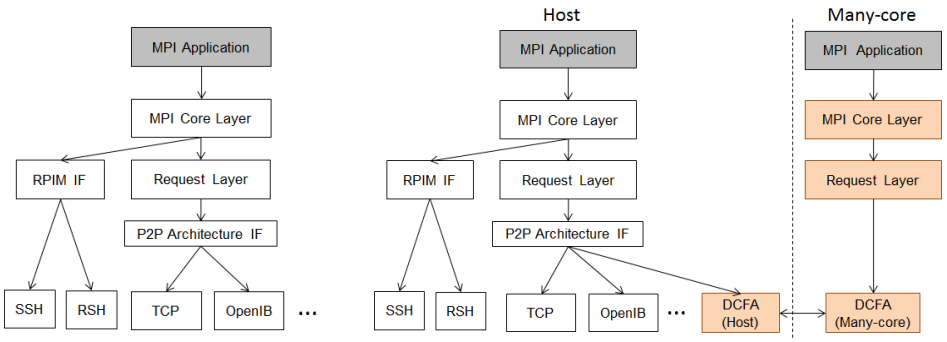
  This interface defines the same InfiniBand operations as on host system. Since the resource structures have been already initialized on host side, the creation operations only create similar structures but do not send commands to HCA. As explained in Section 3.3.2.1, the posting send/receive and polling completion can be also transmitted to HCA directly.

3.4 DCFA-MPI

3.4.1 Design Overview

DCFA-MPI is a light-weight MPI library over the DCFA, to implement direct point-to-point communication between many-core accelerators. It provides a parallel programming environment on the many-core based cluster, in which the many-core units are connected to host CPU via PCI Express and used as floating-point accelerators. DCFA-MPI is based on the YAMPII introduced in Section 2.3 and integrated with the DCFA which provides direct InfiniBand communication for many-cores, the MPI software structure will be introduced in Section 3.4.1.1.

MPI application involves different kinds of operations. The thesis approach is to move the point-to-point communication operations to many-core so that extra cost is minimized. However, many-core has less resource when compared to the host CPU. For example, its cache memory size and memory bandwidth is smaller than those of the host CPU. Therefore, moving all of the operations to the many-core reduces the performance, some operations have to be offloaded to host CPU. The task offloading design will be described in Section 3.4.1.2.



(a) in YAMPII (b) in DCFA-MPI

Figure 3.6: MPI Implementation Structure

3.4.1.1 Software Structure

The DCFA, which provides InfiniBand communication between many-cores, has been integrated into DCFA-MPI as another P2P communication module under the P2P Architecture interface (Figure 3.6(b)).

Because the InfiniBand resource initialization / destruction have to be performed by host CPU, this DCFA implementation is separated to the host part DCFA (Host), and the many-core part DCFA (Many-core). The upper layers are similar with host side, except that the P2P Architecture Interface is merged into DCFA (Many-core) because there is no other communication architecture can be supported. The details of DCFA layer will be introduced in Section 3.4.2.2.

3.4.1.2 Offload Design

A common MPI application always includes several tasks during runtime. They are the process launching task, the process connection task, the floating-point calculation task, the P2P communication task, the collective communication task, and the file or standard I/O task. The many-core only handles the direct point-to-point communication through DCFA, other heavy-weight functions, such as collective communication and communication using user defined data types, are offloaded to the host CPU. Because many-core has less resources when compared to the host CPU, moving all of the operations to the many-core reduces the performance.

Following tasks are decided to be offloaded to host CPU.

- Process launching
- File or standard I/O
- Collective communication
- Communication using user defined data types

Following tasks are handled by many-core.

- Floating point calculation
- P2P communication

3.4.2 Implementation

3.4.2.1 Task offloading

The same with DCFA, DCFA-MPI also uses the DCFA CMD module (Section 3.2) for task offloading. Every process of the many-core MPI application has a dedicated host assist daemon. The dedicated host assist daemon handles all the tasks offloaded to host CPU. The corresponding commands for these offloaded tasks are defined in both many-core application and dedicated host assist daemon, and the operations corresponding for each command is defined in dedicated host assist daemon.

Listing 3.6 shows an example of MPI application, which contains the following 6 steps: i) MPI resource initialization, ii) Reading input parameters, iii) floating-point computing, iv) P2P communication, v) Console output, vi) MPI resource destruction. After compiling with DCFA-MPI, a command is sent to host daemon to get input parameters before MPI application begin. Then go into the MPI application, the step i) is offloaded to host by issuing command to host assist daemon, the result and initialization information are returned. At the step ii), since the parameters have been received before step i), many-core can read the parameters directly. The step iii) and iv) are performed by many-core. The step v) is offloaded to host, the string is printed at host console. Similar with step i), the step vi) also performs the offload mode.

Listing 3.6: A Common MPI application

```
...
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

count = argv[1];
...
for( ; ; ){
  /*floating-point computing*/

  MPI_Isend(sendbuf, count, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD
      , &reqs[0]);
  MPI_Irecv(recvbuf, count, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD
      , &reqs[1]);
  MPI_Waitall(2, reqs, status);
}
if(my_rank == 0){
  printf("finish\n");
}
MPI_Finalize();
```

3.4.2.2 Components in DCFA-MPI

DCFA-MPI defines the DCFA as an architecture specified P2P communication layer, Figure 3.7 shows its detailed components.

- DCFA-MPI (Host) Assist Daemon

  This component is similar with the DCFA (Host) Assist Daemon described in Section 3.3.3.3. Each MPI process will start a daemon to assist the many-core process in managing MPI resources. When it starts, it will create a

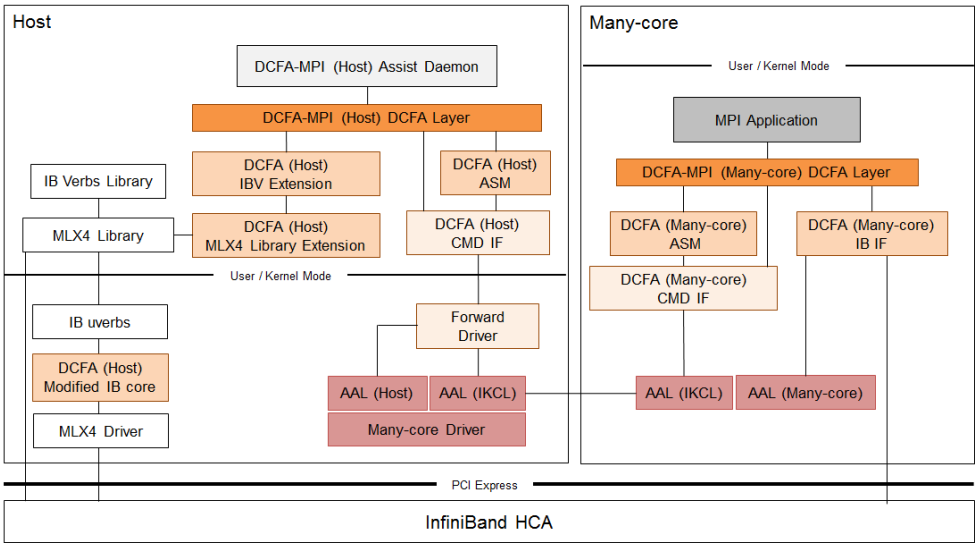


Figure 3.7: Implementation Components in DCFA-MPI

command server, and then wait for the offloaded tasks. When it receives an offloading command, it will perform the appropriate operations, and then return the operating result. When it receives the exit command, it will destroy the command server and then exit immediately.

- DCFA-MPI (Host) DCFA Layer

  The same with other host only P2P layers, the architecture specified communication should be implemented in this layer. For the InfiniBand communication, this layer performs resource initialization/destruction and data transfer. For many-core to many-core communication, the resource initialization/destruction should be performed on host side, therefore, the main task of this component is to initialize/destroy the InfiniBand resources in the shared memory area mapped from many-core memory. Following the implementation of DCFA described in Section 3.3.3, this component should also handle the task offloading and memory mapping. The task offloading is implemented by using DCFA (Host) CMD interface, and the memory mapping is implemented by using the DCFA (Host) ASM module. The detailed processing will be introduced in Section 3.4.2.3. In addition to the tasks for many-core communication, the original host communication is also implemented in order to provide host to host InfiniBand communication.

- DCFA-MPI (Many-core) DCFA Layer

  This layer performs almost the same operations defined in host only P2P layers. However, the buffers such as the InfiniBand internal queue buffer of QP and CQ, and several global buffers which are defined according to the communication protocol, are allocated from the shared memory but not normal many-core memory. The detail of the communication protocol will be discussed in Section 3.4.2.4.

3.4.2.3 The Process from Library Installation to Application Excution

- Library Installation

  The YAMPII Library's installation process generates the static MPI library and the user commands such as mpicc and mpirun. In DCFA-MPI, two static MPI libraries are generated, one for host MPI applications, one for many-core MPI applications. The hos static library is used to link to an executable file for host assist daemon in later installation process. The many-core static library includes not only the MPI objects but also the necessary objects to form a many-core kernel image.

- mpicc

  This command compiles the user source code to an executable MPI application. For DCFA-MPI, the MPI application executable file is actually a kernel image. The mpicc complies the user source code to an object file and then linked with the many-core static library to generate a kernel image.

- mpirun

  This command decides the policy to launch the MPI processes. In DCFA-MPI, it includes the following four steps.

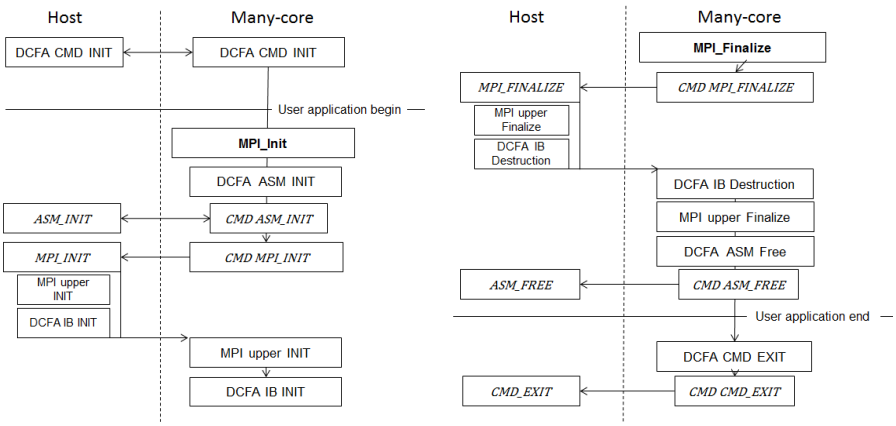
  1. Inserts AAL modules to host kernel to create a many-core device file on each host.
  2. Boots the kernel image generated by mpicc command on each host.
  3. Inserts the Forward Driver to the host kernel on each host.
  4. Launches an assist daemon on each host.

- Inside the Application Excution

  Following pseudo code shows the call stack of an MPI application runs on many-core. When the many-core kernel is loaded and started a compiled image, it first goes into the `main` function which is the entry of the whole many-core kernel. In this function, it first initializes the basic framework of the kernel, and then goes into the user application entry, the `user_main` function. In the `user_main` function, it first initializes a command client which holds a command channel connected to the command server on host side, and then it gets the user input arguments from host side through the command channel and stores them into the variable *argc* and *argv*. These two variables are passed into the entry of MPI application, the `__main` function. After the MPI application returns, the command client exits at the user application layer.

Listing 3.7: Pseudo code of an DCFA-MPI Application

```
/*The entry of many-core kernel*/
int main(void){
  /* kernel initialization */
  ...
  user_main();
```



(a) Initialization (b) Finalization

Figure 3.8: Internal Process Sequence of DCFA-MPI

```
  return 0;
}

/*The entry of user applications*/
int user_main(void){
  int argc = 0, ret;
  char **argv;

  CMD_INIT();
  GET_ARG(&argc, &argv);
  ret = __main(argc, argv);
  CMD_EXIT();
  return ret;
}

/*The entry of MPI application */
int __main(int argc, char **argv) {
  MPI_Init(&argc, &argv);
  ...
  MPI_Finalize();
  return 0;
}
```

- Application Initialization

  Figure 3.8(a) shows the initialization processing. The command module is initialized before the MPI application starts so that later tasks can be easily offloaded to host side. The `MPI_Init` function first initializes the ASM structure, then sends the `MPI_Init` command to start the actual initialization work on host side. After it receives the rank information and some necessary information of the InfiniBand resources, it starts the process to create its own structures for communication.

- Application Finalization

  At the end of an MPI application, the `MPI_Finalize` functions is called to release all the structures for MPI communication (Figure 3.8(b)). In con-

trast to the `MPI_Init`, the host finalization is first performed, and followed by the InfiniBand resource destruction, upper MPI management structure finalization. The ASM is freed at last. After leaving the MPI application, the command client exits.

3.4.2.4 MPI Implementation

- Memory Registration

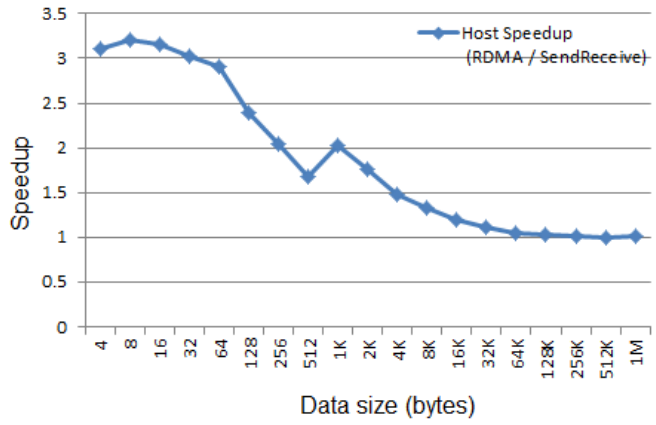
  In InfiniBand implementation, memory registration is required for any memory access such as local read/write, remote read/write. If the buffers used for data transferring are registered/deregistered every time in advance of their use, this part become relatively expensive because this operation locks the memory pages to preserve the virtual to physical memory mapping. Current MPI implementations have been discussed this issue. In the MPICH2 [5], a registration cache [38] is used to reduce the number of registrations and deregistrations [25]. It delays the deregistration and put the buffer into a cache, if the same buffer is reused later, the registration process can be avoided. In the MVAPICH [6], which is currently the most widely used MPI implementation over InfiniBand, the same cache technique and the way to use a pre-registered pool are discussed [37] [27]. The OpenMPI [8] designs a MPool and a Rcache components which provide the memory allocation/deallocation and registration/deregistration service, and the registration cache service respectively.

  In the DCFA-MPI, similar design was discussed. But the cost of registration is much higher than original host operation because the offloading process also includes the overhead of command transferring. Thus, a simple way is chosen for the experimental implementation. A huge pre-registered pool allocated from the shared memory, called USER buffer, is prepared at the initialization stage. Application has to allocate the buffers used for send/receive from this pool. This pool manages the allocation/freeing by itself to avoid the physical address change, and the real release only happens at the MPI finalization stage.

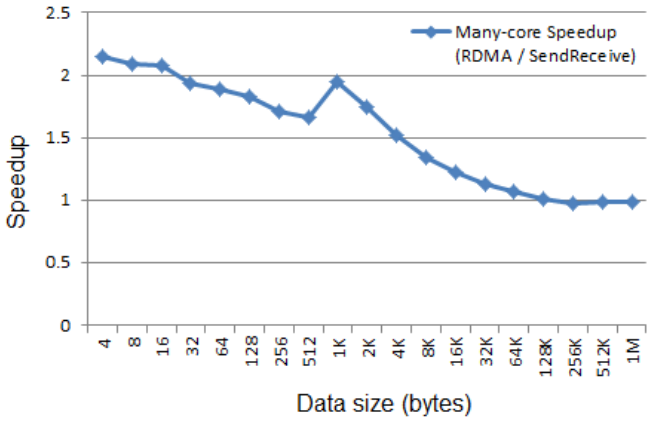
- Communication Protocol

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

  InfiniBand supports both send/receive operations and RDMA write/read operations for communication. Compared with send/receive operations, the RDMA operations has better performance, but the destination buffer address and the Remote Key of its MR are required to be known before the communication. Figure 3.9(a) shows the speedup (RDMA write RTT



(a) Host Speedup

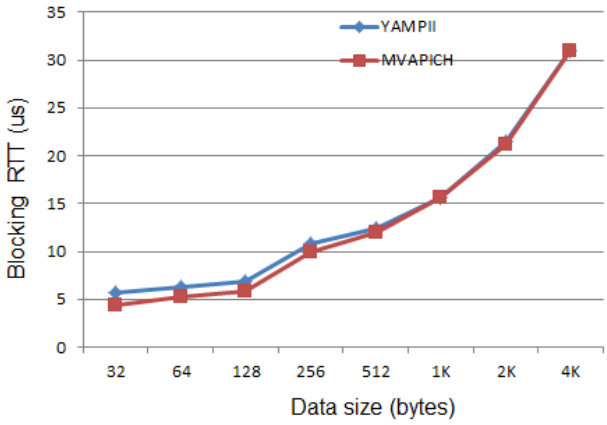


(b) Many-core Speedup

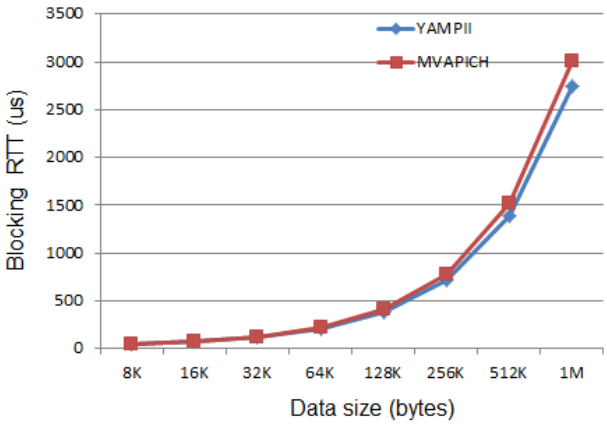
Figure 3.9: Comparison between RDMA write and send/receive

/ send/receive RTT) on hosts, and Figure 3.9(b) shows the value on many-cores implemented by using DCFA. The send/receive is slower than RDMA write for smaller messages, but the gap gradually become smaller with the increase of message size and reaches the same speed from 128Kbytes.

The YAMPII uses RDMA write for smaller data transfer and send/receive for large data transfer, Three global buffer are pre-allocated and pre-registered, the RDMA buffer is used for receiving RDMA packets as a ring buffer, the SEND buffer is used for providing send packets, and the RECEIVE buffer is used for allocating receive packets. In the MPI initialization process, QPs are created and connected for each remote rank, a part of the RDMA buffer is distributed to each connection, and the address, the MR Remote Key and other information to maintain a ring buffer of the RDMA buffer are exchanged. Then several receiver packets are allocated from the RECEIVE buffer and pre-posted to shared receive queue. In communication stage, eager protocol is used for small data transfer, a send packet holding the header information and user data is allocated from the SEND buffer, user data is copied into this packet, then this packet is send to the correct offset of receiver's RDMA buffer whether receiver starts to receive or not. Receiver polls the RDMA buffer until new packet arrived, then it copies data to user buffer. Sender can confirm the completion



(a) Blocking RTT for small messages



(b) Blocking RTT for large messages

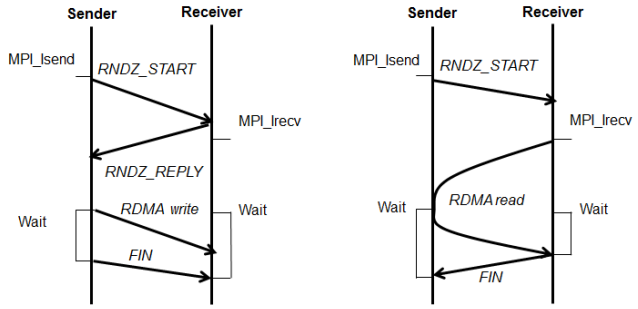
Figure 3.10: Comparison between YAMPII and MVAPICH

by polling the CQ. The send/receive communication is used for large data transfer, the send packet is posted to the send queue, the receiver polls CQ to check if new packet has been arrived. The copy step and the post send step are pipelined by splitting the data to fixed blocks.

Figure 3.10 shows the comparison with MVAPICH, which uses the RDMA write instead of send/receive for large data transfer. The result proves that, for larger messages, using send/receive operations can get the same performance as RDMA write.

However, DCFA-MPI is aimed to implement a zero-copy design. Because the many-core has a lower CPU frequency than host CPU, data copy on many-core may be more expensive than the copy on host CPU.

Since user send/receive buffers can be allocated from the USER buffer to avoid frequent memory registration, the sender side can easily eliminate the data copy from user send buffer to the packet allocated from SEND buffer. Because InfiniBand already provides the scatter/gather technique to transfer data from multi-sources to one destination, the header part can still use the packet allocated from SEND buffer and set as the first scatter/gather element (SGE), then the data part can directly point to the user send buffer and set as the second SGE.



(a) Two-way handshake followed by RDMA write

(b) one-way handshake followed by RDMA read

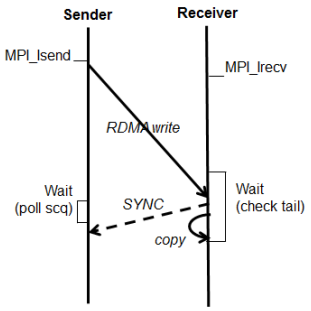
Figure 3.11: Communication Protocols in Current MPI Implementations

It's clear that the send/receive mode cannot achieve better performance by eliminating the copy in receive process. Because, if sender first, it has to wait receiver to post receive the prepared user receive buffer, and then posts send its user send buffer, if receiver first, even if it posted the receive buffer, it has to notify sender side that it's OK to post send now, in other words, the post send/receive operation cannot overlapped in both situations. Thus the RDMA communication mode is considered. Since all communication are changed to RDMA mode, the RECEIVE buffer and shared receive queue can be eliminated.

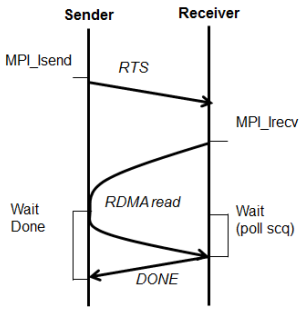
RDMA communication mode contains the RDMA write operation, which writes data from local SGEs to remote buffer, and the RDMA read operation, which reads data from remote buffer to local SGEs. Current MPI implementations use several kinds of rendezvous protocol using either RDMA write or RDMA read, to hide communication latency by overlapping computation with communication. MVAPICH [37] first showed a two-way handshake followed by RDMA write (Figure 3.11(a)), and then optimizes to a one-way handshake followed by RDMA read (Figure 3.11(b)). As explained in this work [37], the RDMA read one achieves better overlapping of computation and communication. MPICH2 [25] also shows a similar protocol using RDMA read in its RDMA Channel. However, the RDMA read modes can only benefit the sender first situation, because even if the receiver arrives first, it has to wait for sender's control packet and then RDMA read the data from sender side.

DCFA-MPI uses another rendezvous protocol [21] in which both RDMA write and RDMA read are used. Following control packet types are defined:

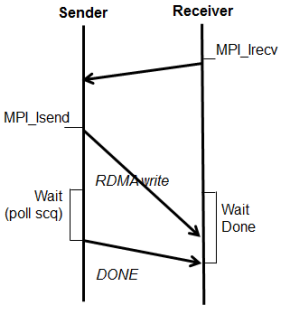
- EAGER packet
  It consists of the MPI header part and followed by the data in user send buffer. A magic is added in the tail of the data part to confirm the completion of data transfer. This packet type is used in Eager protocol.
- Request to Send (RTS) packet
  It consists of the MPI header part and address data of user send buffer. It's used in rendezvous protocol when sender arrives first.
- Request to Receive (RTR) packet
  It consists of the MPI header part and the address data of user receive



(a) Eager Protocol



(b) Sender First Rendezvous Protocol



(c) Receiver First Rendezvous Protocol

Figure 3.12: Communication Protocols in DCFA-MPI

buffer. It's used in rendezvous protocol when receiver arrives first.

– DONE packet
  It only consists of the MPI header part. It's used in rendezvous protocol when sender or receiver finished its RDMA write or RDMA read.
– SYNC packet
  It only consists of the data used to synchronize the credit information of RDMA buffer. The detail will be described in Section 3.4.2.4.

Using above control packet types, DCFA-MPI utilizes the following four communication protocols.

– Eager Protocol
  Figure 3.12(a) shows the process of Eager protocol. The sender decides to use eager protocol for small data transfer. It will RDMA write the EAGER packet and the data to the RDMA buffer in receiver side, and poll the send CQ to confirm its completion. The receiver will wait for the completion of data transfer by checking the tail of received packet if the packet is EAGER type, then copy data to user receive buffer.
– Sender First Rendezvous Protocol
  The Sender will send a RTS packet including its buffer address and the MR Remote Key to receiver, then waits the receiver's DONE packet. When the receiver starts to receive, it must receive the RTS packet, then it starts a RDMA read by using the buffer information in RTS packet, it will send a DONE packet to the sender when the read operation is finished (Figure 3.12(b)).
– Receiver First Rendezvous Protocol
  The receiver will send a RTR packet including the user receive buffer

information to sender, then waits the DONE packet. Sender receives the RTR packet, and then it starts a RDMA write by using the buffer information included in the RTR packet. After the RDMA write is finished, the sender will send a DONE packet to the receiver (Figure 3.12(c)).

- Simultaneous Send/Receive Rendezvous Protocol
  In this situation, the sender will receive a RTR packet after it sent a RTS; the receiver will also receive a RTS packet after it sent a RTR. The sender will disregard the RTR and still wait for receiver's RDMA read, the receiver will RDMA read by using the buffer data included in RTS packet as the process of Sender First Rendezvous Protocol.

- Credit Flow

As introduced in above section, both the Eager data packet and the control packets consume receiver's RDMA buffer. The RDMA buffer is divided into fixed size blocks, each connection can holds a fixed number of the blocks, called credit. When a sender sends out a packet, it will consume a block in receiver's RDMA buffer, and its credit is decremented. When the receiver has finished the processing of a packet, this block become empty, and the credit for the corresponding sender will be incremented. However, since the latest information about the credits is only known by the receiver, a credit flow mechanism to synchronize it is required.

For experimental purpose, DCFA-MPI uses a simplified version of the mechanism introduced in paper [27] [26]. The credit information of local RDMA buffer is added into the header part of each packet, to notify the other side about credit status. If the communication pattern is symmetric, each sender can always update its credit to achieve continuous communication. However, if there is in an asymmetric pattern, the sender may be blocked because all local credits are consumed, thus the receiver must send a explicit packet to notify sender side the latest credit. This explicit packet is defined as SYNC packet in DCFA-MPI.

DCFA-MPI defined a parameter called $MaxCredit$ to determine the maximum number of credits for each connection, and another parameter called $CreditSyncLimit$ to decide when to send a SYNC packet to the other side. For each connection, a process has a set of $<offset_{local}, limit_{local}>$ and a set of $<offset_{remote}, limit_{remote}>$. When the process is a receiver, it uses $<offset_{local}, limit_{local}>$ to represent the credit status of its local RDMA buffer. $offset_{local}$ is the offset of the last received block, $limit_{remote}$ is the maximum offset can be received. When the process is a sender, it uses $<offset_{remote}, limit_{remote}>$ to represent the credit status of remote RDMA buffer. $offset_{remote}$ is the offset of the last sent block in remote RDMA buffer, $limit_{remote}$ is the maximum offset can be sent. Thus current credit number of remote RDMA buffer can be set as follow.

$$credit_{remote} = (limit_{remote} + MaxCredit - offset_{remote})\%MaxCredit$$

For any connected pair of processes, the credit flow control protocol is described as follow.

1. Sets the $MaxCredit$ to N, and $CreditSyncLimit$ to X.

2. When a connection is established, sets both $<offset_{local}, limit_{local}>$ and $<offset_{remote}, limit_{remote}>$ to $<-1, N-1>$.

3. In sending processing, if $offset_{remote} == limit_{remote}$, then the process cannot send any packets. Otherwise adds the value of $credit_{remote}$ into packet and sends packet out, then sets $offset_{remote} = (offset_{remote} + 1)\%MaxCredit$ after sending.

4. After the process receives a packet, updates the local credit status as follow.

   $offset_{local} = (offset_{local}+1)\%MaxCredit, limit_{local} = (limit_{local}+1)\%MaxCredit$

   If $packet.credit_{remote} < X$, then sends a SYNC packet to the other process.

5. The process adds the value of $limit_{local}$ into every sending packet.

6. The process sets $limit_{remote} = packet.limit_{local}$ for every received packet.

Following this protocol, in a rendezvous communication, the $limit_{remote}$ is always updated in both processes, because one process can be updated by the received RTS/RTR packet, the other one can be updated by the returned DONE packet. In the symmetric eager communication, the $limit_{remote}$ is also updated in both processes, because both of them can receive the EAGER packet. In the asymmetric eager communication, the receiver process will send a SYNC packet to the sender process if its $credit_{remote}$ is smaller than $CreditSyncLimit$.

User is responsible for setting the parameter $MaxCredit$ and $CreditSyncLimit$, because the best value of these parameters may not be the same for different system environment and different communication patterns used in target applications. If the $MaxCredit$ is too small, the SYNC packets will be sent frequently, but if it is too large, the memory consumption may become a problem. If the $CreditSyncLimit$ is too small, the sender may have to wait for the SYNC packet for a long time in a high latency network, but if it is too large, the SYNC packets will be sent frequently. The memory consumption problem will be discussed in next section, and the tuning of parameter $CreditSyncLimit$ will be discussed in Section 4.3.1.3.

3.4.3 Scalability Discussion

Although current DCFA-MPI only provides the communication between two many-core units, it is aimed at perfecting the communication over post peta-scale supercomputer with close to a million processors. Thus current scalability limitations and available solutions are discussed here.

3.4.3.1 Queue Pair Connection

Current DCFA-MPI sets up all-to-all QP connections at MPI initialization stage. Since the number of MPI processes may be up to $O(1K)$ or even $O(1M)$, here is

Table 3.1: Comparison of IBA Transport Service Types

	RC	RD	UC	UD
Scalability (Number of QPs)	$O(p^2)$	$O(p)$	$O(p^2)$	$O(p)$
RDMA Operations	yes	yes	no	only RDMA write

The origin of this table is shown in [11]

certainly a runtime bottleneck because this is an $O(p^2)$ operation. The solutions have already been discussed in several papers [30] [36] [18] [15]. The paper [30] [36] introduce a lazy connection strategy, which establish a connection only when a process needs communicate whit another, it especially benefits the applications which haven't all-to-all communication. The paper [18] introduces a locality-aware connection management based on the lazy connection establishment which can also performs good performance with less established connections on even large-scale communication applications, The paper [15] describes another method using the InfiniBand Unreliable Datagram transport (UD) (Table 3.1), in which one QP can be connected with multiple QPs, to reduce the connection resources. However, UD does not support the RDMA features, which deliver much better performance for smaller messages and are needed for zero-copy rendezvous design. Moreover, the Reliable Datagram transport (RD) may also be a possible solution since it supports both $O(p)$ connection and RDMA features.

3.4.3.2 Memory Consumption

A DCFA-MPI communication endpoint, which is associated with a <rank, communicator> pair, always contains a global CQ, a global SEND buffer, a global USER buffer and several connections (Figure 3.13). Each connection contains a QP and a RDMA buffer. As introduced in Section 3.3.1.1, both the CQ and the QP contain a queue buffer and the buffer size is defined by the initialization parameters specified by user application. The SEND buffer, USER buffer and RDMA buffer are pre-allocated and pre-registered buffers, the SEND buffer is used for allocating various control packets, the USER buffer is a experimental design for avoiding dynamic memory registration in many-core side and is used for allocating user send/receive buffers, the RDMA buffer is used for receiving various control packets and the eager send data. Current DCFA-MPI prepares all of them in the MPI initialization.

Let's assume there are p processes and each process only contains one endpoint, the memory size of these buffers is $Buffer_{send}$, $Buffer_{user}$ and $Buffer_{rdma}$ respectively, the memory size of the queue buffers inside CQ and QP is $Buffer_{cq}$ and $Buffer_{qp}$. Therefore, the total memory consumption per process can be described as follow.

$$Buffer_{cq} + Buffer_{send} + Buffer_{user} + p \times (Buffer_{rdma} + Buffer_{qp})$$

The size of $Buffer_{cq}$ and $Buffer_{qp}$ are decided by InfiniBand depending on the InfiniBand configuration specified by user application. We measured the buffer size for the InfiniBand configuration listed in Table 3.2, the $Buffer_{cq}$ is 4Kbytes, and the $Buffer_{qp}$ is 12Kbytes. We assume the $Buffer_{send}$ and $Buffer_{user}$ are fixed to 16Kbytes and 1Mbytes, and then assume the $MaxCredit$ is 64 and each block size of the RDMA buffer is 8Kbytes, thus the $Buffer_{rdma}$ is equal to 512Kbytes. Using above value, for large scale process applications, the trend of memory consumption per process can be predicted as shown in Figure 3.14.

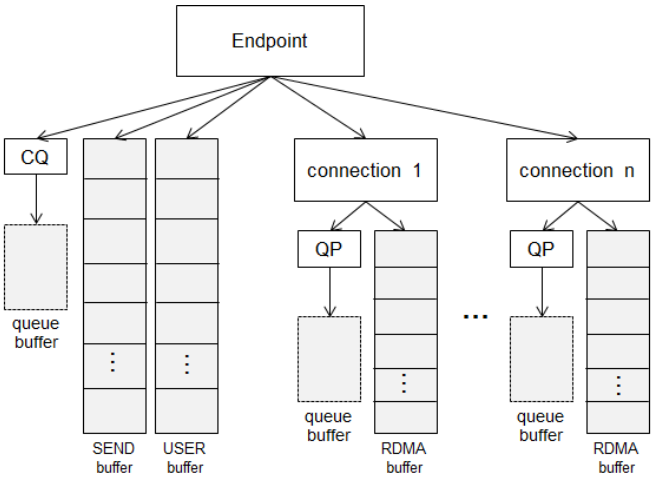


Figure 3.13: Buffer Organization in a DCFA-MPI Endpoint

Table 3.2: DCFA-MPI InfiniBand Configuration

Transport Service Type	RC
Max Send WQ	64
Max Receive WQ	0
Max Send/Receive SGE	2
Max CQE	64

The memory consumption per process grows linearly, and reaches 10^6Mbytes for the MPI application with $O(M)$ processes. Obviously, only the ($Buffer_{rdma}$+ $Buffer_{qp}$) part will increase with the growth of process number. By utilizing lazy connection establishment, the memory consumption of this part can be significantly reduced. The paper [14] shows this method can keep a fixed memory consumption in "no communication" and "allreduce" cases as the process number was increased. However, this method doesn't benefit the application including large-scale communication such as all-to-all communication, tuning the buffer size dynamically [34] might be a solution to improve it.

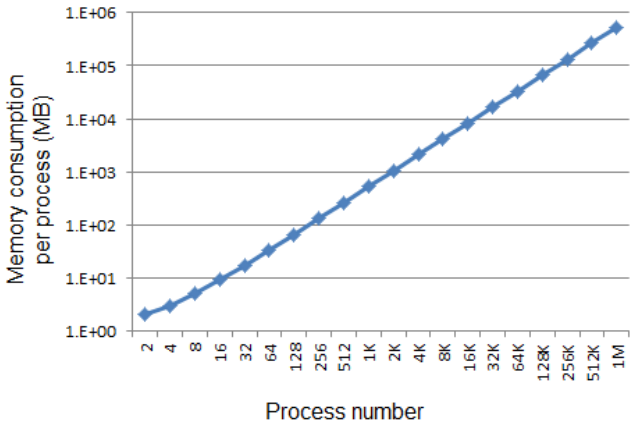


Figure 3.14: Memory Consumption for Large Scale Process

Chapter 4

Evaluation

4.1 Evaluation Environment

We performed all experiments on two Intel Workstations with the configuration given in Table 4.1.

4.2 DCFA

The DCFA evaluation is based on the comparison between the results of the following two experiments: many-core to many-core data transfer over DCFA; host to host data transfer using the InfiniBand Verb API. Let's call them "DCFA" and "host", respectively. The program scenarios of the two experiments have been described in Section 3.3.2.2 and Section 3.3.1.2, and both the Send/Receive and RDMA write are measured. All of the experiments are conducted in Ping-Pong fashion between the two workstations, and the latency result is derived from Round Trip Time.

The relative latency of "DCFA" in both Send/Receive mode and RDMA write mode is shown in Figure 4.1. Because of the overhead of accessing into many-core memory, "DCFA" is always slower than "host". However, when the latency of data transfer between two many-cores is much larger, this overhead can be ignored. "DCFA Send/Receive" got the worst result, 1.50 times slower than "host Send/Receive", when message size is 128bytes; the best result, the same latency with "host Send/Receive", when message size is 128Kbytes. "DCFA RDMA write" got the worst result, 2.22 times slower than "host RDMA write", when message size is 64bytes; the best result, the same latency with "host RDMA write", when message size is 4Kbytes.

Table 4.1: Server Architecture used in the Experiments

Machine	Intel Workstation
M/B	Intel S5520SCR
CPU	Intel Xeon X5680 3.33GHz x 2
InfiniBand HCA	Mellanox MT26428
Many-core Card	Knights Ferry x 1
Operating System	Red Hat Enterprise Linux Server release 6.1
Mellanox OFED Version	1.5.3

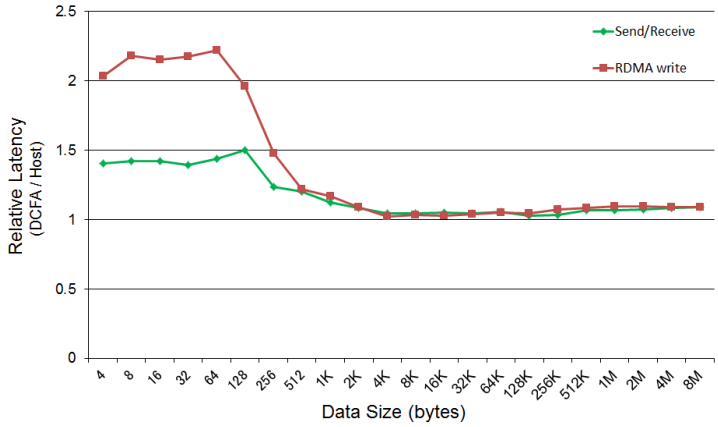


Figure 4.1: DCFA SendReceive/ host SendReceive and DCFA RDMA write/ host RDMA write

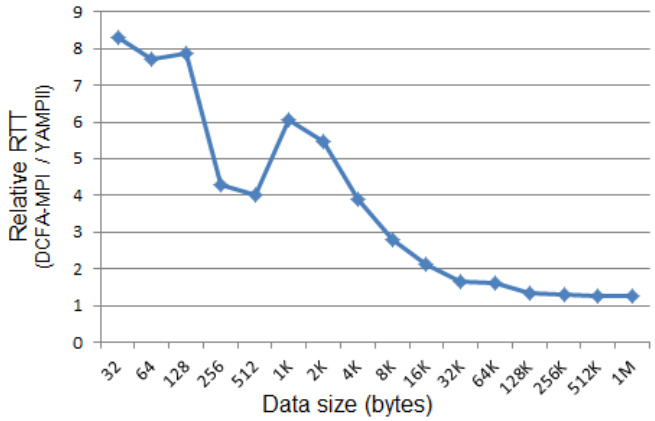
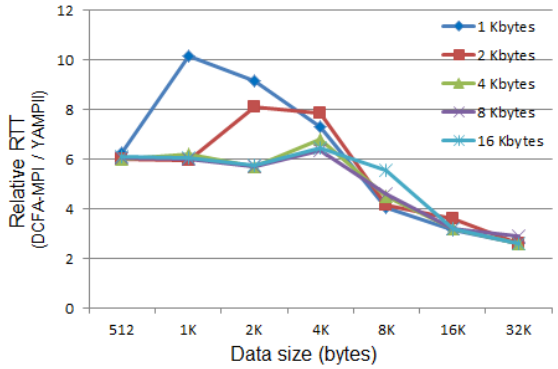


Figure 4.2: Comparison between the RTT of DCFA-MPI and the RTT of YAMPII

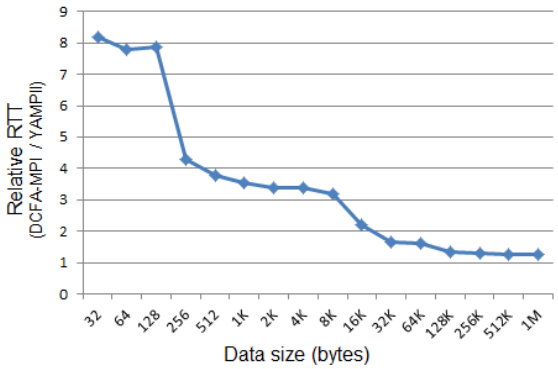
4.3 DCFA-MPI

4.3.1 Performance Optimization

DCFA-MPI is implemented based on a light-weight host MPI implementation called YAMPII. Thus the performance of DCFA-MPI is compared with YAMPII by measuring the Round Trip Time (RTT) of their blocking communication (Figure 4.2). Roughly speaking, the performance of DCFA-MPI is getting closer to YAMPII with the increase of message size. However, the range from 512bytes to 8Kbytes doesn't follow the overall trend, a distinct slowdown is observed in this range, and the range from 128bytes to 256bytes also shows a drastic performance change. The analysis and improvement around the two ranges will be discussed in following two sections. The tuning of parameter *CreditSyncLimit* used in the implementation of credit flow (Section 3.4.2.4) will be discussed in Section 4.3.1.3.



(a) Comparison between the RTT of DCFA-MPI and the RTT of YAMPII in different *EagerUpperLimit*



(b) Comparison between the RTT of DCFA-MPI and the RTT of YAMPII

Figure 4.3: *EagerUpperLimit* Optimization in DCFA

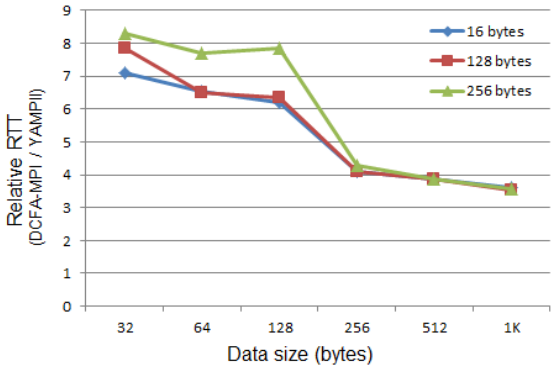
4.3.1.1 Eager Upper Limit

The slowdown in [512bytes - 8Kbytes] may be caused by the value of *EagerUpperLimit* parameter, which determines a message uses eager protocol or not. This parameter is set to 1K in current implementation, if the size of message plus the size of packet header is less than this limit, this message must be sent in Eager protocol, otherwise it must be sent in rendezvous protocol . This value might affect the communication performance with the data size around it, because if the value is too large, the overhead of the extra copy in eager protocol will definitely reduce the performance, but if it is too small, the overhead of handshakes in rendezvous protocol will account for too large proportion. Thus it should be tuned in each execution environment to get the best value.

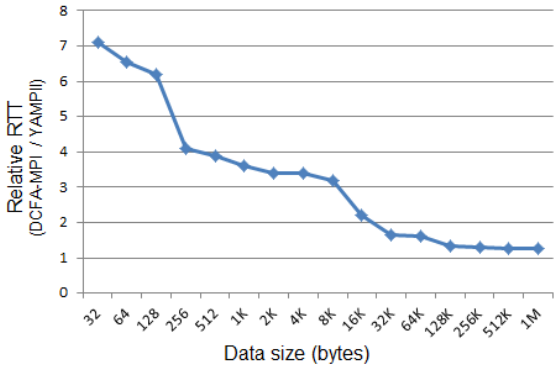
We have tuned it in the environment described in Table 4.1 with the *EagerUpperLimit* 1Kbytes, 2Kbytes, 4Kbytes, and 8Kbytes. As shown in Figure 4.3(a) the most appropriate value of *EagerUpperLimit* should be 4Kbytes or 8Kbytes. The RTT of DCFA-MPI with 8Kbytes *EagerUpperLimit* is also compared with YAMPII, as Figure 4.3(b) shows, the distinct slowdown in [512bytes - 8Kbytes] is solved.

4.3.1.2 InfiniBand Inline Limit

The sharply acceleration in [128bytes - 256bytes] is also noteworthy. This might be affected by the value of $IBInlineLimit$ parameter. In the InfiniBand post send processing, if data is smaller than this parameter, it will be copied to the queue



(a) Comparison between the RTT of DCFA-MPI and the RTT of YAMPII in different $IBInlineLimit$



(b) Comparison between the RTT of DCFA-MPI and the RTT of YAMPII

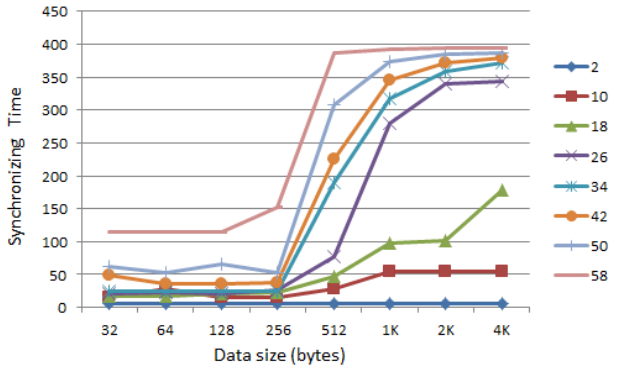
Figure 4.4: $IBInlineLimit$ Optimization in DCFA

buffer of the send queue and then transferred out, otherwise only the address information of data buffer is set into the queue buffer. This option has helped the host communication get better performance for smaller messages, however it may be changed on many-core.

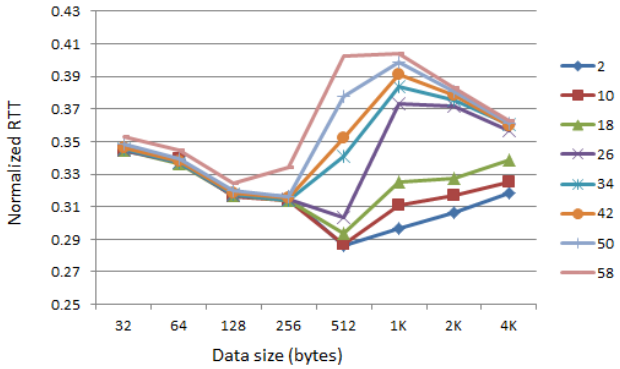
We performed a comparison between the different values of $IBInlineLimit$. It can be summarized from the results shown in Figure 4.4(a) that, the inline option is not suitable for many-core communication. The RTT of DCFA-MPI with 8Kbytes $EagerUpperLimit$ and 16bytes $IBInlineLimit$ is also compared with YAMPII, as Figure 4.3(b) shows, the relative value of the range [128bytes - 256bytes] is reduced from 8x to 7x.

4.3.1.3 Credit Sync Limit

Section 3.4.2.4 has introduced the design of the credit flow used in DCFA-MPI. We tuned it for a simple asymmetric communication application, in which one rank only sends messages and the other one only receives, and the send/receive processing is repeated 400 times. We simply defined $MaxCredit = 64$, thus the value of $CreditSyncLimit$ is measured from 2 to 58. Since the rendezvous protocol communication can always synchronize the credit information, measuring the eager data sizes is enough. Figure 4.5(a) compares the number of the SYNC packets sent in a whole processing, obviously the value 2 got the most stable result. The time of synchronization increases with the increasing of $CreditSyncLimit$,



(a) Synchronizing Time



(b) Normalized RTT

Figure 4.5: Performance of different *CreditSyncLimit* values

especially for the limit value larger than 18 and data size is larger than 512bytes, it seems that almost every send/receive requires a synchronization. Figure 4.5(b) shows the normalized RTT for each limit value, the value 2 also gets the best performance, and larger value gets slower result. This result can be easily explained using the results shown in the first figure, the more synchronization the more overhead.

In the final optimized DCFA-MPI, the value of *EagerUpperLimit*, *IBInlineLimit* and *CreditSyncLimit*2 are set to 8Kbytes, 16bytes and 2 respectively. The relative RTT of the final optimized version is shown in Figure 4.6, Comparing with the previous optimization result, the relative RTT for 32bytes data has been improved from 7x to 6.5x. It presents the worst result, 6.5x slower than YAMPII when data size is 32 bytes, and is getting closer to YAMPII with the increase of data size, and finally gets the best result, only 1.25x slower than YAMPII when data size is larger than 512Kbytes.

4.3.2 Intel MPI + Offload mode

Intel MPI Library also supports several programming modes for Intel MIC Architecture [28]. Since only the MPI + offload mode has implemented the communication mode between two many-cores so far, it's compared with DCFA-MPI in a simple stencil computing benchmark using two processes. Every process is responsible for computing with half of the data, the adjacent elements exchange their value with the other process after local computing finished, thus the communication can be considered to occur only in the two adjacent element

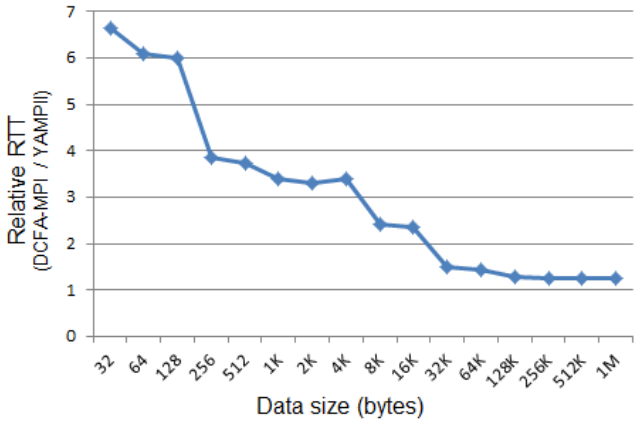


Figure 4.6: Comparison between the RTT of Final Opitimized DCFA-MPI and the RTT of YAMPII

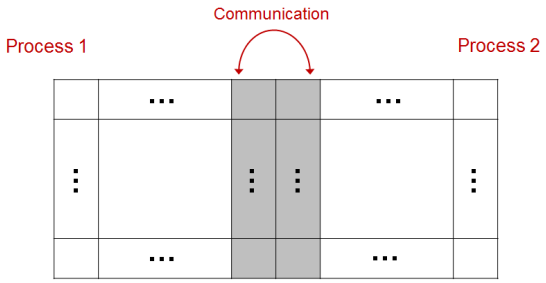


Figure 4.7: Two Processes Stencil Computing

arrays (Figure 4.7). The communication data size is depending on the number of adjacent elements, This benchmark assumes it is the same with the computing problem size, for a N size problem, the communication data size per process is $N \times Size_{Double} = 8Nbytes$.

Listing 4.1: A test MPI application

```
for( ; ; ){
  /*stencil computing*/

  MPI_Isend(sendbuf, count, MPI_DOUBLE, dst, 0,
     MPI_COMM_WORLD, &reqs[0]);
  MPI_Irecv(recvbuf, count, MPI_DOUBLE, src, 0,
     MPI_COMM_WORLD,  &reqs[1]);
  MPI_Waitall(2, reqs, status);
}
```

Listing 4.1 shows the main loop used in the experimental stencil computing benchmark. Due to the computing part of both modes are performed on MIC, this part can be omitted to measure the accurate communication latency. In then DCFA-MPI mode, the whole loop is performed on MIC, the communication latency is only caused by the MPI communication. In the Intel MPI + Offload mode, the stencil computing is offloaded to MIC using pragma and the MPI communication still runs on host, thus the latency consists of the offload data transfer between the host CPU and MIC, and the host MPI communication.

Because so far a native offload program cannot get good performance, following optimization methods are used. The source code is shown in Appendix A.

- Eliminate offload Initialization
  Depending on the time debug information provided from Intel debug tool, a heavy initialization is executed in the first pragma. Thus, the first pragma is removed from the main loop by setting a empty pragma before the loop.

- Persistent Buffer
  The offload pragma always re-allocates/frees the variable data buffers in every copy in/out processing by default. The data transfer overhead become much expensive in stencil computing because several vector arrays are copied in/out in every loop. Some pragma options can be set to allocate the buffers only once and make the data persistent in the whole computing loop, and then the transfer overhead could be significantly reduced.

- Fastest Data transfer over PCI express
  Data is aligned on a 4Kbytes page boundary and the size is a multiple of 4Kbytes.

- Double Buffer Method
  The offload copy in/out can be overlapped with the MPI communication by splitting the variable data buffer into two subset buffers.

Figure 4.8 shows the relative value of the communication latency for DCFA-MPI(MPI on KNF) and Intel MPI on Xeon + Offload to KNF mode. Although the DCFA-MPI is slower than the Intel MPI which performs communication between hosts, its performance is always superior to the Intel MPI on Xeon + Offload to KNF mode. The best result is achieved at 32bytes, 5x faster than offload mode, and with the increasing data size, the gap between two modes is getting smaller and smaller, but the DCFA-MPI also maintains better performance than the offload mode, and gets the worst result 1.12x speedup at 1Mbytes. Because the extra overhead of every offload operation, the DCFA-MPI performs much better performance than then offload mode when data size is small, but with the increasing proportion of MPI communication latency, the effect of the offload overhead can almost be ignored.

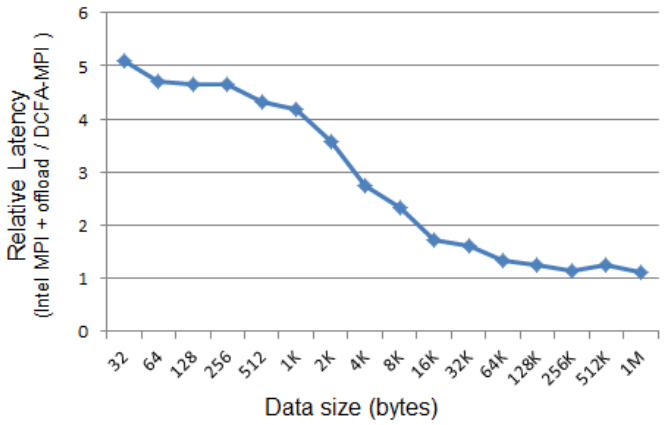


Figure 4.8: Relative Communication Latency in the Stencil Computing Benchmark using DCFA-MPI (the Latency of Intel MPI on Xeon + Offload to KNF / the Latency of DCFA-MPI(MPI on KNF))

Chapter 5

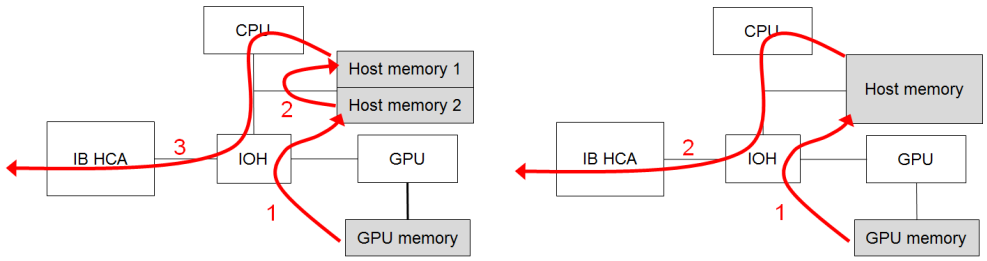
Related Work

5.0.3 GPUDirect

The GPUDirect technology, first released in June 2010, accelerates the communication in GPGPU-based cluster systems [7]. Prior to the release of this technology, each GPU-to-GPU communication had to complete the following steps (Figure 5.1(a)). First, the GPU copies the data from the GPU device memory to the host memory. Second, the host CPU copies the data from the GPU dedicated host memory to the host memory available for the InfiniBand HCA. Finally, the InfiniBand HCA sends data to the remote node. GPUDirect accelerates the processing by eliminating the second step (Figure 5.1(b)). However, the GPUDirect also needs the copy from GPU device memory to host memory, thus GPUDirect incurs high latency than DCFA.

5.0.4 OPIOM

OPIOM [16] stands for Off-Processor IO with Myrinet in parallel file-systems. As a basic operation, the remote read, which performs a data transfer from server disks to the memory space of remote client, consists of three data movements. The first movement is from the disks to the kernel memory space, the second one is from the kernel memory space to user-level application memory space, and the third one is from the user-level application memory space to remote node via Myrinet. OPIOM eliminates the first two movements, so that data can be moved directly from disks to the remote node via Myrinet. The implementation is based on SCSI devices and a Myrinet interface. However, this design is technically possible for all the devices which can provide DMA engines and the PCI address space mapped to its own memory space.



(a) without GPUDirect

(b) with GPUDirect

Figure 5.1: Data Transfer in GPU-to-GPU communication

5.0.5 Intel SCIF

Intel published a Intel MIC Platform Software Stack (MPSS) which consists of an embedded Linux, a minimally modified GCC, and driver software [23] [22]. The Linux system running on MIC co-processor is designed to be booted by a host processor, and is connected to host via the host driver. In this Linux environment, the C, C++, Fortran programs can be easily executed as runs on host, and the OpenMPI and Intel MPI Library are also provided for parallel programming. The MPSS contains a Symmetric Communication Interface (SCIF) [2], which is designed as the communication backbone between the host processors and the MIC co-processors in a heterogeneous computing environment. It provides a uniform API for communication between the host processor and the MIC co-processor, and also between the MIC co-processors.

5.0.6 MCAPI

The Multicore Communications API (MCAPI) [12] provides a message based communication framework for closely distributed embedded systems. It defines a programming API for the communication and synchronization between processing cores. It is scalable for any number of cores, and supports heterogeneity architecture such as CPUs and accelerators. The multicast communication is not supported. Unlike the MCAPI, DCFA provides the inter-node communication for the many-cores accelerators connected via InfiniBand both in widely distributed topology and closely distributed topology.

5.0.7 Mvapich for GPGPU-based clusters

Hao Wang, et al., also introduced a method for how the Message Passing Interface (MPI) uses InfiniBand in GPGPU-based clusters [40]. The communication between two GPUs can be separated into three data transfer stages. The first stage is from the sender's GPU device memory to its host memory; the second is from the sender's host memory to the receiver's host memory; the third is from the receiver's host memory to its GPU device memory. This paper shows an innovative approach to improving the performance of the above process by "offloading" MPI datatype packing and unpacking on to a GPU device for non-contiguous datatypes, and "pipelining" all data transfer stages. Obviously, the communication in GPGPU-based clusters still needs host assists.

5.0.8 Intel MPI Library for Intel MIC Architecture

In the Intel MPI Library for Intel MIC Architecture, the node can be either a host processor or a MIC co-processor, and following three programming modes are supported [28].

- Intel MPI + Offload Mode
  MPI ranks on host processors only and computing are offloaded to MIC co-processors, the communication is still performed between host processors.
- Many-core Hosted Mode
  MPI ranks on MIC co-processors only, all messages are transferred to/from MIC co-processors directly.
- Symmetric Mode
  MPI ranks on both host processors and MIC co-processors, messages can be transferred to/from any core.

To the best of our knowledge, only the Intel MPI + offload mode has implemented the data exchange between two separated MIC co-processors so far. This mode has to move data to MIC co-processors before computing, and move data out before host communication. As the results shown in Section 4.3.2, this mode incurs much high latency than DCFA-MPI.

5.0.9 Intra-MIC MPI Communication using MVAPICH2

The paper [31] introduces an early experience of the intra-MIC communication using MVAPICH2 in April this year. It enhances and tunes an shared memory based design in MVAPICH2 on the Knights Ferry co-processors, and evaluated both the P2P communication and collective communication. This implementation has not implemented the inter-node communication yet, and runs entirely on the Linux environment provided by MPSS.

Chapter 6

Conclusions

6.1 Summary of Research Contributions

This paper has designed a direct communication facility, called DCFA, for many-core architectures, especially the Intel MIC architecture, connected to the host via PCI Express with the InfiniBand HCA. By distributing the internal structures of the HCA to memory areas of the host and the many-core unit, the many-core unit may transfer its data directly to/from a remote many-core unit or a remote host. The implementation of DCFA is based on the Mellanox InfiniBand HCA and Intel's Knights Ferry (KNF), a predecessor of Knights Corner. The evaluation results show that, the latency and throughput of DCFA delivers the same performance as that of host to host data transfer for messages larger than 2Kbytes, and at most 2x worse for very small messages. Though DCFA has been implemented based on the Intel MIC architecture, the DCFA method is applicable to other many-core based accelerators if they are capable of mapping PCI Express memory, issuing interrupts to the host, and writing commands to a PCI Express device.

Moreover, this paper has also implemented an MPI library based on the DCFA, called DCFA-MPI, to provide direct many-core to many-core MPI communication. The MPI initialization, finalization, and other heave functions such as collective communication are offloaded to host CPU, the floating-point computing and point-to-point communication are directly performed on many-core. DCFA-MPI uses a RDMA read plus RDMA write zero-copy design for rendezvous protocol. After tuning several parameter values, it finally delivers the same performance as that of host MPI communication for the messages larger than 64Kbytes, and the worst performance, 6x slower than host MPI communication when data size is very small. The DCFA-MPI(MPI on KNF) has been also compared with the Intel MPI on Xeon + Offload to KNF mode between two KNF co-processors. The evaluation results show that, it achieves the best result, 5x speedup for 32bytes messages, and still keeps at least 1.12x speedup for very large messages. Roughly speaking, the communication performance of DCFA-MPI(MPI on KNF) is always superior to the performance of Intel MPI on Xeon + Offload to KNF mode.

6.2 Future Work

6.2.1 Scalability

This research is aimed at perfecting the communication over post peta-scale supercomputer with close to a million processors, thus the problems around scalability are planned to be solved in future research.

6.2.1.1 Lazy Queue Pair Connection

Current experimental MPI implementation sets up all-to-all QP connections at MPI initialization stage. Since the number of MPI processes may be up to $O(1K)$ or even $O(1M)$, here is certainly a runtime bottleneck because this is an $O(p^2)$ operation. This problem has already been discussed in paper [30] [36] [18] [15]. The paper [30] [36] introduce a lazy connection strategy, which establish a connection only when a process needs communicate whit another, it benefits the applications which haven't all-to-all communication. The paper [18] introduces a locality-aware connection management based on the lazy connection establishment which can also performs good performance with less established connections on even large-scale communication applications, The paper [15] describes another method using the InfiniBand unreliable datagram transport (UD), in which one QP can be connected with multiple QPs, to reduce the connection resources. However, UD does not support the RDMA features, which deliver much better performance for smaller messages and are needed for zero-copy rendezvous design. Based on the above solutions, a scalable connection strategy will be selected and ported to DCFA-MPI.

6.2.1.2 Memory Consumption

A pre-allocated and registered buffer is always prepared for each connection to receive the eager message and other control packets. Current DCFA-MPI prepares all of them in the MPI initialization, then the memory consumption per process is up to $O(p \times BufferSize)$. This problem can be almost solved by lazying the connection establishment because the RDMA buffers are only prepared for established connection, The paper [14] shows this solution can keep a fixed memory consumption in "no communication" and "allreduce" cases as the process number was increased. However, this solution doesn't benefit the application including large-scale communication such as all-to-all communication, thus a mechanism to tuning the buffer size dynamically [34] is planned to be studied and port to DCFA-MPI.

6.2.1.3 Communication thread

The paper [24] introduces a mechanism, in which the communication is offloaded into a communication thread to get better overlapping with computation. Based on this idea, some heavy communication functions such as collective communication could be offloaded to a dedicated many-core process.

6.2.2 Task Balance

In current implementation, a host assist daemon is required for each MPI process runs on many-core, the daemon waits the commands from many-core and executes corresponding functions. However, the computing resources of the host CPU are wasted during the waiting time. Therefore, it's necessary to design a mechanism to balance the tasks distributed into the host and the many-core, for example some computing tasks can be also distributed to host while host is waiting,

6.2.3 Intranode communication

The cooperation of OpenMP and MPI on host processors has been researched for a long time [17] [33] [32]. For the accelerators, the OpenACC [13] provides

the interface to specify loops and regions to be offloaded from host processor to accelerator. Thus similar with the OpenMP + MPI Hybrid Programming, the cooperation of OpenACC and MPI for accelerators may be considered. Moreover, the Intel MIC architecture has limited memory so that data might have to be swapped to host memory, this data movement is also handle by the OpenACC, and should be considered in the MPI implementation.

References

[1] Intel insights at sc11. http://newsroom.intel.com/servlet/JiveServlet/download/38-6968/Intel_SC11_presentation.pdf.

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

[3] Isc - the hpc event. http://www.isc-events.com/isc10/.

[4] Mellanox openfabrics enterprise distribution for linux (mlnx_ofed). http://www.mellanox.com/content/pages.php?pg=products_dyn&product_family=26&menu_section=34.

[5] Mpich. http://www.mcs.anl.gov/mpi/mpich2.

[6] Mvapich. http://mvapich.cse.ohio-state.edu/.

[7] Nvidia gpudirect™ technology - accelerating gpu-based systems. http://www.mellanox.com/pdf/whitepapers/TB_GPU_Direct.pdf.

[8] Openmpi. http://www.open-mpi.org/.

[9] Top500 supercomputing sites. http://www.top500.org.

[10] What is gpu computing? http://www.nvidia.com/object/what-is-gpu-computing.html.

[11] Infiniband™ architecture specification, volume 1, release 1.2.1. http://members.InfiniBandta.org/kwspub/spec/vol1r1_2.zip, 2007.

[12] The Multicore Association™. Multicore communications api working group (mcapi™). http://www.multicore-association.org/workgroup/mcapi.php.

[13] The Multicore Association™. Openacc. http://www.openacc-standard.org/content/about-us.

[14] Goodell David, Gropp William, Zhao Xin, and Thakur Rajeev. Scalable memory use in mpi: a case study with mpich2. In *Proceedings of the 18th European MPI Users' Group conference on Recent advances in the message passing interface*, EuroMPI'11, pages 140–149, Berlin, Heidelberg, 2011. Springer-Verlag.

[15] A. Friedley, T. Hoefler, M. Leininger, and A. Lumsdaine. Scalable High Performance Message Passing over InfiniBand for Open MPI. In *Proceedings of 3rd KiCC Workshop 2007*. RWTH Aachen, Dec. 2007.

[16] P. Geoffray. Opiom: off-processor io with myrinet. In *Proceedings of First IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 261 –268, 2001.

[17] Kaiser Timothy H. and Baden Scott B. Overlapping communication and computation with openmp and mpi. *Sci. Program.*, 9(2,3):73–81, August 2001.

[18] Saito Hideo and Taura Kenjiro. Locality-aware connection management and rank assignment for wide-area mpi. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '07, pages 150–151, New York, NY, USA, 2007. ACM.

[19] Yutaka Ishikawa. Yampii, yet another mpi implementation. *IPSJ SIG Technical Report*, 2004(81):115–120, 2004-07-30.

[20] Kuck David J. Supercomputers for ordinary users. In *Proceedings of the December 5-7, 1972, fall joint computer conference, part I*, AFIPS '72 (Fall, part I), pages 213–220, New York, NY, USA, 1972. ACM.

[21] Rashti Mohammad J. and Afsahi Ahmad. Improving communication progress and overlap in mpi rendezvous protocol over rdma-enabled interconnects. In *Proceedings of the 2008 22nd International Symposium on High Performance Computing Systems and Applications*, HPCS '08, pages 95–101, Washington, DC, USA, 2008. IEEE Computer Society.

[22] James Reinders. Knights corner micro-architecture support. `http://software.intel.com/en-us/blogs/2012/06/05/knights-corner-micro-architecture-support/`.

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

[24] Sameer Kumar, Amith R. Mamidala, Daniel A. Faraj, Brian Smith, Michael Blocksome, Bob Cernohous, Douglas Miller, Jeff Parker, Joseph Ratterman, Philip Heidelberger, Dong Chen, and Burkhard Steinmacher-Burow. Pami: A parallel active message interface for the blue gene/q supercomputer. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, IPDPS'12, pages 763–773, 2012.

[25] Jiuxing Liu, Weihang Jiang, Pete Wyckoff, Dhabaleswar K. Panda, David Ashton, Darius Buntinas, William Gropp, and Brian R. Toonen. Design and implementation of mpich2 over infiniband with rdma support. *CoRR*, cs.AR/0310059, 2003.

[26] Jiuxing Liu and Dhabaleswar K. Panda. Implementing efficient and scalable flow control schemes in mpi over infiniband. In *IPDPS*, 2004.

[27] Jiuxing Liu, Jiesheng Wu, Sushmitha P. Kini, Darius Buntinas, Weikuan Yu, Balasubraman Chandrasekaran, Ranjit M. Noronha, Pete Wyckoff, and Dhabaleswar K. Panda. Mpi over infiniband: Early experiences. Technical report, 2003.

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

[29] Message Passing Interface Forum. Mpi: A message-passing interface standard. `http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html`.

[30] Balaji Pavan, Buntinas Darius, Goodell David, Gropp William, Kumar Sameer, Lusk Ewing, Thakur Rajeev, and Träff Jesper Larsson. Mpi on a million processors. In *Proceedings of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 20–30, Berlin, Heidelberg, 2009. Springer-Verlag.

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

[32] Rolf Rabenseifner, Georg Hager, and Gabriele Jost. Hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes. *Parallel, Distributed, and Network-Based Processing, Euromicro Conference on*, 0:427–436, 2009.

[33] Ashay Rane and Dan Stanzione. Experiences in tuning performance of hybrid mpi/openmp applications on quad-core systems. *Proc. of 10th LCI Intfl Conference on High-Performance Clustered Computing*.

[34] Sur Sayantan, Chai Lei, Jin Hyun-Wook, and Panda Dhabaleswar K. Shared receive queue based scalable mpi design for infiniband clusters. In *Proceedings of the 20th international conference on Parallel and distributed processing*, IPDPS'06, pages 101–101, Washington, DC, USA, 2006. IEEE Computer Society.

[35] Taku Shimosawa, Yutaka Ishikawa, Atsushi Hori, Mitaro Namiki, and Yuichi Tsujita. Design and implementation of development environment for systems software for manycore architecture. *IPSJ SIG Technical Report*, 2011(1):1–7, 2011-07-20.

[36] Galen M. Shipman, Tim S. Woodall, Rich L. Graham, Arthur B. Maccabe, and Patrick G. Bridges. Infiniband calability in open mpi. In *Proceedings of the 20th international conference on Parallel and distributed processing*, IPDPS'06, pages 100–100, Washington, DC, USA, 2006. IEEE Computer Society.

[37] Sayantan Sur, Hyun wook Jin, Lei Chai, and Dhabaleswar K. Panda. Rdma read based rendezvous protocol for mpi over infiniband: design alternatives and benefits. In *In PPoPP f06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles*, pages 32–39, 2006.

[38] Hiroshi Tezuka, Francis O'Carroll, Atsushi Hori, and Yutaka Ishikawa. Pin-down cache: A virtual memory management technique for zero-copy communication, 1998.

[39] Kenneth J. Thurber and Leon D. Wald. Associative and parallel processors. *ACM Comput. Surv.*, 7(4):215–255, December 1975.

[40] Hao Wang, S. Potluri, Miao Luo, A.K. Singh, Xiangyong Ouyang, S. Sur, and D.K. Panda. Optimized non-contiguous mpi datatype communication for gpu clusters: Design, implementation and evaluation with mvapich2. In *Proceedings of IEEE International Conference on Cluster Computing (CLUSTER)*, pages 308 –316, sept. 2011.

[41] Gropp William, Lusk Ewing, Doss Nathan, and Skjellum Anthony. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel Comput.*, 22(6):789–828, September 1996.

Appendix A

A Communication only Stencil Computing Application using Intel MPI + Offload mode

```
/*
 * A communication only stencil computing program using
     Intel MPI + offload mode.
 *
 *
 *   Created on: 2012/07/05
 *       Author: simin
 */
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

#define LEN 1024
#define TIME 400

#define ALIGN_SIZE 4096

#pragma offload_attribute (push, target( mic ))

int *sbuf1, *sbuf2;
int *rbuf1, *rbuf2;
int rank, size, n;

void print_mem(void *data, int len) {
        int i;
        for (i = 0; i < len; i++) {
                printf("%02x ", *((char*) data + i));
        }
        printf("\n");
}

#define min(a,b) \
        ({ typeof (a) _a = (a); \
           typeof (b) _b = (b); \
           _a < _b ? _a : _b; })

#pragma offload_attribute (pop)

int src, dst;
int n, len, buf_len, buf_n;
MPI_Status status[4];
MPI_Request reqs[4];
```

```
/* OPT4: Double Buffer Method */
void double_buffer_loop() {
        int norm;

        norm = 0;

        /* only copy-in sbuf once */
#pragma offload_transfer target(mic:0) \
                 in(sbuf1, sbuf2 : length(buf_n) alloc_if(0)
                     free_if(0) )
        do {
                /* waits for data exchange in sbuf1/rbuf1 */
                if (norm > 0) {
                        MPI_Wait(&reqs[0], &status[0]);
                        MPI_Wait(&reqs[1], &status[1]);
                }

#pragma offload_transfer target(mic:0) \
                 in(rbuf1 : length(buf_n) alloc_if(0)
                    free_if(0) )

                MPI_Irecv(rbuf1, buf_n, MPI_INT, src, 0,
                   MPI_COMM_WORLD, &reqs[0]);

#pragma offload target(mic:0) \
        in(norm) \
        nocopy(rbuf1) \
                out(sbuf1 : length(buf_n) alloc_if(0)
                    free_if(0) )
                {
                        /*stencil computing using rbuf1 and
                           sbuf1, sbuf1 is updated*/
                }

                MPI_Isend(sbuf1, buf_n, MPI_INT, dst, 0,
                   MPI_COMM_WORLD, &reqs[1]);

                /* waits for data exchange in sbuf2/rbuf2 */
                if (norm > 0) {
                        MPI_Wait(&reqs[2], &status[2]);
                        MPI_Wait(&reqs[3], &status[3]);
                }
#pragma offload_transfer target(mic:0) \
                 in(rbuf2 : length(buf_n) alloc_if(0)
                    free_if(0) )

                MPI_Irecv(rbuf2, buf_n, MPI_INT, src, 0,
                   MPI_COMM_WORLD, &reqs[2]);

#pragma offload target(mic:0) nocopy(rbuf2)  \
                in(norm) \
        out(sbuf2 : length(buf_n) alloc_if(0) free_if(0) )
                {
                        /*stencil computing using rbuf2 and
                           sbuf2, sbuf2 is updated*/
                }
```

```
                MPI_Isend(sbuf2, buf_n, MPI_INT, dst, 0,
                    MPI_COMM_WORLD, &reqs[3]);

        } while (++norm < TIME);

        /* wait for last iter */
        MPI_Waitall(4, reqs, status);
}

int main(int argc, char **argv) {
        int j, namelen;
        double time0, time1, t0, t1;

        len = LEN;
        if (argc == 2) {
                len = atoi(argv[1]);
        }

        n = len / sizeof(int);
        if (n < 2)
                return 0;
        len = n * sizeof(int);

        buf_len = len / 2;
        buf_n = n / 2;

        /* OPT3: Fastest Data transfer over PCI express */
        sbuf1 = (int*) _mm_malloc(buf_len, ALIGN_SIZE);
        rbuf1 = (int*) _mm_malloc(buf_len, ALIGN_SIZE);
        sbuf2 = (int*) _mm_malloc(buf_len, ALIGN_SIZE);
        rbuf2 = (int*) _mm_malloc(buf_len, ALIGN_SIZE);

        memset(sbuf1, 1, buf_len);
        memset(sbuf2, 2, buf_len);
        memset(rbuf1, 3, buf_len);
        memset(rbuf2, 4, buf_len);

        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Comm_size(MPI_COMM_WORLD, &size);

        src = (rank + 1) % size;
        dst = (rank + size - 1) % size;

/*
 * OPT1: Eliminate offload Initialization
 * OPT2: Persistent Buffer
 */
#pragma offload_transfer target (mic) \
                nocopy(sbuf1, sbuf2, rbuf1, rbuf2 : length(
                    buf_n) alloc_if(1) free_if(0)) \
                in(rank, size)

        MPI_Barrier(MPI_COMM_WORLD);

        time0 = MPI_Wtime();

        double_buffer_loop();
```

```
        time1 = MPI_Wtime();

/* OPT2: Persistent Buffer */
#pragma offload_transfer target (mic) \
                nocopy(sbuf1, sbuf2, rbuf1, rbuf2 : length(n
                    ) alloc_if(0) free_if(1))

        if (rank == 0) {
                printf("%d\t%lf\n", n, (time1 - time0) *
                    1000 * 1000 / TIME);
        }

        MPI_Finalize();

        /* OPT3: Fastest Data transfer over PCI express */
        _mm_free(sbuf1);
        _mm_free(rbuf1);
        _mm_free(sbuf2);
        _mm_free(rbuf2);

        return 0;
}
```