2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum

Design of Direct Communication Facility for Many-Core based Accelerators

Min Si* and Yutaka Ishikawa*†‡
*Department of Computer Science, University of Tokyo, Tokyo, Japan
Email: {msi@il.is.s, ishikawa@is.s}.u-tokyo.ac.jp
†Information Technology Center, University of Tokyo, Tokyo, Japan
‡RIKEN Advanced Institute for Computational Science, Japan

***Abstract*—A direct communication facility, called DCFA, for a many-core based cluster, whose compute node consists of many-core units connected to the host via PCI Express with Infiniband, is designed and evaluated. Because a many-core unit is a device of the PCI Express bus, it is not capable of configuring and initializing the Infiniband HCA, according to the PCI Express specification. This means that the host has to assist memory transfer between many-core units, and thus extra communication overhead is incurred. In DCFA, the internal structures of the Infiniband HCA are distributed to both the memory space of the host and that of the many-core unit. After the host CPU configures and initializes the HCA, it obtains the addresses of both the HCA and the internal structures assigned by the host. Using the information given by the host and the internal structures assigned in the many-core memory area, the many-core unit may transfer data directly between other many-core units using the HCA without host assists. The implementation of DCFA is based on the Mellanox Infiniband HCA and Intel's Knights Ferry. Preliminary results show that, for large data transfer, the latency of DCFA delivers the same performance as that of host to host data transfer.**

***Keywords*-cluster; direct communication; many-core; infiniband; accelerator;**

I. Introduction

Accelerator-based PC clusters, in which a node is formed by a PC server with floating-point calculation accelerators such as GPGPU, are now widely available. Indeed, three of the peta-scale supercomputers ranked in the November 2011 top500 [1] are GPGPU-based PC clusters. Intel announced an x86-based many-core architecture called Intel® Many Integrated Core (Intel® MIC) architecture at ISC 2010. 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 [2]. Knights Corner will have greater than 50 cores and will be based on Intel's 22nm process technology.

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 GPGPU and the host, and between the host and the remote host. These extra data movements result in communication overhead. GPGPU-based cluster systems follow this execution model because GPGPU is not capable of controlling communication devices, such as the Infiniband Host Channel Adapter (Infiniband HCA).

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. However, current GPGPU-based cluster systems cannot implement this model because GPGPU is not capable of writing commands to communication devices.

Unlike current GPGPUs, a many-core based accelerator such as Knights Corner can write commands to a communication device if the PCI Express device address is given by the host. 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 accelerator-based clusters.

This paper designs a direct communication facility for many-core based accelerators, called DCFA, to implement the standalone parallel execution model. The implementation of DCFA is based on the Mellanox Infiniband HCA and Intel's Knights Ferry, a software development predecessor of Knights Corner. Internal structures of the HCA are carefully distributed to memory areas of both the host and the accelerator so that the accelerator transfers its memory area directly to/from either remote host or remote accelerator. The host CPU configures and initializes the Infiniband HCA.

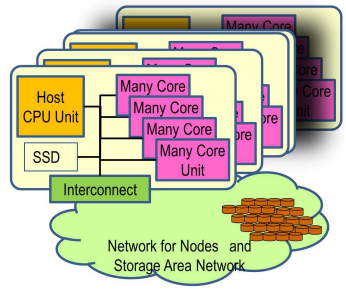
After introducing the background of this paper in the following section, the design of DCFA is presented in Section III. In Section IV, the experimental environment, evaluation scenarios and results are presented. After presenting related work in Section V, this paper will be concluded with a discussion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

II. Background

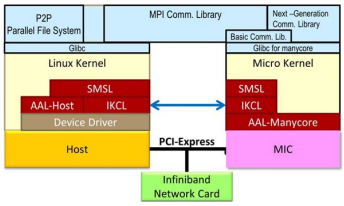
Information Technology Center at the University of Tokyo and RIKEN Advanced Institute for Computational Science have been designing and developing system software for many-core based clusters. The current target platform is based on the Intel MIC architecture as shown in Figure 1(a). Compute nodes, each of which consists of many-core units with a host CPU, are connected by Infiniband.

This architecture has two characteristics: a cluster of many-cores and a cluster of regular PCs because many-cores and host CPUs can share the same interconnect network. Since the throughput of each core of a many-core is lower each

978-0-7695-4676-6/12 $26.00 © 2012 IEEE
DOI 10.1109/IPDPSW.2012.113



(a)



(b)

Figure 1. Target Architecture

core of the host CPU and the limited capacity of memory cache in the many-core, rich system services such as file I/Os should be provided in the host CPU. Utilizing those two types of clusters, two types of execution model have emerged: in the first, number crunching applications run on many-core clusters, and in the second, file I/Os of those applications are performed on host CPUs. Since most number crunching applications do not issue file I/O requests frequently, some other data-intensive applications may run on host CPUs during number crunching applications run on many-cores.

Figure 1(b) depicts the operating system kernel for the target architecture that we have been designing and developing. The Linux kernel runs in the host CPU while a micro kernel runs in many-core units.

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

A micro kernel for many-core units is to be implemented on top of the AAL and SMSL. 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 memory cache of many-cores if such a function were to be executed in many-core units.

At this moment, DCFA has been implemented on top of AAL instead of the micro kernel being designed and implemented.

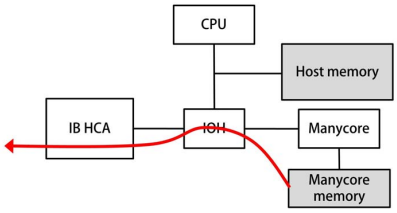


Figure 2. Data Transfer in DCFA

III. Design

DCFA is targeted to implement direct data transfer for many-core architectures (Figure 2). 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.

A. An overview of the Internal Implementation in Infiniband

The communication stack for the Infiniband Architecture (IBA) [3] consists of different layers. The interface presented by the Infiniband HCA to user-level software belongs to the transport layer. A queue-based model 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 instructions needed to transmit data, and the RQ holds instructions 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, an instruction called 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.
- 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 check the CQ to see if any WQR has been finished.

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(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 more like a mechanism to implement the circular structure. If we describe the Infiniband HCA as a producer who produces 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.). This is allocated in host memory.

The IBA also specifies memory management mechanisms. 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 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). 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, and memory locations are registered with a PD. A QP can only access the MRs in the same PD.

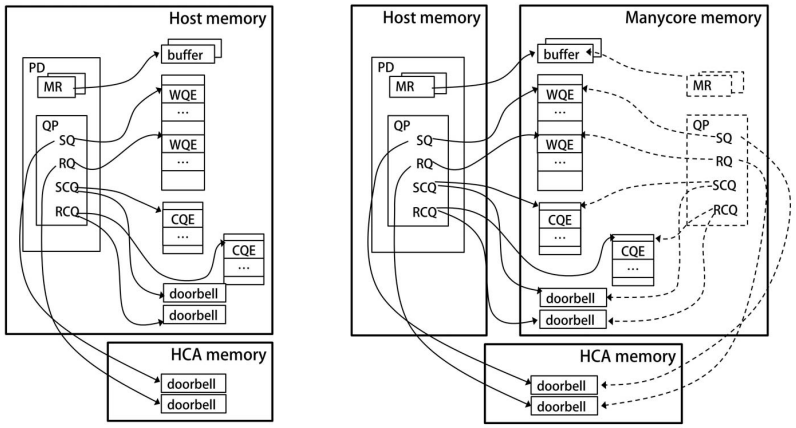
A typical Infiniband communication processing task between two nodes can be described as having the following stages (Figure 4(a)).

1) Device configuration
   Configure the Infiniband HCA by writing configuration commands to the PCI controller.
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 RDMA-able MR is usually 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 CQ, but there is no notification sent to the remote host. The remote host can check 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.

B. Design for Many-core Architectures

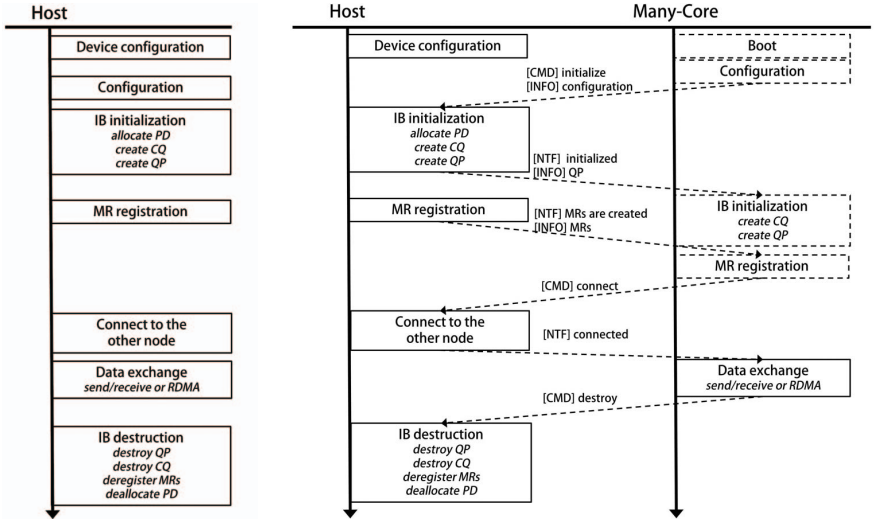
Based on the study of the internal implementation of the Mellanox Infiniband driver, it's clear that if a many-core can access the queue buffers and the doorbell records directly, it can issue commands to an Infiniband HCA. Since AAL, introduced in Section II, has implemented both memory mapping from 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(b) describes 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.



(a) in Original Infiniband (b) in DCFA

Figure 3. Allocation of Internal Structures



(a) in Original Infiniband (b) in DCFA

In (b), CMD (Command), NTF (Notification) and INFO (Information) are communication between the host and the many-core sides, implemented by using IKCL, which was introduced in Section II.

Figure 4. Processing Sequence

At the time this paper is written, we have not yet designed any memory management mechanism on the many-core side. The allocation of MR and PD structures is described as follows.

- Memory Region (MR)
  The data buffers are allocated in the memory area that has been pre-mapped to the many-core memory. The memory registration still runs on the host side. The MR structure defined in the many-core side interface holds all the information returned by the host registration.
- Protection Domain (PD)
  A PD structure is only defined on the host side.

Besides the internal structures, a communication scenario for many-core architectures has also been designed and implemented. The stages of an Infiniband communication process that were described in Section III-A, 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 left on the host side due to their heavy overhead, and consequently, the destruction part also stays. Thus, the only part to be moved to 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 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, then sends configuration information and an ”initialize” command to the host.
3) Infiniband initialization
   After receiving the command and information from the many-core, the host executes this stage. The memory allocation is performed as described at the beginning of Section III-B. The completion notification and QP initialization information which will be used in the data exchange stage are sent to the many-core when completed. After receiving the notification and 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 completion notification and MR registration information which will be used in data exchange stage are sent to the many-core. After receiving the notification and information from the host, the many-core creates its MRs.
5) Connection
   When the many-core decides to connect to the other node, it sends a ”connect” command to the host, and waits for a ”connected” completion notification.
6) Data exchange
   After receiving the ”connected” completion notification, 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 sends a ”destroy” command to the host before the program exits. The host is responsible for releasing all the Infiniband resources.

IV. Evaluation

We performed all experiments on two Intel Workstations with the configuration given in Table I. 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 already been described in Section III-B and Section III-A, 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 5. 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 I
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
Card	Knights Ferry x 1
Operating System	Red Hat Enterprise Linux Server release 6.1
Mellanox OFED Version	1.5.3

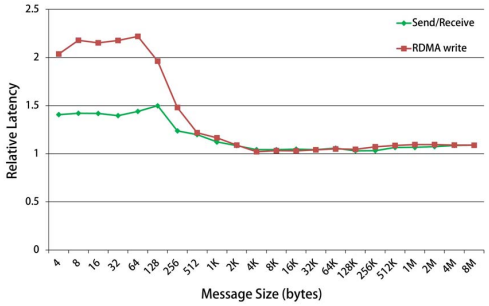


Figure 5. DCFA SendReceive/ host SendReceive and DCFA RDMA write/ host RDMA write

As a future work of this paper, an MPI library over DCFA has been planned now, we hope the comparison result between the MPI on host and the MPI on many-core can be published in our next paper.

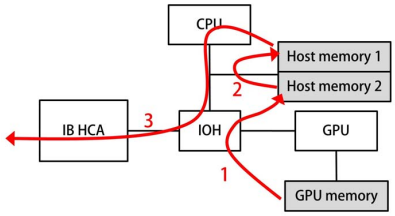
V. Related Work

A. GPUDirect

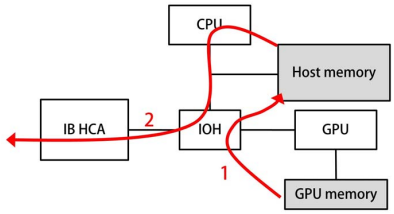
The GPUDirect technology, first released in June 2010, accelerates the communication in GPGPU-based cluster systems [5]. Prior to the release of this technology, each GPU-to-GPU communication had to complete the following steps (Figure 6(a)). First, the GPU copies the data from the GPU device memory 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 6(b)).

B. 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 [6]. 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



(a) without GPUDirect



(b) with GPUDirect

Figure 6. Data Transfer in GPU-to-GPU communication

GPU device for non-contiguous datatypes, and "pipelining" all data transfer stages. Obviously, the communication in GPGPU-based clusters still needs host assists.

C. OPIOM

OPIOM [7] 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.

VI. Concluding Remarks

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 Mellanox 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. Preliminary results show that, for large data transfer, the latency of DCFA delivers the same performance as that of host to host data transfer. 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.

There are three directions for future research, as follows. In one, the same API as the Infiniband Verb API for many-cores is transparently implemented with DCFA extensions. Using this transparent Infiniband Verb API, an MPI library can be ported to many-core architectures.

The second direction involves offloading some heavy functions of the MPI library to the host CPU or one core of the many-core unit. Because, relative to multi-cores, many-cores have small memory caches per core and limited memory bandwidth per core, the footprint in the cache during execution of both the application and the communication library should be minimized. Heavy-weighted functions, such as collective communication and communication using data types, might be considered as candidates for offloading.

The third direction might involve a smart communication layer designed and implemented so as to provide cache-aware rich communication functions. Since this layer does not involve the host CPU, much low latency communication can be accommodated.

VII. Acknowledgments

Grateful acknowledgement is made to Intel for providing the hardware, software and technical support associated with the Intel MIC architecture. They also read the draft paper and gave us comments to improve the paper.

References

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

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

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

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

[6] H. Wang, S. Potluri, M. Luo, A. Singh, X. Ouyang, S. Sur, and D. 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)*, sept. 2011, pp. 308 –316.

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