Open access • Proceedings Article • DOI:10.1109/IPDPSW.2010.5470854

Overlapping computation and communication: Barrier algorithms and ConnectX-2 CORE-Direct capabilities — Source link

Richard L. Graham, Steve Poole, Pavel Shamis, Gil Bloch ...+6 more authors

Institutions: Oak Ridge National Laboratory, Mellanox Technologies

Published on: 19 Apr 2010 - IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum

Topics: InfiniBand, Network interface and Scalability

Overlapping Computation and Communication: Barrier Algorithms and ConnectX-2 CORE-*Direct* Capabilities

Richard L. Graham, Steve Poole
Oak Ridge National Laboratory (ORNL)
Oak Ridge, TN, USA
Email: {rlgraham,spoole}@ornl.gov

Pavel Shamis, Gil Bloch, Noam Bloch,
Hillel Chapman, Michael Kagan, Ariel Shahar,
Ishai Rabinovitz
Mellanox Technologies, Inc.
Yokneam, Israel
Email: {pasha,gil,noam,hillel,michael,ariels,ishai}@mellanox.co.il

Gilad Shainer
Mellanox Technologies, Inc.
Sunnyvale , CA, USA
Email: shainer@mellanox.com

***Abstract*—This paper explores the computation and communication overlap capabilities enabled by the new CORE-*Direct* hardware capabilities introduced in the InfiniBand (IB) Host Channel Adapter (HCA) ConnectX-2. These capabilities enable the progression and completion of data-dependent communications sequences to progress and complete at the network level without any Central Processing Unit (CPU) involvement. We use the latency dominated nonblocking barrier algorithm in this study, and find that at 64 process count, a contiguous time slot of about 80 percent of the nonblocking barrier time is available for computation. This time slot increases as the number of processes participating increases. In contrast, CPU based implementations provide a time slot of up to 30 percent of the nonblocking barrier time. This bodes well for the scalability of simulations employing offloaded collective operations. These capabilities can be used to reduce the effects of system noise, and when using nonblocking collective operations have the potential to hide the effects of application load imbalance.**

Keywords-InfiniBand; CORE-Direct; Offload; Barrier;

I. Introduction

CPU clock speeds have remained essentially constant over the last several years. To keep up with the performance boosts expected as a result of the realization of Moore's law, the number of CPU cores used in high-end systems is rapidly increasing. System size on the Top500 list [1] has changed rapidly, and in November 2009 the top ten systems averaged 134,893 cores, with five systems larger than $100,000$ cores. This rapid increase in core count, and the associated increase in the number of compute threads used in a single job increases the urgency of dealing with system characteristics that impede application scalability.

Scientific simulation codes frequently use collective communications such as broadcasts, and data reductions. The ordered communication patterns used by high performance implementation of collective algorithms present an application scalability challenge. This impediment is further magnified by application load imbalance and system activity, or system noise [2], [3], delaying the collective operations.

CORE-*Direct* functionality, recently added to the InfiniBand ConnectX-2 HCAs by Mellanox Technologies [4], provides hardware support for offloading a sequence of data-dependent communications to the network. Once this sequence of operations is provided to the HCA it is progressed and completed with no CPU involvement, making the CPU available for computation. This functionality is well suited for supporting asynchronous Message Passing Interface (MPI) [5] collective operations. It provides hardware support for overlapping collective communications with application computation, which can be used to improve application scalability.

This paper briefly describes the InfiniBand CORE-*Direct* capabilities, and it is the first to demonstrate how these may be used for overlapping communication with computation. This paper demonstrates how to use the general purpose support for HCA based communication scheduling to implement scalable blocking and nonblocking asynchronous Barrier operations. These are fully progressed at the network level without any CPU involvement. The potential for overlapping nonblocking barrier algorithms with computation is examined.

II. Related Work

Work to delegate communication management, both point-to-point and collective, to processing units other than the main CPU has already been done. A number of studies explored the benefits of HCA-based collective operations, including those described in references [6], [7], [8], [9], [10], [11]. Several analyses of HCA-based broadcast algorithms are available in references [6], [7], [9]. Generally, these all tend to use HCA-based packet forwarding as a means of improving performance of the broadcast operation. Some

of the benefits of offloading barrier, reduce and broadcast operations to the HCA are described in References [8], [12], [10], and [9]. These show that barrier and reduce operations can benefit from reduced host involvement, efficient communication processing, and better tolerance to process skew. Keeping the data transfer paths relatively short in multi-stage communication patterns is appealing, as it offers a favorable payback for moving this work to the network. Even though much research has been done in this area, many problems are still to be solved. As such, these techniques have not gained wide acceptance.

Others studies have shown that the CPU may be used in support of asynchronous collective operation progress. Sancho et. al [13] have dedicated several CPU's to processing collective communications to improve their performance and scalability. Hoefler et. al [14], [15] have implemented non-blocking collectives, and investigated ways to minimize CPU overhead for progressing these collectives. Our approach for supporting both blocking and non-blocking collective operations is aimed at avoiding CPU involvement altogether in progressing these operations.

Amongst all the previous HCA-based collective implementations, only the efforts from Quadrics [16] for Elan 3/4 and IBM [17] for blue gene/P have been widely used. In this work, we study the benefits of collective offload in another popular interconnect technology, InfiniBand.

III. An Overview of InfiniBand

A detailed description of the CORE-*Direct* support and how this is used to implement support for MPI collective operations in Open MPI [18] is described elsewhere [19]. In this section we provide a brief description of these, as well as very recent enhancements to the MPI support.

The InfiniBand Architecture (IBA) [20] defines a communication architecture from the switch-based network fabric to transport layer communication interface for inter-processor communication. Processing nodes are connected as end-nodes to the fabric by Host Channel Adapters (HCAs). Fig. 1 illustrates the IBA specification of the IB support for the Reliable Connection (RC) transport type. With RC, reliable data transfer is implemented by the HCA hardware. Two processes communicate through a pair of IB queues that are created on the HCAs. This pair of send/receive queues is also referred to as a Queue Pair (QP). A communication operation is initiated by posting send, receive, read, write, or atomic Work Queue Elements (WQE) to the QP. Completion of a WQE results in a Completion Queue Event (CQE) being posted to a completion queue. The consumer obtains this event by polling the completion queue. Multiple QP's may share a Completion Queue (CQ). Remote Direct Memory Access (RDMA) write operations generate remote CQEs only if used with Immediate Data.

The general purpose CORE-*Direct* functionality introduced in ConnectX-2 aims to improve application scalability

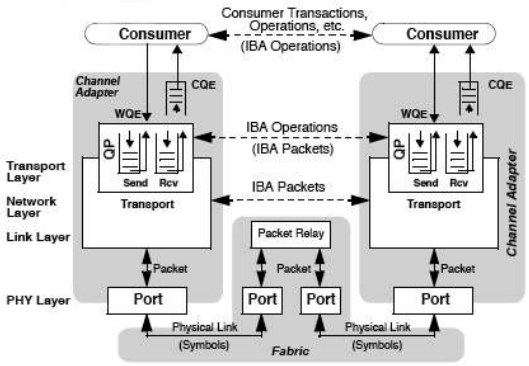

Figure 1. Standard InfiniBand Communication Stack.

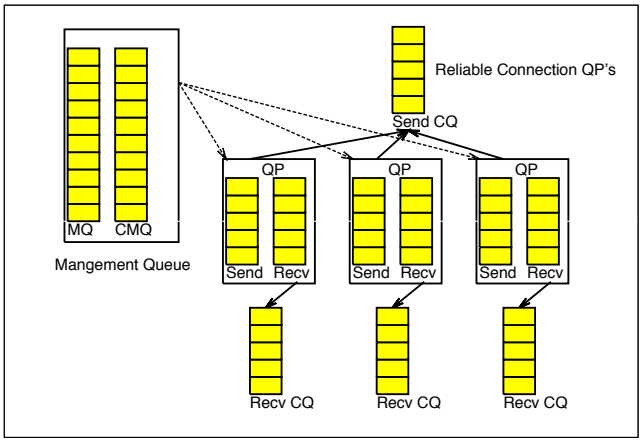

Figure 2. Queue structure, for a group with three communicating processes, used to implement the MPI Collectives on a per communicator basis.

by moving the management of chained network operations to the network card, with CPU involvement in collective operations limited to initiation and completion phases. HCA management of these operations allows for the possibility of overlapping computation and communication, thus reducing job execution time. This may also reduce the performance degradation of process skew when using nonblocking collective operations and the impact of system noise on collective operations, which tends to increase parallel applications job time when using such operations. To accomplish this new

proc 0	proc 1	proc 2	proc 3
Exchange with 1	Exchange with 0	Exchange with 3	Exchange with 2
Exchange with 2	Exchange with 3	Exchange with 0	Exchange with 1

Table I
Communication pattern for a four process recursive doubling barrier

hardware support for wait network tasks, Multiple Work Requests (MWR), and Management Queues (MQ's) are introduced. The following sub-section will describe this new support, and how it is used to implement MPI collective operations.

A. New Hardware Capabilities

The IBA defines several communication tasks, these include send, receive, read, write, and atomic tasks. CORE-*Direct* adds hardware support for cross QP synchronization operations - wait, send_enable, and receive_enable. Wait takes as argument a completion queue and the number of completion tasks to wait for, and can be used to order communications taking place using different QP's. Information about completed tasks consumed by a wait task may not be obtained from a completion queue, and must be inferred from QP completion ordering. For receive completion, this implies that the (MPI) library must keep track of the pre-posted receives, and use a counter to track completions to map a given completion to the appropriate pre-posted receive buffer. For example, if the HCA polls the CQ linked to the receive QP associated with rank X for n wait task completions, on completion, the next n buffers pre-posted to the receive queue will contain the data received by each receive. IB's ordered delivery guarantee makes it possible to identify the arriving data correctly. If a single completion queue was used for more than a single receive queue, there would be no way to correctly identify the source of the arriving data, and would not be able to correctly identify this data.

The Multiple Work Request is a list of InfiniBand communication tasks which the driver posts, in order, to the queues specified by the individual work requests. These tasks include the send, receive, 0-byte RDMA write and wait tasks. An MWR completion entry is posted after the task that is marked with the flag MQE_WR_FLAG_SIGNAL is processed by the HCA. In our implementation of MPI collective algorithms we mark the last task with the MQE_WR_FLAG_SIGNAL flag. The MWR may be used to chain a series of network tasks, and, once posted, the HCAs progress the communication, without using the CPU.

The Management Queue is a queue that is set up to handle Multiple Work Requests. The MQ has both a software handle which the driver uses for processing the MWR, and a hardware management queue (HW-MQ). This HW-MQ is no different than other queues, but serves a coordination role in the context of the MQ construct. When an MQ is created, a completion queue is also created, forcing a one-to-one mapping of MQ to MQ Completion queue, with the current implementation not allowing a single MQ completion queue to service multiple MQs. To help understand what happens when an MWR is posted to an MQ, refer to Figure 2, which shows the queue structure we have used to implement the MPI collective operations. When an MWR is posted to the MQ, the driver posts the individual work requests in-order to the specified QP's with no interleaving of individual tasks from different MWR's. Specifically, wait tasks are posted either to the QP specified in the task, or if a NULL queue is specified, the wait task is posted to the HW-MQ (labeled Management Queue in Figure 2). The send/receive tasks are posted to the specified QP. The driver will also generate additional tasks, based on the structure of the user's MWR. When a send/receive task in the MWR follows a wait task that is posted to the HW-MQ the send/receive task is posted to the specified QP, but is not enabled for send/receive, and it cannot be processed by the HCA until it is enabled. In addition, a send/receive-enable task is posted to the HW-MQ after the wait task, which will cause the corresponding send/receive task to be enabled after the wait task completes.

IV. MPI Collective Design

The new Management-Queue, Multiple Work Request, and the wait tasks are used with the pre-existing IB functionality to implement offloaded, asynchronous collective operations. The queue structure used is displayed in Figure 2. While the current design is aimed at an MPI implementation, a virtually identical design can be used to implement collective operations supported by other communication systems.

Collective communications are managed on a per-communicator basis, to ensure independent progress. Each rank in the communicator uses a single MQ and a RC-QP for each other rank with which it communicates. Receive completion is handled by the wait task, with no user-level access to the receive CQ. The library tracks the receive buffers posted to each QP to retrieve the data for subsequent send tasks. Shared receive queues are not used so that the data source can be identified, without the benefit of CQE. Send tasks are completed asynchronously, out of the critical path using a single send completion queue.

Allocating a set of queues, QPs and MQs per communicator is necessary to provide independent progress between communicators and to properly identify arriving data. The need for per communicator resources in conjunction with the limited on-die resources and caches impacts performance when the number of communicators using the HCA becomes large. However, the ability to offload the progression of collective operations becomes more important as the number of processes participating increases, thus rendering this potential performance degradation relative to the same collective algorithm managed by a CPU less of a problem. The added benefits of being able to overlap computation with communication, allowing the application to make progress on both fronts at the same time seems beneficial in many circumstances, including the many communicator scenario.

As suggested in the previous paragraph, keeping the QP footprint as small as possible is important for maintaining high-performance as the scale of the collective operations increases. We keep the number of queue pairs down by

communicating only with a log_2 number of ranks in the barrier algorithm. We have also demonstrated [21] that we can use the same set of connections for the rooted MPI_Broadcast() algorithm with a log_2 scaling algorithm, for any root in the communicator.

MPI collective operations are implemented using an interdependent sequence of network operations executed by each process in the communicator. Each process participating in a given collective operation executes a different sequence of network operations, with reduction operations also manipulating the data being transferred. These local communication patterns determine the MWR task list used by each process in the communicator. MWR completion is used to determine MPI-level completion. To avoid Receiver-Not-Ready Negative Acknowledgements (RNR-NACK) and the associated retransmission delays we pre-post receive WQE's, and keep track of the receive buffers associated with the WQE, for use by subsequent send tasks. When the MPI application can reuse its send buffer immediately upon return from the call initiating the collective operation, we do not include send completion wait tasks in the MWR tasks list. We do this to avoid the additional network half round trip latency. This is the case for barrier operations that do not include any user data, or when the library keeps an internal copy of the data. However, we always track send completion for proper resource management, keeping this out of the performance critical path, when possible.

The number of tasks posted to the HCA queues using offloaded MWRs is similar to that posted with equivalent main memory based implementations of the same algorithm. The offload based methods use the wait and send/receive-enable tasks in lieu of using main memory CPU cycles to achieve the same effects. The constraint of using the MWR approach is that the full set of communications to be progressed needs to be defined up-front, and therefore IB queue resource limits are more likely to limit the scalability of offloaded based implementations than the main-memory based implementations. However, the limit on the size of an MWR is the same as the limit on the size of a send queue, which is on the order of sixteen thousand entries in the current version of Mellanox's IB stack. For logarithmically scaling algorithms this is not expected to be a problem in the foreseeable future, but for algorithms that require the use of more than a single MWR to describe a given collective operation, synchronization between MWRs may need to be added.

Pre-registered memory is used for task buffers, and large data collective operations are segmented to manage memory usage and allow for pipelining collective operations. Blocks of receive buffers are pre-posted to each QP, to avoid the performance degradation associated with RNR-NACKs.

The blocking and nonblocking barrier operations used to employ a recursive doubling algorithm are described in [22]. The algorithm for N=2^L number of ranks has L steps. At each step $l = 0, 1, ..., L-1$, each rank signals a rank 2^l ranks away, and waits on a signal from the same rank before proceeding to the next step in the algorithm. If the number of process M is not a power of two, and N is the largest power of two smaller than M, each rank $r = N, N+1, ..., M-1$, called an extra rank, is paired with the rank $r - N$. The barrier algorithm for M ranks has an initiation phase where each rank $r = N, N+1, ..., M-1$ signals its partner $r - N$. After the recursive doubling algorithm is executed with the N ranks, each rank paired with an extra rank signals their partner. Implementation details for the blocking and nonblocking operations are similar, with dissimilarities due to the different completion semantics. As an example, the communication pattern for the MPI_Barrier collective operations, is given in Table I.

The communication pattern corresponding to the four process recursive doubling algorithm is shown in Table I. The MWR setup by each rank is shown in Table II. This

proc 0	proc 1	proc 2	proc 3
send to 1	send to 0	send to 3	send to 2
recv wait from 1	recv wait from 0	recv wait from 3	recv wait from 2
send to 2	send to 3	send to 0	send to 1
recv wait from 2	recv wait from 3	recv wait from 0	recv wait from 1

Table II
MWR TASK LIST FOR EACH RANK PARTICIPATING IN A FOUR PROCESS RECURSIVE DOUBLING BARRIER.

proc 0	proc 1	proc 2	proc 3
send to 1 send enabled	send to 0 send enabled	send to 3 send enabled	send to 2 send enabled
send to 2 send not enabled	send to 3 send not enabled	send to 0 send not enabled	send to 1 send not enabled

Table III
QUEUE PAIR TASK LIST FOR EACH RANK PARTICIPATING IN A FOUR PROCESS RECURSIVE DOUBLING BARRIER. A DIFFERENT QP IS USED FOR EACH UNIQUE COMMUNICATION PAIR. SEND TASKS MAY BE POSTED TO THE HCA BUT WILL NOT BE PROCESSED UNTIL THEY ARE MARKED AS ENABLED.

proc 0	proc 1	proc 2	proc 3
send enable 1	send enable 0	send enable 3	send enable 2
MQ recv wait from 1	MQ recv wait from 0	MQ recv wait from 3	MQ recv wait from 2
send enable 2	send enable 3	send enable 0	send enable 1
MQ recv wait from 2	MQ recv wait from 3	MQ recv wait from 0	MQ recv wait from 1

Table IV
MANAGEMENT QUEUE TASK LIST FOR EACH RANK PARTICIPATING IN A FOUR PROCESS RECURSIVE DOUBLING BARRIER. COMPLETION OF THE LAST WAIT OPERATION ON THE MQ SIGNALS MWR COMPLETION.

particular instance of the algorithm uses the send/receive channel semantics for communications. The driver processes the MWR posting tasks to the MQ, and to each of the QP's that will be used to send and receive the data. Table III shows the tasks posted to the QP's for each rank participating in the barrier, and Table IV describes the tasks posted to the MQ, for each of the four ranks participating in the barrier collective operation. The tasks in each queue progress in order, as they become eligible for execution. Sends are executed when enabled, receives are completed as data arrives, and wait events are completed when as tasks complete in the queue specified in the wait task. While each queue is progressed independently, the ordering dependencies brought about by the send-enable task, the wait task, and the arrival of data at the receive queue are what make it possible to build globally ordered communication patterns.

When handling a received send request, the network adapter will fetch the appropriate receive WQE. As we are mainly interested in the CQE written to the CQ to progress the collective operation, we can use alternatives to the send operation. One alternative is to use RDMA write with immediate flag set. This operation does not use receive WQE as the remote data address is defined in the RDMA write request. Still, it generate a CQE. We utilize this optimization in the barrier algorithms.

V. Benchmark Results

In this section we focus primarily on studying the potential for overlapping computation with collective communications. We study the performance of the nonblocking MPIX_Ibarrier() recently voted into the MPI-3 draft standard, and, for comparison, also present the most recent benchmark results for MPI_Barrier(). Since no user data is involved in barrier operations, the performance of a given algorithm is determined by network latency, and the latency of the (MPI) software stack. As such, compared to other collective algorithms which also send user data over the network, the opportunities for overlap are relatively modest. However the simplicity of the algorithm makes it a good first candidate for studying some of the newly developed CORE-*Direct* capabilities. We compare the performance characteristics of barrier operations using the HCA to manage these operations to implementations in which the CPU is used to manage the barrier algorithms.

A. Experimental Setup

The performance measurements were all taken on an eight node, dual socket quad-core, 3.00 Gigahertz Intel Xeon Quad-core X5472 with 32 gigabytes of memory. The system runs Red Hat Enterprise Linux Server 5.1, kernel version 2.6.18-53.el5, a dual port quad data rate ConnectX-2 HCA, and a switch running Mellanox firmware version 2.6.8000. This is pre-release version of the firmware, and provides the first working implementation of the new Management Queue capability.

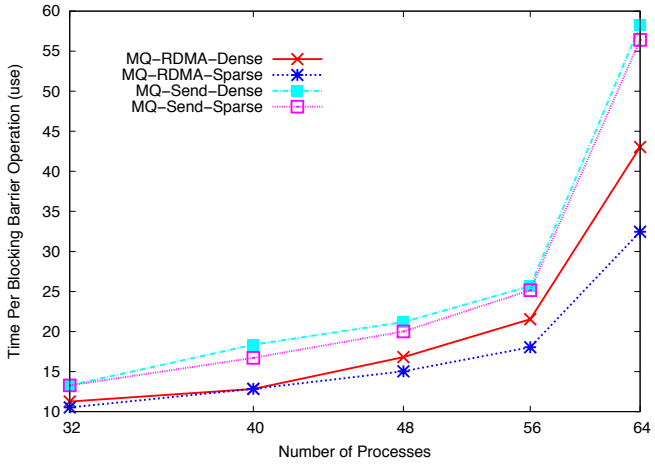

Figure 3. Eight node offloaded MPI_Barrier performance data in units of micro-seconds per call. MQ-RDMA indicates the use of RDMA send with Immediate Data, MQ-Send indicates the use of send with channel semantics, Dense indicates that each rank in the communicator has an RC connection to each other rank, and Sparse indicates RC connections only to the $\log_2$ ranks with which communication takes place.

The prototype offloaded IB collectives are implemented within version 1.5 of the Open MPI code base, as a new collective module. To measure raw barrier time we measure the completion time of a tight loop over barrier calls, and report the average time for the MPI rank 0 rank. Similarly, for the nonblocking collectives we loop over nonblocking barrier initiation and barrier completion. To measure the overlap characteristics of these collective operations, we modify the ideas introduced in the COMB [23] benchmark, adapting them for collective operations. Communication-computation overlap is measured in two ways; (1) Initiating the collective operation and then looping over (work loop, MPI_Test()) until the operation completes, making sure the busy loop does not increase overall operation completion beyond that of the raw operation; (2) Initiating the collective operation, work loop, and then wait for operation completion, with the work loop starting at about 10 percent of the raw completion time, and incrementing this work loop by 10 percent up to about 100 percent of raw completion time. The work loop is created by looping over the "nop" asm instruction.

B. Discussion

MPI_Barrier() latency as a function of process count is presented in Figure 4. In this figure and all subsequent figures MQ is used to label results obtained using HCA offloading approaches, and PTP is used to label results obtained with point-to-point based collective algorithms progressed by the CPU in main memory. While results for the barrier algorithm have been presented previously [19], the current results are significantly improved at the larger

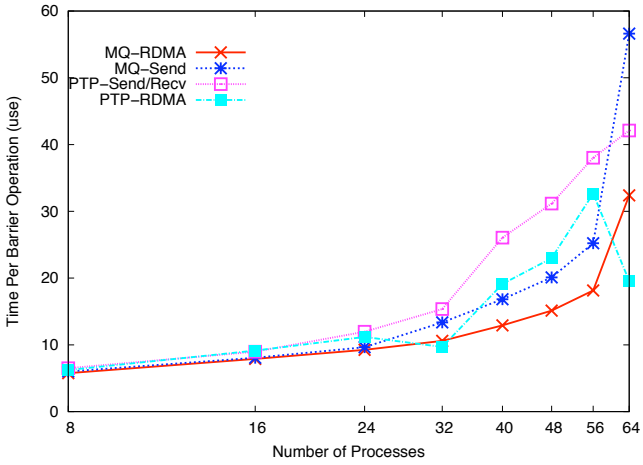

Figure 4. Eight node MPI_Barrier performance data in units of micro-seconds per call. MQ-RDMA uses MWR and RDMA send with immediate data, MQ-send uses MWR and send channel semantics, PTP-RDMA is progressed in main-memory using point-to-point with RDMA semantics polling a memory location for send completion, and PTP-send/recv is progressed in main-memory using point-to-point with send/recv channel semantics.

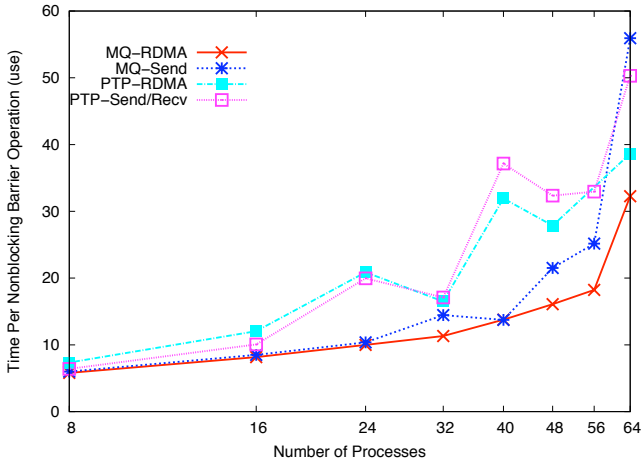

Figure 5. Eight node MPIX_Ibarrier performance data in units of micro-seconds per call. MQ-RDMA uses MWR and RDMA send with immediate data, MQ-send uses MWR and send channel semantics, PTP-RDMA is progressed in main-memory using point-to-point with RDMA semantics polling a memory location for send completion, and PTP-send/recv is progressed in main-memory using point-to-point with send/recv channel semantics.

process counts. This isa result of the reduce number of queue pairs used in the algorithm, decreasing the number of network context resources the HCA needs to manage. This gives the HCA more opportunities to cache such context data and increase performance. In addition, we use immediate data with RDMA write as a means of signaling data arrival, not actually fetching the receive work entry, as a further optimization. Adding these changes reduces the latency of barrier at 64 process from 54 micro-seconds down to 32. Figure 3 displays the results of these experiments. These changes are also used to implement the nonblocking barrier algorithm.

As the results show, over the range of communicator sizes used, the performance of the offloaded barrier is very good compared to the other three approaches used. However, at 64 processes, the CPU based RDMA method out performs the HW-MQ RDMA based approach by polling for completion in main memory, thus reducing the pressure put on the HCA network context cache. It is expected that as the number of ranks participating in the barrier increases, the CPU based approach will also be affected by the HCA caching effects.

The raw performance of the the nonblocking barrier algorithm is shown in Figure 5. The performance of the offloaded based implementation (labeled MQ) is similar to that of the blocking barrier. This is not surprising, as the algorithms are essentially identical. However, the CPU based nonblocking barrier implementations do not perform as well as their blocking counterparts. Since the main-memory based algorithms are used to give a baseline performance expectation for the offloaded collective implementations, and are not a primary focus of this study, we did not thoroughly investigate the observed performance anomalies.

Two measures of overlap are used to study the ability to overlap collective communications with computation. Measuring the overlap by alternating between small work quanta and testing for collective completion, provides an opportunity for both offload based collectives as well as CPU based collective operations to maximize the amount of compute work that can be done. Figure 6 presents the overlap capabilities the nonblocking barrier implementation provides during the recursive doubling algorithm, as a function of process count. Both HCA and CPU based algorithms provide some degree of overlap opportunity, ranging from approximately 30-45% CPU availability at eight processes, and up to around 90% at 64 process count. While we measured up to 20% difference in CPU availability between collective algorithms, all show an opportunity for overlap, with the offloaded based methods having a smoother profile as a function of process count. Since the HCA offloaded methods do not use the CPU to progress the nonblocking barrier operations, it is not surprising that these show a smoother CPU availability profile. In contrast, the CPU based algorithms rely on calls to the MPI library to progress the collective communications, thus being more sensitive to the rate at which calls into the MPI library take place. While high process availability is desirable, a combination of high availability and short collective operation duration is more desirable, thus requiring less compute time to overcome the barrier latency. Such a measure could be CPU availability divided by collective operation duration. The MQ based algorithms perform better using this measure.

From an application perspective, it is far more desirable

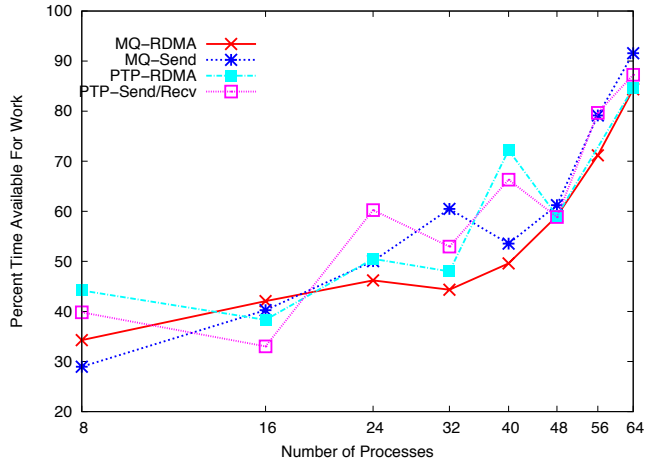

Figure 6. CPU cycles available, as percent of total nonblocking barrier time, for computation for nonblocking barrier. Computed by alternating a busy loop with MPI completion testing, until MPI completion. MQ-RDMA uses MWR and RDMA send with immediate data, MQ-send uses MWR and send channel semantics, PTP-RDMA is p progressed in main-memory using point-to-point with RDMA semantics polling a memory location for send completion, and PTP-send/recv is progressed in main-memory using point-to-point with send/recv channel semantics.

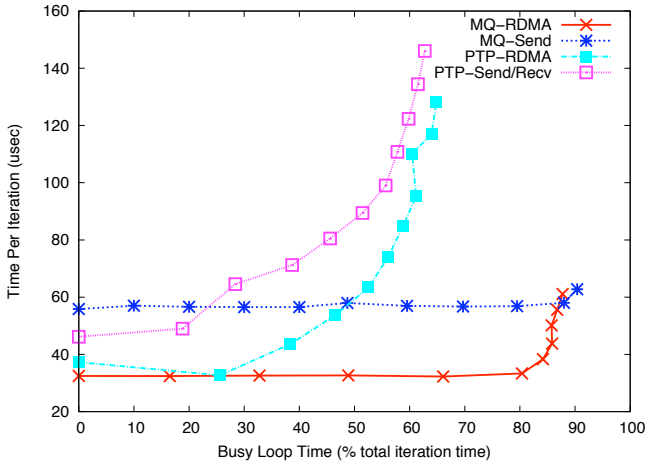

Figure 7. CPU cycles for computation for nonblocking barrier. Computed by first executing a busy loop right after posting the barrier, and then waiting for MPI completion. MQ-RDMA uses MWR and RDMA send with immediate data, MQ-send uses MWR and send channel semantics, PTP-RDMA is progressed in main-memory using point-to-point with RDMA semantics polling a memory location for send completion, and PTP-send/recv is progressed in main-memory using point-to-point with send/recv channel semantics.

to have a single compute time slice per collective operation, rather than many small time slices, making it easier to do useful computation while hiding communication latency. Therefore, we measure how much work can be done after the collective operation is performed and before waiting to complete the operation without impacting its overall completion time. The results are presented in Figure 7. As this figure shows, delegating the progression of the collective operations to the HCA makes the CPU available for relatively long periods of time. The MQ-RDMA based approach provides about 80% CPU availability before starting to impact the nonblocking barrier completion time. This tends to increase as the number of processes involved in the collective increases. On the other hand, the algorithms that rely on the CPU for progression provide at most 30% CPU availability at 64 processes before impacting the collective operation time. This result is not at all surprising, as the CORE-*Direct* functionality is designed with the goal of providing hardware support for overlapping computation and communications.

VI. Conclusions

In this paper we have evaluated the performance characteristics of the latency sensitive blocking and nonblocking barrier algorithms, when using the CORE-*Direct* functionality provided by ConnectX-2. We have shown that this functionality provides support for well performing barrier operations, similar to that of highly optimized CPU based barrier operations. It also provides the necessary support for effectively overlapping computation and communications. The latter characteristic is an effective strategy for mitigating application effects of system noise, and when using nonblocking collective operations, application load imbalance. As such, this capability is a key ingredient to improved scaling for applications using collective operations. Future studies will examine the overlap characteristics of collective operations involving user data, and reduction operations.

Acknowledgment

Research sponsored by the Office of Advanced Scientific Computing Research's FASTOS program; U.S. Department of Energy, and performed at ORNL, which is managed by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725. The HPC Advisory Council (http://www.hpcadvisorycouncil.com) provided computational resource for testing and data gathering.

References

[1] "Top 500 Super Computer Sites." in *http://www.top500.org/*.

[2] R. Mraz, "Reducing the variance of point to point transfers in the ibm 9076 parallel computer," in *Proceedings of the 1994 ACM/IEEE conference on Supercomputing*, November 1994, pp. 620–629.

[3] F. Petrini, D. J. Kerbyson, and S. Pakin, "The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of asci q," in *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, November 2003, p. 55.

[4] "Mellanox Technologies," http://www.mellanox.com/, 2008. [Online]. Available: http://www.mellanox.com/

[5] *MPI: A Message-Passing Standard*, Message Passing Interface Forum, June 2008.

[6] K. Verstoep, K. Langendoen, and H. E. Bal, "Efficient Reliable Multicast on Myrinet," in *the Proceedings of the International Conference on Parallel Processing '96*, 1996, pp. 156–165. [Online]. Available: citeseer.nj.nec.com/verstoep96efficient.html

[7] R. A. Bhoedjang, T. Ruhl, and H. E. Bal, "Efficient Multicast on Myrinet Using Link-Level Flow Control," in *27th ICPP*, 1998. [Online]. Available: citeseer.nj.nec.com/bhoedjang98efficient.html

[8] D. Buntinas, D. K. Panda, and P. Sadayappan, "Fast NIC-Level Barrier over Myrinet/GM," in *Proceedings of IPDPS*, 2001. [Online]. Available: citeseer.nj.nec.com/buntinas01fast.html

[9] W. Yu, D. Buntinas, and D. K. Panda, "High Performance and Reliable NIC-Based Multicast over Myrinet/GM-2," in *Proceedings of the International Conference on Parallel Processing '03*, October 2003.

[10] A. Moody, J. Fernandez, F. Petrini, and D. Panda, "Scalable NIC-based Reduction on Large-Scale Clusters," in *SC '03*, November 2003.

[11] Y. Huang and P. K. McKinley, "Efficient collective operations with ATM network interface support," in *ICPP, Vol. 1*, 1996, pp. 34–43.

[12] D. Buntinas and D. K. Panda, "NIC-Based Reduction in Myrinet Clusters: Is It Beneficial?" in *SAN-02 Workshop (in conjunction with HPCA)*, Feb. 2003.

[13] J. C. Sancho, D. J. Kerbyson, and K. J. Barker, "Efficient offloading of collective communications in large-scale systems," *Cluster Computing, IEEE International Conference on*, vol. 0, pp. 169–178, 2007.

[14] T. Hoefler, A. Lumsdaine, and W. Rehm, "Implementation and performance analysis of non-blocking collective operations for mpi," in *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*. New York, NY, USA: ACM, 2007, pp. 1–10.

[15] T. Hoefler and A. Lumsdaine, "Optimizing non-blocking Collective Operations for InfiniBand," in *Proceedings of the 22nd IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 04 2008. [Online]. Available: http://www.unixer.de/publications/img/hoefler-cac08.pdf

[16] "Quadrics," *http://www.quadrics.com/*.

[17] S. Kumar, G. Dozsa, G. Almasi, P. Heidelberger, D. Chen, M. E. Giampapa, M. Blocksome, A. Faraj, J. Parker, J. Ratterman, B. Smith, and C. J. Archer, "The deep computing messaging framework: generalized scalable message passing on the blue gene/p supercomputer," in *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*. New York, NY, USA: ACM, 2008, pp. 94–103.

[18] E. Garbriel, G. Fagg, G. Bosilica, T. Angskun, J. J. D. J. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. Castain, D. Daniel, R. Graham, and T. Woodall, "Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation," in *Proceedings, 11th European PVM/MPI Users' Group Meeting*, 2004.

[19] R. L. Graham, S. Poole, P. Shamis, G. Bloch, N. Bloch, H. Chapman, M. Kagan, A. Shahar, I. Rabinovitz, and G. Shainer, "Connectx-2 infiniband management queues: First investigation of the new support for network offloaded collective operations," in *Accepted for the 10th IEEE/ACM International Symposium CCGrid*, 2010.

[20] InfiniBand Trade Association, "The InfiniBand Architecture," http://www.infinibandta.org/specs.

[21] R. L. Graham, P. Shamis, G. Bloch, N. Bloch, I. Rabinovitz, H. Chapman, M. Kagan, A. Shahar, and G. Shainer, "Connectx-2 core-direct enabled asynchronous broadcast collective commu nications," in *sumbitted to International Supercomputing Conference - ISC'10*, 2010.

[22] R. L. Graham and G. Shipman, "Mpi support for multicore architectures: Optimized shared memory collectives," in *Proceedings of the 15th European PVM/MPI Users' Group Meeting*, 2008.

[23] W. Lawry, C. Wilson, A. Maccabe, and R. Brightwell, "Comb: a portable benchmark suite for assessing mpi overlap," in *2002 IEEE International Conference on Cluster Computing*, 2002, pp. 472–475.