Scalable Reduction Collectives with Data Partitioning-based Multi-Leader Design*

Mohammadreza Bayatpour
The Ohio State University
bayatpour.1@osu.edu

Sourav Chakraborty
The Ohio State University
chakraborty.52@osu.edu

Hari Subramoni
The Ohio State University
subramoni.1@osu.edu

Xiaoyi Lu
The Ohio State University
lu.932@osu.edu

Dhabaleswar K. (DK) Panda
The Ohio State University
panda.2@osu.edu

ABSTRACT

Existing designs for MPI_Allreduce do not take advantage of the vast parallelism available in modern multi-/many-core processors like Intel Xeon/Xeon Phis or the increases in communication throughput and recent advances in high-end features seen with modern interconnects like InfiniBand and Omni-Path. In this paper, we propose a high-performance and scalable Data Partitioning-based Multi-Leader (DPML) solution for MPI_Allreduce that can take advantage of the parallelism offered by multi-/many-core architectures in conjunction with the high throughput and high-end features offered by InfiniBand and Omni-Path to significantly enhance the performance of MPI_Allreduce on modern HPC systems. We also model *DPML*-based designs to analyze the communication costs theoretically. Microbenchmark level evaluations show that the proposed *DPML*-based designs are able to deliver up to 3.5 times performance improvement for MPI_Allreduce for multiple HPC systems at scale. At the application-level, up to 35% and 60% improvement is seen in communication for HPCG and miniAMR respectively.

CCS CONCEPTS

•Computing methodologies →Massively parallel algorithms;
•Computer systems organization →Parallel architectures;

KEYWORDS

MPI_Allreduce, Data Partitioning, Multi-Leader, SHArP, Collectives, MPI

ACM Reference format:
Mohammadreza Bayatpour, Sourav Chakraborty, Hari Subramoni, Xiaoyi Lu, and Dhabaleswar K. (DK) Panda. 2017. Scalable Reduction Collectives with Data Partitioning-based Multi-Leader Design. In *Proceedings of SC17, Denver, CO, USA, November 12–17, 2017,* 11 pages.
DOI: 10.1145/3126908.3126954

*This research is supported by NSF grants CCF #1565414, CNS #1513120, ACI #1450440, and ACI #1664137.

SC17, Denver, CO, USA

DOI: 10.1145/3126908.3126954

1 INTRODUCTION

Modern supercomputing systems offer sustained multi peta-flop performance and are allowing scientists to scale their parallel applications to tens of thousands of processors. The massive growth in the size and scale of supercomputing systems over the last decade has been driven by the current trends in multi-/many-core architectures and the availability of commodity, RDMA-enabled, and high-performance interconnects such as InfiniBand [10] (IB) and Omni-Path [5].

The Message Passing Interface (MPI) [19] has been the dominant programming model for high-performance computing (HPC) applications for the last couple of decades. The MPI Standard [21] 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. The MPI_Allreduce collective operation is arguably one of the most popular collective operations in use today and accounts for the lion's share of communication time spent by several applications. A 5-year study of production MPI applications by Rabenseifner [24] showed that 37% of time spent in MPI routines was in MPI_Allreduce. While small message allreduce is popular in traditional scientific MPI applications, many applications in newer fields such as deep learning applications extensively use medium and large message reductions [4].

Existing designs for MPI_Allreduce follow a hierarchical strategy where one or two "leader" processes per compute node are responsible for aggregating data from local processes, performing the reduction operation, and communicating the result of partial reduction operation with leaders in other compute nodes. This strategy does not take advantage of the large amount of parallelism available on modern multi-/many-core architectures. For example, Intel's Knights Landing (KNL) processors can have 64 to 72 cores. A designer of a high-performance MPI library targeting such systems should ideally try to take advantage of the increased concurrency available to parallelize the compute intensive and time-consuming reduction operation by distributing the task to more than one or two leaders as is being done today.

On the other hand, modern HPC interconnects like Omni-Path offer very high message throughputs in the small to medium message range and are capable of sustaining multiple concurrent communication operations from different processes. However, these trends do not hold good as the message size increases. For larger messages,

one cannot improve the communication throughput by pumping in more messages to the network concurrently. On such systems, the use of multi-leader based designs can not only improve the performance of computation operations as described above, but it can also significantly improve the communication throughput if the message sizes selected for inter-node data transfers are done with an understanding of the characteristics of the underlying processor and interconnect architecture. These trends are detailed in Section 3.

Further, modern interconnect vendors such as Mellanox have introduced high-performance hardware-based network solutions like Core-direct [12, 13, 18] and SHArP [8] to enable offloading communication and some computation to the fabric. As we can see, while multiple designs that take advantage of the increased parallelism, increased communication throughput and advanced network features are possible, no single design that has been proposed so far has been able to achieve the best performance for different HPC systems. Such HPC systems have varying combinations of host processor and high-performance interconnect. Furthermore, advanced in-network designs like SHArP and Core-direct only focus on enhancing the inter-node communication. In this era where the number of cores per node is increasing dramatically, there is a need for efficient intra-node schemes that work well in conjunction with optimized inter-node communication methods. Thus, a combination of several different communication algorithms that dynamically choose the best algorithm for different message sizes and system sizes is required to extract best possible performance.

These issues lead us to the following broad challenge — ***Can we design novel communication algorithms for Allreduce primitive that takes advantage of the vast amount of parallelism available in modern multi-/many-core architectures as well as high throughput and high-end features exposed by modern interconnects like InfiniBand and Omni-Path to deliver best performance and scalability?***. This broad challenge leads us to the following concrete design challenges.

- Is it possible to effectively use multiple leaders per compute node to load balance the computation and accelerate communication?
- Can the high throughput characteristics of modern interconnects be used in conjunction with multiple leaders to design high-performance communication operations for large message collective operations?
- What performance trends can be observed in emerging hardware such as KNL and Omni-Path and how can they be used to improve collective communication protocols?
- How can we accelerate high-performance and scalable Allreduce algorithms using the SHArP technology?
- Can we design hybrid communication schemes for Allreduce that can utilize the most optimal algorithms for all message sizes?
- What benefits can be observed at the microbenchmark and application levels through the proposed designs?

1.1 Contributions

In this paper, we take up this challenge and present designs for high-performance and scalable MPI_Allreduce collective by exploiting the capabilities of modern multi-/many-core architectures as well as the high communication throughput and advanced features supported by high-performance interconnects like InfiniBand and Omni-Path. We first study the characteristics of the intra-node and inter-node communication in modern HPC systems. With the observed insights, we propose a Data Partitioning-based Multi-Leader (DPML) design to overcome the drawbacks of traditional single-leader based designs by parallelizing both the computation and the communication. We also propose designs that takes advantage of high throughput and high-end features exposed by modern interconnects like InfiniBand and Omni-Path to further enhance the performance of the inter-node collective communications. We evaluate the efficacy of our designs on three different HPC architectures: 1) Xeon + InfiniBand, 2) Xeon + Omni-Path, and 3) Xeon Phi (also known as Knights Landing — KNL) + Omni-Path. We also present a thorough modeling analysis on the design alternatives for MPI_Allreduce collective operation. We evaluate our designs by comparing microbenchmark and application level numbers against state-of-the art MPI libraries such as Intel MPI and MVAPICH2.

Our evaluations show that the DPML design can improve the latency of medium and large message Allreduce by more than 3.5 times. In addition, we show additional performance improvement of up to 80% for small messages with SHArP. We also evaluate our designs with miniAMR [1] and HPCG [2] and show up to 60% and 35% improvement, respectively. To summarize, the major contributions of this paper are:

- Propose, design, and implement multi-/many-core aware Data Partitioning based Multi Leader designs for MPI_Allreduce
- Design multi-leader based collective operations capable of taking advantage of high throughput and advanced offload features offered by modern network interconnects
- Model and analyze proposed DPML framework theoretically
- Study the benefits of the proposed designs with microbenchmarks and application kernels on three different HPC architectures: 1) Xeon + InfiniBand, 2) Xeon + Omni-Path, and 3) KNL + Omni-Path

To the best of our knowledge, this is the first work to analyze the data partitioning-based multi-leader design for MPI collectives and study its performance characteristics with the high throughput and advanced features offered by modern HPC interconnects like InifniBand and Omni-Path.

2 BACKGROUND

In this section, we provide a brief overview of the relevant algorithms and technologies used in this paper.

2.1 Reduction Collectives in MPI

Reduction collectives such as MPI_Reduce and MPI_Allreduce have been widely used in many scientific applications, like minAMR and HPCG. The global reduce functions perform a global reduce operation (such as sum, max, etc.) across all the members of a group. The reduction operation can be either one of a predefined list of operations, or a user-defined operation. MPI_Reduce combines the elements provided in the input buffer of each process in the group, using the operation op, and returns the combined value in the output buffer of the process with rank root [19]. MPI_Reduce returns

the result of the reduction at one node while MPI_Allreduce returns this result at all nodes. MPI_Allreduce is more communication intensive since it involves more communication steps.

State-of-the-art MPI implementations, such as MPICH2 [11], Open-MPI [6], and MVAPICH2 [17] use optimized algorithms to improve the latency of blocking collective operations. Most MPI libraries implement multi-core aware, shared-memory based algorithms for blocking collective operations. For instance, in MVAPICH2, 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" that contains leaders from all nodes. The processes first do a shared-memory based reduction within each compute node to accumulate the data at the leader process. This is followed by an inter-leader reduction operation based on point-to-point MPI operations. Finally, the leader processes perform a shared-memory broadcast to complete the MPI_Allreduce.

2.2 Overview of SHArP

Scalable Hierarchical Aggregation Protocol (SHArP) can optimize the reduction collectives by performing the reduction operations on the data as it traverses a reduction tree in the network, reducing the volume of data as it goes up the tree, instead of using a CPU-based algorithm where the data traverses the network multiple times. Reduction tree, which is a set of network elements organized as hierarchical data objects, describes available data reduction topologies and collective groups. The leaves represent the data sources, and the interior vertices represent aggregation nodes [8]. This technology enables the manipulation of data while it is being transferred within the network, instead of waiting for the data to reach the CPU to operate on this data.

3 COMMUNICATION CHARACTERISTICS OF MODERN ARCHITECTURES

In order to design high-performance and scalable collectives, the performance characteristics and trends of the underlying hardware must be carefully studied. To understand how the multi-/many-core architecture impacts intra-node and inter-node communication, we compare the throughput of different number of communicating pairs relative to one pair of processes. The "osu_mbw_mr" benchmark from the OSU Microbenchmarks Suite [20] was used to determine the throughput. For intra-node numbers, all communicating pairs were placed on the same node whereas for inter-node, the sender processes from each pair were placed on the same node. A detailed description of the hardware and software used in this evaluation is provided in Section 6.1. From the results shown in Figure 1, we can draw several observations.

Intra-node Communication: As shown in Figure 1(a), the relative throughput is very close to the number of pairs even for large messages. This indicates that the shared memory channel can support many concurrent intra-node communication. This also suggests that shallow hierarchies with small depth and large number of children per parent would be better than deeper hierarchies with small number of children.

Inter-node Communication on InfiniBand: As shown in Figure 1(b), multiple processes per node performing concurrent data-transfer can improve the overall throughput of the communication. The relative throughput is close to the number of communicating processes per node, which indicates that increased parallelism for communication would be beneficial for all message sizes.

Inter-node Communication on Omni-Path: Figure 1(c) and 1(d) show similar trends for small messages with Omni-Path. This indicates that concurrent communication can improve performance for small messages. However, the relative throughput is close to 1 for large messages, which shows that naively increasing parallelism would not help. It also suggests that dividing up a large message into smaller chunks and using multiple processes to transfer the individual chunks would improve the overall performance. The message range where maximum throughput can be achieved is dependent on the CPU architecture as well.

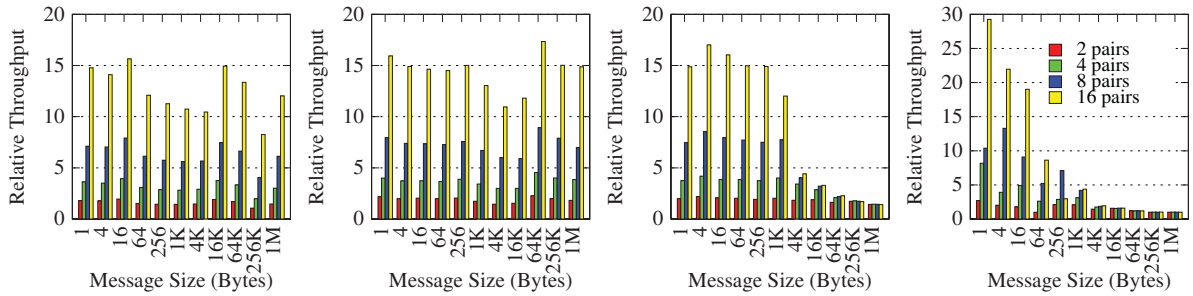
Based on these observations, we can reason about the performance of current reduction algorithms. For example, consider the Recursive Doubling algorithm, which is a flat algorithm where the distance between communicating pairs doubles in each step. This algorithm is optimal in terms of time spent in computation. Thus, it is favored for large messages where computation cost is significant. However, it relies on point-to-point operations for message transfers and incurs the cost of extra-copies. On the other hand, let's consider a hierarchical algorithm where each process concurrently copies the input vector to a "leader process" in the same node through shared memory. From the trends shown in Figure 1(a), it is easy to see that this strategy can offer more throughput due to the large amount of concurrency. However, this strategy requires the leader to perform all $ppn - 1$ (*ppn*: number of processes per node) reductions, which is computationally expensive for large inputs and would place the entire burden on a single process without utilizing the vast parallelism available on modern multi-/many-core architectures. Furthermore, in these approach, only one process per node would participate in the inter-node communication and does not take advantage of the improved throughput obtainable by using concurrent transfers.

4 PROPOSED DESIGNS

In this section, we discuss our proposed designs for implementing high-performance and scalable Allreduce collective. We develop our designs to take advantage of the latest features modern multi-/many-core architectures as well as high-performance interconnects offer to achieve good communication latency and network scalability.

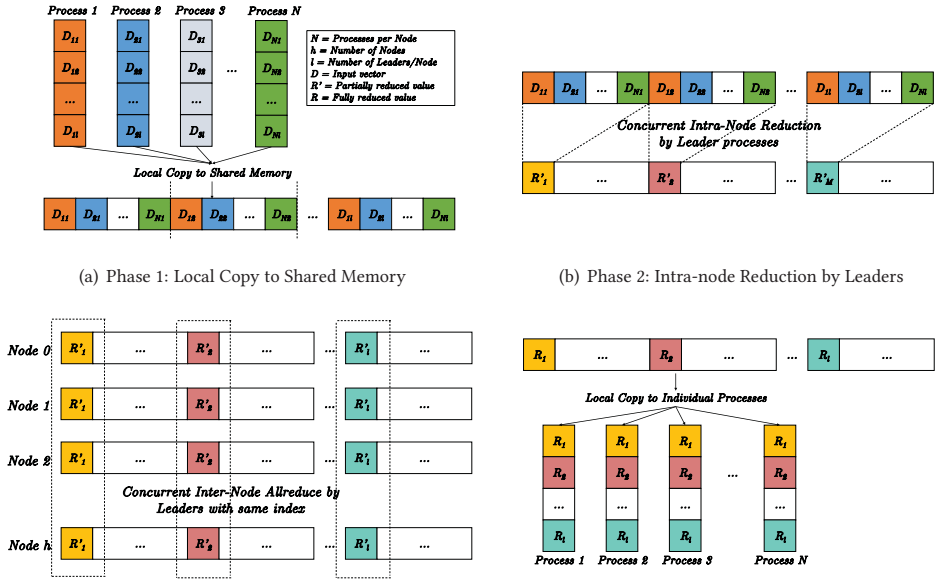
4.1 Data Partitioning-based Multi-Leader Allreduce

As discussed in Section 3, neither hierarchical nor flat designs take advantage of all three features available in current generation hardware: 1) large number of available cores for parallelizing computation, 2) increased concurrency in shared memory, and 3) improved performance for concurrent data transfer through network. Based on this, we design a new scheme called Data Partitioning-based Multi-Leader (DPML) that incorporates all three features. The basic idea behind this algorithm is to designate multiple processes per node as "leaders" which share the computation costs as well as drive concurrent communication. The algorithm consists of four phases, as illustrated in Figure 2.



(a) Shared Memory (KNL) (b) Xeon (Haswell) + IB (EDR - 100Gbps) (c) KNL + Omni-Path (100 Gbps) (d) Xeon (Haswell) + Omni-Path

Figure 1: Relative throughput with different number of pairs of communicating processes (relative to one pair) over various communication channels. (a) shows intra-node and (b-d) shows inter-node communication.



(a) Phase 1: Local Copy to Shared Memory (b) Phase 2: Intra-node Reduction by Leaders

(c) Phase 3: Inter-node Allreduce by Leaders (d) Phase 4: Local Copy to Individual Processes

Figure 2: Different phases in the Data Partitioning-based Multi-Leader (DPML) based Allreduce

Local Copy to Shared Memory: In this phase, each process divides the input data into partitions (one partition per leader) and copies the data into the appropriate locations in a shared memory region. The correct location can be calculated as follows: D_{ij} (jth partition from process with local rank i) is copied to $start\_addr(Leader_j) + i * sizeof(partition)$. This can be viewed as independent gather operations performed by each leader. The goal of this step is to reduce the overall latency by reducing number of intra-node steps and utilize the shared memory in a concurrent fashion.

Intra-node Reduction by Leaders: In this phase, the leader processes perform the reduction on the locally gathered data in parallel. $Leader_j$ is responsible for reducing the partitions $D_{1j}..D_{ppnj}$, requiring $ppn - 1$ reductions. As the data is divided into smaller partitions, each leader shares the load of the computation and can finish the operation much faster compared to single-leader designs. The result of this step is a partially reduced value for each partition.

Inter-node Allreduce by Leaders: In this phase, each leader performs a purely inter-node allreduce of the partially reduced values with leaders of the same rank from other nodes. This allreduce can be implemented via standard allreduce algorithms, such as recursive-doubling or reduce-scatter followed by an allgather. In our evaluation, we use the algorithm dynamically chosen by the MPI library based on message size, system size, and the underlying architecture to achieve the best performance. After this step is completed, each leader process on each node is left with the fully reduced value for the corresponding partition.

Using multiple leaders in this phase is beneficial from two aspects. First, it reduces the latency through concurrent data transfers. Second, it reduces the message size exchanged by each communicating process. As shown earlier, this can improve the performance as large messages do not significantly benefit from multiple concurrent transfers.

Local Copy to Individual Processes: In this phase, the fully reduced value for each partition is copied back to the local processes and combined to obtain the final reduced value. This step is equivalent to concurrent broadcast operations by each leader. Direct shared memory copy is used to reduce the number of steps from $\lceil lg\ ppn \rceil$ to number of leaders. The goal here is the same as that of phase 1.

4.2 Taking Advantage of High Message Rate: Pipelined Data Transfer

The Omni-path network supports very high message rate for small messages [5]. Consequently, the relative throughput obtained for different message sizes for the Omni-Path (shown in Figure 1(c)) shows three different "zones", as described below.

- Zone *A*: In the leftmost zone with the smallest messages, the throughput is largely independent of the message size and scales almost linearly with concurrency. In this range, maximum throughput is limited by the message rate.
- Zone *C*: Similarly in the rightmost zone with the largest messages, the throughput remains unchanged with increase in message size. However in this zone, the throughput is not increased by increasing concurrency. In this range, the throughput is limited by the bandwidth.
- Zone *B*: For medium sized messages, the throughput depends both on concurrency and message size and is not directly limited by either message rate or bandwidth. This zone is a transition zone between Zone *A* and Zone *C*.

These three zones shape the improvement that can be obtained from DPML for any given message size. If the original input message size falls in Zone *B*, the message size used in the inter-node allreduce (Phase 3) can fall inside or closer to Zone *A* depending on the number of leaders. Consequently, the throughput of this step is improved further. For example, a 16KB input can be divided into 1KB sized partitions by 16 leaders, thus benefiting from the increased parallelism. However, the number of leaders in a node can be at most *ppn*. In practice, it is chosen to be smaller than *ppn* to avoid dividing the input into very small chunks. Consequently for very large messages, even with many leaders, the size of message after partitioning can remain in Zone *C*. For example with 16 leaders, a 1MB input message would be divided into 64KB partitions, where the benefit of parallel data transfer is limited. Thus, we need additional designs to extract the best performance of such very large messages.

We achieve this by further partitioning the partially reduced results inside each leader after Phase 2. The required number of such smaller partitions is proportional to the message size and inversely related to the number of leaders. These sub-partitions are then individually reduced with an inter-node allreduce step similar to Phase 3. To get the best overlap, non-blocking allreduce calls are used followed by a waitall. We refer to this design as **DPML-Pipelined**.

4.3 Taking Advantage of Advanced Network Offload: SHArP

As illustrated in Figure 1(b), relative throughput with InfiniBand does not decrease significantly for large messages. Thus, the *DPML-Pipelined* design is not expected to be beneficial. However, for small messages, we can take advantage of the SHArP protocol to improve the performance. As described in Section 2.2, SHArP can be used to offload computation and communication to the InfiniBand switch. We discuss two designs based on this protocol and the trade-offs involved.

Node-level Leader design: Performing reduction using SHArP requires one or more processes per node to call SHArP-specific functions to create SHArP communicators and communicate with the switch. A natural choice would be to use DPML leader processes for this purpose. However, our evaluations show that SHArP can support only a small number of concurrent operations and SHArP communicators. Thus, using all DPML leaders for SHArP communication would severely limit the scalability of the operation. To avoid this, only one leader per node is used to implement the SHArP based reduction. In this design, the leader process needs to gather the data from the processes, transmit them to the switch, and broadcast the result back to the local processes. However, unlike DPML with single leader where the leader is responsible for performing $ppn - 1$ reductions, SHArP distributes the computation among the relevant IB switches.

However, this design suffers from some performance issues. For instance, in multi-socket nodes, the communication latency across sockets is significantly higher than intra-socket communications. Thus, a single leader causes $ppn/2$ processes to incur the extra-communication cost. Both the gather and broadcast phases suffer from this bottleneck.

Socket-level Leader design: Based on the above-mentioned limitations, we propose to use one leader per socket. For most current generation Xeon clusters, this translates to two leaders per node. Each leader is responsible for local processes running on the same socket as itself. This design avoids the expensive inter-socket communication cost while keeping the total number of processes involved in SHArP communication small. This design brings an additional benefit for multi-HCA machines. The leader process selection is HCA-aware, i.e. each leader communicates through its closet HCA to avoid incurring the QPI latency cost.

5 MODELING THE COST OF ALLREDUCE OPERATIONS

To estimate the cost of the operation, we extend the cost model used by Rabenseifner in [25] by treating the cost of shared-memory copies differently from inter-node transfers. Table 1 describes the notations used in the cost model.

5.1 Cost Model

We assume full-duplex mode of communication, i.e. messages going in one-direction do not affect messages in the opposite direction. With this model, cost of Allreduce with power-of-two processes using purely inter-node recursive doubling is

Table 1: Notations used in the Cost Model

Symbol	Description
p	Number of MPI Processes
h	Number of Nodes
l	Number of leader processes per node
n	Size of the input vector in bytes
a	Startup time per inter-node message
b	Transfer time per byte for inter-node messages
a'	Startup time per shared-memory copy
b'	Transfer time per byte for shared-memory copy
c	Computation cost of one reduction operation per byte
k	Number of sub-partitions used in *DPML-Pipeline*

$$T_{r.d} = \lceil lg\ p \rceil (a + nb + nc) \quad (1)$$

5.2 Phases of the Algorithm

The algorithm consists of four phases. Briefly, they are:

(1) **Copy to Local Leaders:** In this phase, all processes on the same node partition the input vector into l parts and copy n/l bytes to the shared memory of each leader processes. Cost of this phase:

$$T_{copy} = l * \left(a' + b'\left(\frac{n}{l}\right)\right) \quad (2)$$

(2) **Intra-node Reduction by Leaders:** In this phase, each leader process performs the reduction operation on the locally gathered data. Each leader is responsible for $\left(\frac{p}{h*l}\right)$ partitions of data. Recall that $p-1$ reduction operations are required for p processes. Hence, the cost is:

$$T_{comp} = \left(\frac{p}{h*l} - 1\right) * n * c \quad (3)$$

(3) **Inter-node Allreduce Operation by Leaders:** After the local reduction, each leader process is left with $\frac{n}{l}$ bytes of data. This data is only partially reduced and must be combined with the corresponding values held by the leader processes in other nodes. To achieve this, each leader performs a purely inter-node allreduce to the corresponding leader in other nodes. Each allreduce operation involves h processes and l such collectives are performed in parallel. Assuming recursive-doubling algorithm is used for the inter-node allreduce, cost of this phase:

$$T_{comm} = \lceil lg\ h \rceil \left(a + \frac{n*b}{l} + \frac{n*c}{l}\right) \quad (4)$$

With *DPML-Pipelined* , the amount of data transferred and computation remain unchanged while number of messages increase by a factor of k. Hence, cost with pipelining:

$$\begin{aligned} T_{comm_k} &= k * \lceil lg\ h \rceil \left(a + \frac{n*b}{l*k} + \frac{n*c}{l*k}\right) \\ &= \lceil lg\ h \rceil \left(a*k + \frac{n*b}{l} + \frac{n*c}{l}\right) \end{aligned} \quad (5)$$

(4) **Local Copy to Individual Processes:** In this phase, each process copies n/l bytes of the fully reduced vector into their receive buffer. This is equivalent to a broadcast of the input vector to all local processes. Cost of this phase:

$$T_{bcast} = l * \left(a' + b'\left(\frac{n}{l}\right)\right) \quad (6)$$

Combining all these phases, we can compute the total cost of the Allreduce:

$$\begin{aligned} T_{allreduce} &= T_{copy} + T_{comp} + T_{comm} + T_{bcast} \\ &= 2 * l * \left(a' + b'\left(\frac{n}{l}\right)\right) + \left(\frac{p}{h*l} - 1\right) * n * c \\ &\quad + \lceil lg\ h \rceil \left(a + \frac{n*b}{l} + \frac{n*c}{l}\right) \end{aligned} \quad (7)$$

5.3 Discussion

Since direct shared memory copies are much faster than inter-node message passing, $a' \ll a$ and $b' \ll b$. Thus, the performance of the operation is dominated by phase 2 and phase 3. Compared to pure recursive doubling, number of communication steps is reduced from $\lceil lg\ p \rceil$ to $\lceil lg\ h \rceil$, a significant improvement for multi/many-core systems. The size of each message is also reduced by a factor of number of leaders (l). As we have shown in Section 3, concurrently sending n/l byte messages from l processes results in better throughput compared to one process sending one n byte message. We can also see that the compute and communication costs become more significant for medium and large messages, where $n \gg 1$. Thus, it is expected that increasing the number of leaders would reduce the latency of medium and large message allreduce.

6 PERFORMANCE EVALUATION

In this section, we describe the experimental setup, provide the results of our experiments, and give an in-depth analysis of these results.

6.1 Experimental Setup

We used the open-source MVAPICH2 MPI library [22] for implementing our designs. To evaluate the proposed designs, we compare them against the state-of-the-art algorithms being used in MVAPICH2-2.2 (referred to as "MVAPICH2") and Intel MPI 2017.1.132 (referred to as "Intel MPI"). To evaluate our designs on various different CPU and interconnect technologies used in modern HPC systems, we use four different clusters equipped with different hardware combinations, as shown in Figure 3. We exclude the combination Knights Landing + InfiniBand due to lack of availability of large clusters with this combination. All experiments are performed in full subscription mode unless specified otherwise. The numbers reported are averages of a minimum of five runs. For microbenchmark level evaluations, at least 1,000 iterations are used.

Cluster A (Xeon + IB w/ SHArP): Cluster A contains 40 compute nodes equipped with the Intel Haswell series of processors using Xeon dual socket, 14-core processors (1,120 cores in total) operating at 2.40 GHz with 128 GB RAM. Each node is equipped with Mellanox MT4115 EDR ConnectX-4 HCAs (100 Gbps data rate) with PCI-Ex Gen3 interfaces. The operating system used is CentOS Linux release 7.2.1511, with kernel version 3.10.0.2827.10.1.el7 and Mellanox OFED version 3.4-2. This cluster was used for all SHArP based evaluations.

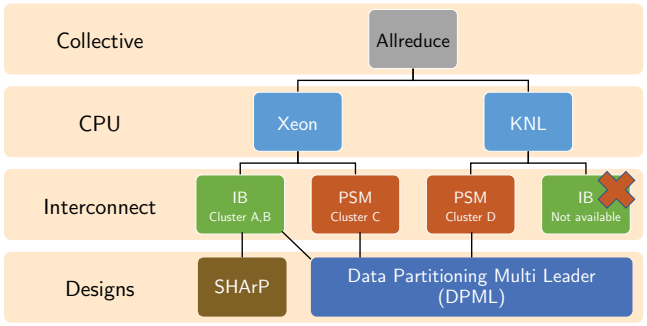


Figure 3: Evaluation space for proposed designs

Cluster B (Xeon + IB w/o SHArP: Cluster B contains 648 Dell PowerEdge C6320 two-socket servers with Intel Xeon E5-2680 v4 (Broadwell, 14 cores, 2.40GHz) processors and 128GB memory. The nodes are connected using Mellanox MT4115 EDR ConnectX-4 HCAs (100 Gbps data rate) with PCI-Ex Gen3 interfaces.

Cluster C (Xeon + Omni-Path): Cluster C has 752 compute nodes equipped with dual socket Intel Haswell 14-core CPUs running at 2.3GHz and 128 GB DDR4 memory. The nodes are connected with Intel Omni-Path Host Fabric Interface adapters (100 Series) capable of (100 Gbps data rate) over PCI-Ex Gen3 interface.

Cluster D (KNL + Omni-Path): Cluster D includes 508 Intel Xeon Phi 7250 KNL compute nodes (68 cores per node, 4 hardware threads per core, 1.4 GHz) housed in 9 racks. Each KNL is a self-hosted node with 96 GB of DDR4 RAM, running CentOS 7 and supporting a KNL-compatible software stack. The nodes include 112GB of local solid state drive (SSD). The interconnect is a 100Gb/sec OmniPath network: a fat tree topology of eight core-switches and 320 leaf switches with 5/4 oversubscription. The KNL nodes were configured in "cache" mode where the 16 GB High-Bandwidth memory (MCDRAM) is used as a direct-mapped L3 cache. The maximum number of processes used on KNL is set to 64 to avoid oversubscribing the physical cores.

6.2 Impact of Number of Leaders on Performance

As described in Section 4.1, the number of leaders in DPML can significantly affect the time taken by the individual phases and the overall performance of the collective. To study the impact of number of leaders, we compare the latency of MPI_Allreduce with different number of leaders on different hardware platforms. To preserve any impact on the compute time, SHArP is not used in this evaluation. MPI_SUM and MPI_FLOAT are used as the operation and the datatype respectively. Figures 4 and 5 show the results on Xeon + InfiniBand whereas Figures 6 and 7 illustrate the performance on Xeon + Omni-Path and KNL + Omni-Path architectures, respectively.

As we can see from the results, increasing the number of leaders for small messages (smaller than 1KB) does not improve performance and sometimes causes slight degradation. This is expected as for such small messages the benefit from parallelizing the computation is tiny. However for medium and large messages the benefit of reduced compute time and increased throughput due to concurrent communication becomes more significant. Consequently, larger number of leaders lead to better performance for these range of messages. For instance, with 512KB message size, Cluster B (Xeon+IB) and Cluster C (Xeon+Omni-Path) respectively show 4.9 times and 4.3 times lower latency with 16 leaders compared to single leader per node.

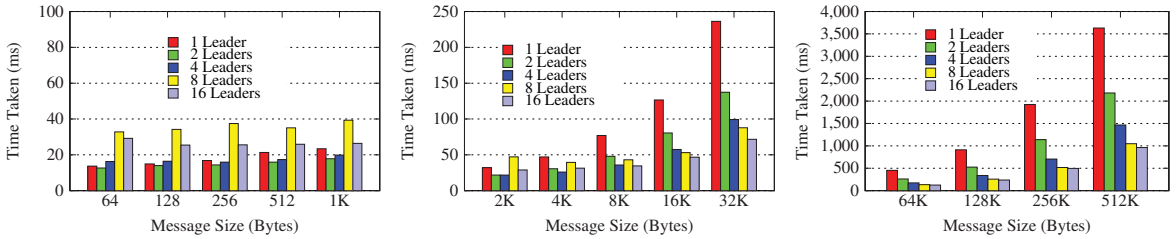
6.3 Impact of SHArP on Communication Performance

We described two SHArP based designs in Section 4.3: Node-level-Leader, and Socket-level-Leader. Note that the two designs are equivalent when there is only a single process per node. However, with multiple processes per node, the socket-leader design take the intra-node communication characteristics into account and is expected to provide better performance. To verify this, we evaluate these two designs with different number of processes with 16 nodes. Figure 8 shows the results from this experiment. In a single process per node scenario, the SHArP-based design performs up to 2.5 times faster than the default Host-based design. However, the benefit is negligible when the message size increases to 2KB and the host-based design outperforms SHArP with 4KB messages. This trend is observed with larger number of processes per node as well and shows that SHArP is more effective at small message sizes. With 4 processes per node, the node-leader and the socket-leader designs perform up to 80% and 100% faster than the default host-based scheme. At full subscription with 28 processes per node, we observe benefits of up to 46% and 73% respectively over the default. This illustrates that while the node-based design is more suitable for small number of processes per node, the socket-based design performs better with larger number of processes per node.

6.4 Comparison with State-of-the-art MPI Libraries

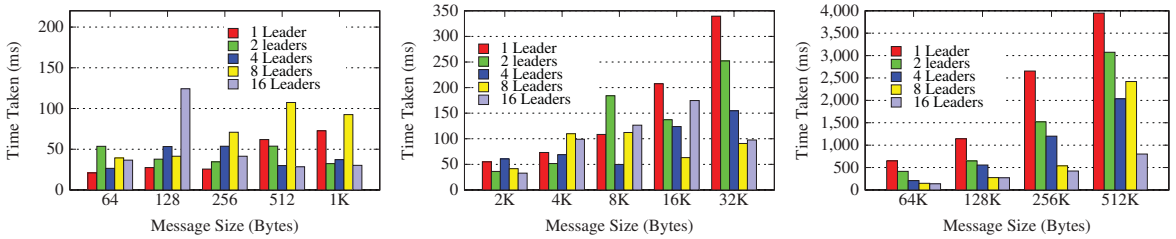
While several general observations can be drawn from the experiments described in Section 3, the optimal number of leaders depend on the message size, the number of processes, and the underlying hardware. For example, for 8KB messages, the best performance is obtained using 4 leaders on both Clusters A and B but with 16 leaders on Clusters C and D. While using 16 leaders is almost always the best choice for large messages that fall in the Zone *C* (as described in Section 3), for medium messages that fall in the transition Zone *B* it is more difficult to predict the optimal configuration. Based on this, we performed empirical evaluation of different configurations on the four clusters and chose the best configuration for each message size.

We compared the performance of MPI_Allreduce with our design against two state-of-the art MPI libraries - Intel MPI and MVAPICH2. These MPI libraries include the capability to choose the appropriate algorithm or configuration based on various factors like message size, number of processes per node, CPU and interconnect being used etc. Thus, we compared the best configuration (number of leaders) of the proposed algorithm against the best algorithm chosen by the MPI library. The results for the different clusters are shown in Figure 9. Intel MPI was not available on Custer A and B and hence was not included for these two. From Figures 9(a) and 9(b), we can see that the proposed design outperforms the default MVAPICH2 by up to 3.59 times on Cluster A and by up to 3.08 times on Cluster B. Similarly for Cluster C and D, the proposed design performs up



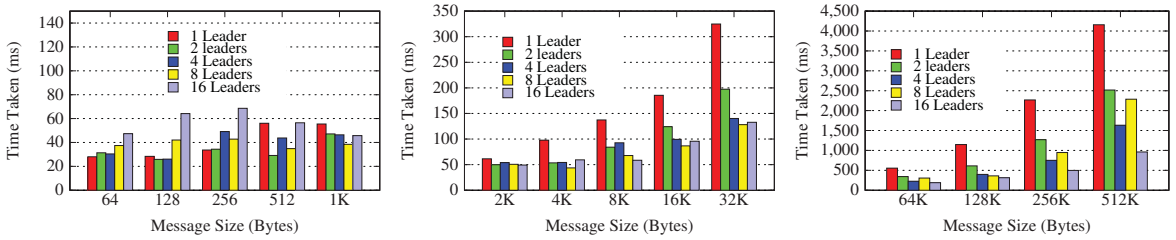
(a) Small Messages (b) Medium Messages (c) Large Messages

Figure 4: Impact of different number of leaders on MPI_Allreduce at 448 processes on Cluster A (16 nodes, 28 ppn)



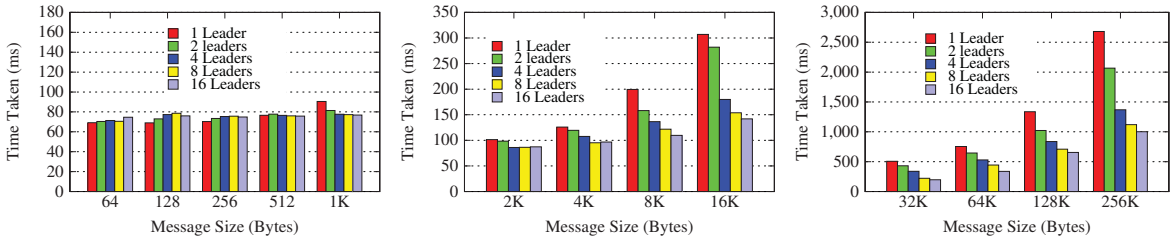
(a) Small Messages (b) Medium Messages (c) Large Messages

Figure 5: Impact of different number of leaders on MPI_Allreduce at 1,792 processes on Cluster B (64 nodes, 28 ppn)



(a) Small Messages (b) Medium Messages (c) Large Messages

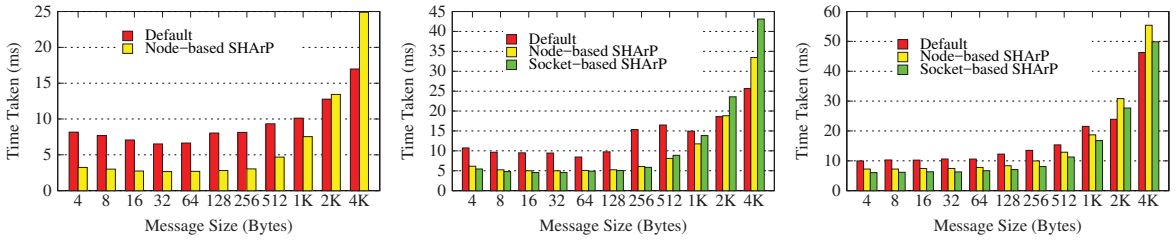
Figure 6: Impact of different number of leaders on MPI_Allreduce at 1,792 processes on Cluster C (64 nodes, 28 ppn)



(a) Small Messages (b) Medium Messages (c) Large Messages

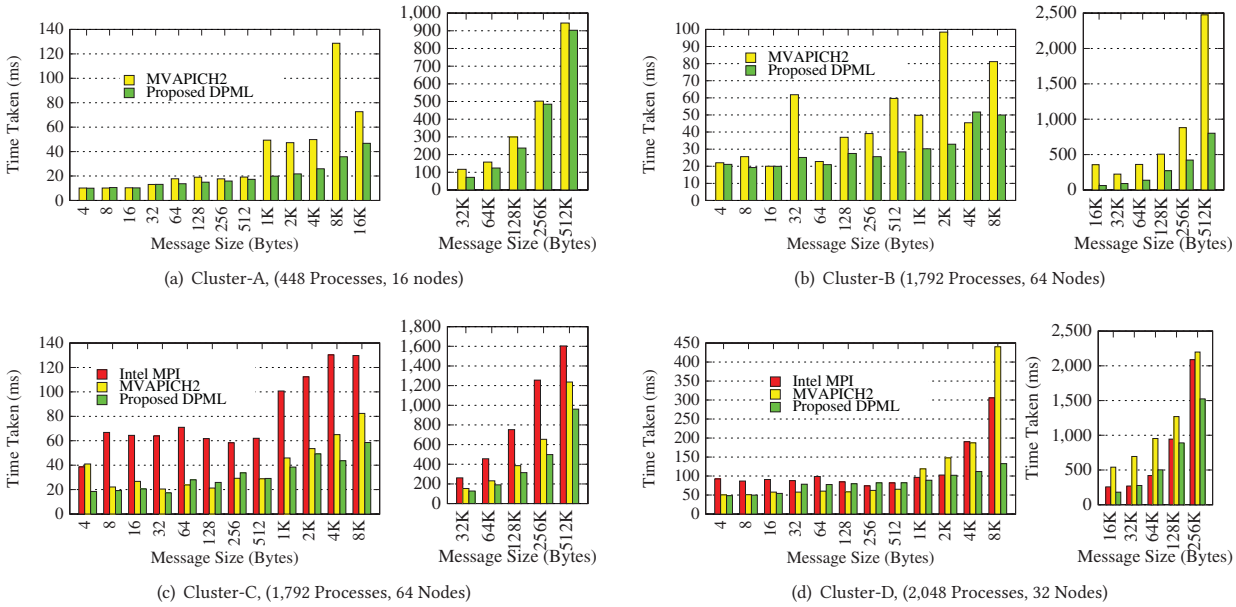
Figure 7: Impact of different number of leaders on MPI_Allreduce at 1,024 processes on Cluster D (32 nodes, 32 ppn)

to 2.98 and 2.3 times compared to Intel MPI and up to 1.4 times and 3.31 times compared to MVAPICH2. Figure 10(a) compares the performance of the proposed design at scale, with 10,240 processes on 160 nodes on Cluster D. Our proposed DPML designs shows very good scalability and outperforms MVAPICH2 and Intel MPI by up to 207% and 48% respectively.



(a) 1 Process Per Node (b) 4 Processes Per Node (c) 28 Processes Per Node

Figure 8: Performance comparison of different SHArP-based designs with 16 nodes on Cluster A



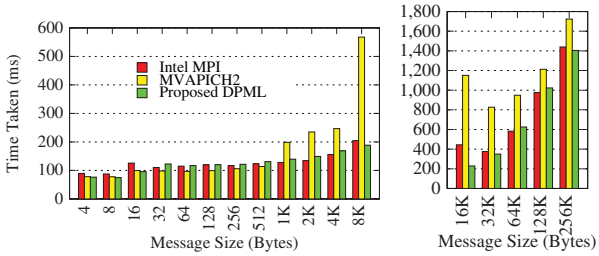
(a) Cluster-A, (448 Processes, 16 nodes)

(b) Cluster-B (1,792 Processes, 64 Nodes)

(c) Cluster-C, (1,792 Processes, 64 Nodes)

(d) Cluster-D, (2,048 Processes, 32 Nodes)

Figure 9: Performance comparison of MPI_Allreduce with proposed DPML design and state-of-the-art MPI libraries



(a) Cluster-D, (10,240 Processes, 160 Nodes)

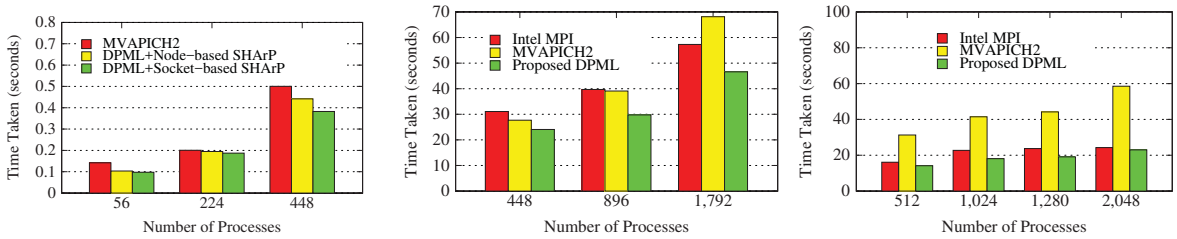
Figure 10: Latency comparison of MPI_Allreduce at large scale with the proposed DPML design

6.5 Performance of HPCG

High-Performance Conjugate Gradient (HPCG) [2] is a popular benchmark used to compare the HPC systems. As HPCG performs small message allreduce, we expect to see benefit with SHArP. Thus, we compare the DDOT timing with SHArP-based designs with 56, 224, and 448 processes with 28 processes per node. As shown in Figure 11(a), the results show the weak scaling nature of HPCG as average execution time increases going from 56 to 448 processes. Compared to the default host-based scheme, the node-leader and socket-leader based designs yield up to improvement of 35% in average execution time in the case of 56 processes and up to 10% improvement in the case of 224 processes. The loss in percentage improvement is expected owing to the fact the *count* argument for MPI_Allreduce going from 56 to 448 processes remains the same and hence, the percentage time spent in MPI_Allreduce by HPCG also correspondingly decreases. Please note that only Mellanox-based systems (like cluster-A) support SHArP. Thus, we only show HPCG numbers for cluster-A.

6.6 Performance of miniAMR

3D stencil calculation with Adaptive Mesh Refinement (miniAMR) [1] is a popular benchmark that mimics commonly found workloads



(a) DDOT Timing Variable of HPCG on Cluster A (28 ppn)

(b) MiniAMR on Cluster C (28 ppn)

(c) MiniAMR on Cluster D (32 ppn)

Figure 11: Performance comparison of applications (MiniAMR and HPCG) with proposed DPML and state-of-the-art MPI libraries

and communication patterns in AMR applications. In this benchmark, a large number of MPI_Allreduce calls take place and the input size of some of these calls can be increased by increasing the number of the processes in the job and also by increasing the frequency of Mesh Refinement. To evaluate the Allreduce designs, we have looked at the average of *Overall Mesh Refinement time* across all the processes in the job. In our evaluations, we set the frequency of Mesh Refinement to one thousand which makes this operation to take more than 98% of overall application time. As we can see in Figure 11, the proposed design shows up to 40% benefit over MVAPICH2 and up to 20% over Intel MPI in Cluster C. On Cluster D, the proposed design provides performance improvements of up to 20% for Intel MPI and up to 60% for MVAPICH2. As miniAMR performs allreduce with relatively large messages, we see good benefit with DPML as expected. Due to technical issues we were not able to get a full set of numbers on cluster-A and cluster-B for miniAMR.

7 RELATED WORK

There has been a number of studies for improving the performance of MPI collective operations. The related studies can be broadly summarized into three categories.

Modeling and redesigning collective algorithms: Many research works have been done to model and redesign collective algorithms in the literature. Rabenseifner analyzed the cost of reduce and allreduce operations in [25] and proposed a new algorithm based on this analysis. In this work, we extend the model used by differentiating the cost of shared memory transfers and propose new algorithms geared towards modern hardware architectures. Authors in [23] attempted to improve collective communication performance by analyzing and extending the point-to-point communication models, such as Hockney [9], LogP/LogGP [3, 7], and PLogP [14] to collective operations. Based on these analysis, they also introduced an optimized tree-based broadcast algorithm, split-binary. Authors in [26] presented their work on improving the performance of collective operations in MPICH. For each collective operation, they selected multiple algorithms depending on the message size and number of processes. Compared to the related work in this group, our models focus on analyzing the performance characteristics of reduction costs at CPU and InfiniBand Switch sites. Based on these analysis, we propose new data partitioning based multi-leader design with hardware-based SHArP offloading mechanism.

Exploiting hardware offloading mechanisms: Bloch et al. described the Scalable Hierarchical Aggregation Protocol (SHArP) technology in [8], which is designed to offload collective operation processing to the network. Kandalla et al. [13] exploits the CoreDirect collective offload feature to offload lists of communication operations to the network interface. Such an interface is used to develop non-blocking collective operations.

Shared memory-based collectives: Li et al. [15, 16] investigated the design and optimizations of MPI collectives for NUMA nodes. They developed performance models for collective communication using shared memory and several algorithms for various collectives. Zhang et al. [27, 28] take advantage of shared memory in order to efficiently handle the communications between virtual machines which are in a given node. Their proposed design enables MPI applications running in virtualized mode to have efficient intra-node communication on SR-IOV enabled InfiniBand clusters.

8 CONCLUSION AND FUTURE WORK

The performance of MPI collectives such as MPI_Allreduce plays an important role in many scientific applications. Existing designs for MPI_Allreduce operation have significant drawbacks for communication on modern multi-/many-core architectures, as it can not fully exploit data parallelism in collectives and the recent advancements in communication performance and network offload support introduced in high-performance interconnects like InfiniBand and Omni-Path. To address these challenges, this paper proposed, modeled, and analyzed a multi-/many-core aware Data Partitioning based Multi Leader design for MPI_Allreduce. We also proposed additional designs to take advantage of high throughput and advanced offload features such as SHArP offered by modern network interconnects. We also studied the benefits of the proposed designs with microbenchmarks and application kernels on three different HPC architectures: 1) Xeon + InfiniBand, 2) Xeon + Omni-Path, and 3) KNL + Omni-Path. Microbenchmark level evaluation showed that the proposed *DPML*-based designs were able to deliver up to 3.5 times performance improvement for MPI_Allreduce. At the application-level, up to 35% and 60% improvements were seen for HPCG and miniAMR, respectively. In the future, we plan to explore the designs for other collectives with SHArP and we also plan to investigate the designs for non-blocking collectives with SHArP. Also, we would like to explore the possibilities of exploiting DPML approach for other blocking and non-blocking collectives as well.

REFERENCES

[1] Mantevo Applications . http://mantevo.org/packages/

[2] 2015. The High Performance Conjugate Gradients Benchmark. (2015). http://hpcg-benchmark.org/

[3] Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, and Chris Scheiman. 1995. 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 (SPAA '95)*. ACM, New York, NY, USA, 95–105. DOI: http://dx.doi.org/10.1145/215399.215427

[4] Ammar Ahmad Awan, K Hamidouche, A Venkatesh, and DK Panda. 2016. Efficient Large Message Broadcast using NCCL and CUDA-Aware MPI for Deep Learning. In *Proceedings of the 23rd European MPI Users' Group Meeting*. ACM, 15–22.

[5] Mark S Birrittella, Mark Debbage, Ram Huggahalli, James Kunz, Tom Lovett, Todd Rimmer, Keith D Underwood, and Robert C Zak. 2015. Intel® Omni-path Architecture: Enabling Scalable, High Performance Fabrics. In *High-Performance Interconnects (HOTI), 2015 IEEE 23rd Annual Symposium on*. IEEE, 1–9.

[6] Open MPI : Open Source High Performance Computing. 2017. http://www.open-mpi.org. (2017).

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

[8] Richard L. Graham, Devendar Bureddy, Pak Lui, Hal Rosenstock, Gilad Shainer, Gil Bloch, Dror Goldenerg, Mike Dubman, Sasha Kotchubievsky, Vladimir Koushnir, Lion Levi, Alex Margolin, Tamir Ronen, Alexander Shpiner, Oded Wertheim, and Eitan Zahavi. 2016. Scalable Hierarchical Aggregation Protocol (SHArP): A Hardware Architecture for Efficient Data Reduction. In *Proceedings of the First Workshop on Optimization of Communication in HPC (COM-HPC '16)*. IEEE Press, Piscataway, NJ, USA, 1–10. DOI: http://dx.doi.org/10.1109/COM-HPC.2016.6

[9] Roger W. Hockney. 1994. The Communication Challenge for MPP: Intel Paragon and Meiko CS-2. *Parallel Comput.* 20, 3 (March 1994), 389–398. DOI: http://dx.doi.org/10.1016/S0167-8191(06)80021-9

[10] IB 2017. InfiniBand Trade Association. http://www.infinibandta.com. (2017).

[11] J. Liu and W. Jiang and P. Wyckoff and D. K. Panda and D. Ashton and D. Buntinas and B. Gropp and B. Tooney. 2004. High Performance Implementation of MPICH2 over InfiniBand with RDMA Support. In *IPDPS*.

[12] Krishna Kandalla, Hari Subramoni, Karen Tomko, Dmitry Pekurovsky, Sayantan Sur, and Dhabaleswar K. Panda. 2011. High-Performance and Scalable Non-Blocking All-to-All with Collective Offload on InfiniBand Clusters: A Study with Parallel 3D FFT. *Comput. Sci.* 26 (June 2011), 237–246. Issue 3-4. DOI: http://dx.doi.org/10.1007/s00450-011-0170-4

[13] K. Kandalla, U. Yang, J. Keasler, T. Kolev, A. Moody, H. Subramoni, K. Tomko, J. Vienne, and D. K. Panda. 2012. 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)*.

[14] Thilo Kielmann, Henri E. Bal, and Kees Verstoep. 2000. Fast Measurement of LogP Parameters for Message Passing Platforms. In *Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing (IPDPS '00)*. Springer-Verlag, London, UK, UK, 1176–1183.

[15] Shigang Li, Torsten Hoefler, Chungjin Hu, and Marc Snir. 2014. Improved MPI Collectives for MPI Processes in Shared Address Spaces. *Cluster Computing* 17, 4 (Dec. 2014), 1139–1155. DOI: http://dx.doi.org/10.1007/s10586-014-0361-4

[16] Shigang Li, Torsten Hoefler, and Marc Snir. 2013. NUMA-aware Shared-memory Collective Communication for MPI. In *Proceedings of the 22Nd International Symposium on High-performance Parallel and Distributed Computing (HPDC '13)*. ACM, New York, NY, USA, 85–96. DOI: http://dx.doi.org/10.1145/2462902.2462903

[17] J. Liu, W. Jiang, P. Wyckoff, D. K. Panda, D. Ashton, D. Buntinas, W. Gropp, and B. Toonen. 2004. Design and Implementation of MPICH2 over InfiniBand with RDMA Support. In *Proceedings of Int'l Parallel and Distributed Processing Symposium (IPDPS '04)*.

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

[19] Message Passing Interface Forum 1994. *MPI: A Message-Passing Interface Standard*. Message Passing Interface Forum.

[20] OSU Micro-Benchmarks. 2017. http://mvapich.cse.ohio-state.edu/benchmarks. (2017).

[21] MPI3 2012. MPI-3 Standard Document. http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf. (2012).

[22] MVAPICH2 2017. MVAPICH: MPI over InfiniBand, Omni-Path, Ethernet/iWARP, and RoCE. http://mvapich.cse.ohio-state.edu/. (2017).

[23] J. Pjesivac-Grbovic, T. Angskun, G. Bosilca, G. E. Fagg, E. Gabriel, and J. J. Dongarra. 2005. Performance Analysis of MPI Collective Operations. In *19th IEEE International Parallel and Distributed Processing Symposium*. 8 pp.–. DOI: http://dx.doi.org/10.1109/IPDPS.2005.335

[24] Rolf Rabenseifner. 1999. Automatic MPI Counter Profiling of all Users: First Results on a CRAY T3E 900-512. In *Proceedings of the message passing interface developerfis and userfis conference*, Vol. 1999. 77–85.

[25] Rolf Rabenseifner. 2004. Optimization of Collective Reduction Operations. In *International Conference on Computational Science*. Springer, 1–9.

[26] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. 2005. Optimization of Collective Communication Operations in MPICH. *Int. J. High Perform. Comput. Appl.* 19, 1 (Feb. 2005), 49–66. DOI: http://dx.doi.org/10.1177/1094342005051521

[27] J. Zhang, X. Lu, J. Jose, M. Li, R. Shi, and D. K. D. K. Panda. 2014. High Performance MPI Library over SR-IOV enabled InfiniBand Clusters. In *2014 21st International Conference on High Performance Computing (HiPC)*. 1–10. DOI: http://dx.doi.org/10.1109/HiPC.2014.7116876

[28] Jie Zhang, Xiaoyi Lu, Jithin Jose, Rong Shi, and Dhabaleswar K. (DK) Panda. 2014. *Can Inter-VM Shmem Benefit MPI Applications on SR-IOV Based Virtualized Infiniband Clusters?* Springer International Publishing, Cham, 342–353. DOI: http://dx.doi.org/10.1007/978-3-319-09873-9_29