2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS) | 978-1-6654-4066-0/20/$31.00 ©2021 IEEE | DOI: 10.1109/IPDPS49936.2021.00053

# Designing High-Performance MPI Libraries with On-the-fly Compression for Modern GPU Clusters*

Q. Zhou, C. Chu, N. S. Kumar, P. Kousha, S. M. Ghazimirsaeed, H. Subramoni and D. K. Panda
*Department of Computer Science and Engineering, The Ohio State University*
{zhou.2595, chu.368, senthilkumar.16, kousha.2, ghazimirsaeed.3, subramoni.1, panda.2}@osu.edu

***Abstract*—While the memory bandwidth of accelerators such as GPU has significantly improved over the last decade, the commodity networks such as Ethernet and InfiniBand are lagging in terms of raw throughput creating. Although there are significant research efforts on improving the large message data transfers for GPU-resident data, the inter-node communication remains the major performance bottleneck due to the data explosion created by the emerging High-Performance Computing (HPC) applications. On the other hand, the recent developments in GPU-based compression algorithms exemplify the potential of using high-performance message compression techniques to reduce the volume of data transferred thereby reducing the load on an already overloaded inter-node communication fabric. The existing GPU-based compression schemes are not designed for "on-the-fly" execution and lead to severe performance degradation when integrated into the communication libraries.**

**In this paper, we take up this challenge and redesign the MVAPICH2 MPI library to enable high-performance, on-the-fly message compression for modern, dense GPU clusters. We also enhance existing implementations of lossless and lossy compression algorithms, MPC and ZFP, to provide high-performance, on-the-fly message compression and decompression. We demonstrate that our proposed designs can offer significant benefits at the microbenchmark and application-levels. The proposed design is able to provide up to 19% and 37% improvement in the GPU computing flops of AWP-ODC with the enhanced MPC-OPT and ZFP-OPT schemes, respectively. Moreover, we gain up to 1.56x improvement in Dask throughput. To the best of our knowledge, this is the first work that leverages the GPU-based compression techniques to significantly improve the GPU communication performance for various MPI primitives, MPI-based data science, and HPC applications.**

***Index Terms*—GPU, Compression, GPU-Aware MPI, HPC, Dask**

## I. Introduction

Message Passing Interface (MPI) [1] is a popular parallel programming model for developing parallel scientific applications. HPC and data science applications usually use communication libraries such as MPI to efficiently parallelize the code and achieve high throughput. Although the well-designed GPU-aware MPI libraries can efficiently leverage massive parallelism and high-bandwidth memory on modern GPU architectures to progress a large amount of data, the data movement is still the performance bottleneck on these systems. Over the last decade, the researchers have significantly optimized data transfers in MPI for GPU-resident data [2]–[5]. However, it has reached the limit since the interconnect and network components have not kept up the same pace with processing power and communication bandwidth provided by the modern GPU architectures. The latest NVIDIA Ampere GPU architecture provides up to 1,555 GB/s peak memory bandwidth and more than 75 GB/s intra-node communication bandwidth [6]. However, the fastest commodity network such as Mellanox InfiniBand (IB) HDR, which is widely used in top supercomputers [7], can only achieve 25 GB/s bandwidth. Figure 1 depicts the situation on Sierra supercomputer [8]. Although the network interconnects can handle small messages, they become a bottleneck for large message transfers. These architectural issues prevent us from efficiently scaling HPC and data science applications to larger GPU systems. Therefore, it is vital to explore new techniques to improve the performance of large message transfers between GPUs.

*This research is supported in part by NSF grants #1818253, #1854828, #1931537, #2007991, #2018627, and XRAC grant #NCR-130002.

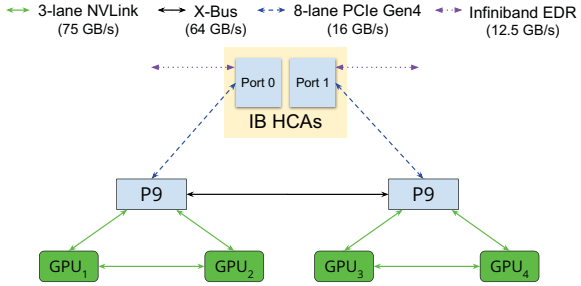
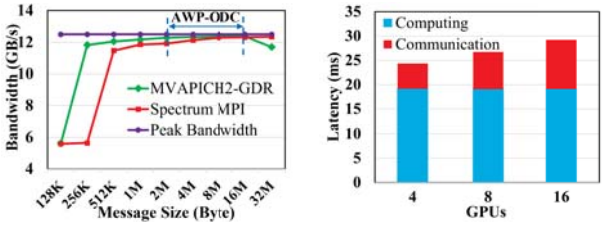



Fig. 1. Disparity between intra-node and inter-node GPU communication on Sierra OpenPOWER supercomputer. P9: IBM POWER9 processor. IB HCAs: InfiniBand Host Channel Adapters. Courtesy [9]

### A. Motivation and Challenges

Figure 2(a) shows the inter-node communication bandwidth with two state-of-the-art MPI libraries: Spectrum MPI and MVAPICH2-GDR. As can be seen in this figure, these libraries are well optimized to saturate the bandwidth of the IB network [5], [9] for large messages. Although MPI libraries are well optimized to saturate the network bandwidth, the communication time at the application-level is still one of the main bottlenecks for the performance of many HPC and data science applications. To clarify this, Figure 2(b) shows computation versus communication time for a representative sample application, AWP-ODC [10]. As can be seen in this figure, the communication time is still significant as we conduct experiments with larger problem sizes and more GPUs. Since the inter-node communication bandwidth is already saturated, we should seek other ways to reduce the communication

978-1-6654-4066-0/21/$31.00 ©2021 IEEE
DOI 10.1109/IPDPS49936.2021.00053




(a) Inter-node D-D Bandwidth (b) AWP-ODC time breakdown

Fig. 2. Motivating Example: production-quality and optimized CUDA-Aware MPI libraries can saturate IB EDR network while the communication time remains a significant bottleneck for HPC applications e.g. AWP-ODC. The message range for AWP-ODC is 2M to 16M as shown in Figure (a).

time of the HPC applications. This leads up to the following challenge: ***What are the other techniques—besides improving the communication bandwidth—that can be used to reduce the communication time of the HPC and data science applications on modern, dense GPU clusters?***

One way to improve communication performance is by taking advantage of compression techniques. Compression provides the opportunity to reduce the data size being transferred. This way, we can lower the pressure on the network and the interconnect with limited bandwidth. Various compression techniques have been proposed in the literature. They are divided into two groups: CPU-based algorithms and GPU-based algorithms. The first four rows of Table I show CPU-based algorithms with no GPU support. These algorithms have low throughput compared to GPU-based designs [11]–[14]. On the other hand, existing GPU-based compression schemes such as GFC [11], MPC [12], SZ [13], and ZFP [14] are typically focusing on achieving high compression ratio and not absolute high performance. In this paper, we address this question: ***What are the significant overhead of existing GPU-based compression algorithms that make them inefficient for HPC and data science applications?***

Some works in literature [15], [16] try to improve the communication performance of HPC applications by integrating compression mechanisms into MPI libraries. The main limitation of these efforts is that they only consider CPU-based compression techniques. As mentioned earlier, these techniques usually have low throughput. There have also been some works [11], [12], [17], [18] that study the efficiency and applicability of GPU-based compression algorithms for HPC applications. However, these designs are inefficient as they do not perform *on-the-fly* message compression and impose significant overhead in the critical path of communications. As can be seen in the last column of Table I, GPU-based compression schemes such as GFC, MPC, SZ, and ZFP do not have efficient support for MPI applications. These limitations—in the current compression schemes—motivate us to address this question in our paper: ***How can we design efficient on-the-fly message compression schemes to improve the performance of HPC and data science applications on the modern GPU clusters?***

TABLE I
COMPARISON BETWEEN DIFFERENT COMPRESSION TECHNIQUES

| Compression Designs | Lossless | Lossy | GPU Support | Single Precision | Double Precision | High Throughput | MPI Support |
|---|---|---|---|---|---|---|---|
| **FPC [19]** | ✓ | ✗ | ✗ | ✗ | ✓ | ✗ | ✓ |
| **fpzip [20]** | ✓ | ✓ | ✗ | ✓ | ✓ | ✗ | ✗ |
| **ISOBAR [21]** | ✓ | ✗ | ✗ | ✓ | ✓ | ✗ | ✗ |
| **SPDP [22]** | ✓ | ✗ | ✗ | ✓ | ✓ | ✗ | ✗ |
| **GFC [11]** | ✓ | ✗ | ✓ | ✗ | ✓ | ✓ | ✗ |
| **MPC [12]** | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ | ✗ |
| **SZ [13]** | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ |
| **ZFP [14]** | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ |
| **Proposed MPC-OPT** | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ |
| **Proposed ZFP-OPT** | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |

### B. Contribution

In this paper, we take up these challenges and redesign the MVAPICH2 MPI library to enable high-performance, on-the-fly message compression for modern GPU clusters. To the best of our knowledge, this is the first work that leverages GPU-based compression techniques to significantly improve the GPU communication performance for various MPI primitives, and MPI-based data science and HPC applications. To summarize, this paper makes the following main contributions:

- We conduct a thorough analysis of the state-of-the-art GPU-based compression algorithms, including the lossless and lossy algorithms and analyze the effectiveness of the two GPU-based compression schemes, MPC and ZFP, on the modern GPU systems.
- We propose a framework to integrate GPU-based compression algorithms to the communication library MVAPICH2. Later, we evaluate the critical overheads of these algorithms and identify regions that degrade the performance when they are integrated into MPI libraries.
- We propose enhancements to the MPC lossless compression scheme (MPC-OPT), to tackle the limitations of the MPC when it is integrated into MPI library. The proposed MPC-OPT scheme improves the GPU utilization by performing *on-the-fly* compression and avoiding expensive memory allocation and extra copies in MPC.
- We propose enhancements to the lossy ZFP compression scheme (ZFP-OPT), to tackle the limitations of ZFP when it is integrated into MPI library. The proposed ZFP-OPT scheme takes advantage of a caching mechanism to reduce the number of CUDA function calls.
- We use OSU Microbenchmark suite to evaluate the point-to-point communication and show that the compression design can achieve up to 83% improvement. We also enhance the Microbenchmark to use real data set and get up to 85% improvement for collective operations.
- We evaluate the effectiveness of MPC-OPT and ZFP-OPT through various application studies and show up to 19% and 37% improvement of GPU computing flops in the AWP-ODC application with MPC-OPT and ZFP-OPT, respectively. We also show up to 1.56× improvement in Dask throughput.

## II. Analysis of GPU compression algorithms

In this section, we discuss the analytic models to realize how *on-the-fly* message compression can benefit communication and what are key factors to benefit from it. Next, we assess the existing GPU-based lossless and lossy compressors on the modern GPU clusters and provide insights into their features.

### A. Analytical Models

We present the cost models based on the ones discussed in [15]. Table II summarizes the notations used in this paper.

TABLE II
Notations used in this paper

| Symbol | Definition |
|---|---|
| $T$ | Overall latency of GPU communication (*seconds*) |
| $T'$ | Overall latency of GPU communication with compression (*seconds*) |
| $T_s$ | Setup time for communication (*seconds*) |
| $T_{compr}$ | Execution time of compression kernel (*seconds*) |
| $T_{decompr}$ | Execution time of decompression kernel (*seconds*) |
| $T_{oh\_compr}$ | Overheads related to compression (*seconds*) |
| $T_{oh\_decompr}$ | Overheads related to decompression (*seconds*) |
| $S_m$ | Size of original message (*Bytes*) |
| $S'_m$ | Size of compressed message (*Bytes*) |
| $B$ | Network Bandwidth between GPUs ($GB/s$) |
| $CR$ | Compression ratio |

A typical communication cost can be realized as follows:

$$T = T_s + \frac{S_m}{B} \tag{1}$$

With compression, the cost of message transfer is reduced. But there are overheads of compression and decompression kernels and related overheads such as extra memory allocation for compressed data. In other words, we have:

$$\begin{aligned} T' &= T_s + T_{compr} + T_{oh\_compr} + \frac{S'_m}{B} + T_{decompr} + T_{oh\_decompr} \\ &= T_s + T_{compr} + T_{oh\_compr} + \frac{S_m}{CR \times B} + T_{decompr} + T_{oh\_decompr} \end{aligned} \tag{2}$$

Ideally, if the overheads related to compression and decompression can be small enough to be ignored, the communication cost can be simplified to equation (3) below:

$$T' = T_s + T_{compr} + \frac{S_m}{CR \times B} + T_{decompr} \tag{3}$$

Based on these models, we summarize the key factors addressed in this paper to achieve high-performance *on-the-fly* message compression as follows.

- Intuitively, a higher compression ratio implies a smaller message size to be exchanged over the network.
- Lower overhead in the compression and decompression processes can better utilize the GPU resources and help benefit communication even with low compression ratio.

### B. Assessment of Existing Compression Libraries

In this paper, we pick two publicly available GPU-based compression libraries that have proven applicable for scientific applications. First, Massively Parallel Compression (MPC) [12], an open-source GPU-based algorithm, is used to represent the lossless compression algorithm. The key technique of MPC is to determine the similarity between consecutive floating-point numbers and compress them accordingly. A *dimensionality* value is used to determines the position of the prior value in the same chunk to predict the current value [12]. Recent studies [17], [23] also demonstrate that error-bounded lossy compression methods are acceptable for many HPC applications. In this work, we use ZFP [14] as it provides a rich set of interfaces, and support parallel compression using GPUs. The current parallel implementation in ZFP supports CUDA-enabled fixed-rate compression. In the fixed-rate mode, each d-dimensional array value is deconstructed into $4^d$ independent blocks. The fixed compressed bits (i.e., rate) per block are amortized over these blocks to achieves a compressed rate in bits/value. For instance, 16 bits/value for 32-bit single-precision floating-point data can yield a compression ratio of 2, i.e., half the data size. Next, we used eight representative HPC datasets, reported in [12], with single-precision floating-points, various data sizes, and unique value distribution to understand the performance and compression ratio of MPC and ZFP on the modern GPU architecture, an NVIDIA V100 GPU.

Table III shows the best compression ratio of MPC achieved on the eight datasets with fine-tuned dimensionality. MPC yields a compression ratio of less than 2, i.e., the reduced size is less than 50%, for most of the datasets, which matches the results reported in [12]. Nonetheless, the lowest throughput of compressor and decompressor is 168.91 Gb/s that is sufficient to be used for the commodity network as mentioned before.

TABLE III
Performance and Compression Ratio of MPC and ZFP

| Dataset | Size (MB) | Unique vals (%) | $TP_{compr}$ ZFP (Gb/s) | $TP_{decompr}$ ZFP (Gb/s) | CR-ZFP | $TP_{compr}$ MPC (Gb/s) | $TP_{decompr}$ MPC (Gb/s) | CR-MPC |
|---|---|---|---|---|---|---|---|---|
| msg_bt | 128 | 92.9 | 469.29 | 735.56 | 2 | 206.01 | 189.14 | 1.339 |
| msg_lu | 93 | 99.2 | 451.48 | 743.52 | 2 | 211.88 | 191.05 | 1.444 |
| msg_sp | 16 | 98.9 | 421.88 | 709.34 | 2 | 204.93 | 174.58 | 1.352 |
| msg_sppm | 16 | 10.2 | 280.36 | 395.08 | 2 | 199.68 | 174.31 | 8.951 |
| msg_sweep3d | 60 | 89.8 | 334.65 | 571.19 | 2 | 207.14 | 211.25 | 1.537 |
| obs_error | 30 | 18.0 | 447.22 | 717.36 | 2 | 209.25 | 187.35 | 1.301 |
| obs_info | 9.1 | 23.9 | 536.88 | 739.07 | 2 | 194.18 | 168.91 | 1.440 |
| num_plasma | 17 | 0.3 | 585.80 | 822.01 | 2 | 197.94 | 185.52 | 1.348 |

For ZFP, we used the 1D array type with the number of total floating-point values as dimension size. Table III exhibits the achieved performance and compression ratio using ZFP for the same eight datasets. Here, we used the rate 16 to compress the data where each 32-bit floating-point value can be compressed to only 16 bits. As expected, a fixed compression ratio and high compression and decompression throughput are observed.

## III. Proposed GPU-based On-the-fly Message Compression Framework

In this section, we describe the framework that integrates GPU-based compression algorithms into the MPI communication middleware.

### A. Proposed Compression Framework

Figure 3 depicts the high-level framework for integrating compression algorithms into a communication middleware. For GPU communication, the original and compressed data

reside in GPU memory on both the sender and receiver sides. As discussed in Section II, the GPU-based compression algorithms generate and process information of control parameters to launch the compression and decompression kernels with desired configurations. In the proposed framework, the sender keeps the information as a header on the system memory and forwards it to the receiver. Hence, the receiver can only perform decompression once the header is received. This requirement incurs one extra message exchange between the sender and receiver. For the standard rendezvous protocol commonly used in modern communication middleware, there is a handshake between the sender and receiver before the real data transfer. Specifically, the sender sends a Request-To-Send (RTS) packet, and the receiver may respond to a Clear-To-Send (CTS) packet, then the sender can initiate the data transfer. In the proposed framework, we piggyback the compression-related header information into the RTS packet to avoid extra message exchanges. The header carries various information such as whether the compression is used or not, used compression algorithm on the sender side, compressed data size, and so on. Accordingly, a receiver can perform the decompression with corresponding kernel modules and configurations. In short, the CPU is responsible for processing control/header information, synchronizing with the GPU compression/decompression kernels, and progressing the communication while GPU is performing compression and decompression.

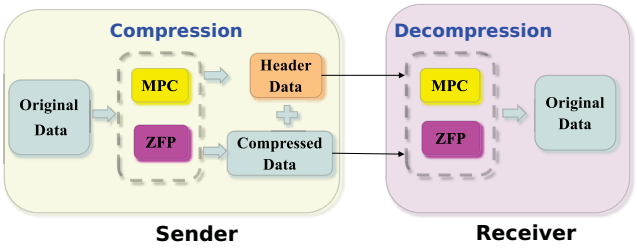



Fig. 3. Framework of integrating compression algorithms into MPI library. Compression algorithm MPC and ZFP are integrated into the MPI library. Rendezvous protocol is used to send the header data and compressed data.

Figure 4 depicts the detailed data flow in the proposed framework. There are seven steps involved in point-to-point communication as follows. In ①, on the sender side, if the data to be sent resides in GPU memory, and the data size exceeds a pre-defined threshold, the sender process launches the compression kernel on GPU with the desired control parameters specific to a compression algorithm. For example, MPC's control parameters include the total number of floating-point values and dimensionality [12]. For ZFP, the control parameters include the rate, i.e., compressed bits per floating-point value, and dimensions of data array [14]. Also, ZFP has well-defined APIs and data structures, such as data field *zfp_field* and bitstream *zfp_stream*, to contain the required control parameters. Accordingly, the proposed framework constructs such information and passes it to the ZFP's interface *zfp_compress* to launch the compression kernel on GPU.

As shown in Figure 4, the control parameters are represented as $A$ in the header data. In step ②, GPU executes the desired compression kernel to compress the original data. Note that a temporary intermediate buffer is required to store the compressed data on GPU. Also, MPC requires an extra buffer *d_off*, which stores a temporary array with initial value "-1" used for synchronization between GPU thread blocks. Thus, the number of elements of this array is equal to the number of GPU thread blocks used in compression. The values in the array are used as flags inside the compression kernel to indicate whether each thread block has finished compression. Upon the completion of the compression kernel, extra information is generated to decompress and restore the data. In step ③, the kernel returns the compressed data size, and we concatenate such information, i.e., represented as $B$, in the header data. Note that MPC uses naive data copy schemes such as *cudaMemcpy* to copy this information from compressed GPU buffer to the header on the host memory. For ZFP, the compression interface *zfp_compress* returns compressed size without an extra copy as the compression ratio is predictable [14]. Once the compression is completed, the sender piggybacks the header data in the RTS packet and sends it to the receiver.

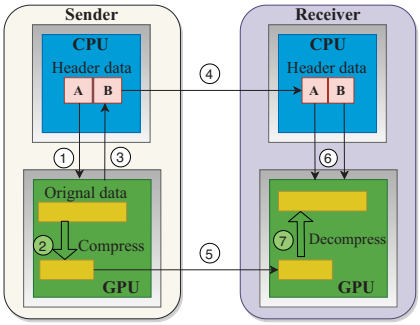



Fig. 4. Data flow of GPU communication with compression. There are seven steps: 1) Launch compression kernel with control parameters 2) Run compression kernel on GPU 3) Returned compressed size 4) Send header data with RTS packet 5) Send compressed GPU data 6) Launch decompression kernel with header data 7) Run decompression kernel to restore the data.

Algorithm 1 provides a high-level overview of the compression framework on the sender side. First, we extract compression algorithm $L$ from $A$ (Line 1). If the compression algorithm is MPC, we launch MPC kernel $MPC\_compress()$ and give $S$, $A$, and $d\_off$ as its input (Line 6). If the compression algorithm is ZFP, we create instances of $zfp\_stream$ and $zfp\_field$ and attach control parameters to them (Lines 8 and 9). Then, we launch ZFP kernel $ZFP\_compress()$ with $S$, $zfp\_stream$, and $zfp\_field$ as input (Line 10). The outputs of both MPC and ZFP are the compressed data $M$ and the size of compressed data $B$. Finally, we attach compression information $(A, B)$ to $RTS$ (Line 11).

Upon receiving the RTS packet with compression control information, the receiver stores the header data in a receive request data structure. If the header indicates that the data is compressed, the receiver prepares a temporary buffer on the GPU with the corresponding size to receive the incoming

**Algorithm 1:** On-the-fly Compression Framework: Sender

**Input :** Send buffer $S$, control parameters $A$, $RTS$ without compression information
**Output:** Compressed data $M$, $RTS$ with compression information

```
Extract compression algorithm L from A;
Allocate GPU buffer for M;
if L==MPC then
    Allocate GPU buffer for d_off;
    Initialize d_off to -1;
    (B,M)=MPC_compress(S,A,d_off);
else if L=ZFP then
    Construct zfp_stream and zfp_field;
    Attach A to zfp_stream and zfp_field;
    (B,M)=ZFP_compress(S,zfp_stream,zfp_field);
Attach compression information (A, B) to RTS;
```

compressed data. In step ⑤, the sender sends the compressed data to the target remote address using existing communication protocol [2], [5], [24] once a CTS packet is received.

Once the receiver collects all the data from the sender, it launches the desired decompression kernel with the corresponding configuration according to the control parameters suggested in the header (in step ⑥). For MPC, similar to the compression kernel, the decompression kernel needs an extra device buffer *d_off* for intra-kernel synchronization purposes. Finally, in step ⑦, the decompression kernel decompresses the received data and restores the original data on the user buffer, and the communication with *on-the-fly* message compression completes. Algorithm 2 shows the high-level design of the compression framework on the receiver size.

**Algorithm 2:** On-the-fly Compression Framework: Receiver

**Input :** received $RTS$, received compressed data $M$
**Output:** received data $O$

```
Extract control parameters A from RTS;
Extract compression algorithm L from A;
Allocate GPU buffer for O;
if L==MPC then
    Allocate GPU buffer for d_off;
    Initialize d_off to -1;
    O=MPC_decompress(A,d_off);
else if L==ZFP then
    Construct zfp_stream and zfp_field based on control
      parameter B;
    O = ZFP_decompress(A,zfp_stream,zfp_field);
```

### B. Analysis of the Compression Framework

First, we evaluate the proposed compression framework using the OSU Micro-Benchmark suite (OMB) on the TACC Longhorn cluster. As shown in Figure 5, we can observe poor performance from the naïve integration. Here, the overhead of the compression and decompression process outweighs the reduced communication time. As suggested in equation (2), we would like to reduce the overhead $T_{oh\_compr}$ and $T_{oh\_decompr}$ as much as possible to reduce the overall latency incurred by the compression and decompression process. Therefore, we investigate the possible overheads in using the existing GPU-based compression libraries and summarize them as follows.

In the proposed naïve integration, using existing compression libraries incur overheads when *preparing* the compression

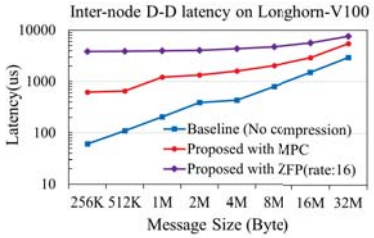
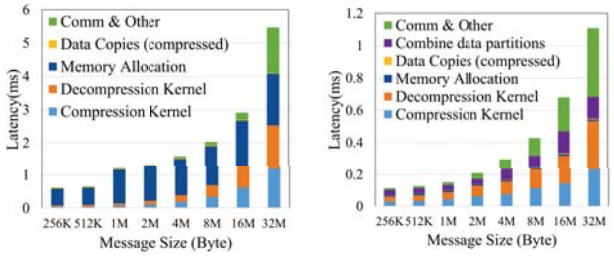



Fig. 5. Latency of naively integrating the compression algorithms.




(a) MPC-Original (b) MPC-OPT

Fig. 6. Breakdown of overall inter-node latency using MPC before and after optimization on Longhorn.

and decompression kernel. For instance, existing compression algorithms require allocating a temporary device buffer to store the compressed data on both the sender and receiver sides, which brings significant overhead. For MPC, extra overheads exist for allocating a device buffer *d_off* array, as indicated above. Furthermore, MPC uses *cudaMemcpy* to retrieve the compressed data size from GPU to the host memory.

In the following sections, we tackle these issues and propose two compression schemes, MPC-OPT and ZFP-OPT, that integrate well with the communications libraries and achieve high-performance *on-the-fly* message compression.

## IV. Proposed MPC-OPT Scheme

Towards overcoming the challenges identified above, we propose several optimizations to mitigate the overheads incurred by the existing GPU compression libraries when used for *on-the-fly* message compression.

### A. Profiling of the Compression Framework with MPC

First, we report the performance profiling of inter-node GPU communication with the proposed naive integration using MPC algorithm as shown in Figure 6(a). As shown in the figure, memory allocations, such as *cudaMalloc* calls, occupy 83.4% and 28.3% of overall time for 256KB and 32MB messages, respectively. The compression and decompression kernels take from 11.7% to 46.3% of the entire process. As indicated in Section III-A, MPC requires copying the compressed data size, which is a 4-bytes unsigned integer number, from GPU memory to host, using API like *cudaMemcpy*, that consistently spends nearly 20us due to the driver and synchronization overhead. These extra costs in memory allocations and data movements are far exceeding the benefits of transferring the compressed message using MPC.

### B. Proposed Optimization Metrics in MPC-OPT

Based on the profiling result, we propose the following optimizations to reduce the overheads discussed previously.

1) To avoid the overheads of allocating temporary device buffers, we build a pre-allocated GPU buffer pool. Such GPU buffers are allocated at the initialization phase, e.g., *MPI_Init*, to remove it from the critical communication paths. On the sender side, each send operation picks an available device buffer in the pool to store the compressed message and returns the resource into the buffer pool when the send request is completed. Similarly, the receiver also gets the free buffer from the pool for the decompression process. Note that such a buffer pool can be dynamically increased and decreased on demand to satisfy the application's requirement. Currently, the buffer size is fixed in the memory pool. To make the memory more efficient, such a memory pool could be further enhanced to support varying buffer sizes to accommodate different message sizes in the future.
2) Similarly, we build a buffer pool to avoid the memory allocation for *d_off* used in MPC. Sender and receiver can re-use the GPU buffers from their pools for *d_off*.
3) To eliminate expensive memory copies using APIs like *cudaMemcpy*, we employ low-latency GDRCopy schemes [25] to copy the information of compressed data size from GPU memory to host. This can reduce the cost from 20$\mu$s to 1-5$\mu$s.

Note that the optimization methods mentioned above can be applied to any compressor including ZFP. Moreover, we propose an advanced optimization solution for MPC to hide compression latency by overlapping kernels. MPC, by design, always exploits the maximum GPU thread blocks, i.e., Streaming Multiprocessors (SM), to execute the compression and decompression kernels for any data size. However, it wastes the GPU resources and occurs higher scheduling overhead. For example, we observed that on the datasets shown in Table III, the compression/decompression runtime of using half of the available SMs is roughly the same as using full GPU. Based on our study, this design would result in a higher overhead when using more thread blocks in a single kernel as MPC kernels employ the busy waiting scheme for intra-kernel synchronization between the thread blocks. To mitigate such synchronization overhead, we propose the data partitioning and multi-stream flow to decompose the compression and decompression kernels. Figure 7 depicts an example of four streams to illustrate the kernel decomposition that executes multiple compression kernels for four independent data partitions. Here, each kernel only uses one-fourth of the maximum GPU thread blocks, and this is adjustable. As a result, multiple kernels can still utilize GPU resources in parallel with lower intra-kernel synchronization overhead. After all the kernels are done with compression, the compressed data of each partition are merged in a single contiguous buffer. To avoid conflicts, as shown in Figure 7, these combine operations follow a fixed order because we only realize the compressed

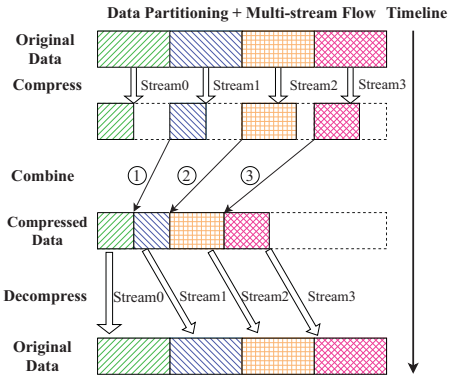



Fig. 7. Data partitioning and multi-stream flow for MPC.

size and corresponding offsets upon the kernel completion. In the proposed framework, we store the number of partitions as well as compressed data sizes of all partitions in the header and piggyback it with the RTS packet. Hence, the receiver can use the same number of streams to decompress the data in parallel for each partition.

**Algorithm 3:** Proposed MPC-OPT Scheme: Sender side

**Input :** Send buffer $S$, control parameters $A$, $RTS$ without compression information, Preallocated buffer pool $P$, number of partitions $N$
**Output:** Compressed data $M$, $RTS$ with compression information

```
1  Extract compression algorithm L from A;
2  if L==MPC then
3      Get preallocated GPU buffer from P for M;
4      Get preallocated GPU buffer from P for d_off;
5      Initialize d_off to -1;
6      [S_1, ..., S_N]= divide S to N partitions;
7      (B_i,M_i)=MPC_compress(S_i,A_i,d_off_i); //Runs in parallel with
         CUDA Streams
8      M = combine [M_1,...,M_N];
9      Attach N and [B_1,...,B_N] to control parameters A;
10     Attach compression information (A, B) to RTS;
```

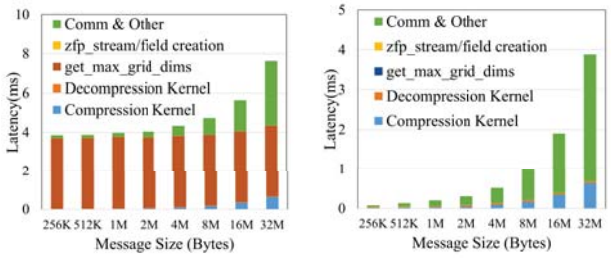
Note that we have verified that such partitioning has a negligible impact on the final compression ratio. Algorithm 3 shows a high-level design of MPC-OPT on the sender side. This algorithm is similar to Algorithm 1 except that it avoids extra memory allocations and instead, uses preallocated buffer $P$ (Lines 3 and 4). Moreover, it parallelizes data compression by data partitioning and using CUDA streams (Lines 6 and 7). As discussed earlier, we do the same optimizations on the receiver side for decompression in MPC-OPT. To achieve better performance, we fine-tune the number of partitions used for different message sizes based on the experimental results.

As shown in Figure 6(b), the compression and decompression time gets reduced significantly after applying the proposed optimization methods. We observe up to 4X improvement compared to the naive integration in Figure 5.

## V. Proposed ZFP-OPT Scheme

### A. Profiling of the Compression Framework with ZFP

In addition to applying optimization methods described in the previous section, we report the profile results when using ZFP for message compression, as depicted in Figure 8(a), to




(a) ZFP-Original (b) ZFP-OPT

Fig. 8. Breakdown of overall inter-node latency using ZFP before and after optimization on Frontera Liquid.

illustrate the performance issues. In this experiment, we used 16 bits/value rate and one-dimensional array type for compression. The creation of *zfp_stream* and *zfp_field* using CPU processes on both the sender and receiver sides only takes 9us. The dominant overhead comes from the *get_max_grid_dims* function, which is used to determine the GPU hardware configuration such as the number of available SMs. In this function, *cudaGetDeviceProperties* is called per message to retrieve the maximum grid dimension supported. Such CUDA call incurs significant driver overhead that takes nearly 1840$\mu$s. However, this information is static for a given GPU architecture.

### B. Proposed Optimization Methods in ZFP-OPT

The CUDA API *cudaGetDeviceProperties* returns all the device properties, which is a time-consuming operation. However, as ZFP kernel only requires the maximum grid dimensions information, which is static for a given GPU at runtime. Therefore, we modify the function to use *cudaDeviceGetAttribute* to get the required information only once and cache the information as static values. As a result, the run time of this function gets reduced to only approximately 1$\mu$s from 4,000$\mu$s. Figure 8(b) shows the significantly reduced overall latency after applying all the proposed optimization schemes. As can be seen, the majority of time is spent on compression, decompression, and reduced communication. Moreover, we can observe that ZFP yields significantly low decompression time since it is a lossy algorithm.

## VI. Microbenchmark Results and Analysis

We conduct experiments on four GPU-enabled clusters, Longhorn and Liquid subsystems, on the Texas Advanced Computing Center (TACC) Frontera supercomputer, Lassen system in Lawrence Livermore National Laboratory, and the RI2 cluster of The Network Based Computing Lab at The Ohio State University. Longhorn, Lassen and RI2 systems are equipped with NVIDIA Tesla V100 GPUs whereas Frontera Liquid is equipped with NVIDIA Quadro RTX 5000 GPUs. The detailed specifications of Longhorn [26], Liquid submerged systems [27] and Lassen [28] systems are available in their respective specification sheets. The RI2 cluster is equipped with Xeon Broadwell E5-2680 v4, 2.4GHz Nodes with 14 Cores/Socket and 128GB of CPU Memory per node. Each Node is equipped with 1 NVIDIA Tesla V100 GPU that is connected to the CPU via the PCIe Host Bridge. The nodes on RI2 are interconnected with IB-EDR(one way 100Gb/s). The proposed framework was implemented on top of MVAPICH2-GDR [29], a highly-optimized GPU-Aware MPI library. We used osu_latency in OSU Micro-Benchmark suite (OMB) to evaluate the intra-node and inter-node point-to-point communications. We run osu_bcast and osu_allgather to evaluate collective communications. We reported the average latency over 1,000 iterations with 100 warm-up runs.

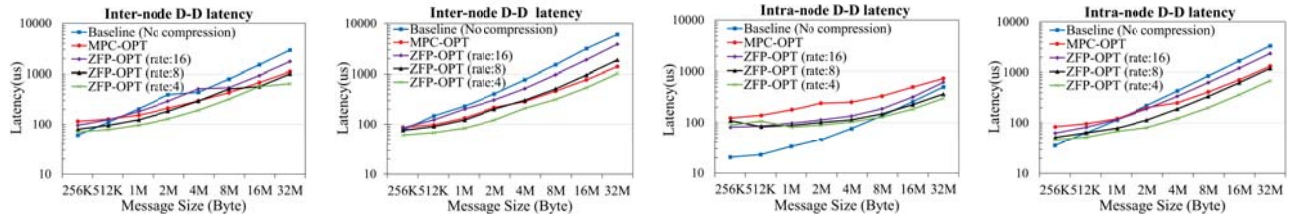
### A. Point-to-Point Communication Enhancements

*1) Inter-node GPU Communication:* Figure 9(a) shows the inter-node latency of message size 256KB to 32MB on Longhorn between two V100 GPUs across the IB EDR network. Since we focus on compression for large message transfer, we do not show the result for messages smaller than 256KB. MPC-OPT shows benefits compared to the baseline from 1MB message and achieves up to 62.5% reduced latency for 32MB. ZFP-OPT(rate:4) can achieve up to 78.3% reduced latency at 32MB. Figure 9(b) shows the inter-node latency between two RTX GPUs over FDR network on Frontera Liquid system. MPC-OPT yields reduced latency up to 77.1% at 32MB while ZFP-OPT(rate:4) can reduce the overall latency up to 83.1% at 32MB.

*2) Intra-node GPU communication:* Next, we evaluate the performance of the proposed schemes on faster interconnects such as PCIe and NVLink. Figure 9(c) shows the intra-node latency between two V100 GPUs with NVLink on Longhorn. Using MPC-OPT has not yielded any benefit because the high-speed 3-lane NVLink used here can transfer the raw data faster than the entire compression and decompression procedure. Although the performance of ZFP-OPT is significantly better than MPC-OPT, it only improves the communication of message size larger than 8MB. More specifically, ZFP-OPT(rates:4/8) improves performance up to 40.5% and 27.7% at 32MB, respectively. Figure 9(d) shows the intra-node latency between two RTX GPUs with PCIe bus on Frontera Liquid. Unlike NVLink, the bandwidth of PCIe link between GPUs is lower than the throughput of compression using MPC-OPT and ZFP-OPT. Hence, MPC-OPT and ZFP-OPT can reduce the latency up to 60.6% and 79.8%, respectively.

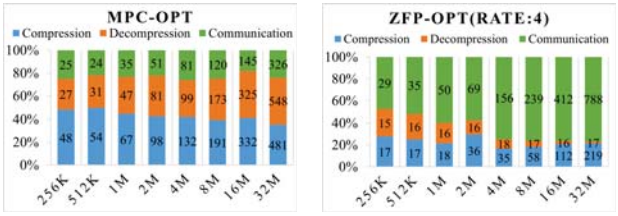
*3) Latency breakdown:* Figure 10(a) and Figure 10(b) show the inter-node latency breakdown of compression, decompression and communication for MPC-OPT and ZFP-OPT(rate:4) on Frontera Liquid. The compression/decompression time includes all overheads on the sender/receiver side. These overheads increase with the message size in MPC-OPT. However, for ZFP-OPT, the decompression time is nearly constant from 256KB to 32MB. The compression/decompression time in ZFP-OPT is lower than that of MPC-OPT since ZFP-OPT has higher throughput. In MPC-OPT, due to the high compression ratio on dummy data, the communication time is lower than that of ZFP-OPT.

Based on these experiments, we can summarize that ZFP-OPT can successfully reduce the traffic pressure on almost all interconnects with all message sizes except the high-speed



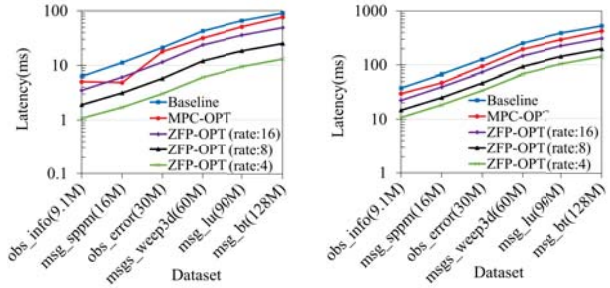

(a) Longhorn - Inter node (b) Frontera Liquid - Inter node (c) Longhorn - Intra Node (d) Fronter Liquid - Intra Node

Fig. 9. Inter-node and Intra-node point-point latency of GPU communication on Longhorn and Frontera Liquid. On Frontera Liquid, MPC-OPT and ZFP-OPT start to show benefits from 512KB for inter-node latency. With a lower rate, ZFP-OPT can achieve better improvement due to a higher compression ratio. For intra-node, such threshold is around 2MB for MPC-OPT and 1MB for ZFP-OPT. On Longhorn, inter-node latency shows a similar trend. For intra-node, ZFP-OPT(rate:4/8) shows improvement from message size larger than 8M. MPC-OPT does not show improvement between 256KB and 32MB due to the relatively low throughput of MPC kernel compare to intra-node interconnect.




(a) MPC-OPT (b) ZFP-OPT

Fig. 10. Inter-node latency breakdown of compression, decompression and communication time for MPC-OPT and ZFP-OPT(rate:4) on Frontera Liquid. The number shown in the bars are the latency results for corresponding parts.




(a) MPI_Bcast(8node,2ppn) (b) MPI_Allgather(8node,2ppn)

Fig. 11. Latency of MPI_Bcast and MPI_Allgather for 8 nodes, 2ppn on Frontera Liquid.

NVLink. On the other hand, MPC-OPT can be useful for input data that yields a high compression ratio over slow IB and PCIe networks.

### B. Collective Communication Enhancements

This section evaluates the impact of the proposed framework on the performance of collective operations. We modified OMB to transfer data from real datasets in Table III.

*1) MPI_Bcast:* Figure 11(a) shows the latency of MPI_Bcast on 8 nodes with 2 ppn (2GPUs/node) on Frontera Liquid. MPC-OPT reduces the latency from 15% on msg_bt to 57% on msg_sppm. The maximum improvement is due to the highest compression ratio on msg_sppm. ZFP-OPT achieves nearly the same improvement for different datasets with a given rate. ZFP-OPT(rate:4) reduces the latency by 85%.

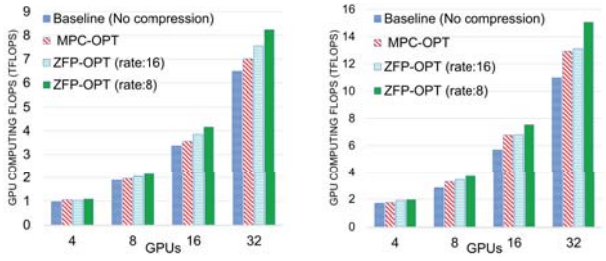
*2) MPI_Allgather:* Figure 11(b) shows the latency of MPI_Allgather for GPU data for 8 nodes with 2 ppn on Frontera Liquid. MPC-OPT can improve the performance from 20% on msg_bt to 30% on msg_sppm. ZFP-OPT provides nearly constant improvement for a given rate and it reduces the latency up to 73%. We observed the same trends for MPI_Bcast and MPI_Allgather with four nodes and 1 ppn. However, due to lack of space, we do not include the graphs.

## VII. Application Results and Analysis

### A. AWP-ODC

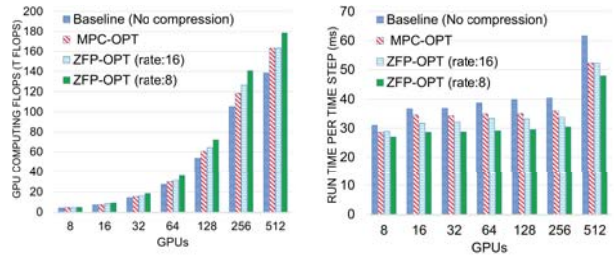
We use the Anelastic Wave Propagation software (AWP-ODC-OS) which is a GPU-enabled application to simulate wave propagation in a 3D viscoelastic or elastic solid [10]. We make minor modifications to the application to enable using CUDA-Aware MPI primitives, e.g., passing device buffer directly to MPI_Isend without an explicit copy. In our experimental analysis, we use a single moment source as the input along with a mesh file of dimensions 320x320x2048. AWP-ODC-OS reports the averaged run time per time step and GPU computing flops. We show weak scaling results in this section.

Figure 12 depicts the results on the Frontera Liquid. ZFP-OPT(rate:8) is able to achieve up to 37% improvement compared to the baseline over 64 GPUs with 4 GPU/node. We also observe that more speedup can be achieved for ZFP-OPT with a lower rate due to a higher compression ratio. However, it would generate incorrect output as it exceeds the lowest precision AWP-ODC can tolerate. On the other side, MPC-OPT can achieve up to 19% improvement compared to the baseline on 64 GPUs with 4 GPUs/node. This result demonstrates that MPC-OPT can provide a high compression ratio for the data produced by AWP-ODC. In our experiments, we observed that MPC-OPT can yield the compression ratio as low as 3 and as high as 31. The highest compression ratio is achieved at the initialization steps when the elements in the data are highly duplicated. On Lassen cluster, we test larger scale up to 512 GPUs on 128 nodes shown in Figure 13. In Figure 13(a), MPC-OPT achieves 18% improvment on 512 GPUs with 4 GPUs/node. ZFP-OPT(rate:8) achieves 35% improvement on 128 GPUs with 4 GPUs/node. Figure 13(b) depicts the similar improvement for run time per time step. With MPC-OPT, the run time improves by 15% on 512 GPUs with 4 GPUs/node. ZFP-OPT(rate:8) improves the run time by 26% on 128 GPUs with 4 GPUs/node. This result shows the scalability of the proposed framework.




(a) 2GPUs/node (b) 4GPUs/node

Fig. 12. Weak scaling of AWP-ODC on Frontera Liquid (higher is better).



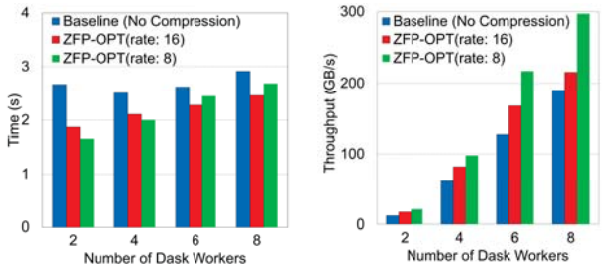

(a) GPU Computing Flops (TFlops) (higher is better) (b) Run time per time step (ms) (lower is better)

Fig. 13. Weak scaling of AWP-ODC with 4GPUs/node on Lassen. The scaling trends are similar for 1 GPU/node and 2 GPUs/node. The graphs are not included to avoid redundancy.

### B. Data Science using Dask

Dask [30] is a popular Python-based parallel and distributed computation framework that enables high-performance data science. The MVAPICH2-GDR library can be used to support communication between Dask workers using GPUs as computing devices. Dask typically exchanges large amounts of data between workers—typically messages range between 8MB to 1GB. We use a modified version of Dask with a new communication backend MPI4Dask [31] over MVAPICH2-GDR. We show the benefits of compression in a Dask-based application benchmark. This benchmark creates a cuPy array and distributes its chunks across Dask workers. The benchmark then adds these distributed chunks to their transpose, forcing the GPU data to move over the network. The following operations are performed:

$$y = x + x.T; \quad y = y.persist(); \quad wait(y); \tag{4}$$




(a) Execution Time-1GPU/node (Lower is better) (b) Aggregate Throughput-1GPU/node (Higher is better)

Fig. 14. Sum of cuPy Array and its Transpose (cuPy Dims: 10K×10K, Chunk size: 1K): Performance Comparison between Dask w/ MVAPICH2-GDR and Dask w/ MVAPICH2-GDR + ZFP-OPT(rate: 8/16) on the RI2 Cluster.

Results for the application benchmark—sum of cuPy array with its transpose—are shown in Figure 14. Figure 14(a) shows execution time where we observe an average speedup of 1.18× for 2 – 8 Dask workers when using ZFP-OPT(rate:8). Throughput comparison in Figure 14(b) depicts that with ZFP-OPT(rate:8) we achieve an aggregate throughput of 297.4 GB/s, outperforming the baseline by 1.56× for 8 dask workers.

To summarize, we show that MPC-OPT is suitable for applications—that exchange large messages with high compression ratio like the AWP-ODC. For ZFP-OPT, the fixed compression ratio helps to predict the reduced message size and can benefit more applications not only from the traditional HPC domain, but also for emerging workloads from data science. Nevertheless, one must carefully select the appropriate rate value to obtain the optimal reduced communication while maintaining acceptable accuracy.

## VIII. Related Work

To reduce communication latency, researchers have developed CPU-based online data compression. Jeannot et al. [32], [33] proposed an online CPU-based compression library AdOC for Gbits Ethernet. Other researchers have co-designed the MPI library with the CPU-based lossless compression algorithms. Ke et al. [15] designed a cMPI library that compresses the messages using CPU inside the MPI library. Filgueira et al. [16] designed a CoMPI library that integrated several compression algorithms into the MPI library for various primitives and demonstrates the efficacy of *on-the-fly* message compression for many CPU-based applications. Camata et al. [34] accelerated the MPI broadcast primitive using FPC compressor [19]. Shan et al. [35] has improved MPI_Reduce on many-core architecture with two parallel data compression schemes using OpenMP. With the high computing power of advanced GPU architectures, many lossless compression algorithms have significantly accelerated with the enhanced GPU version [11] [12]. However, they are only applied for offline processing and have not been integrated with communication libraries. On the other side, the lossy compression algorithms, SZ [13] and ZFP [14] have shown error-bounded accuracy and high compression throughput for the very large-scale HPC applications in recent study [17].

## IX. Conclusion

In this paper, we present a high-performance framework to leverage GPU-based compression algorithms, including lossless and lossy compressors, to perform GPU-assisted *on-the-fly* message compression. We integrate the proposed framework into the MVAPICH2-GDR library to benefit the HPC and data science applications. Moreover, we optimize the framework and existing GPU-based compression implementations by decomposing and overlapping the compression/decompression kernels to achieve the low-overhead *on-the-fly* message compression. The proposed framework demonstrates up to 85% reduced latency at the benchmark-level on the Frontera Liquid subsystem with NVIDIA RTX 5000 GPUs and IB FDR networks. At the application-level evaluation, the proposed design

yields up to 37% higher GFLOPs for the AWP-ODC, and up to 1.56x improvement in Dask throughput. As future work, we plan to explore the dynamic design to automatically determine the use of compression or selection of different algorithms for specific communication calls base on the compression costs and communication time assisted by real-time monitor like OSU INAM [36]. Also, we plan to study various GPU-based compression algorithms and explore the designs to accelerate various communication patterns like Alltoall and Allreduce.

## ACKNOWLEDGMENT

The authors thank Prof. Yifeng Cui for guiding conducting experiments with AWP-ODC. We thank our colleague Kawthar Shafie Khorassani for helping enable AWP-ODC using CUDA-Aware MPI primitives. We also thank Dr. Aamir Shafi for guiding conducting experiments with Dask.

## REFERENCES

[1] "MPI-3 Standard Document," http://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf.

[2] S. Potluri, K. Hamidouche, A. Venkatesh, D. Bureddy, and D. K. Panda, "Efficient Inter-node MPI Communication Using GPUDirect RDMA for InfiniBand Clusters With NVIDIA GPUs," in *42nd International Conference on Parallel Processing (ICPP), 2013*.

[3] C.-H. Chu, P. Kousha, A. A. Awan, K. S. Khorassani, H. Subramoni, and D. K. Panda, "NV-Group: Link-Efficient Reduction for Distributed Deep Learning on Modern Dense GPU Systems," in *Proceedings of the 34th ACM International Conference on Supercomputing*, 2020.

[4] X. Luo, W. Wu, G. Bosilca, T. Patinyasakdikul, L. Wang, and J. Dongarra, "ADAPT: An Event-based Adaptive Collective Communication Framework," in *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, 2018.

[5] S. S. Sharkawi and G. A. Chochia, "Communication protocol optimization for enhanced GPU performance," *IBM Journal of Research and Development*, vol. 64, no. 3/4, pp. 9:1–9:9, 2020.

[6] NVIDIA, "Whitepaper: NVIDIA A100 Tensor Core GPU Architecture," https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf, 2020, Accessed: March 2, 2021.

[7] E. Strohmaier, J. Dongarra, H. Simon, and M. Meuer, "TOP 500 Supercomputer Sites," http://www.top500.org, 1993, Accessed: March 2, 2021.

[8] Lawrence Livermore National Laboratory, "Sierra," https://computation.llnl.gov/computers/sierra, 2018, Accessed: March 2, 2021.

[9] K. S. Khorassani, C.-H. Chu, H. Subramoni, and D. K. Panda, "Performance Evaluation of MPI Libraries on GPU-enabled OpenPOWER Architectures: Early Experiences," in *International Workshop on OpenPOWER for HPC (IWOPH 19) at the 2019 ISC High Performance Conference*, 2018.

[10] Y. Cui, K. B. Olsen, T. H. Jordan, K. Lee, J. Zhou, P. Small, D. Roten, G. Ely, D. K. Panda, A. Chourasia, J. Levesque, S. M. Day, and P. Maechling, "Scalable earthquake simulation on petascale supercomputers," in *SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, 2010, pp. 1–20.

[11] M. A. O'Neil and M. Burtscher, "Floating-Point Data Compression at 75 Gb/s on a GPU," in *Fourth Workshop on General Purpose Processing on Graphics Processing Units*, March 2011.

[12] A. Yang, H. Mukka, F. Hesaaraki, and M. Burtscher, "MPC: A Massively Parallel Compression Algorithm for Scientific Data," in *IEEE Cluster Conference*, September 2015.

[13] S. Di and F. Cappello, "Fast Error-bounded Lossy HPC Data Compression with SZ," in *International Parallel and Distributed Processing Symposium(IPDPS)*, 2016.

[14] P. Lindstrom, "Fixed-rate compressed floating-point arrays," *IEEE Transactions on Visualization and Computer Graphics*, vol. 20, 08 2014.

[15] Jian Ke, M. Burtscher, and E. Speight, "Runtime Compression of MPI Messanes to Improve the Performance and Scalability of Parallel Applications," in *SC '04: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, 2004, pp. 59–59.

[16] R. Filgueira, D. Singh, A. Calderón, and J. Carretero, "CoMPI: Enhancing MPI Based Applications Performance and Scalability Using Run-Time Compression," in *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, Sep. 2009, pp. 207–218.

[17] S. Jin, P. Grosset, C. M. Biwer, J. Pulido, J. Tian, D. Tao, and J. P. Ahrens, "Understanding GPU-Based Lossy Compression for Extreme-Scale Cosmological Simulations," *ArXiv*, vol. abs/2004.00224, 2020.

[18] K. Zhao, S. Di, X. Liang, S. Li, D. Tao, Z. Chen, and F. Cappello, "Significantly Improving Lossy Compression for HPC Datasets with Second-Order Prediction and Parameter Optimization," in *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*, 2020, p. 89–100.

[19] M. Burtscher and P. Ratanaworabhan, "High Throughput Compression of Double-Precision Floating-Point Data," in *2007 Data Compression Conference*, 2007.

[20] P. Lindstrom and M. Isenburg, "Fast and efficient compression of floating-point data," *IEEE Transactions on Visualization and Computer Graphics*, vol. 12, no. 5, pp. 1245–1250, 2006.

[21] E. R. Schendel, Y. Jin, N. Shah, J. Chen, C. S. Chang, S. Ku, S. Ethier, S. Klasky, R. Latham, R. Ross, and N. F. Samatova, "Isobar preconditioner for effective and high-throughput lossless data compression," in *2012 IEEE 28th International Conference on Data Engineering*, 2012, pp. 138–149.

[22] S. Claggett, S. Azimi, and M. Burtscher, "SPDP: An Automatically Synthesized Lossless Compression Algorithm for Floating-Point Data," in *2018 Data Compression Conference*, 2018, pp. 335–344.

[23] G. Sun, S. Kang, and S.-W. Jun, "BurstZ: A Bandwidth-Efficient Scientific Computing Accelerator Platform for Large-Scale Data," in *Proceedings of the 34th ACM International Conference on Supercomputing*, 2020.

[24] R. Shi, S. Potluri, K. Hamidouche, J. Perkins, M. Li, D. Rossetti, and D. K. Panda, "Designing Efficient Small Message Transfer Mechanism for Inter-node MPI Communication on InfiniBand GPU Clusters," in *2014 21st International Conference on High Performance Computing (HiPC)*, Dec 2014, pp. 1–10.

[25] Davide Rossetti, "A fast GPU memory copy library based on NVIDIA GPUDirect RDMA technology," https://github.com/NVIDIA/gdrcopy, Accessed: March 2, 2021.

[26] "Longhorn - Texas Advanced Computing Center Frontera - User Guide," https://portal.tacc.utexas.edu/user-guides/longhorn.

[27] "Liquid Submerged System - Texas Advanced Computing Center, Frontera - Specifications," https://www.tacc.utexas.edu/systems/frontera.

[28] Lawrence Livermore National Laboratory, "Lassen — high performance computing," https://hpc.llnl.gov/hardware/platforms/lassen, 2018, Accessed: March 2, 2021.

[29] Network-Based Computing Laboratory, "MVAPICH: MPI over InfiniBand, Omni-Path, Ethernet/iWARP, and RoCE," http://mvapich.cse.ohio-state.edu/, 2001, Accessed: March 2, 2021.

[30] M. Rocklin, "Dask: Parallel computation with blocked algorithms and task scheduling," in *Proceedings of the 14th Python in Science Conference*, K. Huff and J. Bergstra, Eds., 2015, pp. 130 – 136.

[31] "MPI4Dask User Guide," http://hibd.cse.ohio-state.edu/static/media/hibd/dask/mpi4dask-0.1-userguide.pdf.

[32] E. Jeannot, B. Knutsson, and M. Bjorkman, "Adaptive online data compression," in *Proceedings 11th IEEE International Symposium on High Performance Distributed Computing*, 2002, pp. 379–388.

[33] E. Jeannot, "Improving middleware performance with adoc: an adaptive online compression library for data transfer," in *19th IEEE International Parallel and Distributed Processing Symposium*, 2005.

[34] J. Camata, M. Burtscher, W. Barth, and A. Coutinho, "Accelerating MPI Broadcasts using Floating-Point Compression," Aug. 2010. [Online]. Available: https://www.academia.edu/12231991/Accelerating_MPI_Broadcasts_using_Floating_Point_Compression

[35] H. Shan, S. Williams, and C. W. Johnson, "Improving MPI Reduction Performance for Manycore Architectures with OpenMP and Data Compression," in *2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, 2018, pp. 1–11.

[36] P. Kousha, B. Ramesh, K. Kandadi Suresh, C. Chu, A. Jain, N. Sarkauskas, H. Subramoni, and D. K. Panda, "Designing a profiling and visualization tool for scalable and in-depth analysis of high-performance gpu clusters," in *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, 2019, pp. 93–102.