Scalable MPI Collectives using SHARP: Large Scale Performance Evaluation on the TACC Frontera System

Bharath Ramesh, Kaushik Kandadi Suresh, Nick Sarkauskas, Mohammadreza Bayatpour, Jahanzeb Maqbool Hashmi, Hari Subramoni, Dhabaleswar K. Panda
Department of Computer Science and Engineering
The Ohio State University
Columbus, USA
{ramesh.113, kandadisuresh.1, sarkauskas.1, bayatpour.1, hashmi.29, subramoni.1, panda.2}@osu.edu

***Abstract*—The Message-Passing Interface (MPI) is the *de-facto* standard for designing and executing applications on massively parallel hardware. MPI collectives provide a convenient abstraction for multiple processes/threads to communicate with one another. Mellanox's HDR InfiniBand switches provide Scalable Hierarchical Aggregation and Reduction Protocol (SHARP) capabilities to offload collective communication to the network and reduce CPU involvement in the process. In this paper, we design and implement SHARP-based solutions for MPI_Reduce and MPI_Barrier in MVAPICH2-X. We evaluate the impact of proposed and existing SHARP-based solutions for MPI_Allreduce, MPI_Reduce, and MPI_Barrier operations have on the performance of the collective operation on the 8th ranked TACC Frontera HPC system.**

Our experimental evaluation of the SHARP-based designs show up to 5.4X reduction in latency for Reduce, 5.1X for Allreduce, and 7.1X for Barrier at full system scale of 7,861 nodes over a host-based solution.

***Index Terms*—SHARP, MPI, MPI collectives, MPI_Allreduce, MPI_Reduce, MPI_Barrier**

I. Introduction

Super-computing systems have grown in size and scale over the last decade. Two key drivers fueling the growth of supercomputers are the current trends in multi-/many-core architectures and the availability of commodity, RDMA-enabled, and high-performance interconnects such as InfiniBand [1] (IB). Such HPC systems are allowing scientists and engineers to tackle grand challenges in various scientific domains. Users of HPC systems rely on parallel programming models to parallelize their applications and obtain performance improvements.

Message Passing Interface (MPI) [2] is a very popular parallel programming model for developing parallel scientific applications. The MPI Standard [3] offers primitives for various point-to-point, collective, and synchronization operations. Collective operations defined in the MPI standard offer a very convenient abstraction to implement group communication operations. Owing to their ease of use and performance portability, collective operations are widely used across various scientific domains. Due to this wide use, the performance of collective operations has a significant impact on the overall performance of high-end applications running on modern HPC systems.

Historically, researchers have proposed several solutions that take advantage of specific features offered by modern high-end interconnects such as InfiniBand [4] to offload collective operations to the network and thereby achieve excellent performance and scalability on very large High Performance Computing (HPC) systems. One such example is the use of InfiniBand Hardware Multicast feature to offload the MPI_Bcast collective operation [5]. However, the multi-cast based solution had the significant disadvantage of being "unreliable" and was limited in its capability to implement other collective operations such as MPI_Allreduce, MPI_Reduce, and MPI_Barrier that are some of the most frequently used operations in scientific applications.

Recognizing this need, interconnect vendors such as Mellanox introduced high-performance hardware-based network solutions like core-direct [6], [7], [8] and SHARP [9] to enable offloading of more complex communication and some computation to the fabric. However, the compute capability offered by Core-direct is quite limited. Scalable Hierarchical Aggregation Protocol (SHARP) [9] is a new solution in this space that can offload large amount of computation to the network switch. For instance, while SHARP-based collectives can provide excellent performance and scalability for small message sizes [9], they have some performance issues and limitations that need to be handled intelligently to achieve the best performance for larger message sizes and process counts [10].

While researchers have addressed some of these challenges in earlier scientific literature, there exists no scholarly work that systematically studies the impact SHARP-based

*This research is supported in part by NSF grants #1931537, #1450440, #1664137, #1818253, #2007991 and XRAC grant #NCR-130002

978-1-6654-1561-3/20/$31.00 ©2020 IEEE
DOI 10.1109/ExaMPI52011.2020.00007

network offload can have on the performance of important collective operations like MPI_Allreduce, MPI_Reduce, and MPI_Barrier at very large scales on modern HPC systems.

II. Contributions

In this paper, we take up this challenge and do a thorough study of the performance impact SHARP-based collective algorithm designs can have on the performance of MPI collective operations on the 8th ranked TACC Frontera system, which provides a grand total of 7,861 compute nodes and the capability of running 440,216 processes at full system scale. We also propose enhanced SHARP-based designs for MPI_Reduce and MPI_Barrier in MVAPICH2-X. To summarize, this paper makes the following important contributions:

- Design and implement SHARP-based solutions for MPI_Reduce and MPI_Barrier in MVAPICH2-X
- Evaluate the impact the proposed SHARP-based solutions have on the performance of MPI_Reduce and MPI_Barrier at scale on Frontera
- Evaluate the impact existing SHARP-based solutions have on the performance of MPI_Allreduce at scale on Frontera
- Perform a careful analysis of the benefits of SHARP-based collective operations

Our experimental evaluation shows that our proposed designs are able to deliver up to 5.4X reduction in latency for Reduce, 5.1X for Allreduce, and 7.1X for Barrier at full system scale of 7,861 nodes.

III. Background

A. Reduction Collectives and MPI_Barrier

The reduction collectives in MPI (i.e. MPI_Allreduce and MPI_Reduce) are used widely in applications. Both collectives use a reduction operation in order to combine elements in the input buffer. The operation can be one of the predefined operations (sum, min, etc.) or it can be user-defined. MPI_Reduce returns the result of the operation to only the root while MPI_Allreduce will broadcast the result to all ranks.

MPI Implementations will optimize most collective operations, including the reduction collectives, to utilize a shared-memory channel for transferring data between processes within the same node for relatively small message sizes ($\leq$ 2,048 bytes). For the reduction collectives, each node designates a "leader" process who gets the intermediate result and then uses the network fabric to reduce its result with other node-leaders. The final result is then returned on the root process in the case of MPI_Reduce, and broadcast using the shared-memory channel on each node in the case of MPI_Allreduce.

MPI_Barrier is a relatively simple collective. A process that calls it will block until all other processes in the communicator also call MPI_Barrier. This allows the application to synchronize all processes, only allowing itself to continue until all processes are ready.

B. Mellanox SHARP

Mellanox Scalable Hierarchical Aggregation and Reduction Protocol (SHARP) is a technology that allows reduction collectives to be offloaded to the network. At a high level, a subset of processes within MPI communicators (one per node, one per socket, etc.) are used to form a SHARP group. This group is used to define end nodes in a SHARP tree in which these nodes (leaves in the tree) feed in data that is to be reduced and traversed upwards. Each non-leaf node in the tree is called an Aggregation Node (AN). ANs are responsible for performing the reduction operation. When the data reaches the root of the tree, it is then distributed. The benefit of using SHARP is that CPU time is freed for application use. Additionally, the application does not need to wait for the data to reach the CPU to perform the reduction application, minimizing the effect of system noise. An example of using ANs in a tree based fashion is shown in Figure 1.

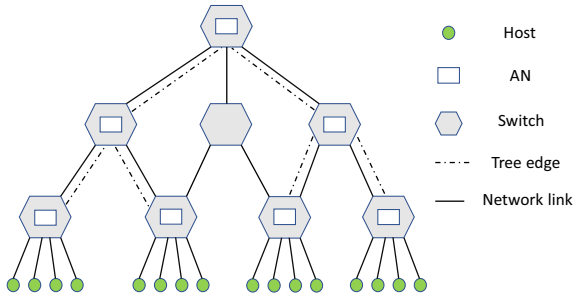


Fig. 1: Sharp sample logical tree. The circles represent hosts, hexagons represent switches, rectangles represent Aggregation nodes (ANs), dotted lines represent the reduction tree and other lines represent network links. Not all switches function as ANs in a reduction tree, as shown in the figure

IV. Design and Implementation

In this section, we describe the design and implementation of Hierarchical SHARP-based MPI_Reduce, MPI_Allreduce and MPI_Barrier in MVAPICH2-X using shared memory within the node and using SHARP-based support or native point to point operations for inter-node operations.

A. Shared-memory collectives

Most MPI libraries utilize shared-memory and/or kernel-assisted copy mechanisms to optimize intra-node transfers. In this paper, we do not consider evaluating using kernel-assisted mechanisms such as CMA and XPMEM, since our focus is on smaller messages sizes ($\leq$ 2,048 bytes) for which shared memory implementations have the same relative performance [11]. MVAPICH2-X uses a cache-aligned shared memory region created using the `mmap` unix system call. On systems with multiple sockets per node, MVAPICH2-X creates a separate shared-memory region for each socket in the node

Algorithm 1 Implementing MPI_Reduce on top of SHARP allreduce

Definition of Variables
  $sendbuf$: Pointer to the send buffer
  $recvbuf$: Pointer to the receive buffer
  $root$: Rank of the root process in MPI_Reduce
  $comm\_ptr$: Communicator passed to MPI_Reduce
  $rank$: Logical rank of process in $comm\_ptr$
  tmp_buf : Temporary buffer to store results
Pseudo Code for MPI_Reduce
  Initialize sharp reduce_spec
  Shared memory reduce to root process on each node
  if $rank \neq root$ **then**
    $reduce\_spec.recvbuf \leftarrow tmp\_buf$
  else
    $reduce\_spec.recvbuf \leftarrow recvbuf$
  end if
  sharp_do_allreduce($comm\_ptr.sharp\_comm$, $reduce\_spec$)

in order to build hierarchical intra-node collectives that can minimize inter-socket traffic, thereby reducing latency. The processes are then grouped in a communicator for simplicity of use in other MPI operations. A set of flags are maintained for each process to aid synchronization operations.

B. Hierarchical SHARP-based MPI_Reduce

In MVAPICH2-X, a hierarchical algorithm is implemented for MPI_Reduce in two steps. The first step involves an intra-node reduction operation to a designated root process on each node. This is followed by an inter-node reduction operation (which could use algorithms such as binomial, k-nomial, and others), with one process per node participating in the step. However, at larger scales, a software-based tree algorithm would not necessarily obtain the best performance, especially considering hardware-based offload protocols such as SHARP. Since SHARP currently only has support for allreduce, barrier and not reduce, we implemented MPI_Reduce on top of the allreduce SHARP primitive to take advantage of hardware-based offload. Algorithm 1 shows the pseudo-code implementation of the proposed sharp based MPI_Reduce. Since an MPI_Allreduce is semantically equivalent to every process calling MPI_Reduce, MPI_Reduce can be implemented on top of supported SHARP primitives by just ignoring the receive buffer on every non-root process. In the first phase, we do an intra-node reduction using shared-memory on a designated root process on every node. For the inter-node step, we call SHARP APIs to utilize network offload capabilities. The SHARP APIs provide a "reduce_spec" structure that is filled with details on the datatype, operations, aggregation modes, send buffer, receive buffer, and others. As described in Algorithm 1, setting the receive buffer of "reduce_spec" on non-root processes and then calling SHARP based Allreduce will complete the reduction operation.

C. Hierarchical SHARP-based MPI_Allreduce

For this paper, we primarily evaluate a socket-aware hierarchical Allreduce algorithm. In MVAPICH2-X, this algorithm is implemented in five steps. First, we perform an intra-socket reduction operation to a designated root process on each socket as shown in Figure 2(a). Then, socket-leaders within the node perform an inter-socket reduction as shown in Figure 2(b). At this stage, one designated root process on every node contains the reduced data for that node. The third step, depicted in Figure 2(c), involves an inter-node Allreduce operation. The algorithm used could either be based on point-to-point operations tuned for a particular message/system size or use other primitives such as the ones provided by SHARP. Figure 2(d) shows the next step, which involves a broadcast operation across socket-leaders. At the end of the 4th step, socket level leaders have the final result of the allreduce operation. Finally, an intra-socket broadcast is performed among processes within a socket after which the final result is obtained at every process. This is depicted in Figure 2(e).

D. Hierarchical SHARP-based MPI_Barrier

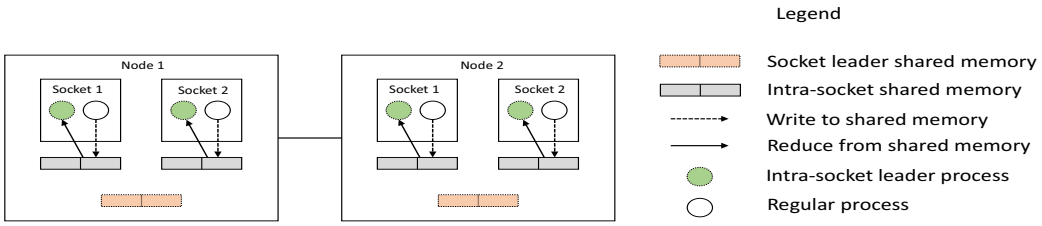
The implementation of the MPI_Barrier primitive in MVAPICH2-X is similar to the implementation of MPI_Allreduce, since an Allreduce can essentially be viewed as a Barrier without compute or any data buffer to be moved. In the first step, a root process on each socket polls an array of "gather" flags on shared-memory. These flags are set by other non-root processes and then reset by the root process thereby indicating that the gather operation is complete. This is followed by an inter-socket gather phase. The third step involves an inter-node barrier, either implemented using a point-to-point pairwise exchange or using SHARP. The last two steps mirror the first two steps by performing an inter-socket broadcast followed by an intra-socket broadcast using shared-memory.

V. Experimental Results and Evaluation

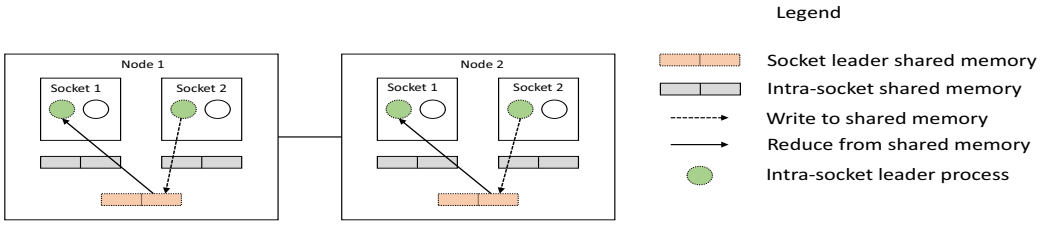
In this section, we describe the experimental setup used to conduct our experiments. An in-depth analysis of the results is also provided to correlate design motivations and observed behavior. All results reported here are averages of multiple (five) runs to discard the effect of system noise.

A. Experimental Setup

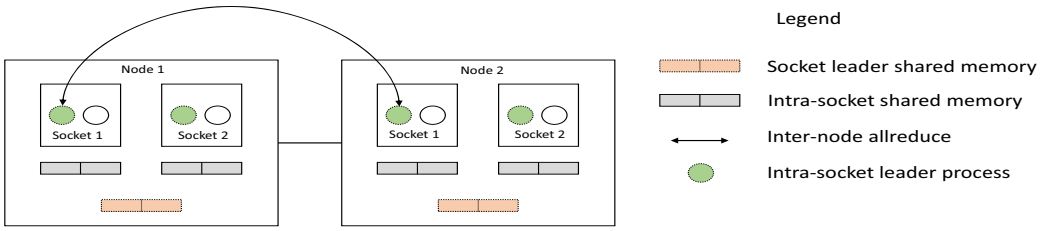
We ran all evaluations on the 8th ranked Frontera super-computer at the Texas Advanced Computing Center (TACC) [12]. The cluster comprises of dual-socket Intel Xeon 8280 ("Cascade Lake") nodes. Each node contains a two-socket motherboard, with 192 GB of DDR-4 RAM and each socket containing a 28-core processor running at 2.7 Ghz. The interconnect is arranged in a fat-tree topology and contains Mellanox InfiniBand HDR adapters providing 100 Gb/s of bandwidth between compute nodes and 200 Gb/s between switches in the fabric. The nodes run the RedHat Enterprise Linux (RHEL) 7 operating system with kernel version 3.10.0-1127.13.1.el7.x86_64. We compare an optimized version of



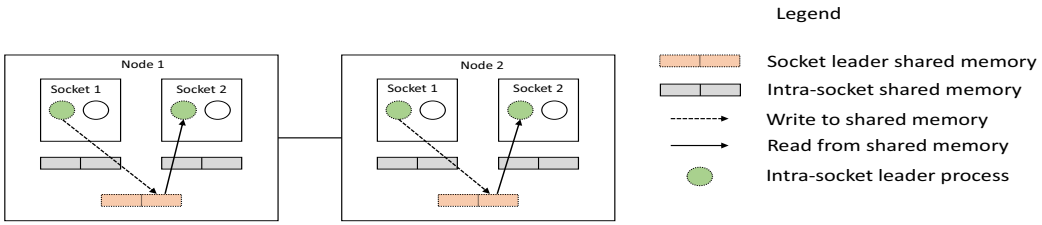
(a) **First phase**: It involves an intra-socket reduction in parallel across nodes to a leader process designated as the root. The socket-level leaders then participate in the second phase of the algorithm



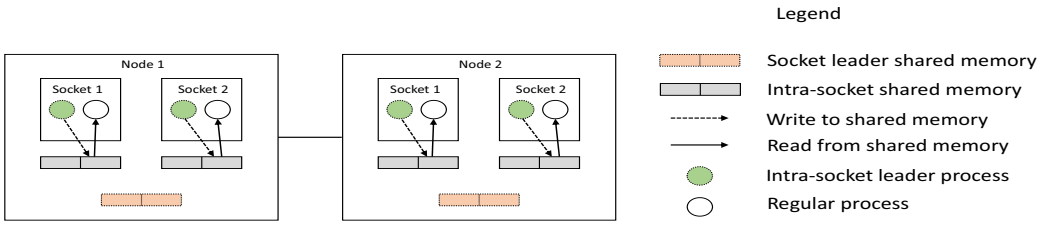
(b) **Second phase**: In this phase, the socket-leaders within each node participate in an intra-node reduction using their common shared memory, to a specific root process on each node



(c) **Third phase**: Node-level leaders perform an inter-node allreduce. The inter-node algorithm either uses SHARP (if enabled) or a tuned algorithm for a particular message and system size



(d) **Fourth phase**: The fourth phase mirrors the second phase. The node-level leader broadcasts results to other socket level leaders within the node using shared-memory. After this phase, all socket leaders have the final result of the allreduce operation



(e) **Fifth phase**: The final phase involves an intra-socket broadcast using the shared-memory channel. This completes the allreduce operation

Fig. 2: Five phases in the socket-aware algorithm for MPI_Allreduce in MVAPICH2-X taking four processes per node as an example

MVAPICH2-X 2.3 with and without SHARP enabled, compiled with GCC v8.3.0. We use the Dynamic Connected (DC) transport protocol support in MVAPICH2-X [13] to achieve good scale-out performance with a large number of processes.

B. Micro-Benchmark Level Evaluation

We primarily show results from the Micro-Benchmarks listed below.

1) OSU Micro Benchmarks: We show results using the OSU Micro Benchmarks (OMB) suite v5.6.3 [14]. We report the average latency on running 1,000 iterations of osu_allreduce and osu_barrier with 100 warm-up iterations (iterations that are skipped entirely) over five different runs. The osu_allreduce and osu_barrier tests report the average latency among all the processes for the given run-time parameters above. For osu_reduce, we run 1,000 iterations over five runs and choose to report the maximum latency of all processes, since the upper-bound of reduction operations is defined by the time the root takes to exit. Reducing the maximum latency also has the potential to reduce skews in application code, which is beneficial as well. We report numbers up to a message size of 2,048 bytes, where applicable. The latency is always reported in micro-seconds (*us*), unless specified otherwise.

C. Results for MPI_Reduce

This sub-section describes the experimental results and scaling for MPI_Reduce over different node counts, processes per node (ppn) counts and message sizes.

1) Scaling trends with one process per node: First, we evaluate how MPI_Reduce scales with one process per node and different node counts for MVAPICH2-X and MVAPICH2-X with SHARP. We report the maximum latency since this is often the bottleneck in MPI_Reduce calls at the root process. We observe that MVAPICH2-X without SHARP has a close to linear scaling with an increase in node count. For instance, its latency for a 16 byte message starts at around 3.17*us* for 4 nodes going all the way up to 126.12*us* for 7,861 nodes (full system size). This highlights the limitation of software-only optimizations of MPI_Reduce. On the other hand, MVAPICH2-X with SHARP demonstrates close to flat scaling up to 4,096 nodes, with the latency hovering between 1.86*us* - 6.9*us* for the same message size of 16 bytes. We see a jump, with the absolute value of MVAPICH2-X-SHARP increasing 2-fold when the node count is 7,861. This is usually attributed to variation when the entire cluster is being used at once. The results are demonstrated in Figures 3(a) and 3(b), which represent the absolute latency across the system. We can also infer the speedup of MVAPICH2-X-SHARP for the sample message size of 16 bytes. When analyzing the speedup of MVAPICH2-X-SHARP over MVAPICH-X, we observe linear scaling with MVAPICH2-X and close to flat scaling with MVAPICH2-X-SHARP, with speedups of up to 9.5X for a message size of 16 bytes. The scaling trend is similar for messages up to 2,048 bytes.

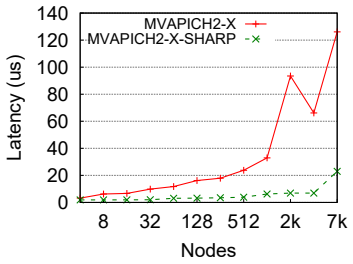
2) Scaling trends with multiple processes per node: In general, the expected trends for multiple processes per node are very similar to trends seen in the single process per node scenarios. However, due to the involvement of shared-memory in the hierarchical collective step as described in section IV, the max latency for the reduction operation goes up. This in turn reduces the scaling efficiency from the ideal case of one process per node (ppn). To evaluate the impact of increasing processes per node, we take a sample message size of 16 bytes and show how an increase in processes per node affects scaling. This is described in Figures 4(a), 4(b), 5(a), and 5(b). The X-axis represents different node counts starting from 4 to 1,024 nodes. The Y-axis indicates latency for the MPI operation. Figures 3(a), 3(b) represent 1 process per node, 4(a), 4(b) represent 2 processes per node and 5(a), 5(b) represent 16 processes per node numbers for two chosen message sizes of 16 bytes and 2,048 bytes. Due to a lack of time available to run large scale tests in the cluster, we only ran experiments up to 1024 nodes with the process per node counts greater than 1. In Summary, we see that MVAPICH2-X-SHARP still has a speedup of up to 3.6X for multi processes per node scenarios, but with up to 2.6X decrease in scaling efficiency when compared to the one process per node case.

3) Scaling trends with increasing message size: Figures 3(c), 4(c), and 5(c) show the impact of message size on the latency of MVAPICH2-X with SHARP. We show a specific set of node/process per node counts to simplify the amount of data shown in the paper and also because they are representative of the entire set of node/process per node pairs. We observe that the latency for MVAPICH2-X-SHARP remains largely the same up to the inline size of 64 bytes, after which it gradually increases. The speedup over MVAPICH2-X follows a similar trend as seen in sub-sections V-C1 and V-C2, with MVAPICH2-X-SHARP showing large benefits of up to 3.6X over the baseline of MVAPICH2-X for 2 processes per node scenarios. We observe a variation in numbers, where the latency of MVAPICH2-X reduces with increase in message size in certain cases. This is attributed to the fact that certain message sizes are not tuned properly, which results in a decrease in latency with an increase in message size for some cases. However, the trends for MVAPICH2-X-SHARP will still remain largely the same due to the limitations of pure software-based approaches.

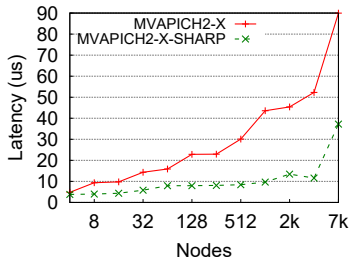
D. Results for MPI_Allreduce

This sub-section describes the experimental results and scaling for MPI_Allreduce over different node counts and number of processes.

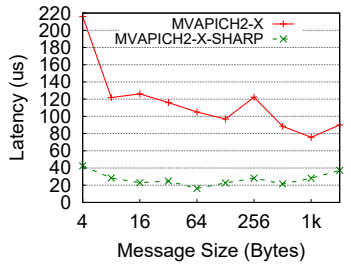
1) Scaling trends with one process per node: Similar to MPI_Reduce, we evaluate how MPI_Allreduce scales with one process per node for node counts from 4 to 7,861 (full system scale) for both MVAPICH2-X and MVAPICH2-X-SHARP. We report the average latency for all experiments pertaining to MPI_Allreduce. The trends would be the same with minimum and maximum latency since Allreduce is a balanced collective, where the differences in latency arise only due to the final



(a) Scaling of MPI_Reduce for 1ppn, 16 bytes up to 7,861 nodes. MVAPICH2-X-SHARP provides a benefit of up to 13.5X over MVAPICH2-X

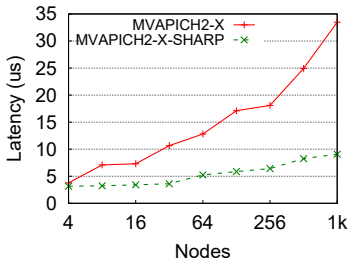


(b) Scaling of MPI_Reduce for 1ppn, 2,048 bytes up to 7,861 nodes. MVAPICH2-X-SHARP is up to 4.5X better than MVAPICH2-X

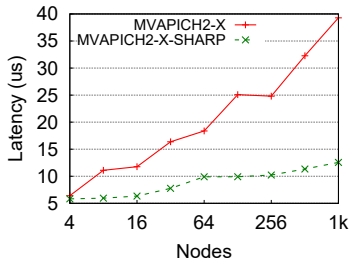


(c) Message size scaling of MPI_Reduce for 1ppn, 7,861 nodes. MVAPICH2-X-SHARP provides a flat latency curve for message sizes from 4 to 2,048 bytes

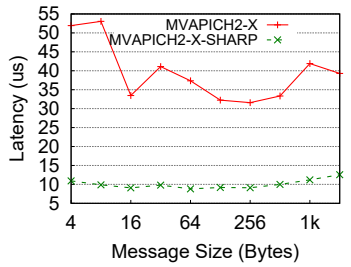
Fig. 3: Performance of MPI_Reduce with MVAPICH2-X and MVAPICH2-X-SHARP for one process per node (ppn) across various message and system sizes



(a) Scaling of MPI_Reduce for 2ppn, 16 bytes up to 1,024 nodes. MVAPICH2-X-SHARP provides a benefit of up to 3.5X over MVAPICH2-X

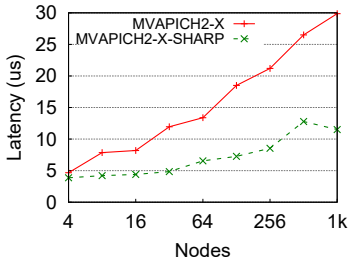


(b) Scaling of MPI_Reduce for 2ppn, 2,048 bytes up to 7,861 nodes. MVAPICH2-X-SHARP is up to 4X better than MVAPICH2-X

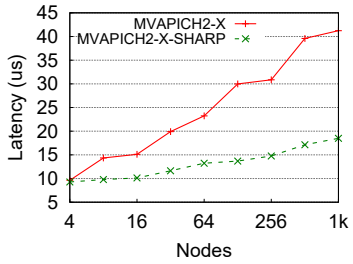


(c) Message size scaling of MPI_Reduce for 2ppn, 1,024 nodes. MVAPICH2-X-SHARP provides a flat latency for message sizes from 4 to 2,048 bytes

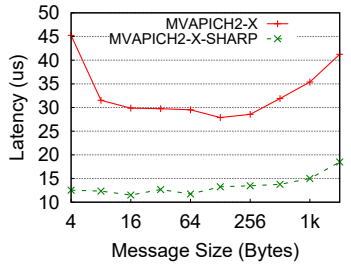
Fig. 4: Performance of MPI_Reduce with MVAPICH2-X and MVAPICH2-X-SHARP for 2 processes per node (ppn) across various message and system sizes



(a) Scaling of MPI_Reduce for 16ppn, 16 bytes up to 1,024 nodes. MVAPICH2-X-SHARP provides a benefit of up to **X** over MVAPICH2-X

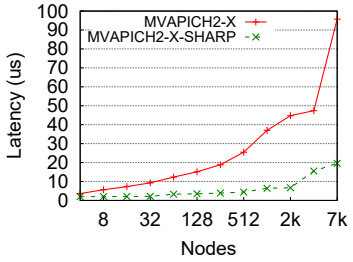


(b) Scaling of MPI_Reduce for 16ppn, 2,048 bytes up to 1,024 nodes. MVAPICH2-X-SHARP provides a benefit of up to **X** over MVAPICH2-X

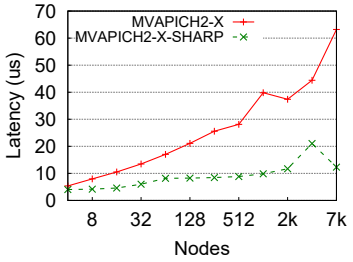


(c) Message size scaling of MPI_Reduce for 16ppn, 1,024 nodes. MVAPICH2-X-SHARP provides a flat latency curve for message sizes from 4 to 2,048 bytes

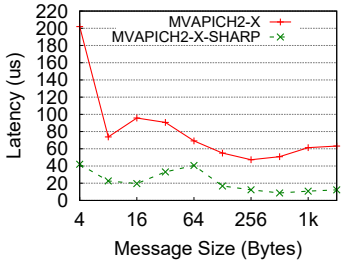
Fig. 5: Performance of MPI_Reduce with MVAPICH2-X and MVAPICH2-X-SHARP for 16 processes per node (ppn) across various message and system sizes



(a) Scaling of MPI_Allreduce for 1ppn, 16 bytes up to 7,861 nodes. MVAPICH2-X-SHARP provides a benefit of up to 6.69X over MVAPICH2-X

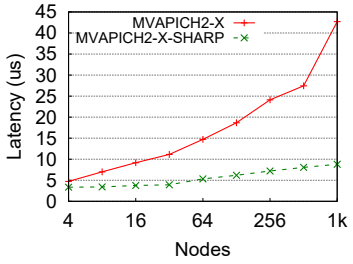


(b) Scaling of MPI_Allreduce for 1ppn, 2,048 bytes up to 7,861 nodes. MVAPICH2-X-SHARP is up to 5.1X better than MVAPICH2-X

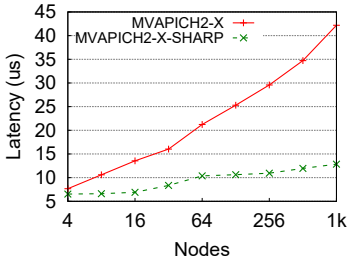


(c) Message size scaling of MPI_Allreduce for 1ppn, 7,861 nodes. MVAPICH2-X-SHARP provides a flat latency curve for message sizes from 4 to 2,048 bytes

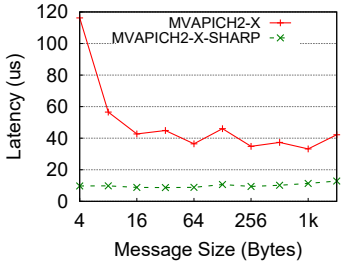
Fig. 6: Performance of MPI_Allreduce with MVAPICH2-X and MVAPICH2-X-SHARP for one process per node (ppn) across various message and system sizes



(a) Scaling of MPI_Allreduce for 2ppn, 16 bytes up to 1,024 nodes. MVAPICH2-X-SHARP provides a benefit of up to 4.8X over MVAPICH2-X

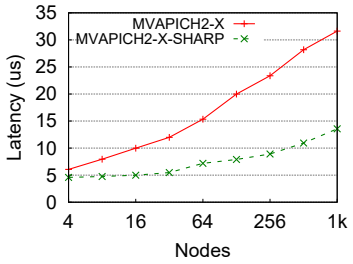


(b) Scaling of MPI_Allreduce for 2ppn, 2,048 bytes up to 7,861 nodes. MVAPICH2-X-SHARP is up to 3.2X better than MVAPICH2-X

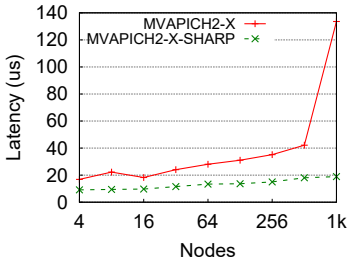


(c) Message size scaling of MPI_Allreduce for 2ppn, 1,024 nodes. MVAPICH2-X-SHARP provides a flat latency curve for message sizes from 4 to 2,048 bytes

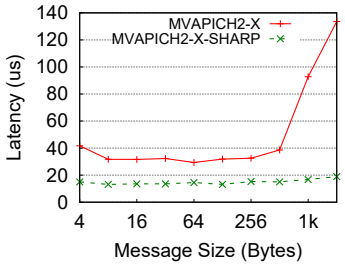
Fig. 7: Performance of MPI_Allreduce with MVAPICH2-X and MVAPICH2-X-SHARP for 2 processes per node (ppn) across various message and system sizes



(a) Scaling of MPI_Allreduce for 16ppn, 16 bytes up to 1,024 nodes. MVAPICH2-X-SHARP provides a benefit of up to 2.5X over MVAPICH2-X

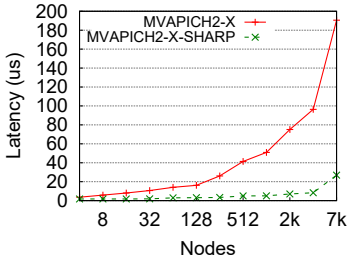


(b) Scaling of MPI_Allreduce for 16ppn, 2,048 bytes up to 1,024 nodes. MVAPICH2-X-SHARP provides a benefit of up to 7.06X over MVAPICH2-X

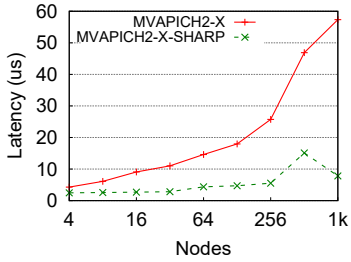


(c) Message size scaling of MPI_Allreduce for 16ppn, 1,024 nodes. MVAPICH2-X-SHARP provides a flat latency curve for message sizes from 4 to 2,048 bytes

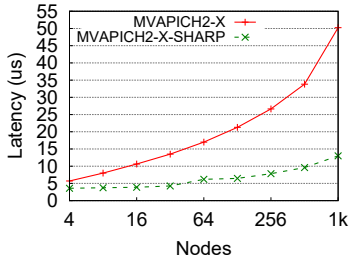
Fig. 8: Performance of MPI_Allreduce with MVAPICH2-X and MVAPICH2-X-SHARP for 16 processes per node (ppn) across various message and system sizes



(a) Scaling of MPI_Barrier for 1ppn, up to 7,861 nodes. MVAPICH2-X-SHARP provides a benefit of up to 11.5X over MVAPICH2-X



(b) Scaling of MPI_Barrier for 2ppn, up to 1,024 nodes. MVAPICH2-X-SHARP provides a benefit of up to 7.3X over MVAPICH2-X



(c) Scaling of MPI_Barrier for 16ppn, up to 1,024 nodes. MVAPICH2-X-SHARP provides a benefit of up to 3.8X over MVAPICH2-X

Fig. 9: Performance of MPI_Barrier with MVAPICH2-X and MVAPICH2-X-SHARP for 1 processes per node (ppn), 2ppn and 16ppn across various system sizes

broadcast phase (As per the socket-aware algorithm described in section IV). Results for absolute numbers across node counts are shown in Figures 6(a) and 6(b) for message sizes of 16 and 2,048 bytes respectively. The X-axes represent increasing node counts, the Y-axes represent the Latency in *us* and two lines colored red and green correspond to MVAPICH2-X and MVAPICH2-X-SHARP respectively. We observe that MVAPICH2-X scales linearly with an increase in node count, starting at 3.66*us* for 4 nodes to 61*us* for 7,861 nodes for a message size of 2,048 bytes. We observed random jumps at certain node counts, which is most likely system noise. MVAPICH2-X-SHARP shows close to flat scalability similar to MPI_Reduce, with the latency remaining around 3-8*us* up to 2,048 nodes and jumping up to 40*us* at 7,861 nodes. We see that MVAPICH2-X-SHARP obtains a speed-up of up to 6.69X versus MVAPICH2-X (at 2,048 nodes for 16 byte messages) and up to 5.1X for 2,048 byte messages (at 7,861 nodes).

2) Scaling trends with multiple processes per node: We evaluate the scaling of MPI_Allreduce with multiple processes per node by comparing the latency for different processes per node for a fixed message sizes of 16 bytes and 2,048 bytes as well as increasing node counts. The absolute latency for different node counts are depicted in Figure 6(a) and 6(b) for 16 and 2,048 byte messages respectively. The X-axis represents increasing node counts, Y-axis represents the latency in *us* and different colored lines are used to show absolute latency of MVAPICH2-X and MVAPICH2-X-SHARP. Due to limited time, we could only run multi processes per node experiments up to 1,024 nodes. We observe that scaling efficiency reduces when increasing the number of processes per node, primarily due to intra-node operations taking more time in the process. At 1,024 nodes, we observe an improvement over MVAPICH2-X of 9X for one process per node jobs, 4X for two processes per node and 2X for 16 processes per node. We observed a decrease in scaling efficiency for four processes per node runs. Unfortunately, due to limited time available on the cluster, we were not able to re-run the experiments. We are planning to do it during the next maintenance window.

3) Scaling trends with increasing message size: Figures 6(c), 7(c), and 8(c) show the impact of message size on the latency of MVAPICH2-X with SHARP. The X-axes represent message size in bytes, Y-axes represent latency in *us* and 2 lines represent MVAPICH2-X and MVAPICH2-X-SHARP respectively. As in MPI_Reduce, we show specific node/process per node counts because they are representative of the entire set of node/process per node pairs. We observe that the latency remains fairly flat for larger messages up to 2,048 bytes when compared to the 4 byte latency with a jump of only 1.2X in most node/process counts. The speed-up over MVAPICH2-X follows a similar trend as seen for MPI_Reduce, with MVAPICH2-X-SHARP showing large benefits of up to 9X over the baseline of MVAPICH2-X.

E. Results for MPI_Barrier

This sub-section evaluates the impact of MPI_Barrier for various processes per node and node counts.

1) Scaling trends with one process per node: For MPI_Barrier, we report the average latency across all processes, similar to what we do for MPI_Allreduce. Just as in the case of MPI_Allreduce, MPI_Barrier shows the same trends for minimum and maximum latency as well. For the one process per node scenario, we evaluate the impact of increasing node counts from 4 nodes up to 7,861 nodes (full system scale) on the latency of both MVAPICH2-X and MVAPICH2-X-SHARP. We observe that MVAPICH2-X-SHARP has near-flat scaling with an increase in node count up to 4,096 nodes, with an average latency between 1.7*us* - 8.2*us*. The numbers at 7,861 nodes increase in the same way as observed for MPI_Allreduce and MPI_Barrier. The experimental results show that the speedup obtained by MVAPICH2-X-SHARP over MVAPICH2-X goes up to a factor of 11.5X at large scales. These results are shown in Figure 9(a). The X-axis in the graph represents increasing node counts, Y-axis represents the latency in *us* and red and green lines represent MVAPICH2-X and MVAPICH2-X-SHARP respectively.

2) Scaling trends with multiple processes per node: We use 2 processes per node and 16 processes per node for a case study of the impact of multiple processes per node on the latency of a SHARP based Barrier implementation in MVAPICH2-X. The trends observed are similar to the ones seen in MPI_Allreduce and MPI_Reduce, with MVAPICH2-X-SHARP providing massive benefits at large scales. MVAPICH2-X-SHARP shows up to 7.3X and 3.8X improvement over MVAPICH2-X for 2 processes per node and 16 processes per node respectively. The reduction in scaling improvement over one process per node runs is explained by the fact that shared-memory based operations add to the intra-node latency in the phased hierarchical algorithm implemented by MVAPICH2-X. These results are shown in Figures 9(b) and 9(c), with the X-axis representing node counts up to 1,024 and Y-axis representing latency in *us*.

VI. Related work

The original sharp paper [15] by Mellanox introduced the SHARP technology, details reasons for several aspects of its design, as well as showed an initial evaluation of reduce, allreduce, and barrier using native benchmarks and MPI level benchmarks.

Bayatpour et al. [16] created novel designs for reduction collectives. These designs select multiple leader processes per node which share computation costs as well as drive concurrent communication. Computation costs are reduced using SHARP with node-level or socket-level leaders.

Kandalla et al. [17] designed new MPI_Iallreduce algorithms using Mellanox CORE-Direct technology to offload communication costs to the network. The proposed designs were demonstrated to overlap communication and computation in the Preconditioned Conjugate Gradient solver routine in the Hypre software library.

Kumar et al. [18] accelerated MPI_Allreduce's computation operations on the Blue Gene/Q supercomputer by taking advantage of each core's Quad Processing SIMD unit.

Mellanox enhanced the SHARP protocol with new technology that ships with the latest Infiniband HDR adapters called Streaming Aggregation in [19]. Reduction trees can be defined to use the existing low-latency reduction or use the new streaming-aggregation capability whose protocol is optimized to increase bandwidth for reduction operations.

Using shared memory for designing collectives is a well-researched topic. Li et al. [20], [21] developed performance models for the collectives using shared memory as well as investigated the design and optimizations of shared memory collectives with NUMA nodes. Zhang et al. [22], [23] use shared memory to handle the communication between virtual machines running on the same node. Their proposed design enables MPI applications running in a virtualized environment have efficient intra-node communication using SR-IOV.

Much work regarding the modeling and redesigning of collective algorithms has been done in literature. Rabenseifner [24] proposed new algorithms for reduce and allreduce which were designed based on the results of an analysis. [25] improved upon collective communication performance by extending point-to-point communication models, such as Hockney [26], LogP/LogGP [27], [28], and PLogP [29] to collective operations. They also introduced "split-binary", an optimized tree-based broadcast algorithm. [30] improved the performance of collectives in MPICH. For each collective, they selected multiple algorithms depending on the message size and number of processes.

VII. Conclusion

In this paper, we Designed, and implemented SHARP-based solutions for MPI_Reduce and MPI_Barrier in MVAPICH2-X. We evaluated the impact the proposed SHARP-based solutions have on the performance of MPI_Reduce and MPI_Barrier at scale on Frontera. We also evaluated the impact that existing SHARP-based solutions have on the performance of MPI_Allreduce at scale on Frontera. We then performed a careful analysis of the benefits of SHARP-based collective operations.

Our experimental evaluation showed that our proposed designs deliver up to 5.4X reduction in latency for Reduce, 5.1X for Allreduce, and 7.1X for Barrier at full system scale of 7,861 nodes over a host-based solution.

As part of future work, we aim to do a more comprehensive evaluation of the SHARP-based collective operations with a larger number of processes per node and even larger scales. The proposed SHARP-based solutions for MPI_Reduce and MPI_Barrier will be available with future releases of the MVAPICH2-X MPI library.

VIII. Acknowledgements

We would like to thank John Cazes from TACC for providing us the necessary access to run our experiments on Frontera at scale.

References

[1] "InfiniBand Trade Association," http://www.infinibandta.com, 2017.
[2] *MPI: A Message-Passing Interface Standard*, Message Passing Interface Forum, Mar 1994.
[3] "MPI-3 Standard Document," http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf, 2012.
[4] "InfiniBand Trade Association, InfiniBand Architecture Specification, Volume 1, Release 1.0," http://www.infinibandta.com.
[5] A. Mamidala, L. Chai, H.-W. Jin, and D. K. Panda, "Efficient SMP-Aware MPI-Level Broadcast over InfiniBand's Hardware Multicast," in *Communication Architecture for Clusters (CAC) Workshop*, 2006.
[6] M. Venkata and R. Graham and J. Ladd and P. Shamis and I. Rabinovitz and 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.
[7] 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," *Comput. Sci.*, vol. 26, pp. 237–246, June 2011. [Online]. Available: http://dx.doi.org/10.1007/s00450-011-0170-4
[8] K. Kandalla, U. Yang, J. Keasler, T. Kolev, A. Moody, H. Subramoni, K. Tomko, J. Vienne, and D. K. Panda, "Designing Non-blocking Allreduce with Collective Offload on InfiniBand Clusters: A Case Study with Conjugate Gradient Solvers," in *IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, 2012.

[9] R. L. Graham, D. Bureddy, P. Lui, H. Rosenstock, G. Shainer, G. Bloch, D. Goldenerg, M. Dubman, S. Kotchubievsky, V. Koushnir, L. Levi, A. Margolin, T. Ronen, A. Shpiner, O. Wertheim, and E. Zahavi, "Scalable Hierarchical Aggregation Protocol (SHArP): A Hardware Architecture for Efficient Data Reduction," in *Proceedings of the First Workshop on Optimization of Communication in HPC*, ser. COM-HPC '16. Piscataway, NJ, USA: IEEE Press, 2016, pp. 1–10.

[10] M. Bayatpour, S. Chakraborty, H. Subramoni, X. Lu, and D. K. D. Panda, "Scalable Reduction Collectives with Data Partitioning-based Multi-leader Design," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '17. New York, NY, USA: ACM, 2017, pp. 64:1–64:11. [Online]. Available: http://doi.acm.org/10.1145/3126908.3126954

[11] J. Vienne, "Benefits of cross memory attach for mpi libraries on hpc clusters," *ACM International Conference Proceeding Series*, 07 2014.

[12] D. Stanzione, J. West, R. T. Evans, T. Minyard, O. Ghattas, and D. K. Panda, "Frontera: The evolution of leadership computing at the national science foundation," in *Practice and Experience in Advanced Research Computing*, ser. PEARC '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 106–111. [Online]. Available: https://doi.org/10.1145/3311790.3396656

[13] H. Subramoni, K. Hamidouche, A. Venkatesh, S. Chakraborty, and D. K. Panda, "Designing MPI Library with Dynamic Connected Transport (DCT) of InfiniBand: Early Experiences," in *Supercomputing*, ser. Lecture Notes in Computer Science. Springer International Publishing, 2014.

[14] O. Micro-Benchmarks, http://mvapich.cse.ohio-state.edu/benchmarks, 2017.

[15] R. L. Graham, D. Bureddy, P. Lui, H. Rosenstock, G. Shainer, G. Bloch, D. Goldenerg, M. Dubman, S. Kotchubievsky, V. Koushnir, L. Levi, A. Margolin, T. Ronen, A. Shpiner, O. Wertheim, and E. Zahavi, "Scalable hierarchical aggregation protocol (sharp): A hardware architecture for efficient data reduction," in *2016 First International Workshop on Communication Optimizations in HPC (COMHPC)*, 2016, pp. 1–10.

[16] M. Bayatpour, S. Chakraborty, H. Subramoni, X. Lu, and D. K. D. Panda, "Scalable reduction collectives with data partitioning-based multi-leader design," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '17. New York, NY, USA: ACM, 2017, pp. 64:1–64:11. [Online]. Available: http://doi.acm.org/10.1145/3126908.3126954

[17] K. Kandalla, U. Yang, J. Keasler, T. Kolev, A. Moody, H. Subramoni, K. Tomko, J. Vienne, B. R. de Supinski, and D. K. Panda, "Designing non-blocking allreduce with collective offload on infiniband clusters: A case study with conjugate gradient solvers," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, 2012, pp. 1156–1167.

[18] S. Kumar, A. Mamidala, P. Heidelberger, D. Chen, and D. Faraj, "Optimization of mpi collective operations on the ibm blue gene/q supercomputer," *Int. J. High Perform. Comput. Appl.*, vol. 28, no. 4, p. 450–464, Nov. 2014. [Online]. Available: https://doi.org/10.1177/1094342014552086

[19] R. L. Graham, L. Levi, D. Burredy, G. Bloch, G. Shainer, D. Cho, G. Elias, D. Klein, J. Ladd, O. Maor, A. Marelli, V. Petrov, E. Romlet, Y. Qin, and I. Zemah, "Scalable hierarchical aggregation and reduction protocol (sharp)tm streaming-aggregation hardware design and evaluation," in *High Performance Computing*, P. Sadayappan, B. L. Chamberlain, G. Juckeland, and H. Ltaief, Eds. Cham: Springer International Publishing, 2020, pp. 41–59.

[20] S. Li, T. Hoefler, and M. Snir, "NUMA-aware Shared-memory Collective Communication for MPI," in *Proceedings of the 22Nd International Symposium on High-performance Parallel and Distributed Computing*, ser. HPDC '13. New York, NY, USA: ACM, 2013, pp. 85–96.

[21] S. Li, T. Hoefler, C. Hu, and M. Snir, "Improved MPI Collectives for MPI Processes in Shared Address Spaces," *Cluster Computing*, vol. 17, no. 4, pp. 1139–1155, Dec. 2014.

[22] J. Zhang, X. Lu, J. Jose, M. Li, R. Shi, and D. K. D. K. Panda, "High Performance MPI Library over SR-IOV enabled InfiniBand Clusters," in *2014 21st International Conference on High Performance Computing (HiPC)*, Dec 2014, pp. 1–10.

[23] J. Zhang, X. Lu, J. Jose, R. Shi, and D. K. D. Panda, *Can Inter-VM Shmem Benefit MPI Applications on SR-IOV Based Virtualized Infiniband Clusters?* Cham: Springer International Publishing, 2014, pp. 342–353.

[24] R. Rabenseifner, "Optimization of Collective Reduction Operations," in *International Conference on Computational Science*. Springer, 2004, pp. 1–9.

[25] J. Pjesivac-Grbovic, T. Angskun, G. Bosilca, G. E. Fagg, E. Gabriel, and J. J. Dongarra, "Performance Analysis of MPI Collective Operations," in *19th IEEE International Parallel and Distributed Processing Symposium*, April 2005, pp. 8 pp.–.

[26] R. W. Hockney, "The Communication Challenge for MPP: Intel Paragon and Meiko CS-2," *Parallel Comput.*, vol. 20, no. 3, pp. 389–398, Mar. 1994.

[27] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken, "LogP: Towards a Realistic Model of Parallel Computation," in *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1993, pp. 262–273.

[28] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman, "LogGP: Incorporating Long Messages into the LogP Model&Mdash;One Step Closer Towards a Realistic Model for Parallel Computation," in *Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA '95. New York, NY, USA: ACM, 1995, pp. 95–105.

[29] T. Kielmann, H. E. Bal, and K. Verstoep, "Fast Measurement of LogP Parameters for Message Passing Platforms," in *Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing*, ser. IPDPS '00. London, UK, UK: Springer-Verlag, 2000, pp. 1176–1183.

[30] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of Collective Communication Operations in MPICH," *Int. J. High Perform. Comput. Appl.*, vol. 19, no. 1, pp. 49–66, Feb. 2005.