2011 19th Annual IEEE Symposium on High Performance Interconnects

# Designing Non-blocking Broadcast with Collective Offload on InfiniBand Clusters: A Case Study with HPL*

K. Kandalla[1], H. Subramoni[1], J. Vienne[1], S. Pai Raikar[1], K. Tomko[2], S. Sur[1], and D. K. Panda[1]

[1] *Department of Computer Science and Engineering*
*The Ohio State University*
{kandalla, subramon, viennej, pai, surs, panda}@cse.ohio-state.edu

[2] *Ohio Supercomputer Center*
*Columbus, Ohio*
{ktomko}@osc.edu

***Abstract*—The upcoming MPI-3.0 standard is expected to include non-blocking collective operations. Non-blocking collectives offer a new MPI interface, using which an application can decouple the initiation and completion of collective operations. However, to be effective, the MPI library should provide a high-performance and scalable implementation. One of the major challenges in designing an effective non-blocking collective operation is to ensure progress of the operation while processors are busy in application-level computation. The recently introduced Mellanox ConnectX-2 InfiniBand adapters offer a task-offload interface (CORE-Direct) that enables communication progress without requiring CPU cycles.**

**In this paper, we present the design of a non-blocking broadcast operation (MPI_Ibcast) using the CORE-Direct offload interface. Our experimental evaluations show that our implementation delivers near perfect overlap, without penalizing the latency of the MPI_Ibcast operation. Since existing MPI implementations do not provide non-blocking collective communication, scientific applications have been modified to implement collectives on top of MPI point-to-point operations to achieve overlap. HPL is an example of an application use-case scenario for non-blocking collectives. We have explored the benefits of our proposed network offload based MPI_Ibcast implementation with HPL and we observe that HPL can achieve its peak throughput with significantly smaller problem sizes, which also leads to an improvement in its run-time by up to 78%, with 512 processors. We also observe that our proposed designs can minimize the impact of system noise on applications.**

## I. Introduction

The fastest super-computing systems currently offer sustained peta-flop performance and allow scientists to scale their parallel applications to tens of thousands of processors. The Message Passing Interface (MPI) [1] has been a popular programming model for the High Performance Computing applications running on these systems for the last couple of decades. The current MPI Standard, MPI-2.2, defines a set of collective operations that are used to communicate data among a group of participating processes. Currently, MPI only defines blocking collective operations, i.e. the application has to wait until the collective call completes. This limits the overall performance and scalability of various scientific parallel applications. In addition, collectives may also get delayed due to system noise [2], [3], further affecting application performance. For these reasons, some applications employ non-blocking collective operations by re-implementing them on top of MPI point-to-point operations. The High-performance Linpack (HPL) benchmark [4] is used to generate the Top500 list [5] and it uses MPI_Isend, MPI_Irecv and MPI_Test calls to implement non-blocking broadcast algorithms of varying complexities. Due to sustained interest by the community, the upcoming MPI-3 standard will include non-blocking collective communication operations. Using these non-blocking collectives, applications can be re-engineered to achieve communication/computation overlap.

### A. Motivation

The real benefits offered by various MPI implementations are likely to be the key driver for acceptance of non-blocking collectives. Simplistic designs that require the host processors to progress the MPI library are not portable and may also minimize the potential for overlap. In this context, real benefits of non-blocking collectives can only be achieved with corresponding network support. It is also believed that network based solutions can be resilient to system noise because they require little intervention from the host processors.

InfiniBand is a very popular switched interconnect standard being used by almost 41% of the Top500 Super-computing systems [5]. Since InfiniBand is so widely used, efficient support of non-blocking collectives in MPI implementations on InfiniBand is critical. Mellanox recently introduced network offload capabilities in their ConnectX-2 [6] adapter. Using this feature, generic lists of communication tasks can be offloaded to the network interface [7]. Such an interface eliminates the need for the host processor to progress communication and provides a low-level mechanism which can be leveraged to design non-blocking collective communication algorithms. However, in order to leverage the full benefits of this low-level mechanism, MPI libraries must be designed in a highly efficient manner.

In this paper, we propose fully functional designs for the MPI_Ibcast operation based on InfiniBand ConnectX-2's CORE-Direct network-offload feature. We have integrated our design into MVAPICH2 [8], a popular MPI implementation for InfiniBand, iWARP and RoCE technologies. MVAPICH2 is currently used by more than 1,600 organizations in 62 countries worldwide.

*This research is supported in part by U.S. Department of Energy grants #DE-FC02-06ER25749, #DE-FC02-06ER25755 and contract #DE-AC02-06CH11357; National Science Foundation grants #CCF-0621484, #CCF-0702675, #CCF-0833169, #CCF-0916302 and #OCI-0926691; grant from Wright Center for Innovation #WCI04-010-OSU-0; grants from Intel, Mellanox, Cisco, QLogic, and Sun Microsystems; Equipment donations from Intel, Mellanox, AMD, Obsidian, Advanced Clustering, Appro, QLogic, and Sun Microsystems.

978-0-7695-4537-0 2011

DOI 10.1109/HOTI.2011.14

### B. Contributions

We list the important contributions of this paper below:

1) Our proposed network-offload based MPI_Ibcast designs offer full communication/computation overlap without penalizing the latency.
2) We demonstrate that by using our proposed design, it is possible to improve throughput of applications that use MPI_Ibcast.
3) We show that our proposed MPI_Ibcast implementation minimizes the impact of system noise on the performance of parallel applications.
4) HPL is typically run at full scale, with large problem sizes to measure the peak throughput of supercomputers. We observe that with our proposed MPI_Ibcast implementation, HPL can achieve the peak throughput with much smaller problem sizes, which also leads to improved run-times (benefits of up to 78%).

## II. Background

In this section we give the necessary background information for our work.

### A. InfiniBand and ConnectX-2 Network Interface

Current generation InfiniBand QDR network cards and switches can deliver 36 Gbps end-to-end bandwidth and about 1.0 to 1.5 $\mu$s latency. The ConnectX-2 [6] network interface is the latest adapter from Mellanox. Along with all of the standard InfiniBand features, it offers a new network offloading feature called CORE-Direct [9]. Using this feature, arbitrary lists of send, receive and wait operations can be created. These lists can then be posted to a work-request queue to be further processed by the network card. The network adapter executes it and eliminates the need for the host processor to progress the communication tasks. Using such task-lists, non-blocking collective operations may be designed by upper-level libraries.

### B. Broadcast Algorithms in MVAPICH2

State-of-the-art open-source MPI implementations, such as MPICH2 [10], Open-MPI [11] and MVAPICH2 use optimized algorithms to improve the latency of blocking collective operations. MVAPICH2 implements multi-core aware, shared-memory based algorithms for blocking collective operations. The processes that are within a compute node are grouped within a "shared memory communicator". One process per node is designated as a leader and participates in a "leader communicator" which contains leaders from all nodes. The broadcast operation is scheduled across these communicators to achieve low communication latency. The "leader" phase of the algorithm may use either the k-nomial algorithm or the scatter-allgather algorithm. For the intra-node phase, we can either use the shared-memory buffers or use point-to-point operations which are implemented through shared-memory channels. It is also possible to design the intra-node phases through loop-back, where the NIC moves the data between the source and the destination buffers. However, the latency of such an approach will be higher.

### C. High Performance Linpack (HPL)

The HPL benchmark is used to generate the semi-annual Top500 list [5]. The benchmark records the time required for LU factorization of a dense matrix. Based on the time, it calculates the rate of floating point operations of the system. The benchmark follows a 2-D block decomposition strategy for load balancing. The total number of processors are split into a $PxQ$ grid. The key use of broadcast operation in HPL is to forward a panel for factorization of the blocks owned by individual processors. Since panel factorization is in the critical path of this algorithm, the benchmark implements a "look-ahead" strategy. When the broadcast operations are overlapped, computation on the processors can proceed while the broadcast operation is still in progress. HPL relies on various virtual panel broadcast topologies to implement non-blocking versions of the *One-to-All* broadcast operation through MPI_Isend, MPI_Irecv and MPI_Iprobe operations. Several developers have invested significant time and effort in tuning the benchmark to overlap the DGEMM computation with the broadcast operations to hide the latency of the broadcast operation. However, since the host processors are required to progress the broadcast operation, this directly affects the peak throughput that HPL can achieve.

A high performance implementation of MPI_Ibcast based on InfiniBand's network offload feature could potentially help to improve the performance of HPL, as it requires very little intervention from the host processors.

### D. Impact of System Noise

Several researchers have demonstrated the impact of system noise on the performance of parallel applications [2], [3]. The impact of noise is higher at larger scales, because the delays tend to get propagated across various tasks in the job. Hoefler et al, quantified the impact of noise on various host-based collective operations in [2] and concluded that rooted collectives (such as MPI_Bcast) are less sensitive to noise when compared to dissemination based collective operations (such as MPI_Allgather), because they used tree-based algorithms. However, large message algorithms for MPI_Bcast operations involve the allgather phase and hence, and it is necessary to study its behavior in the presence of system noise. The allgather phase may be implemented either through the recursive doubling algorithm, or the ring algorithm, both of which are known to be susceptible to system noise. However, with network based implementations, the network can independently execute the schedules, with little intervention from the host processors. Such designs may be helpful in minimizing the impact of system noise on the performance of collective operations.

## III. Relation to Previous Work

Improving computation and communication overlap in parallel applications has traditionally been a topic of great

interest [12]. Hoefler et. al. proposed using host based techniques for designing non-blocking collective operations [13]. However, host based techniques, do not offer performance portability and may not deliver complete overlap. In [14], Hemmert et. al. demonstrate the benefits of using triggered operations and counting events provided by the Portals 4.0 message passing interface. Additionally, Beckman at. al. [15] studied the impact of noise on the performance of collectives by injecting noise. We use a subset of these parameters in our experiments.

Graham et. al. reported early experiences with the CORE-Direct software API in [9]. Subramoni et. al. proposed communication primitives for blocking collective operations with the CORE-Direct in [7]. In [16], we designed a scalable network offload based MPI_Ialltoall implementation and demonstrated up to 23% improvement with a parallel 3D FFT library. Venkata et. al. proposed blocking and non-blocking implementations of broadcast based on the CORE-Direct interface and reported results based on micro-benchmark based evaluations with 64 processes [17]. In our paper, we propose efficient blocking and non-blocking designs for the broadcast operation that scales up to 512 processes and study the benefits with HPL and also study the benefits of using network based collectives with system noise. We would like to note that our designs are applicable to any InfiniBand cluster that uses the ConnectX-2 interface along with the CORE-Direct software.

## IV. Designing Offload-based Broadcast Algorithms

We described the communication protocols we use for small and large messages with the CORE-Direct interface in [16]. We pre-post buffers to minimize the latency of small messages and we rely on InfiniBand's *Receiver-Not-Ready*(RNR) feature for large messages. In this section, we discuss our proposed designs for the network-offload based algorithms for the broadcast operation.

### A. Small and Medium Message Length Broadcasts

For small messages, we use a network-offload based k-nomial tree algorithm to minimize the communication latency. Depending on its logical location in the k-nomial bcast tree, each process creates a different task-list. However, the task-lists of intermediate processes in the tree need to be constructed carefully to maintain data consistency across different levels of the tree. One obvious choice is to force the MPI library to wait till the data is received before creating the task-list with the send-tasks to the other processes in the sub-tree. However, such a design will only work for a blocking version of the broadcast operation.

In Figure 1, we describe our proposed multi-send feature, which can be used to design the MPI_Ibcast operation in a fully asynchronous manner. Assume Rank i is an intermediate process in the k-nomial tree. Rank i receives data from its parent process and broadcasts the same data

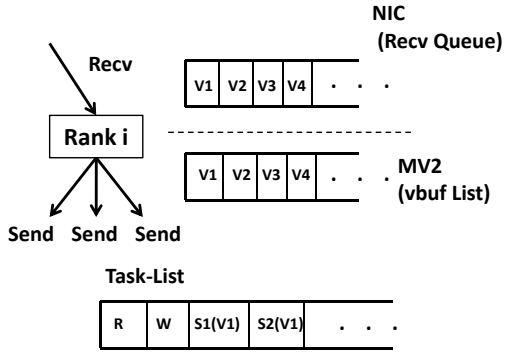


Figure 1. Offload Knomial-Tree Broadcast

buffer to other processes in its sub-tree. When the message arrives on the offload recv queue, the InfiniBand network interface is guaranteed to place this packet into the vbuf that is at the head of this list, $v1$. We internally maintain a list of vbufs "MV2-vbuf-list", inside the MVAPICH2 library, which mirrors the list of pre-posted vbufs on the small message offload-recv-queue. Since vbuf $v1$ will contain the data once the recv task is complete, we can use the same vbuf to create the subsequent send-tasks. This design allows us to create the task-list of the k-nomial algorithm inside the MPI_Ibcast function, even before the broadcast has even started. In Figure 1, we show the task-list to be comprised of the recv-task $R$, a wait-task $W$, and the send-tasks $S1(v1)$, $S2(v1)$, and so on. The wait-task is a NIC-level operation and ensures that the send-tasks are executed only after the recv-task $R$ has been executed and this should not be confused with an MPI-level wait operation. Once the task-list has been posted, we de-queue vbuf $v1$ from the list so that we handle the next MPI_Ibcast operation correctly.

### B. Large Message Length Broadcasts

For large messages, we have designed a network-offload-based scatter-allgather algorithm. We use the binomial exchange for the scatter phase and the ring algorithm for the allgather phase of the operation. The binomial exchange is a special case of the k-nomial algorithm, where the degree of the tree is (k=2). In the ring algorithm, suppose there are $N$ processes, each process $i$ executes $N$ iterations and it communicates with processes $i-1$ and $i+1$ in each step. Since we use this algorithm only for large messages, we dynamically register the user buffers and use the RNR mechanism for flow control. This simplifies the design, because we can directly use the registered user buffers to create the send/recv tasks. The wait tasks can be used as necessary to ensure data consistency across different steps of the algorithm.

### C. Algorithm Design Choices

As discussed in Section II-B, in MVAPICH2, we schedule the entire broadcast operation across the leader and the inter-node communicators. We can design offload-bcast algorithms in a similar manner. We have the following choices:

1) *Flat-Offload (FO)*: Use network-offload algorithms directly on the given communicator, without considering the node-level topology.
2) *Two-level-Offload-Host (TOH)*: Use network-offload-based mechanisms for the leader exchange phase and complete the intra-node phases through send/recv during the MPI_Wait operation.
3) *Two-level-Offload-Offload (TOO)*: Use network-offload algorithms for both the leader exchange and the intra-node phases.

Suppose we consider the *FO* approach, with $N$ processes, the allgather phase of the scatter-allgather will involve $N$ steps. The current ConnectX-2 interface allows us to post task-lists of fixed sizes, which makes this algorithm hard to scale, unless we rely on a light-weight offload progress thread [16]. Since this approach does not consider the node-level topology, its latency is also higher. So, we do not evaluate this design for the rest of this paper.

We believe the *TOO* approach can deliver better overlap, because it does not involve any intervention of the host processors after the task-list has been posted. However, this alternative might not deliver the best communication latency, because the intra-node phase relies on the network-loop-back. In the *TOH* approach, since the intra-node phases of the broadcast operation are done through shared-memory channels, we can expect its latency to be better for small messages. However, since the library performs the intra-node phase during the MPI_Wait operation, we may see higher overheads due to the MPI_Wait operation for large messages and hence, lower overlap. Hence, for our proposed MPI_Ibcast operation, we use the *TOH* approach for small messages to achieve good communication latency and for large messages, we can either use *TOH* or the *TOO* approaches, depending on whether we want to optimize for latency or overlap.

## V. Experimental Evaluation

### A. Experimental Setup

Each node of our 512-core testbed has eight Intel Xeon cores running at 2.53 Ghz with 12 MB L3 cache. The cores are organized as two sockets with four cores per socket. Each node also has 12 GB of memory and Gen2 PCI-Express bus. They are equipped with MT26428 QDR ConnectX-2 HCAs with PCI-Ex interfaces. We used a 171-port Mellanox QDR switch, with 11 leafs, each having 16 ports. Each node is connected to the switch using one QDR link. The HCA as well as the switches use the latest firmware. The operating system used is Red Hat Enterprise Linux Server release 5.4 (Tikanga), with the 2.6.18-164.el5 kernel version. OFED version 1.5.1 is used on all machines, and the OpenSM version is 3.3.7. We would like to note that there were non-deterministic race-conditions while running LibNBC's basic threaded version and LibNBC's real-time thread option was crashing the nodes, while running with super-user permissions.

### B. Benchmark Suite

In this paper, we use modified versions of the OSU Micro-Benchmarks, which are a part of the MVAPICH2 software package. We use the *osu_bcast* benchmark to measure the average latency of the different implementations of the MPI_Bcast operation across various message/system sizes.

**Overlap Benchmark:** In this benchmark, we perform floating point matrix-matrix operations by invoking the cblas_dgemm function supported by the Intel MKL Library (10.2.1.017), between the MPI_Ibcast and the MPI_Wait operations. We measure the overall time required for completion and compute the GFLOPS rating for the given case and compare it against the theoretical peak FLOPS rating for our system. In the first experiment, we fix the message size and vary the matrix size $N$ gradually between the values 10 and 8K and we measure the throughput. In the second experiment, we decide the DGEMM problem size such that the compute time is almost equal to the communication latency of the MPI_Ibcast operation for a given message length. We repeat this step across different message sizes and measure the throughput achieved. For the Host-Based-Test version, we also introduce MPI_Test calls at different frequencies to examine its impact on the overlap percentage. However, since the computation in DGEMM is not a linear relationship, we interleave the MPI_Test calls with multiple calls to cblas_dgemm, with adjusted matrix dimensions. This way, the overall compute operation remains equivalent to making one cblas_dgemm with matrix of size $N$.

In the following figures, we use the following acronyms: MVAPICH2-Default-Bcast refers to the latency of MPI_Bcast in MVAPICH2; TOO and TOH refer to latency of Two-level-Offload-Offload and Two-Level-Offload-Host broadcast algorithms, respectively. We refer to libNBC's results as Host-Based-Test. In Figures 4 and 5, Host-Based-Test-X indicates that we invoke the MPI_Test call $X$ times to progress the communication, while performing compute tasks. We also report the average results from multiple runs to eliminate any experimental errors.

### C. Communication Latency

In Figures 2 and 3, we compare the latency of the default multi-core aware host-based MPI_Bcast operation in MVAPICH2, with the Host-based-Test approach along with our proposed Network-Offload based broadcast algorithms, for different message and system sizes. For the Host-based-Test approaches, we post the NBC_Ibcast operation, immediately followed by the NBC_Wait() operation to measure the latency when no overlap is attempted. We can observe that the latency of the Offload-Host mechanism is better than that of both Offload-Offload and Host-Based-Test approaches. The Offload-Host approach also performs comparably with the default broadcast algorithm in MVAPICH2. As discussed in Section IV-C, since the *Offload-Offload* design has to entirely rely on the network loop-back operation for both

the inter-node and the intra-node phases of the broadcast operation, this leads to higher communication latency. However, the rest of the designs can leverage the shared-memory channels for the intra-node communication steps. We would also like to note that the performance of *Offload-Offload* is much better when compared to MVAPICH2's broadcast algorithm completely executed over network loop-back.

### D. Computation/Communication Overlap

In this section, we discuss the impact of various non-blocking Bcast designs on the overall throughput of DGEMM, a matrix-matrix multiplication program. In Figure 4, we keep the message length of the MPI_Ibcast operation constant at 128 KB and we vary the value of $N$, for different system sizes. We compare it to the estimated theoretical peak of the the system, based on the CPU clock frequency (2.53 GHz) and the fact that the Xeon processors used have four floating point units (10 GF double precision per core). We can see that with our proposed Offload-based MPI_Ibcast, we can achieve a higher throughput for smaller values of N. However, as the value of N is increased beyond 3,000, the throughput achieved with the Host-based-Test version is also comparable. However, this behavior may not remain consistent for different message lengths. This also requires the DGEMM matrix operation to be broken down carefully and places a higher burden on the application developers. In Figure 5, we compare the throughput for different message lengths. We can see that we consistently achieve the highest throughput with MPI_Ibcast based on network-offload, when compared to the Host-based-Test alternatives. The throughput with the Host-Based-Test approach drops down to about 60% of the peak throughput as the message length of the MPI_Ibcast operation is increased beyond 1MB.

## VI. HPL Results

In this section, we discuss our experiences with running the HPL Benchmark with our proposed network-offload based MPI_Ibcast operations and we compare it with HPL's 1-ring broadcast implementation. For all our experiments, we have fixed the value of $NB$ to be 200 and look-ahead depth as 2. The process grids were chosen as $16x8$, $16x16$ and $16x32$ for 128, 256 and 512 processes, respectively. For brevity, we are excluding the results obtained with some of the other HPL's broadcast implementations, because we observed similar trends. We also study HPL's behavior with libNBC's non-blocking broadcast implementation, we refer to these results as HPL-Host for the rest of this section.

### A. HPL Throughput

In Figure 6, we compare the normalized throughput of the HPL benchmark with different problem sizes, $N$, across different system sizes. The values of N are expressed in terms of percentage of the overall physical memory available, for a given system size. We can see that for smaller values of N, HPL with our proposed MPI_Ibcast performs significantly better than HPL with the 1-ring broadcast algorithm. With 256 processes, and 10% overall memory, we see a difference of up to 13% when compared with HPL's 1-ring implementation. As the problem size is increased, we can see that HPL with Offload-Bcast performs better than HPL with 1-ring-Bcast and Host-Bcast.

For a given system size, suppose that HPL-1ring achieves its peak throughput of X GFLOPS, with a memory consumption of Y%. In Figure 7, we use the bar-graphs to compare the problem sizes required by HPL-Offload and HPL-Host to achieve the same performance of X GFLOPS. We also depict this peak throughput data by using the line-graphs in the same figure. We can see that for all system sizes, HPL-Offload is able to achieve or surpass the peak throughput achieved by HPL-1ring with smaller values of N. For example, with 512 processes, the peak throughput achieved by HPL-1ring is 4,400 GFlops, with a problem size of N=248,700 doubles, which is about 60% of the overall memory for the given system size. We can observe that HPL-Offload can achieve the same throughput of 4,400 GFlops, with a problem size of N=143,500 doubles, which is just about 20% of the overall system memory. We can also see that HPL-Host achieves its peak throughout of 4,017 GFlops, with N=258,000, which is about 63% of the overall memory. This trend is consistent with our observations with the overlap benchmarks in Figure 4. Hence, we can conclude that *HPL-Offload can achieve the peak throughput that HPL-1ring achieves with much smaller problem sizes.*

In Table I, we compare the run-times of the HPL benchmark across different system sizes, with varying problem sizes and broadcast implementations. We express the problem size N, as a fraction of the total system memory consumed and the run-times are expressed in seconds. We can co-relate the information presented in Figure 7 and Table I to make the following observation. With 512 processes, HPL-1ring with a problem size that consumes 60% of memory requires 2,332 s to achieve its peak throughput. Similarly, HPL-Host requires 2,591 s to achieve its peak throughput with 60% consumption of memory. However, HPL-Offload can achieve the same or higher performance with just about 20% memory and a run-time of 448 s.

### B. HPL Performance Analysis

In Figure 8(a), we compare the fraction of application run-time used towards performing various compute and MPI-level tasks, with 256 processes and problem sizes of 144,000 and 177,000 doubles. We can see that for both the problem sizes, HPL with Offload-Bcast spends a lesser fraction of time inside the MPI library, because all of the broadcast operation has been offloaded to the network. In Figures 8(b) and (c), we take a closer look at the time HPL spends within the MPI library, for the same problem sizes. We can observe that HPL with 1-ring-Bcast spends nearly 40% of its MPI time in progressing the broadcast operation. However, HPL with Offload-Bcast spends a significantly lesser fraction of

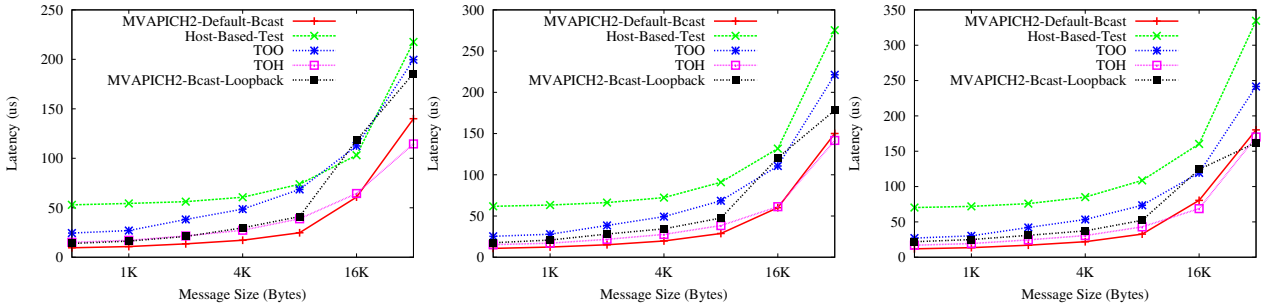


Figure 2. Small Message Latency Comparison: (a) 64 Processes, (b) 128 Processes and (c) 256 Processes

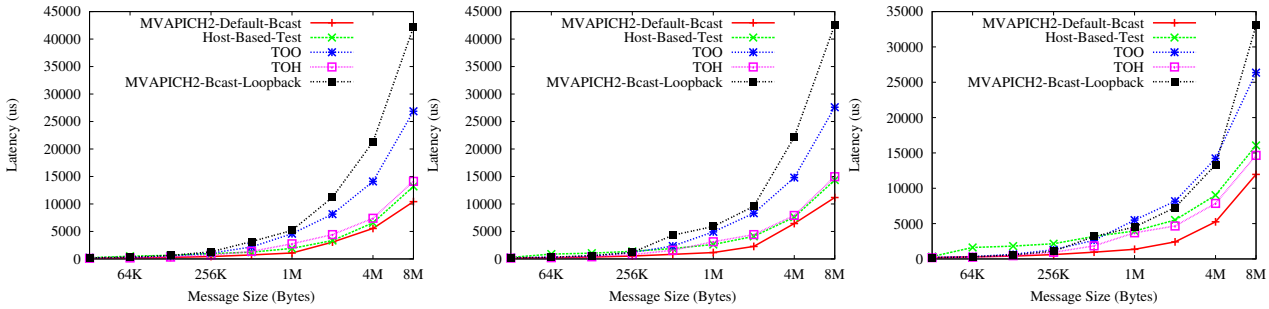


Figure 3. Large Message Latency Comparison: (a) 64 Processes, (b) 128 Processes and (c) 256 Processes

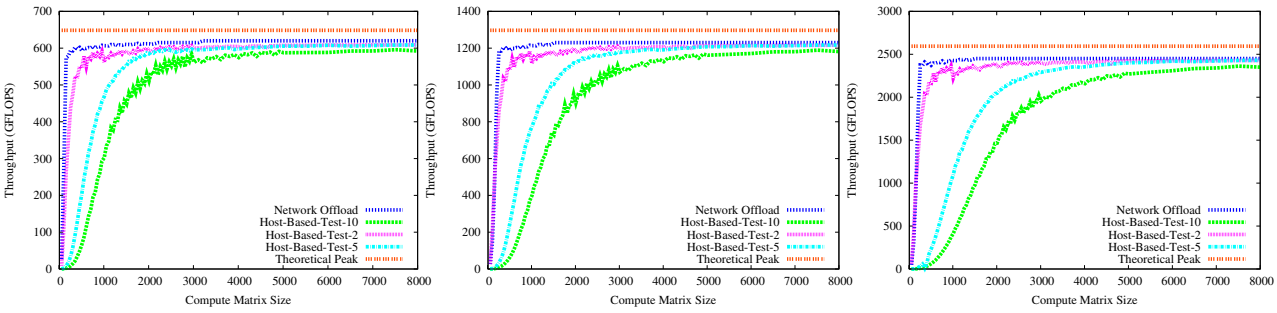


Figure 4. Overlap DGEMM Comparison Fixed Message Length: (a) 64 Processes, (b) 128 Processes and (c) 256 Processes

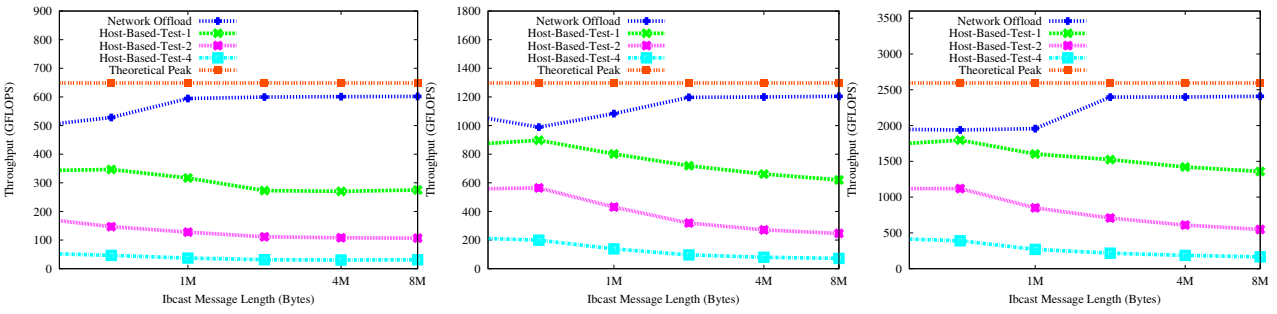


Figure 5. Overlap DGEMM Comparison: (a) 64 Processes, (b) 128 Processes and (c) 256 Processes

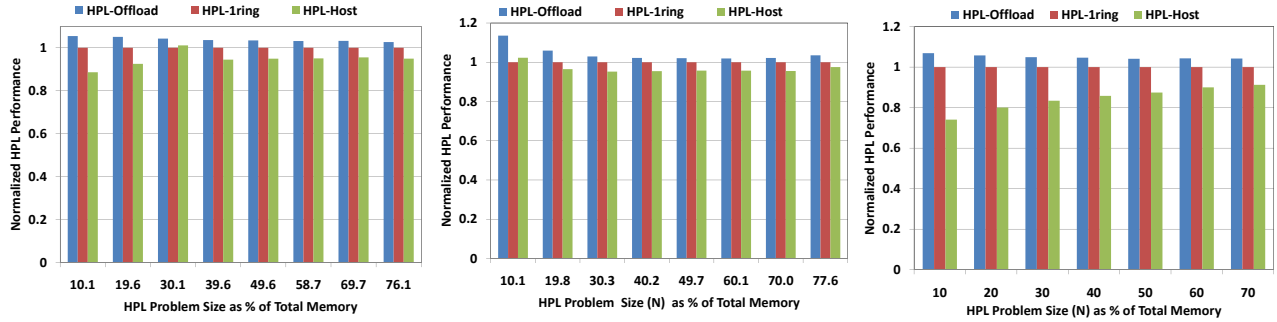


Figure 6. HPL Performance: (a) 128 Processes, (b) 256 Processors and (c) 512 Processes

Table I
APPLICATION RUN TIME (SECONDS)

| Job Size | 64 | | | 128 | | | 256 | | | 512 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Bcast Implementation | Offload | 1ring | Host | Offload | 1ring | Host | Offload | 1ring | Host | Offload | 1ring | Host |
| App Size (Memory %) | | | | | | | | | | | | |
| 10 | 60.5 | 62.6 | 69.8 | 85.0 | 89.7 | 101.2 | 125.4 | 142.4 | 139.2 | 169.3 | 180.9 | 244.3 |
| 20 | 154.5 | 158.8 | 173.6 | 217.5 | 228.5 | 247.0 | 320.7 | 339.6 | 352.1 | 448.2 | 473.2 | 592.0 |
| 30 | 287.5 | 293.6 | 317.7 | 405.1 | 422.1 | 451.1 | 591.1 | 608.4 | 639.1 | 807.1 | 846.5 | 1,015.7 |
| 40 | 426.1 | 434.4 | 467.2 | 610.4 | 632.6 | 669.4 | 893.6 | 913.1 | 956.5 | 1,239.0 | 1,298.2 | 1,511.8 |
| 50 | 603.6 | 614.5 | 658.1 | 841.7 | 870.8 | 917.4 | 1,212.1 | 1,238.1 | 1,292.1 | 1,708.6 | 1,778.8 | 2,034.9 |
| 60 | 798.0 | 811.9 | 866.4 | 1,079.6 | 1,113.4 | 1,171.1 | 1,603.1 | 1,634.6 | 1,707.5 | 2,234.2 | 2,332.8 | 2,590.5 |
| 70 | 998.7 | 1,014.1 | 1,079.8 | 1,387.7 | 1,434.1 | 1,501.1 | 1,998.7 | 2,042.2 | 2,137.8 | 2,796.9 | 2,915.1 | 3,194.3 |
| 77 | 1,162.1 | 1,178.7 | 1,267.4 | 1,581.7 | 1,622.6 | 1,702.1 | 2,337.7 | 2,420.5 | 2,481.5 | NA | NA | NA |

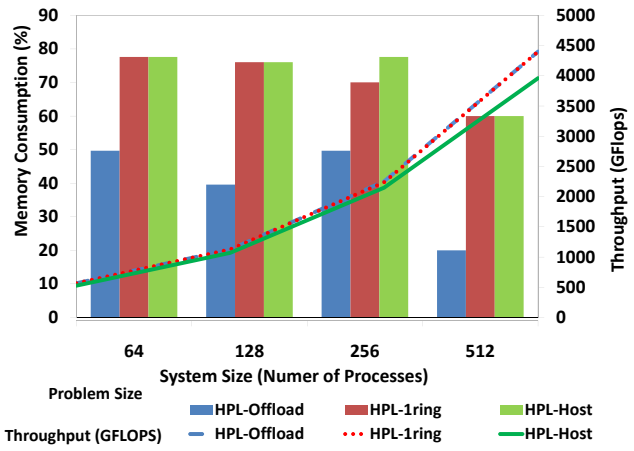


Figure 7. HPL Performance Summary

time in the HPL-Ibcast-wait operation. We can also observe that as the problem size is increased, the fraction of time inside the HPL-Ibcast-wait operation further reduces. This is because the compute phases of HPL are longer and this allows for higher communication/computation overlap.

## VII. IMPACT OF SYSTEM NOISE

In this section, we study the impact of system noise on the performance of applications that overlap computation with the MPI_Ibcast operation, by comparing out proposed solution with the Host-based-Test approach. We rely on a simple daemon that performs matrix-matrix multiplication operations that can be used to inject noise with durations and frequencies and we schedule this daemon on each core on all the nodes. In this experiment, we use our overlap benchmark described in Section V-B, with 256 processes and a message length of 2MB. We vary the noise duration from about 30 $\mu$s to about 250 $\mu$s and the noise frequency between 20Hz and 1KHz. We expect the noise to affect the latency of the host-based implementations of MPI_Ibcast, because it interferes with progressing the communication. This implies that a part of the broadcast algorithm needs to be completed in the MPI_Wait operation, leading to higher MPI-level overheads and lower application throughput. However, with our proposed network offload based designs, since the host processors are not involved in progressing the collective operation, the latency of the MPI_Ibcast operation is not affected. This implies that the network based collective algorithms can be resilient to system noise. In Table II, we compare the DGEMM throughput for different noise durations and frequencies. We note that the throughput of DGEMM with the offload-based solution is about 2,397 GFLOPS and with the Host-Based-Test approach is about 1,631 GFLOPS under quiet conditions. We can see that with the network based MPI_Ibcast design, the throughput varies by only a small amount, when compared with the Host-based-Test version. This small degradation is expected, because our daemon will also affect the compute tasks.

## VIII. CONCLUSIONS AND FUTURE WORK

In this paper we design a high-performance scalable, non-blocking implementation of the *One-to-All* Broadcast operation exploiting InfiniBand's Network Offload feature. Our designs deliver near perfect communication/computation overlap, without penalizing the communication latency. We observe that our proposed MPI_Ibcast implementation al-

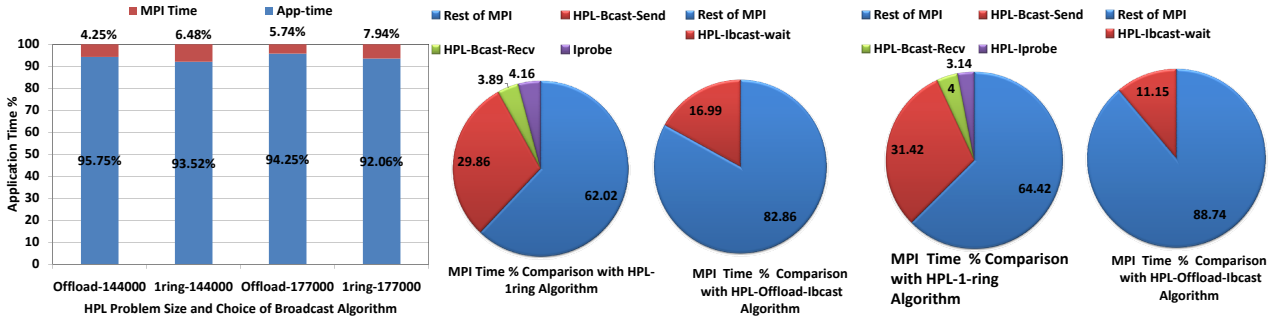


Figure 8. (a) MPI-Time Vs App-Time with 256 Processes; (b)MPI % Time Comparison with N=144,000 and (c) MPI % Time Comparison with N=177,000

Table II
DGEMM THROUGHPUT WITH SYSTEM NOISE(GFLOPS)

| Noise Duration ($\mu$s) | Quiet | | 50 | | 150 | | 250 | |
|---|---|---|---|---|---|---|---|---|
| MPI_Ibcast algorithm (s) | Offload | Host | Offload | Host | Offload | Host | Offload | Host |
| Noise Frequency (Hz) | | | | | | | | |
| Quiet | 2,397 | 1,631 | - | - | - | - | - | - |
| 1K | - | - | 2,378 | 1,593 | 2,347 | 1,568 | 2,302 | 1,501 |
| 40 | - | - | 2,396 | 1,614 | 2,391 | 1,601 | 2,384 | 1,585 |
| 20 | - | - | 2,395 | 1,618 | 2,393 | 1,593 | 2,389 | 1,611 |

lows the HPL benchmark to achieve its peak throughput with significantly smaller problem sizes, which also improves HPL's run-times by up to 78%. We also observe that our proposed designs can minimize the impact of system noise on applications when compared to host-based implementations of the MPI_Ibcast operation.

## REFERENCES


[1] MPI Forum, "MPI: A Message Passing Interface," in *www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf*.

[2] T. Hoefler, T. Schneider and A. Lumsdaine, "Characterizing the Influence of System Noise on Large-Scale Applications by Simulation," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov. 2010.

[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 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2003.

[4] J. Dongarra, "The LINPACK Benchmark: An Explanation," in *Proceedings of the 1st International Conference on Supercomputing*. London, UK: Springer-Verlag, 1988, pp. 456–474.

[5] Top500, "Top500 Supercomputing systems," Oct 2010, .

[6] Mellanox Technologies, "ConnectX-2 Architecture," http://www.hpcwire.com/features/Mellanox-Rolls-Out-Next-Iteration-of-ConnectX-57046327.html.

[7] H. Subramoni, K. Kandalla, S. Sur and D K. Panda, "Design and Evaluation of Generalized Collective Communication Primitives with Overlap using ConnectX-2 Offload Engine," in *18th Annual IEEE Symposium on High Performance Interconnects (HotI)*, 2010.

[8] "MVAPICH2: High Performance MPI over InfiniBand, RoCE and iWARP," http://mvapich.cse.ohio-state.edu.

[9] R. Graham, S. Poole, P. Shamis, G. Bloch, N. Boch, H. Chapman, M. Kagan, A. Shahar, I. Rabinovitz, G. Shainer, "Overlapping Computation and Communication: Barrier Algorithms and Connectx-2 CORE-Direct Capabilities ," in *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium, Workshops*, 2010.

[10] MPICH2, http://www.mcs.anl.gov/research/projects/mpich2/.

[11] Open-MPI, http://www.open-mpi.org/.

[12] J. C. Sancho, K. J. Barker, D. J. Kerbyson, and K. Davis, "Quantifying the Potential Benefit of Overlapping Communication and Computation in Large-Scale Scientific Applications," ser. SC. New York, NY, USA: ACM, 2006.

[13] T. Hoefler, J. Squyres, W. Rehm, and A. Lumsdaine, " A Case for Non-blocking Collective Operations ," in *Frontiers of High Performance Computing and Networking . ISPA 2006 Workshops, Lecture Notes in Computer Science*, vol. 4331/2006, 2006, pp. 155–164.

[14] K. Hemmert, B. Barrett and K. Underwood, "Using Triggered Operations to Offload Collective Communication Operations," in *Recent Advances in the Message Passing Interface*, ser. Lecture Notes in Computer Science, vol. 6305. Springer Berlin / Heidelberg, 2010, pp. 249–256.

[15] P. Beckman, K. Iskra, K. Yoshii, S.Coghlan, A.Nataraj, "Benchmarking the effects of operating system interference on extreme-scale parallel machines," *Cluster Computing*, vol. 11, pp. 3–16, March 2008.

[16] K. Kandalla, H. Subramoni, K. Tomko, D. Pekurovsky, S. Sur and D. K. Panda, "High-Performance and Scalable Non-Blocking All-to-All with Collective Offload on InfiniBand Clusters: A Study with Parallel 3D FFT ," in *International Supercomputing Conference(ISC), Hamburg, Germany*, June,2011.

[17] M. Venkata, R. Graham, J. Ladd, P. Shamis, I. Rabinovitz, F. Vasily and G. Shainer, "ConnectX-2 CORE-Direct Enabled Asynchronous Broadcast Collective Communications," in *Proceedings of the 25th IEEE International Parallel and Distributed Processing Symposium, Workshops*, 2011.