Lock-free Asynchronous Rendezvous Design for MPI Point-to-Point Communication

RAHUL KUMAR, AMITH R. MAMIDALA, MATTHEW J. KOOP, GOPAL SANTHANARAMAN, DHABALESWAR K. PANDA

Technical Report
OSU-CISRC-6/08-TR36

Lock-free Asynchronous Rendezvous Design for MPI Point-to-Point Communication *

Rahul Kumar, Amith R. Mamidala, Matthew J. Koop, Gopal Santhanaraman, and Dhabaleswar K. Panda

Network-Based Computing Laboratory
Department of Computer Science and Engineering
The Ohio State University
{kumarra, mamidala, koop, santhana, panda}@cse.ohio-state.edu

Abstract. Message Passing Interface (MPI) is the most commonly used method for programming distributed-memory systems. Most MPI implementations use a rendezvous protocol for transmitting large messages. One of the features desired in a MPI implementation is the ability to asynchronously progress the rendezvous protocol. This is important to provide potential for good computation and communication overlap to applications. There are several designs that have been proposed in previous work to provide asynchronous progress. These designs typically use progress helper threads with support from the network hardware to make progress on the communication. However, most of these designs use locking to protect the shared data structures in the critical communication path. Secondly, multiple interrupts may be necessary to make progress. Further, there is no mechanism to selectively ignore the events generated during communication. In this paper, we propose an enhanced asynchronous rendezvous protocol which overcomes these limitations. Specifically, our design does not require locks in the communication path. In our approach, the main application thread makes progress on the rendezvous transfer with the help of an additional thread. The communication between the two threads occurs via system signals. The new design can achieve near total overlap of communication with computation. Further, our design does not degrade the performance of non-overlapped communication. We have also experimented with different thread scheduling policies of Linux kernel and found out that the round robin policy provides the best performance. With the new design we have been able to achieve 20% reduction in time for a matrix multiplication kernel with MPI+OpenMP paradigm on 256 cores.

1 Introduction

Cluster based computing is becoming quite popular for scientific applications due to its cost effectiveness. The Message Passing Interface (MPI) is the most commonly used method for programming distributed memory systems. Many applications use MPI point-to-point primitives to send large messages. Typical MPI implementations use a rendezvous protocol for transmitting large messages. The rendezvous protocol involves a handshake to negotiate buffer availability and then the message transfer takes place. The message transfer usually occurs in a zero-copy fashion.

* This research is supported in part by DOE grants DE-FC02-06ER25755 and DE-FC02-06ER25749, NSF Grants CNS-0403342 and CCF-0702675; grants from Intel, Sun MicroSystems, Cisco Systems, and Linux Networks; and equipment donations from Intel, AMD, Apple, IBM, Microway, PathScale, SilverStorm and Sun MircroSystems.

One of the features desired in a quality MPI implementation is the ability to asynchronously progress the rendezvous protocol. This is important to provide potential for good computation and communication overlap to the applications. Many modern network interfaces offload network processing to the NIC and thus are capable of handling communication without the intervention of CPU. MPI provides non-blocking semantics so that the application can benefit from computation and communication overlap. The benefits of non-blocking semantics depend on the ability to achieve asynchronous progress. Thus, it is important to address this issue in the MPI implementation.

There are several designs that have been proposed previously to provide asynchronous progress. These designs typically use an additional thread to handle incoming rendezvous requests. For example, in [1], a RDMA read based threaded design is proposed to provide asynchronous progress. Though the basic approach has been proven to achieve good computation and communication overlap, there are several overheads associated with the implementation of the design. First, the existing design uses locking to protect the shared data structures in the critical communication path. Second, it uses multiple interrupts to make progress. Third, there is no mechanism to selectively ignore the events generated. In this paper, we propose an enhanced asynchronous rendezvous protocol which overcomes these limitations. Specifically, our design does not require locks in the communication path. In our approach, the main application thread makes progress on the rendezvous transfer with the help of an additional thread. The communication between the two threads occurs via system signals.

We have incorporated our design in MVAPICH [2], a popular MPI implementation over InfiniBand. The new design can achieve almost total overlap of communication with computation. Further, our design does not reduce the performance of non-overlapped communication. We have also experimented with different thread scheduling policies of Linux kernel and found out that round robin policy provides the best performance. With the new design we have been able to achieve 20% reduction in time for a matrix multiplication kernel with MPI+OpenMP paradigm on 256 cores.

2 Background

2.1 InfiniBand Overview

The InfiniBand Architecture [3] (IBA) defines a switched network fabric for interconnecting compute and I/O nodes. InfiniBand supports two types of communication semantics. They are called *Channel* and *Memory* semantics. In channel semantics, the sender and the receiver both explicitly place work requests to their Queue Pair (QP). After the sender places the send work request, the hardware transfers the data in the corresponding memory area to the receiver end. In memory semantics, Remote Direct Memory Access (RDMA) operations are used instead of send/receive operations.

InfiniBand supports event handling mechanisms in addition to polling. In InfiniBand, the Completion Queue (CQ) provides an efficient and scalable mechanism to report completion events to the application.The CQ can provide completion notifications for both send and receive events as well as many asynchronous events. In the polling mode, the application uses an InfiniBand verb to poll the memory locations associated with the completion queue. In the asynchronous mode, the application does not need to continuously poll the CQ to look for completions. The CQ will generate an interrupt when a completion event is generated. Further, IBA provides a mechanism by which only “solicited events” may cause interrupts. In this mode, the application

can poll the CQ, however on selected types of completions, an interrupt is generated. This mechanism allows interrupt suppression and thus avoid unnecessary costs (like context-switch) associated with interrupts.

2.2 Overview of MVAPICH Communication Protocols

MPI communication is often implemented using two general protocols:

Eager protocol: In this protocol, the sender process sends the message eagerly to the receiver. The receiver needs to provide buffers in advance for the incoming messages. This protocol has low startup overhead and is used for small messages.

Rendezvous protocol: The rendezvous protocol involves a handshake during which the buffer availability is negotiated. The message transfer occurs after the handshake. This protocol is used for transferring large messages. In the rendezvous protocol, the actual data can be transferred using RDMA write or RDMA read over InfiniBand. Both these approaches can achieve zero copy message transfer. MVAPICH [2] currently has both these modes for transferring data in the rendezvous protocol.

3 Existing Asynchronous Rendezvous Protocol

In this Section, we first explain the existing implementation for achieving asynchronous progress in the rendezvous protocol. The basic design was proposed in [1] and used InfiniBand's RDMA read capability together with IBA's event notification mechanism. Figure 1 (left) provides an overview of the approach. As shown in the figure, the main idea in achieving asynchronous progress is to trigger an event once a control message arrives at a process. This interrupt invokes a callback handler which processes the message and makes progress on the rendezvous. The required control messages which triggers the events in the existing scheme are: a) RNDV_START and b) RNDV_FINISH. In addition, the RDMA read completion also triggers a local completion event. This design provides good ability to overlap computation and communication via asynchronous progress. For example, if an application is busy doing computation, the callback handler can make progress via the interrupt mechanism. However, there are a couple of important details that arise in implementing the approach.

One main issue in the existing approach is the overhead of interrupt generation. As explained above, a total of three interrupts are generated for every rendezvous transfer of data. This can potentially degrade the performance for medium messages using this protocol. Further, it is not easy to provide for a mechanism to selectively ignore the events generated by the control messages. This feature can be used whenever the main application thread is already making progress and is expecting the control messages. Another important issue which cannot be overlooked is the overhead of locking/unlocking shared data structures. In this paper, we take into account all these issues and propose a new implementation alternative. Specifically, we aim to:

- Avoid using locks for shared data structures
- Reduce the number of events triggered by the control messages
- Provide for an ability for the process to selectively ignore the events generated

4 The Proposed Design

As explained above, the existing design has several limitations. In this Section, we explain our new approach of achieving asynchronous progress. Figure 1 (right) explains

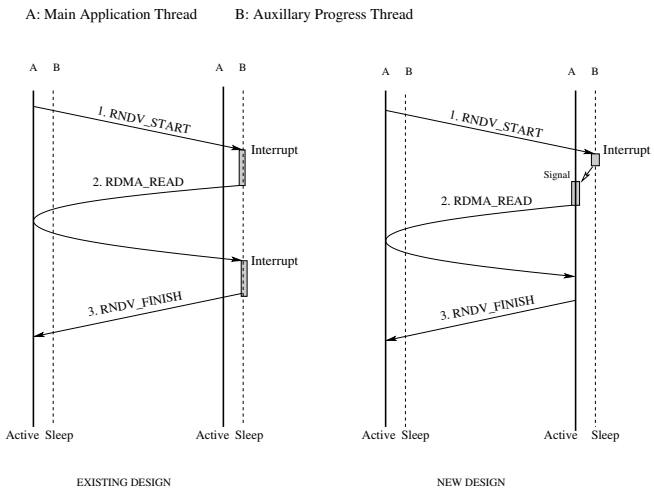


Fig. 1. Asynchronous Rendezvous Protocol Implementations

the basic idea in the new implementation. In our approach, each process creates an auxiliary thread at the beginning. The auxiliary thread waits for RNDV_START control message. As seen from the figure, the RNDV_START control message issued by the sender interrupts the auxiliary thread. This thread in turn sends a signal to the main thread to take the necessary action. This is different from the earlier approach where the auxiliary thread made progress on the rendezvous communication. Since, only one thread is involved with communication data structures, no locking mechanism is required for the data structures. In the second step, the main thread issues the RDMA read for the data transfer. After issuing RDMA read, the main thread resumes to perform the computation. Unlike the existing approach, the RDMA read completion does not trigger any interrupt in our design. We believe this interrupt does not help in overlap in Single Program Multiple Data (SPMD) programming model where each process performs the same task and the load is equally balanced. This was also observed in our experiments as can be seen in Figure 2. The figure shows the normalized execution time of mpi implementation of matrix multiplication kernel on 4 nodes for different matrix dimensions. Triggering of the interrupt on RDMA read completion can be easily added to the protocol if required. In our design, the main thread sends the RNDV_FINISH message soon after it discovers the completion of RDMA read.

There are several benefits of this new design. First, locks are avoided thus reducing contention for shared resources. Also, in our design the signal from the auxiliary thread is disabled by the main thread when it is not expecting a message from any process. By doing so, the main thread is not unnecessarily interrupted by an unexpected message since it does not have the receive buffer address to make progress on the communication. The main thread also disables signal if it is already inside the MPI library and making communication progress. Since the main thread can disable the interruption from the auxiliary thread, the execution time of the application is unaffected if rendezvous protocol is not used by the application. Also, the signal is enabled only if a non-blocking receive has been posted and not for blocking receives. Also, at most of the time the auxiliary thread is waiting for interrupts from the NIC and does not perform any communication processing. Therefore, as the auxiliary thread is I/O bound the dynamic priority of the thread is very high which helps in scheduling it quickly. Finally, the new design also cuts down the number of interrupts to one thus improving the communication performance.

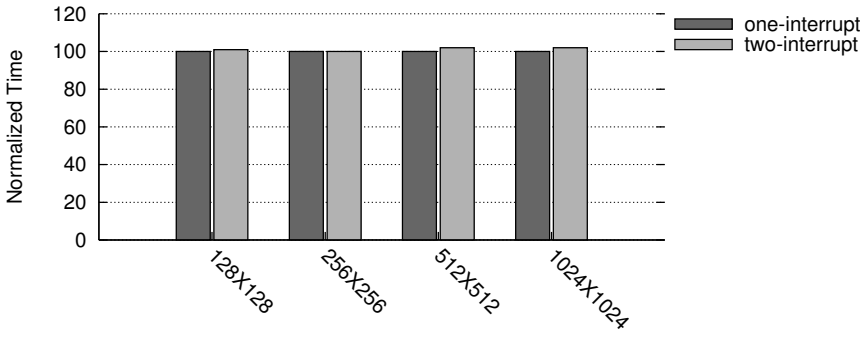


Fig. 2. Additional interrupt provides no improvement

5 Experimental Evaluation

The experiments were conducted on 64 node InfiniBand Linux cluster. Each machine has a dual 2.33 GHz Intel Xeon "Clovertown" quad-core processors for a total of eight cores per node. Each node is connected by DDR network interface card MT25208 dual-port Memfree HCA by Mellanox [4] through a switch. InfiniBand software support is provided through OpenFabrics/Gen2 stack [5], OpenFabrics Enterprise Edition 1.2.

5.1 Comparison with existing design

Figure 3 shows the performance of basic bandwidth micro-benchmark. We used OSU Benchmarks [6] for the experiment. The legend 'no-async' refers to the basic RDMA read based rendezvous protocol without any enhancements for asynchronous progress, 'existing-async' refers to the existing asynchronous progress design proposed in [1] and 'new-async' refers to the proposed design described in Section 4. Figure 3 shows

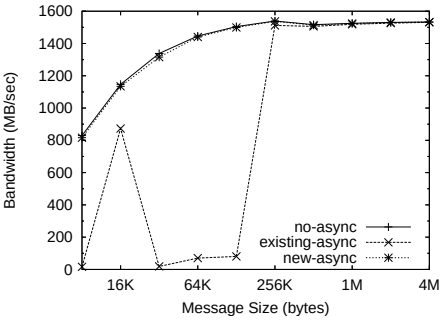


Fig. 3. Bandwidth for large messages

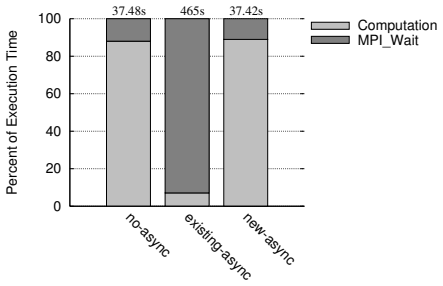


Fig. 4. NAS-SP Normalized Execution Time

that the bandwidth of the proposed design closely matches with the base bandwidth numbers, which matches our expectations. However, with the old design the bandwidth is very low. In the bandwidth test, the receiver posts several requests and waits for

the completion of all the pending messages. As several rendezvous start messages are received by the process, the auxiliary thread is continuously interrupted. Also, since the main thread is not involved in computation, both the threads concurrently poll the MPI library. The main thread cannot make any progress, however, it hinders the auxiliary thread from being scheduled on the processor. Therefore, due to exhaustion of CPU resources by the main thread the bandwidth performance is affected. The bandwidth performance is also nondeterministic as it depends on the scheduler to schedule the auxiliary process quickly. The effects of schedule is discussed in Section 5.3.

The performance of the new design is very similar to the base bandwidth performance since the main thread disables interrupts from the auxiliary thread when it is already inside the MPI library.

The poor performance of the existing design can be seen not only on micro-benchmarks but also in the performance of SP NAS Parallel Benchmark [7] application as can be seen in Figure 4. It can be seen from the figure that with the old design most of the execution time is wasted in MPI_Wait. In the remaining evaluations we do not show the performance of the old design. We found that the old design performs well when using an extra-core, however, it performs poorly when a single processor is assigned per process.

5.2 Overlap Performance

Figures 5 and 6 show the overlap performance of the proposed design. Sandia Benchmark [8] (SMB) has been used to evaluate the overlap capability of the implementation. Overlap potential at the receiver and at the sender have been shown in Figures 5 and 6 respectively. Since the base design and the proposed design employ RDMA read, al-

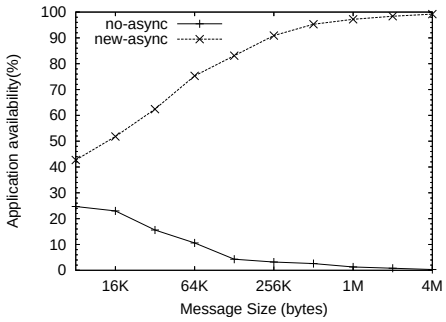


Fig. 5. Application availability at Receiver

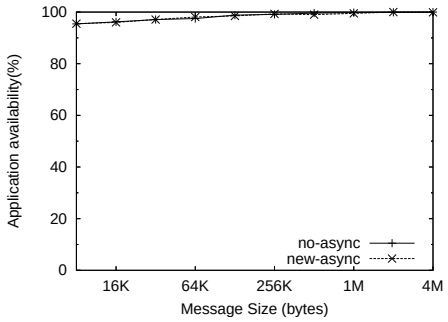


Fig. 6. Application availability at Sender

most total overlap is achieved at sender for both protocols. However, at the receiver the base RDMA read based protocol offers no overlap, as expected. The proposed design is able to achieve increasing overlap with increasing message size and reaches almost 100% overlap for messages greater than 1MB.

5.3 Effect of Scheduling algorithm

Figures 7 and 8 show the effect of scheduling algorithm on the overlap performance of the new design. Results for the default Linux scheduler, FIFO and Round robin have been compared. For each of the executions with different scheduling algorithm, the auxiliary thread is assigned the highest possible priority so that it is scheduled as soon as it is interrupted. Figure 7 shows the results for different message sizes. We observe that

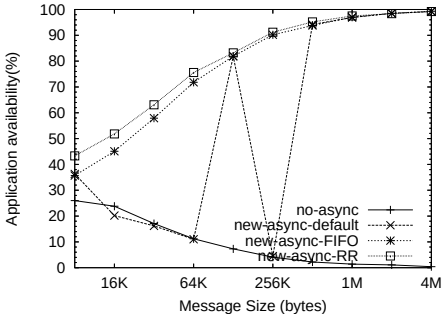


Fig. 7. Effect of scheduling algorithm

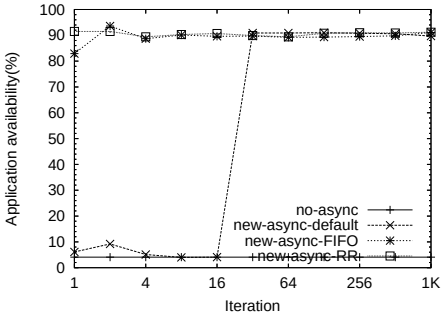


Fig. 8. Effect of increased time of execution

with the default scheduling algorithm, the performance is not consistent for all message sizes. At some message sizes the auxiliary thread is not scheduled on the processor on being interrupted. However, with FIFO scheduling algorithm the performance improves and is best for round-robin algorithm.

Figure 8 shows the overlap performance for 256KBytes message with increased number of iterations in each execution. From the figure, it is observed that with the default scheduling algorithm the performance of the design improves after a certain time interval. We feel that the improved performance is due to the dynamic priority scheme of Linux scheduling algorithm. Since the auxiliary thread hardly uses the CPU and is mostly waiting for completion events it is assigned a high dynamic priority which helps increase its performance. However, for FIFO and round robin the performance is optimal even for low number of iterations.

5.4 Application Performance

In this Section we use a matrix multiplication kernel to evaluate the application performance of the proposed design. The kernel uses Cannon's algorithm [9] and employs both MPI and OpenMP [10] programming models. The kernel requires the number of processes to be a perfect square. Since we wanted to use all 64 nodes of the cluster we could only use 4 cores per node in our experiments. However, since each thread is affined to a single core, the presence of the remaining unused cores of the nodes do not improve or affect the performance of the design. OpenMP programming model is used within the node and MPI is used for inter-node communication.

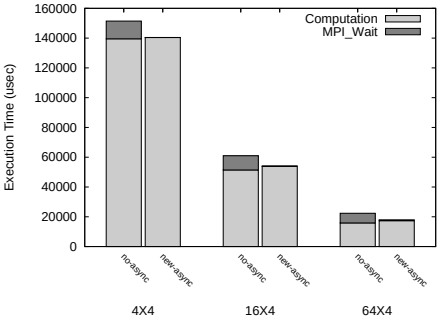


Fig. 9. Matrix Multiplication: Varying system size

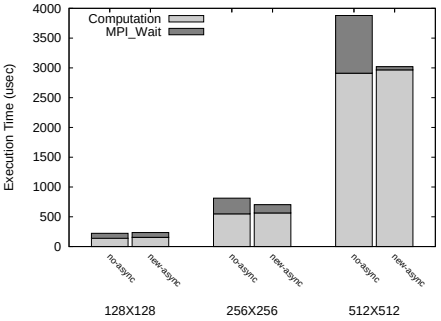
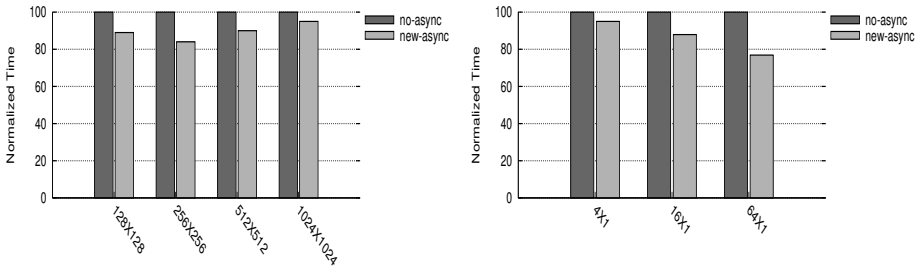


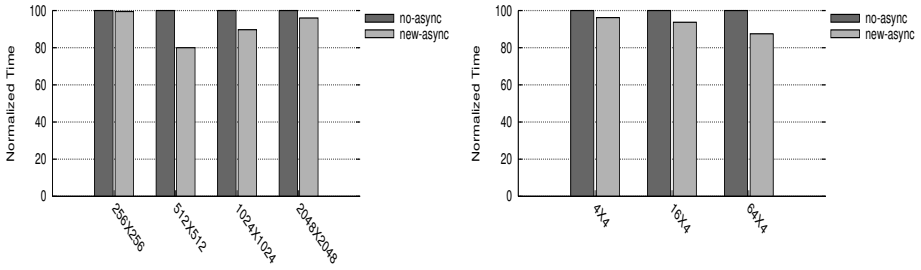
Fig. 10. Matrix Multiplication: Varying problem size

Figure 9 shows the application performance with increasing system sizes for a square matrix of dimensions 2048 elements. Each element of the matrix is a double datatype occupying eight bytes. As can be seen from the figure, the MPI_Wait time can be reduced by using the proposed design. Figure 10 shows the performance for increasing problem size on four nodes and dividing the work of each node among four of its cores using OpenMP. Reductions in MPI_Wait time can also be seen with different problem sizes. For matrix of 128X128 dimensions, no improvement is observed as the message communication is of size 4K Bytes which does not employ rendezvous protocol.



(a) Varying System Size (b) Varying Problem Size

Fig. 11. Matrix Multiplication: MPI x1 configuration



(a) Varying System Size (b) Varying Problem Size

Fig. 12. Matrix Multiplication: MPI x4 configuration

The performance of the new design has also been evaluated for MPI programming model without using OpenMP. The MPI implementation of matrix multiplication kernel was run in two configurations. In the first configuration, all the processes are involved in inter-node communication. This is achieved by launching only one process per node. In the second configuration, four processes per node are launched. In this setup, some of the process employ shared memory for communication whereas some processes are involved in inter-node communication. So the processes which use shared memory communication cannot achieve any overlap. The results for the first configuration can be

seen in Figures 11(a) and 11(b). The results for the second configuration can be seen in Figures 12(a) and 12(b). In Figure 12(a), no improvement is observed for matrix of dimensions 256X256. This is because the size of message transfer is less than 8KB. Messages of sizes lower than 8KB were not using rendezvous protocol in the experiments. Considerable improvement is observed for all the other problem sizes and system sizes. However, the improvement in performance is lower than when MPI+OpenMP programming model is used. This is because of decreased percentage of communication time (of the corresponding execution time) than the MPI+OpenMP program.

6 Related Work

Several studies have been done to show the importance of overlap capability in MPI library. Brightwell et al. [11] show the ability of applications to benefit from such features. Eicken et al. [12] propose for hardware support for active messages to provide communication and computation overlap. In our design we provide a mechanism to achieve overlap with the current hardware capability. Schemes to achieve overlap in one-sided communication have been proposed in [13]. Sur et al. [1] propose thread based rendezvous protocol which employs locks for protection. However, in our design we propose a lock free mechanism to achieve overlap.

7 Conclusions and Future Work

There are several designs that have been proposed in the past to provide asynchronous progress. These designs typically use progress helper threads with support from the network hardware to make progress on the communication. However, most of these designs use locking to protect the shared data structures in the critical communication path. Secondly, multiple interrupts may be necessary to make progress. Further, there is no mechanism to selectively ignore the events generated during communication.

In this paper, we proposed an enhanced asynchronous rendezvous protocol which overcomes these limitations. Specifically, our design does not require locks in the communication path. In our approach, the main application thread makes progress on the rendezvous transfer with the help of an additional thread. The communication between the two threads occurs via system signals. The new design achieves almost total overlap of communication with computation. Further, our design does not reduce the performance of non-overlapped communication. We have also experimented with different thread scheduling policies of Linux kernel and found out that round robin policy provides the best performance. With the new design we have been able to achieve 20% reduction in time for a matrix multiplication kernel with MPI+OpenMP paradigm on 256 cores. In future, we plan to carry out scalability studies of this new design for a range of applications and system sizes.

References

1. Sur, S., Jin, H.W., Chai, L., Panda, D.K.: RDMA Read Based Rendezvous Protocol for MPI over InfiniBand: Design Alternatives and Benefits. In: Symposium on Principles and Practice of Parallel Programming, (PPOPP '06). (March 2006)
2. Network-Based Computing Laboratory: MVAPICH: MPI over InfiniBand and iWARP. http://mvapich.cse.ohio-state.edu
3. InfiniBand Trade Association: InfiniBand Architecture Specification. http://www.infinibandta.com

4. Mellanox: Mellanox Technologies. http://www.mellanox.com
5. OpenFabrics Alliance: OpenFabrics. http://www.openfabrics.org/
6. OSU Micro-Benchmarks. http://mvapich.cse.ohio-state.edu/benchmarks/
7. NAS Parallel Benchmarks (NPB). www.nas.nasa.gov/Software/NPB/
8. Sandia National Laboratories: Sandia MPI Micro-Benchmark Suite. http://www.cs.sandia.gov/smb/
9. Kumar, V., Grama, A., Gupta, A., Karypis, G.: Introduction to parallel computing: design and analysis of algorithms. Benjamin-Cummings Publishing Co., Inc. (1994)
10. OpenMP. http://openmp.org/wp/
11. Brightwell, R., Underwood, K.D.: An Analysis of the Impact of MPI Overlap and Independent Progress. In: ICS '04: Proceedings of the 18th annual international conference on Supercomputing. (March 2004)
12. Eicken, T.V., Culler, D.E., Goldstein, S.C., Schauser, K.E.: Active messages: a mechanism for integrated communication and computation. In: ISCA '92: Proceedings of the 19th annual international symposium on Computer architecture, New York, NY, USA, ACM (1992)
13. Nieplocha, J., Tipparaju, V., Krishnan, M., Panda, D.K.: High performance remote memory access communication: The armci approach. Int. J. High Perform. Comput. Appl. (2006)