

# Designing In-network Computing Aware Reduction Collectives in MPI

Bharath Ramesh, Goutham Kalikrishna Reddy Kuncham, Kaushik Kandadi Suresh, Rahul Vaidya, Nawras Alnaasan, Mustafa Abduljabbar, Aamir Shafi, Hari Subramoni, Dhabaleswar K. (DK) Panda

*Department of Computer Science and Engineering  
The Ohio State University  
Columbus, USA*

{ramesh.113, kuncham.2, kandadisuresh.1, vaidya.84, alnaasan.1, abduljabbar.1, shafi.16, subramoni.1, panda.2}@osu.edu

**Abstract**—The Message-Passing Interface (MPI) provides convenient abstractions such as MPI\_Allreduce for inter-process collective reduction operations. With the advent of deep learning and large-scale HPC systems, it is ever so important to optimize the latency of the MPI\_Allreduce operation for large messages. Due to the amount of compute and communication involved in MPI\_Allreduce, it is beneficial to offload collective computation/communication to the network to allow the CPU to work on other important operations and provide maximal overlap/scalability. NVIDIA’s HDR InfiniBand switches provide in-network computing features using the Scalable Hierarchical Aggregation and Reduction Protocol (SHARP) for this purpose with two protocols targeted at different message ranges: 1) Local Latency Tree (LLT) for small messages, and 2) Streaming aggregation tree (SAT) for large messages. In this paper, we first analyze the overheads involved in using SHARP-based reductions with SAT in an MPI library using micro-benchmarks. Next, we propose designs for large message MPI\_Allreduce by fully utilizing the capabilities provided by the SHARP runtime while overcoming various bottlenecks. The efficacy of our proposed designs is demonstrated using micro-benchmark results. We observe up to 89% improvements over MVAPICH2-X and HPC-X for large message reductions.

**Index Terms**—HPC, InfiniBand, MPI, In-network computing, NVIDIA SHARP, Allreduce

## I. INTRODUCTION

Modern HPC systems span thousands of nodes with hundreds of cores per node for massively parallel compute tasks. Running massively parallel applications and benchmarks that fully utilize these systems requires robust distributed communication runtimes. In the context of HPC, the Message Passing Interface (MPI) has been the de-facto standard for distributed communication and provides convenient abstractions for point-to-point and collective communication. The appearance of CPU-only clusters in the top 500 list [1], such as the #2 ranked Fugaku system with 7,630,848 cores [2] has made it even more important for MPI libraries to optimize collective performance at scale. Achieving high scale-up and scale-out efficiency for collectives on these systems requires communication runtimes to adapt and optimize their performance by taking hardware and software factors into account.

\*This research is supported in part by NSF grants #1818253, #1854828, #1931537, #2007991, and #2018627

Large-message reductions (MPI\_Allreduce/MPI\_Reduce) are dominant collectives, especially in Deep Learning (DL) and Molecular Dynamics (MD) applications [3], [4], [5]. Nevertheless, as the message sizes increase, the communication overhead related to MPI\_Allreduce also grows substantially. Transferring large messages between nodes takes more time, resulting in higher latency and potentially slower overall execution time. To address this issue, optimizing MPI\_Allreduce for large messages can effectively reduce the associated communication overhead. Furthermore, large messages can consume a significant amount of memory on each node, leading to additional overhead in terms of intermediate buffering at the MPI level.

One of the most popular techniques to address the compute and memory requirements of MPI reductions is offloading compute/communication operations to network adapters and switches. Such a mechanism immediately releases host resources and enables the MPI implementation to free up large intermediate send buffers. This allows the CPU’s compute to overlap with communication, while also providing higher throughput at scale. High-end interconnect vendors have provided multiple hardware-level features to enable low latency and scalability. However, these do not cater to large-message reductions. Take for example NVIDIA’s CORE-Direct[6], which allowed offloading reduction as well as communication operation to network adapters and switches. CORE-direct compute capabilities quickly became a bottleneck for large message reductions. NVIDIA’s InfiniBand adapters provide hardware-based Multicast support [7], which allows one process to send data to multiple processes in one step. NVIDIA’s Scalable Hierarchical Aggregation and Reduction Protocol (SHARP) enables the use of in-network computing to offload reduction operations to the network, while also supporting efficient hardware-based communication features such as Multicast. With the introduction of streaming aggregation trees (SAT)[8], the SHARP runtime streams data over the network while performing reductions at line rates in a topology-aware fashion. Since the reduction is done on the switches, data is reduced on-the-fly (i.e. combined) and then sent over to the intended destination processes with minimal redundancy.

SHARP achieves high scalability[9], [10], [8] and enables the CPU to focus on other compute tasks.

Prior work on reduction collectives that utilize SHARP for inter-node operations have primarily been designed for message sizes that are in the small/medium range (between 4 bytes and 64K bytes) [9], [11]. These solutions do not work well for large messages as they rely on distinct intra-node and inter-node phases and do not utilize streaming aggregation features provided by NVIDIA’s InfiniBand interconnect. In this paper, we propose designs for reduction collectives in MPI using the streaming aggregation tree protocol in the NVIDIA SHARP runtime by analyzing and solving various bottlenecks. The overall idea of the design is to use all available CPU cores to perform compute operations within the node, while also having optimal network utilization by using SHARP for inter-node operations in a non-blocking fashion.

## II. BACKGROUND

### A. Zero-copy intra-node transport mechanisms

XPMEM/Cross-Partition Memory allows processes to share their address space with other processes, by enabling processes to map regions of memory that belong to other processes into their own address space. The other process can then read/write to the remote process’s shared address region without any kernel involvement. This mechanism supports load/store access to a remote process’s memory, thereby enabling fine-grained reduction operations and one-copy writes to the remote process. The term zero-copy refers to the transport requiring zero additional copies than the minimum required to transfer data.

### B. Nvidia SHARP

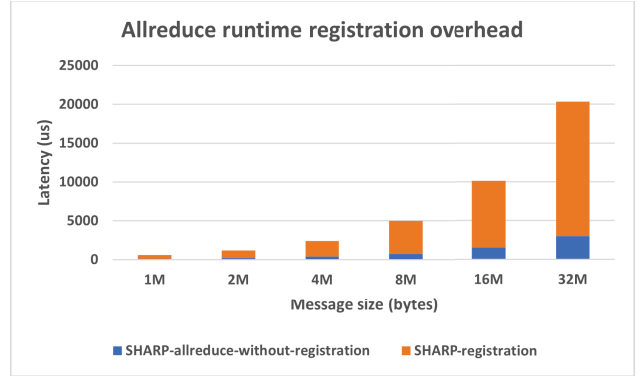
Scalable Hierarchical Aggregation Protocol (SHARP) technology offloads the processing of collective reduction operations to the network. The reduction tree structure is hierarchical, with leaves representing the data source and vertices/non-leaf nodes representing aggregation nodes. Reduction operations are performed by the Aggregation Nodes (AN). Upon reaching the root of the tree, the data is distributed by the hardware. SHARP broadly provides two protocols for message transfers – Low Latency Trees (LLT), and Streaming Aggregation Trees (SAT). LLTs are optimized for smaller messages and follow a latency optimized protocol through the use of hierarchical tree-based algorithms, whereas SATs are bandwidth optimized and perform pipelined ring-based reductions.

## III. MOTIVATION

We motivate the need for our designs by highlighting the advantages as well as bottlenecks involved in the use of streaming aggregation for reduction collectives.

### A. High overheads at runtime

Similar to InfiniBand verbs APIs [12], the SHARP runtime requires buffers to be registered with the network adapter using `sharp_coll_reg_mr`, which involves pinning pages and is hence



**Figure 1:** Overheads for a two process allreduce with SHARP.

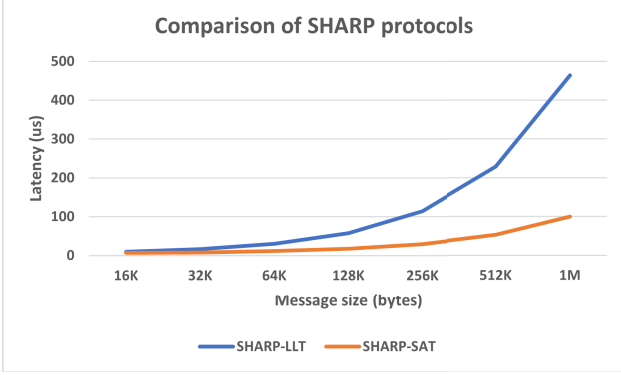
very expensive. Figure 1 shows the breakdown of a SHARP-based allreduce using streaming aggregation. The orange bars represent the registration cost and the blue bars represent the latency of an allreduce between two processes using SHARP. As shown in the figure, the orange bars clearly dominate the overall latency of the allreduce for large messages. This becomes a problem when an allreduce is called repeatedly on the same buffer as every iteration involves a registration call, which is redundant after the first iteration for a given buffer. This shows the need for caching metadata on buffers and the registration process to amortize the cost.

### B. Resource limitations in multi-core scenarios

The usage of streaming aggregation involves acquiring locks on switch-level resources, which are limited[8]. Moreover, there are also hardware limitations placed on the number of processes within the node that can use SHARP-based collectives. This quickly becomes a problem when scaling up within the node, since not all processes can use the SHARP runtime. Prior work[11], [9] has proposed the usage of leader-based designs for MPI collectives to alleviate such problems, by only allowing a designated leader process to use the SHARP runtime for small message reductions. However, designing an algorithm that works for large messages requires an analysis of the capabilities provided by streaming aggregation while being aware of resource limitations. This motivates the need for resource-aware leader-based reduction designs that reap the benefits of using streaming aggregation trees given by the SHARP runtime while also performing well for jobs with many processes per node.

### C. Better bandwidth utilization

Figure 2 shows an allreduce collective latency comparison of the low latency tree (LLT) protocol versus the streaming aggregation tree (SAT) protocol in the SHARP runtime. As shown in the figure, the orange line, which represents SAT significantly outperforms the LLT scheme with an increase in message size. This implies significantly better bandwidth utilization over the network using SAT over LLT, which



**Figure 2:** Allreduce latency numbers on two processes

is very important for large messages. The latency shown by SHARP-SAT is very close to the point-to-point latency between two processes, indicating wire speed reduction of data. This motivates the need for designs that make use of the significant gains in bandwidth and scalability provided by SAT in the SHARP runtime while taking resource limitations into account.

#### IV. CONTRIBUTIONS

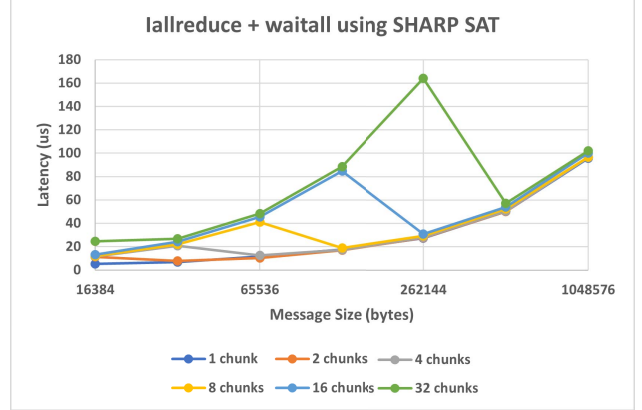
This paper proposes reduction collective designs that use SHARP’s streaming aggregation tree for inter-node compute and communication. We conduct an analysis of streaming aggregation capabilities, and bottlenecks involved in using it for reduction collectives. The proposed design overcomes resource constraints and bottlenecks in the NVIDIA SHARP runtime and outperforms the state-of-the-art NVIDIA HPC-X library [13] for reduction collectives. The paper makes the following contributions:

- 1) Identify registration overheads involved in the use of SHARP streaming aggregation for large messages and propose solutions to address them.
- 2) Analyze the impact of chunking reductions when using streaming aggregation for different message sizes to empirically determine ways to overlap intra-node reductions with SHARP-based reductions.
- 3) Propose an algorithm for large message allreduce that overcomes bottlenecks and resource constraints in the SHARP runtime by making efficient use of node and network level resources.
- 4) Evaluate the proposed design by comparing it against state-of-the-art MPI libraries using micro-benchmarks. Our proposed designs outperform state-of-the-art libraries by up to **89%** for a message size of 256K.

#### V. DESIGN AND IMPLEMENTATION

##### A. Analyzing the impact of chunking streaming aggregation operations

A Naïve algorithm for an allreduce that involves intra-node and inter-node phases would be to perform a local intra-

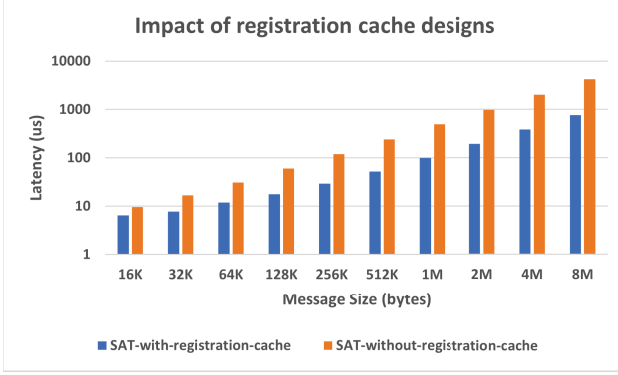


**Figure 3:** Analyzing the impact of chunking one iallreduce call into multiple calls

node reduce, followed by an inter-node allreduce, and finally an intra-node broadcast. The problem with this approach is that it leads to under-utilization of the network for large messages, as the inter-node step has to wait for a large message intra-node reduction to complete, which can take significant time as the number of processes and the message size scale. An intuitively better approach would be to chunk the intra-node and inter-node operations into multiple steps, with the primary goal being pipelining intra-node and inter-node transfers/reductions. This necessitates the need to study the behavior of SAT when chunking multiple transfers and offloading them to the network. We study this with a simple experiment that involves varying the number of non-blocking iallreduces for every message size, followed by a waitall. For instance, a 1M allreduce could be represented as two 512K iallreduces followed by a waitall. Figure 3 shows the impact of running iallreduce + waitall on eight nodes with one process per node. The graph shows that the latency of combining multiple smaller non-blocking SHARP reductions that are greater than or equal to 16K (which is the streaming aggregation threshold on the system) is the same as the latency of one single large iallreduce. This observation is important as it shows that it is safe to divide reductions into chunks that are greater than or equal to 16K without any loss in performance. Hence, the design only needs to empirically determine the size of intra-node reductions that have to be completed before issuing an inter-node operation using SHARP.

##### B. Registration cache design

As described in Figure 1, the cost of registration forms a significant portion of the allreduce runtime and has to be amortized in order to achieve good latency for reductions. To alleviate this bottleneck, we use self-balancing binary search trees that index each address that is registered with the SHARP runtime. The first call to allreduce involves an insertion operation, which finishes in  $O(\log n)$  time (where  $n$  is the number of unique buffer entries registered with



**Figure 4:** Impact of registration cache designs using OSU micro-benchmarks on a 8 nodes, 1 PPN allreduce

SHARP) plus the cost of registration. Subsequent calls to allreduce using the same buffer will only involve querying the tree to get the right registration metadata, which is a  $O(\log n)$  operation and hence avoids redundant and expensive registration calls. The performance improvements obtained from the registration cache design is shown in Figure 4. We observe up to **6X** benefits against a design without caching using micro-benchmarks for an 8 node, 1 PPN allreduce. This is due to avoiding repeated expensive registration calls to the SHARP runtime. The performance improvements increase significantly as the message size for the allreduce increases.

#### Algorithm 1: Common routines

**Inputs :**  
*sendbuf* — sendbuf from MPI\_Allreduce  
*recvbuf* — recvbuf from MPI\_Allreduce  
*count* — Number of elements in sendbuf/recvbuf  
*datatype* — Size of data type contained in sendbuf/recvbuf  
*num\_chunks* — Tuned number of chunks to decide the size of upper-level chunks in each step

**Function:**  
*exchange\_buffer\_information* (*sendbuf*, *recvbuf*)  
**begin**  
  *region* = *get\_shared\_mem\_region*()  
  *region*[*local\_rank*].*sendbuf* = *sendbuf*  
  *region*[*local\_rank*].*recvbuf* = *recvbuf*  
**end**

**Function:**  
*reduce\_chunk* (*local\_rank*, *local\_size*, *dest*, *count*, *datatype*, *op*, *offset*)  
**begin**  
  **for** *src\_rank*  $\leftarrow$  1 **to** *local\_size* **do**  
    **if** *src\_rank* == *local\_rank* **then**  
      *src\_addr*  $\leftarrow$  *sendbuf*  
    **end**  
    **else**  
      *src\_addr*  $\leftarrow$  *get\_remote\_sendbuf*(*src\_rank*)  
    **end**  
    *src\_addr*  $\leftarrow$  *src\_addr* + *offset*  
    *do\_local\_reduce\_op*(*src\_addr*, *dest*, *count*, *datatype*)  
  **end**  
**end**

#### Algorithm 2: Allreduce algorithm

**Inputs :**  
*sendbuf* — sendbuf from MPI\_Allreduce  
*recvbuf* — recvbuf from MPI\_Allreduce  
*count* — Number of elements in sendbuf/recvbuf  
*datatype* — Size of data type contained in sendbuf/recvbuf  
*num\_chunks* — Tuned number of chunks to decide the size of upper-level chunks in each step

**Function:**  
*MPI\_Allreduce* (*sendbuf*, *recvbuf*, *count*, *datatype*, *op*)  
**begin**  
  **if** *is\_leader*(*local\_rank*) **then**  
    *tmpbuf*  $\leftarrow$  *get\_tmpbuf\_from\_pool*()  
    *memcpy*(*tmpbuf*, *sendbuf*, *count* \* *sizeof*(*datatype*))  
    *reqs*  $\leftarrow$  *get\_reqs\_from\_pool*(*num\_chunks*)  
    *exchange\_buffer\_information*(*tmpbuf*, *recvbuf*)  
  **end**  
  **else**  
    *exchange\_buffer\_information*(*sendbuf*, *recvbuf*)  
  **end**  
  *do\_local\_barrier*()  
  **for** *chunk*  $\leftarrow$  0 **to** *num\_chunks* **do**  
    *offset*  $\leftarrow$  *chunk\_size* \* *chunk* + (*chunk\_size*/*local\_size*) \* *local\_rank*  
    *chunk\_count*  $\leftarrow$  *chunk\_size*/*type\_size*  
    *dest*  $\leftarrow$  *get\_leader\_tmpbuf*() + *offset*  
    *reduce\_chunk*(*local\_rank*, *local\_size*, *dest*, *count*, *datatype*, *op*, *offset*)  
    **if** *is\_leader*(*local\_rank*) **then**  
      *sharp\_allreduce\_nb*(*tmpbuf*, *recvbuf* + *chunk*\**chunk\_size*, *chunk\_count*, *datatype*, *op*, *reqs*[*chunk*], *leader\_comm*)  
    **end**  
  **end**  
  **if** *is\_leader*(*local\_rank*) **then**  
    *sharp\_allreduce\_waitall*(*reqs*, *num\_chunks*)  
  **end**  
  *do\_local\_barrier*()  
  *root* = *leader*  
  *do\_local\_bcast*(*recvbuf*, *count*, *datatype*, *root*)  
**end**

#### C. Allreduce design

The overall idea of the algorithm is to use XPMEM-based transfers within the node while pipelining them with the SHARP runtime for any inter-node transfers based on the observations on chunking SHARP operations and resource constraints. Note that this only works for commutative reduction operations. CPU cores do most of the intra-node reductions, while inter-node reductions are done by the SHARP runtime. One process on each node is designated as a leader. Only leader processes interact with the SHARP runtime. Algorithm 2 shows the implementation details of our algorithm. The major steps involved in our design for MPI\_Allreduce are described below.

1) *Exchanging buffer information and registration:* The first step of our algorithm involves exchanging buffer information between processes within a node, as well as any registration operations related to the XPMEM runtime. The term *local\_rank* is used to denote the relative rank of a process

within the node (as opposed to the mpi rank, which represents the rank amongst all processes in the entire job). The function `exchange_buffer_information` shown in Algorithm 1 shows the pseudo code for this step. We use a shared memory region to store a per-local rank structure that contains metadata about the send/receive buffers that are part of the MPI\_Allreduce call. At the start of each allreduce, all ranks write metadata into a designated location in the shared region. This information is later used in the reduction phase, which is explained in V-C3. At the end of this step, all processes have access to the send/receive buffers of every other process through XPMEM calls that attach to remote memory and map regions to their own address space. These calls are amortized using a software cache.

2) *Maintaining a pool of temporary buffers:* We maintain a pool of temporary buffers on every leader process. The temporary buffers serve as a location for processes to write their reduced results into. The temporary buffer initially contains a copy of the send buffer from the leader processes. All processes then store their locally reduced results into the temporary buffers. The temporary buffers are then used as send buffers for a SHARP-based inter-node allreduce. Streaming aggregation has resource constraints and the SHARP runtime only allows a subset of processes to execute reduction operations on the network. Using a temporary buffer that all processes write to allows the leader process to use it for an allreduce operation using the SHARP runtime, thereby working around the imposed constraints.

3) *Chunking reductions:* The function `reduce_chunks` shown in Algorithm 1 explains the code for chunking reduction operations. Each process within the node is responsible for the reduction of a chunk of a buffer. If the number of processes is PPN, then every send buffer is divided into PPN chunks, to form chunks of size `message_size/PPN`. The idea here is to use all processes on the node to perform reductions on buffers of size `message_size/PPN`. Given a `local_rank`, we follow a linear strategy to map chunks to processes i.e. a process with `local_rank i` gets the  $i^{\text{th}}$  chunk. To perform the reduction, process  $i$  within the node loads the  $i^{\text{th}}$  chunk of the send buffer from every process (including itself) and reduces the vector it represents with the leader process's `tmpbuf`. The reduced result is stored in the `tmpbuf` at an appropriate offset represented by the chunk the process is responsible for. The loads/stores are possible through the use of XPMEM, as explained in V-C1. This step is followed by a local barrier after which the reduced result is used by the leader process to call an inter-node reduction using SHARP.

While reducing a chunk of the send buffer by using every process achieves great parallelism, waiting until the entire reduction is complete before the inter-node reduction is called leads to the under-utilization of the network. For this purpose, we have an outer loop that performs another higher level of chunking (we call this upper-level chunking), with the upper-level `chunk_size` being a tunable parameter. The idea is to send a portion of the send buffer into the chunked reduction routine explained in the previous paragraph, and changing the offset at

every step based on the `chunk_size`. The optimal `chunk_size` is empirically determined and depends on the system's CPU and network architectures.

4) *Using non-blocking SHARP operations for inter-node transfers:* As soon as a reduced chunk is available, leader processes on every node perform an allreduce using SHARP. We first query our registration cache for SHARP operations to see if the buffers are already registered. If not, they are added to the registration cache. The output structure from the registration process is then used in the call to a SHARP allreduce. We use the non-blocking variant of SHARP allreduce so that the leader process can also participate in the reduction, which is only possible due to the truly asynchronous nature of the SHARP runtime. To account for the upper-level chunking described in V-C3, we use a request for each upper-level chunk. Once the reduction of all upper-level chunks is done, the leader process then waits on all SHARP allreduce requests. At the end of this step, the leader process on every node contains the final reduced result.

5) *Local broadcast:* Since the inter-node step is complete and the leader process on every node contains the final reduced result, the last step in the algorithm involves an intra-node XPMEM-based broadcast with the leader process as the root. To prevent unnecessary skews, all processes perform another intra-node barrier after this step.

## VI. EXPERIMENTAL EVALUATION

In this section, we describe the experimental setup and show the micro-benchmark level evaluations of the proposed designs in an MPI library.

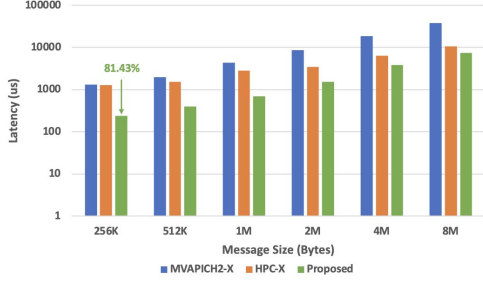
### A. Experimental setup

We implemented our proposed design in the MVAPICH2-X [14] MPI library. We compare the performance of the proposed schemes with MVAPICH2-X v2.3 and HPC-X MPI libraries. We use the `osu_allreduce` benchmark from the OSU micro-benchmarks v7.1 suite [15] for all our benchmark-level evaluations. All experiments are run an average of five times to account for any variability in results.

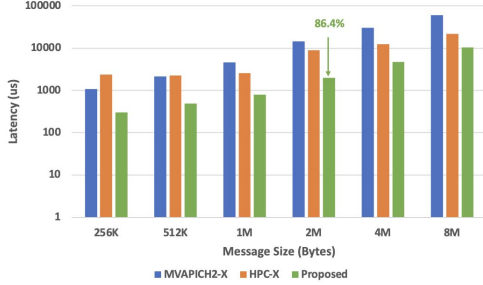
We ran our experiments on two clusters, both with an InfiniBand HDR interconnect. The first cluster (MRI) has 12 NVIDIA A-100 equipped nodes with each node containing a two-socket AMD EPYC 7713 64-Core Processor. The second cluster (HPCAC) contains 32 nodes, each equipped with a ConnectX-6 HCA and dual-socket Xeon Gold 6138 20-core processors. Both systems have NVIDIA quantum-2 switches that support streaming aggregation trees for offloading reductions. We demonstrate results starting from 16 processes per node, as our design relies on sufficient cores being available within the node to perform local reductions.

### B. Performance of MPI\_Allreduce on MRI

This section presents the performance of MPI\_Allreduce on the MRI cluster, specifically on two, four, and eight nodes, with message sizes ranging from 256KB to 8MB. Figures 5(a), and 5(b) illustrate the two node evaluations for 32

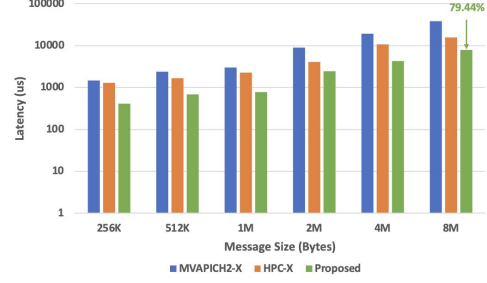


(a) osu\_allreduce on 2 Nodes, 32 PPN

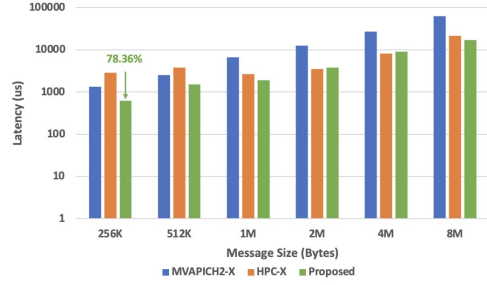


(b) osu\_allreduce on 2 Nodes, 64 PPN

**Figure 5:** Comparison of the proposed design against HPC-X and MVAPICH2-X using osu\_allreduce on two nodes for messages ranging from 256K bytes to 8M bytes on the MRI cluster.

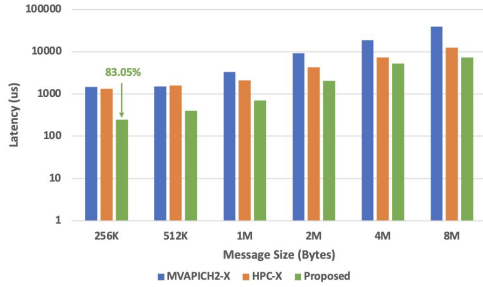


(a) osu\_allreduce on 8 Nodes, 32 PPN

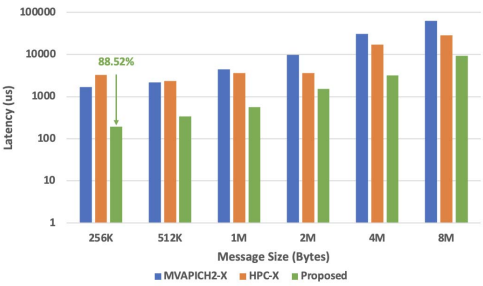


(b) osu\_allreduce on 8 Nodes, 64 PPN

**Figure 7:** Comparison of the proposed design against HPC-X and MVAPICH2-X using osu\_allreduce on eight nodes for messages ranging from 256K bytes to 8M bytes on the MRI cluster.



(a) osu\_allreduce on 4 Nodes, 32 PPN



(b) osu\_allreduce on 4 Nodes, 64 PPN

**Figure 6:** Comparison of the proposed design against HPC-X and MVAPICH2-X using osu\_allreduce on four nodes for messages ranging from 256K bytes to 8M bytes on the MRI cluster.

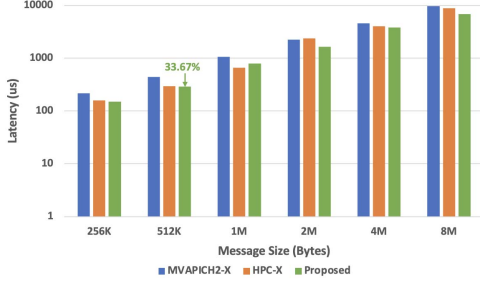
and 64 processes per node (PPN). At 32 PPN, we achieve improvements of up to 81.25% compared to HPC-X and up to 81.43% compared to the MVAPICH2-X MPI library. With 64 PPN, our proposed scheme outperforms MVAPICH2-X and HPC-X libraries by 86.4% and 87.28%, respectively. The improvements at this scale are due to the perfect overlap of inter-node reductions with intra-node reductions, and complete utilization of all CPU cores for intra-node reductions.

On 4 nodes, we observe improvements of up to 83.43% compared to MVAPICH2-X and 88.74% compared to HPC-X as shown in Figures 6(a) and 6(b). Similarly, on eight nodes (Figure 7(a)) with 32 PPN, we see improvements of up to 79.44% and 68.11% when compared to MVAPICH2-X and HPC-X MPI libraries respectively. As shown in Figure 7(b), we observe improvements of up to 73.25% and 78.36% compared to MVAPICH2-X and HPC-X respectively on 8 nodes, 64 PPN. We observed a performance degradation for message sizes close to the L2 cache boundary and are investigating possible reasons why. We suspect this might be due to sub-optimal cache utilization and a lack of NUMA awareness in our implementation.

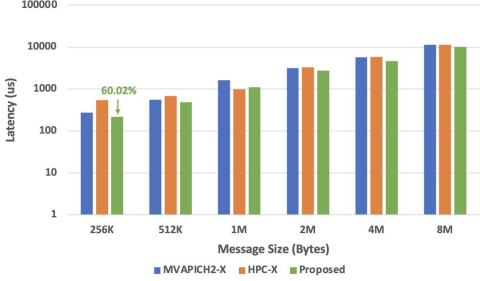
### C. Performance of MPI\_Allreduce on HPCAC

This section examines the performance of MPI\_Allreduce on the HPCAC cluster across two, four, and eight nodes, with message sizes ranging from 256KB to 8MB. We only had access to a subset of eight nodes since our designs rely on the XPMEM kernel module. Figures 8(a) and 8(b) depict



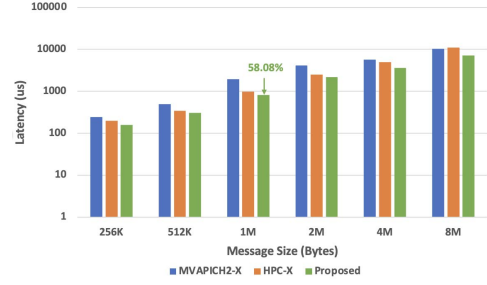


(a) osu\_allreduce on 2 Nodes, 16 PPN

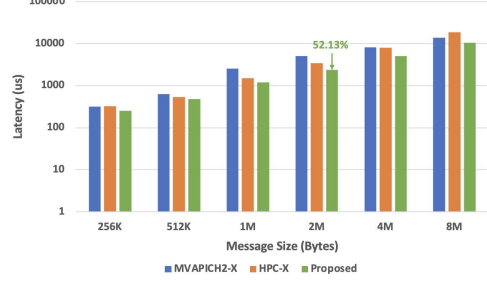


(b) osu\_allreduce on 2 Nodes, 32 PPN

**Figure 8:** Comparison of the proposed design against HPC-X and MVAPICH2-X using osu\_allreduce on two nodes for messages ranging from 256K bytes to 8M bytes on the HPCAC cluster.

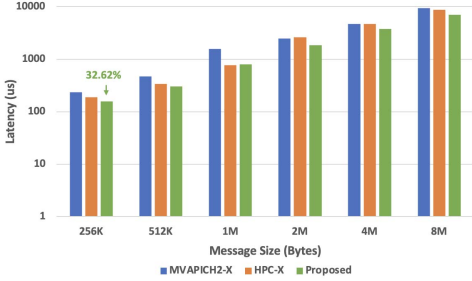


(a) osu\_allreduce on 8 Nodes, 16 PPN

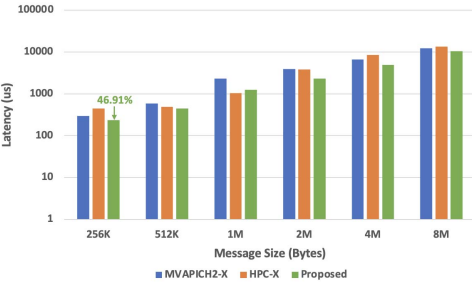


(b) osu\_allreduce on 8 Nodes, 32 PPN

**Figure 10:** Comparison of the proposed design against HPC-X and MVAPICH2-X using osu\_allreduce on eight nodes for messages ranging from 256K bytes to 8M bytes on the HPCAC cluster.



(a) osu\_allreduce on 4 Nodes, 16 PPN



(b) osu\_allreduce on 4 Nodes, 32 PPN

**Figure 9:** Comparison of the proposed design against HPC-X and MVAPICH2-X using osu\_allreduce on four nodes for messages ranging from 256K bytes to 8M bytes on the HPCAC cluster.

the evaluations for 16 and 32 processes per node (PPN) on 2 nodes. At 16 PPN, our proposed scheme demonstrates improvements of up to 22.7% compared to HPC-X and up to 33.67% compared to the MVAPICH2-X MPI library. With 32 PPN, our proposed scheme surpasses MVAPICH2-X and HPC-X libraries by 32.16% and 60.02% respectively.

On 4 nodes (Figures 9(a) and 9(b)), we observe enhancements of up to 48.72% compared to MVAPICH2-X and 46.91% compared to HPC-X. Likewise, on 8 nodes (Figure 10(a)) with 16 PPN, we achieve improvements of up to 58.08% and 36.4% relative to MVAPICH2-X and HPC-X MPI libraries respectively. On 8 nodes (Figure 10(b)) with 32 PPN, we note improvements of up to 53.80% and 43.26% compared to MVAPICH2-X and HPC-X respectively.

## VII. RELATED WORK

Researchers have focused on optimizing various MPI collectives and improving their performance. Bayatpour et al. [16] proposed reduction collective designs for large messages and significantly improved the performance of MPI\_Allreduce on modern multi-core architectures. They chunk buffers across processes and have a dedicated root process that performs communication using one-sided operations to overlap intra and inter-node phases. Our proposed design uses all cores for compute and does not require a dedicated process to perform communication operations. In [11], Bayatpour et al. propose DPML, a multi-leader-based approach to perform MPI\_Allreduce on medium messages. DPML used multiple

communicators to perform simultaneous reduction operations and uses shared memory within the node, limiting its performance for large messages.

Hashmi et al. [17] re-designed MPI collective algorithms based on XPMEM (Shared Address Spaces) and showed significant performance gains on modern multi-core systems. Kandalla et al. [18] proposed novel designs for MPI\_Iallreduce using CORE-Direct technology to offload communication to the network. The new designs were evaluated on Preconditioned Conjugate Gradient solver routine in the Hypr library to show communication and computation overlap. Researchers have also used shared memory to design various collectives. Zhang et al., [19] [20] explored shared memory utilization to facilitate communication between virtual machines, running on the same node. Li et al., [21] developed performance models for collectives utilizing and investigated the design and optimization of shared memory collectives targeting NUMA nodes.

## VIII. CONCLUSION AND FUTURE WORK

The advent of large-scale HPC systems with thousands of nodes and millions of cores have made it ever so important to optimize communication performance to achieve high performance and scalability. Reduction collectives are very commonly used in various domains and require optimizations with both compute and communication. In-network computing features provided by modern interconnects such as InfiniBand have enabled high throughput and scalability only achievable through hardware. In this paper, we propose an algorithm for reduction collectives in MPI that utilize the in-network feature of streaming aggregation provided by the SHARP runtime for large messages. We analyze various bottlenecks and propose a solution that works well for a large number of processes within the node while also having good scale-out performance across nodes. Our proposed designs show up to **89%** over state-of-the-art MPI libraries for large message allreduce. As future work, we plan to evaluate application use cases and scalability at higher node counts, as resources become available.

## REFERENCES

- [1] H. Meuer, E. Strohmaier, J. Dongarra, and H. Simon, "TOP 500 Supercomputer Sites," <http://www.top500.org>.
- [2] T. Shimizu, "Supercomputer Fugaku: Co-designed with application developers/researchers," in *2020 IEEE Asian Solid-State Circuits Conference (A-SSCC)*, 2020, pp. 1–4.
- [3] R. Salomon-Ferrer, D. A. Case, and R. C. Walker, "An overview of the Amber biomolecular simulation package," *WIREs Computational Molecular Science*, vol. 3, no. 2, pp. 198–210, 2013. [Online]. Available: <https://wires.onlinelibrary.wiley.com/doi/abs/10.1002/wcms.1121>
- [4] J. Rasley, S. Rajbhandari, O. Ruwase, and Y. He, "DeepSpeed: System Optimizations Enable Training Deep Learning Models with Over 100 Billion Parameters," in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 3505–3506. [Online]. Available: <https://doi.org/10.1145/3394486.3406703>
- [5] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in *Advances in Neural Information Processing Systems* 32. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: <http://papers.nips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf>
- [6] 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 *2010 18th IEEE Symposium on High Performance Interconnects*, 2010, pp. 40–49.
- [7] J. Liu, A. Mamidala, and D. Panda, "Fast and scalable MPI-level broadcast using InfiniBand's hardware multicast support," in *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, 2004, pp. 10–.
- [8] R. L. Graham, L. Levi, D. Burreddy, 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.
- [9] B. Ramesh, K. K. Suresh, N. Sarkauskas, M. Bayatpour, J. M. Hashmi, H. Subramoni, and D. K. Panda, "Scalable MPI Collectives using SHARP: Large Scale Performance Evaluation on the TACC Frontera System," in *2020 Workshop on Exascale MPI (ExaMPI)*, 2020, pp. 11–20.
- [10] R. L. Graham, D. Burreddy, P. Lui, H. Rosenstock, G. Shainer, G. Bloch, D. Goldenberg, 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.
- [11] M. Bayatpour, S. Chakraborty, H. Subramoni, X. Lu, and D. K. 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*. ACM, 2017, p. 64.
- [12] P. MacArthur, Q. Liu, R. Russell, F. Mizero, M. Veeraraghavan, and J. Dennis, "An integrated tutorial on InfiniBand, Verbs and MPI," *IEEE Communications Surveys Tutorials*, vol. PP, pp. 1–1, 08 2017.
- [13] NVIDIA, "NVIDIA HPC-X," <https://developer.nvidia.com/networking/hpc-x>.
- [14] MVAPICH2-X: Unified MPI+PGAS Communication Runtime over OpenFabrics/Gen2 for Exascale Systems, <http://mvapich.cse.ohio-state.edu/>.
- [15] OSU Micro-benchmarks, <http://mvapich.cse.ohio-state.edu/benchmarks/>.
- [16] M. Bayatpour, J. Maqbool Hashmi, S. Chakraborty, H. Subramoni, P. Kousha, and D. K. Panda, "SALA: Scalable and Adaptive Designs for Large Message Reduction Collectives," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, 2018, pp. 12–23.
- [17] J. M. Hashmi, S. Chakraborty, M. Bayatpour, H. Subramoni, and D. K. Panda, "Design and Characterization of Shared Address Space MPI Collectives on Modern Architectures," in *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2019, pp. 410–419.
- [18] 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.
- [19] 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)*, 2014, pp. 1–10.
- [20] J. Zhang, X. Lu, J. Jose, R. Shi, and D. K. Panda, "Can Inter-VM Shmem Benefit MPI Applications on SR-IOV Based Virtualized Infiniband Clusters?" in *Euro-Par 2014 Parallel Processing*, F. Silva, I. Dutra, and V. Santos Costa, Eds. Cham: Springer International Publishing, 2014, pp. 342–353.
- [21] 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: Association for Computing Machinery, 2013, p. 85–96. [Online]. Available: <https://doi.org/10.1145/2493123.2462903>