IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Received 25 April 2023, accepted 29 May 2023, date of publication 1 June 2023, date of current version 19 June 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3281834

RESEARCH ARTICLE

Exploring Approximate Communication Using Lossy Bitwise Compression on Interconnection Networks

YAO HU, (Member, IEEE)
Research Institute for Digital Media and Content, Keio University, Hiyoshi Campus, Yokohama, Kanagawa 223-8523, Japan
e-mail: huyao0107@gmail.com

This work was supported by the Japan Society for the Promotion of Science (JSPS) KAKENHI under Grant 21K11859.

ABSTRACT The use of approximate communication has emerged as a promising approach for enhancing the efficiency of communication in parallel computer systems. By sending incomplete or imprecise messages, approximate communication can significantly reduce communication time. In this study, we examine application-level techniques for approximate communication to enable high portability on high-performance interconnection networks. Specifically, we focus on lossy compression of floating-point data, which is frequently exchanged between compute nodes in parallel applications. Our approach involves a simple application scenario where a source process compresses a communication dataset and a destination process decompresses it in an MPI parallel program. We use two bitwise procedures for compression: lossy *bit-zip* compression and lossless *bit-mask* compression. Our aim is to transmit the largest possible amount of approximate data with the least possible compression overhead. Additionally, we explore error check and correction techniques to ensure bit-flip fault tolerance for the compressed data during transmission. We implement our scheme in several communication-intensive MPI applications and demonstrate that our approximate communication approach effectively speeds up total execution time while staying within a specified quality-of-result *error bound*.

INDEX TERMS Interconnection network, parallel computing, approximate communication, message passing interface (MPI), lossy compression.

I. INTRODUCTION

Recent parallel computers in high-performance computing (HPC) systems face a primary concern with the network bandwidth, as its annual improvement is modest compared to the computation power in compute nodes. This highlights the strong need to improve the network bandwidth on high-performance interconnection networks. In parallel applications, a considerable amount of floating-point datasets are frequently exchanged between compute nodes via an interconnection network. Reducing the redundancy of communication data is a way to virtually increase the network bandwidth. Approximate computing [1] is gaining traction in this context, as it introduces a new trade-off between the quality and speed, allowing parallel applications to improve system efficiency while retaining an acceptable level of accuracy.

A significant fraction of applications running on HPC systems primarily utilize MPI (Message Passing Interface) parallelism to explore execution efficiency. Figure 1 reports that a reasonably high number of parallel applications spend more than half of their time in MPI [2]. Moreover, the fraction of time spent on communication increases significantly with the number of processes [3]. This large communication overhead limits the scalability of parallel applications. Hence, improving the MPI communication speed enhances the execution performance of applications.

Some scientific applications in parallel computing produce similar communication data repeatedly [4], which can be compressed by each compute node to reduce traffic. This can be achieved by sending only the difference information from the previous data. Data compression can result in shorter

The associate editor coordinating the review of this manuscript and approving it for publication was Mohamed Elhoseny.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License.
For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

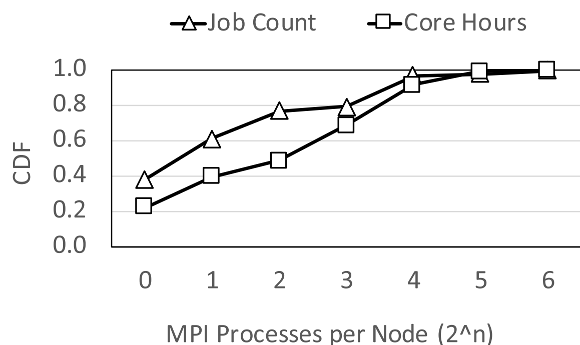


FIGURE 1. Cumulative distribution function (CDF) of resource usage and time on a large IBM BG/Q supercomputing system (Mira) [2].

communication times. Lossy compression, unlike lossless compression, is a data encoding method that uses inexact approximations and partial data discarding to represent the content. Lossy compression trades off data precision or quality to further reduce data size for storage, handling, and transmission. In this work, the compression ratio is a measure of how effectively floating-point data can be compressed. It is defined as the ratio of the number of bits used to represent the original floating-point data to the number of bits used to represent the compressed data. The compression ratio of lossy compression is generally higher than that of lossless compression because of the limited data precision loss. However, heavy data compression can harm MPI communication in high-performance interconnection networks, which are latency-sensitive and have inter-process communication times less than one microsecond [5].

In this research, we utilize an application-level fast lossy compression technique on floating-point data to enhance the effective communication network bandwidth on high-performance interconnection networks. In this work, we use the IEEE 754 standard [6] for representing and manipulating scientific floating-point numbers widely used in recent parallel computer systems. IEEE 754 offers the following advantages in comparison to alternative standards or methods like Bfloat, Posit, and RNS for digital data representation. Firstly, IEEE 754 provides a well-defined format for representing floating-point numbers, which ensures that computations can be carried out consistently across different platforms and programming languages. Secondly, IEEE 754 defines a set of arithmetic operations that are guaranteed to be accurate and consistent, even in the presence of rounding errors and other numerical issues. Thirdly, IEEE 754 provides a wide range of precision options, allowing users to choose the level of precision that best suits their needs. This flexibility is important in scientific and engineering applications, where the precision requirements may vary widely depending on the problem being solved. The efficacy of our lossy compression algorithm is higher for floating-point numbers with greater precision.

Our parallel programs compress the IEEE 754 floating-point datasets before sending them from the source side, and then exchange them with relevant processes, and ultimately decompress them at the receiver side. Note that, the scope of this work is limited to communication datasets that are transferred between compute nodes on high-performance interconnection networks. Our compression method involves two bitwise procedures: lossy *bit-zip* compression and lossless *bit-mask* compression. The *bit-zip* compression relies on value predictions for floating-point values, similar to the SZ method introduced in [7], which is often the case for intermediate and final floating-point results generated by certain scientific programs. The *bit-mask* compression explores the bit-level locality to enhance the compression ratio based on the outcomes of the *bit-zip* compression. We offer fine granularity to the MPI implementation, which generates a bit stream encapsulated in a byte array, corresponding to an MPI unsigned char type. In the whole lossy compression process, we maintain a specified *error bound* to adjust to the precision requirements of a target MPI application, which introduces a new trade-off between the compression ratio and precision.

Additionally, we develop a bit-flip recovery scheme that is optimized for a specific bit error rate (BER) to provide fault tolerance for the compressed data transmitted over high-performance interconnection networks. To achieve this, we employ application-level error check and correction techniques such as CRC (Cyclic Redundancy Check) [8] to detect any bit-flip errors and Hamming code [9] to correct them, as necessary.

Our main contributions in this work are as follows:

- We developed and evaluated a lossy bitwise compression algorithm at the application level for floating-point MPI communication data on high-performance interconnection networks.
- We explored application-level error detection and correction techniques to safeguard compressed data from precision loss during transmission on high-performance interconnection networks.
- Our evaluation results demonstrate that our lossy *bit-zip* compression method, supplemented by the lossless *bit-mask* compression technique, effectively enhances the execution performance of MPI applications while maintaining a specified *error bound* for the compressed data.

The remainder of the paper is structured as follows: We review related work in Section II. In Section III, we describe our lossy bitwise floating-point compression algorithm. Section IV presents our evaluation methodology and results. Finally, in Section V, we summarize our findings.

II. RELATED WORKS

One of the most widely used lossy compression algorithms is the discrete cosine transform (DCT) [10]. It is primarily utilized to compress multimedia data, such as audio, video, and images. A well-known lossy compression algorithm of the transform type is the fast Fourier transform (FFT) [11]. It transforms a signal from its original domain, often time or space, to a representation in the frequency domain and vice

versa. FFT offers a compression ratio similar to DCT [12]. Another more complex lossy compression algorithm, called ISABELA [13], achieves a higher compression ratio by transforming the data layout, such as sorting, cubic B-spline fitting, and window splitting. A fast lossy compression scheme [14] simply truncates the 16, 24, or 32 least significant bits to save total link energy. However, compression of near-zero small floating-point values does not guarantee an *error bound*.

In recent years, prediction techniques based on preceding data elements have garnered significant attention for lossy compression. A predictive type of lossy compression is applied to differential pulse-code modulation (DPCM) [15]. DPCM serves as a signal encoder that uses the pulse-code modulation (PCM) baseline but adds functionalities based on sample prediction of the signal. Linear predictive coding (LPC) [16] is another lossy compression algorithm that finds extensive use in speech coding and synthesis. In audio and speech processing, it uses the information of a linear predictive model to represent the spectral envelope of a digital speech signal in compressed form. While primarily designed for lossless compression, FPZIP [17] also supports lossy compression to achieve high compression ratios. FPZIP predicts the data by using a subset of encoded data and maps the difference to an integer number. To achieve a higher compression ratio, ZFP [18] divides 3-D floating-point arrays into small, fixed-size blocks of dimensions. As a result, it often outperforms FPZIP. FPC [19], [20], [21] is a proposed high-speed compressor for double-precision floating-point data. In FPC, the data is predicted using FCM/DFCM (differential-finite-context-method predictor) by selecting values closer to the true ones. Bitwise XOR operations are then performed between the predicted and true values. Finally, the leading zeros in the result are compressed to fewer bits (e.g., 4). SZ [7], [22], [23], [24] is proposed for lossy compression, which predicts data using three curve-fitting models: preceding-neighbor fitting model, linear-curve fitting model, and quadratic-curve fitting model. The primary idea behind SZ is to use linear predictive coding for predictable data and complicated binary analysis for unpredictable data. Its compression time is shorter than that of ZFP. Another work [4] presented a similar idea of floating-point data compression for FPGA-based high-performance computing by using a one-dimensional polynomial predictor. They also showed two different encoding methods, i.e., performance-oriented and area-oriented, which can achieve different compression ratios.

There is typically a trade-off between compression latency overhead and compression ratio in the aforementioned lossy compression algorithms. Historically, most of these algorithms have been optimized for the purpose of saving compact data in storage. Even for more recent applications in HPC and cloud computing, the main objective is still to compress data for storage [25], such as storing checkpoint images while accepting a minor decrease in the quality of results [26]. Our objective differs significantly from that of existing compression algorithms. Specifically, compressing data for inter-process communication in parallel programs is much more sensitive to latency than compressing data for storage purposes. To the best of our knowledge, our study is the first attempt to apply a lossy data compression algorithm at a program level to the inter-process communication datasets generated in parallel MPI applications on high-performance interconnection networks.

III. LOSSY BITWISE FLOATING-POINT COMPRESSION

This section proposes a lossy bitwise floating-point compression algorithm for inter-process communication data on high-performance interconnection networks. The algorithm consists of two stages. In the first stage, it employs a lossy *error-bounded bit-zip* compression scheme [27] to take advantage of the continuity of numerical floating-point data. In the second stage, it applies a lossless *bit-mask* compression technique to the outcomes of the first stage. This technique explores the bit-level locality to enhance the compression performance while maintaining the *error bound*.

Furthermore, we introduce a fault tolerance mechanism to safeguard the compressed data from precision loss during transmission, assuming that bit flips may occur on high-performance interconnection networks. Lastly, we discuss the integration of our design and implementation into MPI applications.

A. LOSSY BIT-ZIP COMPRESSION

We utilize distinct compression strategies for two kinds of data values: *predictable* and *unpredictable*. If the data can be linearly predicted within the *error bound*, the former is compressed using a sequence of pre-defined, abbreviated bit-string symbols. On the other hand, the latter is compressed by rounding the IEEE 754 floating-point data bits and removing the least significant bits (LSBs), while keeping the *error bound* intact.

1) LINEAR PREDICTION

We use a linear predictor to compress floating-point communication data, subject to a user-defined *error bound*. This bound allows for adjusting the balance between the quality of the results and the compression ratio. Given the abundance of prior research on data compression, we expand the linear predictor [4], [7], [28] for our lossy floating-point compression. Communication datasets that exhibit continuity typically feature locality, which increases the likelihood of achieving a high compression ratio. The linear predictor forecasts each floating-point number, enabling us to attain a high compression ratio and minimize communication traffic on high-performance interconnection networks.

Initially, we perform a straightforward conversion of the input floating-point values' bit strings as a preprocessing step such that each element in the converted dataset $d = \{d_1, d_2, \ldots, d_m\}$ is greater than or equal to *zero*, i.e., $d_i \geq 0$. This conversion flips the *sign* bits of negative floating-point values to *zero*. Subsequently, we associate

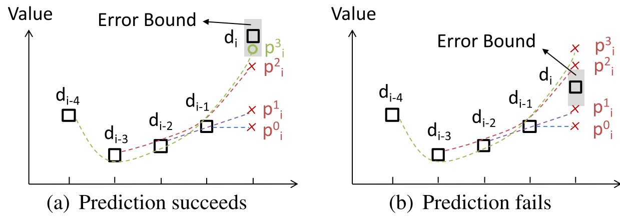


FIGURE 2. ***Error-bounded* linear prediction.**

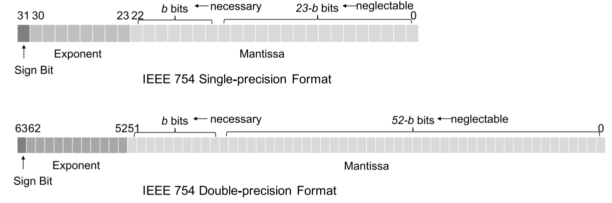


FIGURE 3. **IEEE 754 floating-point data format. The *bit-cut* compression algorithm retains the necessary *b* bits while discarding some of the least significant bits (LSBs) that are negligible.**

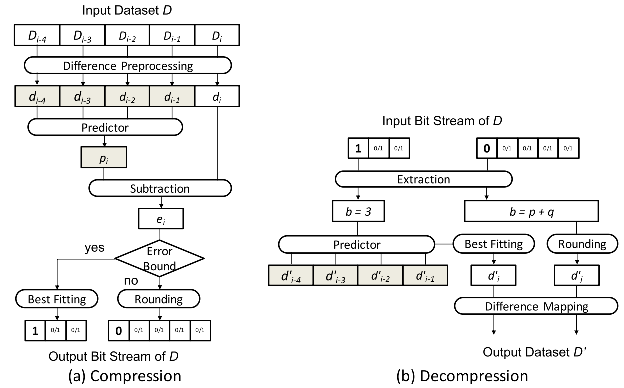


FIGURE 4. **Block diagram of the lossy *bit-zip* compression/decompression for 1-D floating-point data array.**

d_i with a pre-defined, abbreviated bit-string symbol if it can be predicted by its previous data within the *error bound*. We employ a flexible linear predictor for this lossy compression. For a one-dimensional numerical dataset d, the linear predictor predicts each element d_i using its $n+1$ preceding elements, i.e., $\{d_{i-(n+1)}, d_{i-n}, \ldots, d_{i-1}\}$.

We adopt a standard polynomial prediction, p_i, that varies based on n. Notably, we can designate q bits to express a maximum of 2^q predictions. Because the hit ratios for the predictions $n < 4$ are relatively high (see Section IV-A for detailed analysis), we limit ourselves to the first four predictions (i.e., $q = 2$). The predictions for the first four values of n are as follows:

$$p_i^0 = d_{i-1} \; (n = 0) \tag{1}$$

$$p_i^1 = 2 \times d_{i-1} - d_{i-2} \; (n = 1) \tag{2}$$

$$p_i^2 = 3 \times d_{i-1} - 3 \times d_{i-2} + d_{i-3} \; (n = 2) \tag{3}$$

$$p_i^3 = 4 \times d_{i-1} - 6 \times d_{i-2} + 4 \times d_{i-3} - d_{i-4} \; (n = 3) \tag{4}$$

In Fig. 2, we compute the differences between the true value d_i and each of the predicted values (p_i^0, p_i^1, p_i^2, and p_i^3). We then determine the value of *bestfit* as *argmin*($|p_i^0 - d_i|$, $|p_i^1 - d_i|$, $|p_i^2 - d_i|$, $|p_i^3 - d_i|$). Next, we calculate $|p_i^{bestfit} - d_i|$ and check if it satisfies the *error bound*. For simplicity, we refer to the *error bound* as the absolute error bound (E). If the prediction satisfies the error bound, i.e., $|p_i^{bestfit} - d_i| \leq E$, we represent the prediction using a predefined bit-string symbol. Note that we append a flag bit (*1*) to the beginning of the bit string to identify a linear prediction.

2) BIT CUT

If a value prediction is successful at a source node, the corresponding floating-point value is converted into a three-bit representation. In case of prediction failure, we discard the least significant bits (LSBs) of the IEEE 754 floating-point expression of the value to achieve a high compression ratio while maintaining the specified *error bound*. In other words, we only retain the required b bits in the *mantissa* to satisfy the *error bound*, as illustrated in Fig. 3. Using Equations 5 - 7, we determine the value of b for the floating-point value d_i based on the specified *error bound* E.

$$2^{-x} \leq E < 2^{-x+1} \; (x > 0) \tag{5}$$

$$2^y \leq d_i < 2^{y+1} \tag{6}$$

$$b = x + y \; (b = 0 \; if \; x + y < 0) \tag{7}$$

For example, for a double-precision floating-point value, the minimum required number of bits for the lossy *error-bounded* compression is $1 + 11 + b = 12 + b$, and the corresponding compression ratio is $64/(12 + b)$.

3) ENCODING AND DECODING

Figure 4 depicts the block diagrams of lossy *bit-zip* compression and decompression. The encoded data bits are concatenated into a continuous output bit stream, following their original order in the input dataset. This implies that our lossy compression scheme does not necessitate displacement information, thus minimizing communication overhead.

To reconstruct the data from the received bit stream in the decompression process, we first identify the leading bit of each data piece as either *0* or *1*. We also need to determine the bit length of the data piece to decode it. If the leading bit is *1*, the data piece corresponds to the linearly predicted data, and its bit length is always three. We decode the linearly predicted data by performing a calculation similar to Equations 1 - 4. On the other hand, if the leading bit is *0*, the bit length of the data piece is $12 + b$ for double-precision floating-point values. The value of b is crucial to compute the total bit length of the data piece for its decoding. Note that $b = x + y$ (Equation 7), where x is determined by the *error bound* E, and y is determined by the *exponent* bits. The lost LSBs due to compression are filled with $(1000\ldots)_2$, and the number of supplemental bits is $64 - (12 + b) = 52 - b$.

After the decoding phase, we convert the decoded dataset to the final decompressed dataset through a simple difference

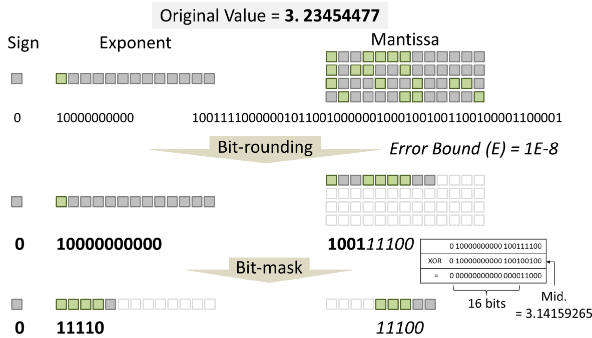


FIGURE 5. **Cooperation of the lossy *bit-zip* compression and the lossless *bit-mask* compression. In this example, a 64-bit floating-point value is compressed to 11 bits, resulting in a significant reduction in the number of bits required.**

mapping procedure, which is the inverse operation of the difference preprocessing performed in the compression phase.

B. LOSSLESS BIT-MASK COMPRESSION

We have observed that even though approximate IEEE 754 floating-point values can be compressed in the *bit-cut* step (Section III-A2), they still exhibit some leading bit sequences that repeat. Building on this observation, we propose to use a dictionary-based compression technique to replace these repeating occurrences with a series of predefined short bit-string symbols.

1) MASK SELECTION

We propose a *bit-mask* pattern to further enhance the compression ratio while minimizing the cost and decompression penalty. This technique exploits common bit sequences by identifying matches based on a few remembered bit positions. This approach is similar to previous works [29], [30]. The number of bit changes allowed in a match limits the effectiveness of the *bit-mask* compression, as more bit changes lead to more matching sequences. To balance cost and benefit, we use the median value of the dataset to create a matching sequence, which acts as a bit mask. The bit mask replaces a few bit strings with predefined shorter bit-string symbols, resulting in a compression technique that provides both high compression ratio and fast decompression.

The *bit-mask* compression technique is dictionary-based, ensuring lossless compression, and is employed as a supplement to the lossy *bit-zip* compression scheme. While the lossy *bit-zip* compression scheme discards some least significant bits (LSBs) to maintain the *error bound*, the lossless *bit-mask* compression technique compresses a few most significant bits (MSBs) without sacrificing any data precision, as illustrated in Fig. 5. By working together, the lossy *bit-zip* compression and lossless *bit-mask* compression techniques achieve a superior compression ratio for floating-point values.

Encoding 1: Linear Prediction

Compression Type	Prediction Type
1	(2 bits)

Encoding 2: Bit-cut

Compression Type	Exponent	Mantissa
0	(11 bits)	(*b* bits)

Encoding 3: Bit-mask (All Bits Match)

Compression Type	Bit-mask Flag	Bit-mask Type
0	1…1 (*f* bits)	0

Encoding 4: Bit-mask (Initial Bits Match)

Compression Type	Bit-mask Flag	Bit-mask Type	Bit-mask Position	Leftover Mantissa
0	1…1 (*f* bits)	1	(2 bits)	(*b-x* bits)

FIGURE 6. **The generic encoding scheme in our lossy bitwise compression algorithm.**

2) ENCODING AND DECODING

The compression algorithm we use employs a generic encoding scheme, which is depicted in Fig. 6. This scheme applies to both the lossy *bit-zip* compression and the lossless *bit-mask* compression. The first bit of the generic encoding indicates the compression type: a *0* means either the *bit-cut* compression or the *bit-mask* compression, while a *1* means the linear-prediction compression. Additionally, the generic encoding requires at least one flag bit set to *1* to identify the *bit-mask* encoding. The number of flag bits (f) needed depends on the maximum value (d_{max}) of the dataset (d). This ensures that the leading *exponent* bit(s) of each value in the dataset do not conflict with the all-*1* flag bits. The value of f is determined as follows:

$$f = \begin{cases} 1 & d_{max} < 2^{2^{10}-1023} \\ 2 & 2^{2^{10}-1023} \leq d_{max} < 2^{2^{10}+2^{9}-1023} \\ 3 & 2^{2^{10}+2^{9}-1023} \leq d_{max} < 2^{2^{10}+2^{9}+2^{8}-1023} \end{cases}$$

It is sufficient to use up to three *bit-mask* flag bits ($f \leq 3$) in most cases ($d_{max} < 2^{769}$). Additionally, we use one type bit to represent two types of the *bit-mask* compression: *0* indicates that no mismatch occurs for all bits after XORing the value and the bit mask, while *1* indicates that no mismatch occurs for initial bits before the bit location where mismatch, i.e., *1*, occurs. The following two position bits represent the bit location: *00*, *01*, *10*, and *11* indicate that no mismatch occurs for the initial 12, $12 + 2^1 = 14$, $12 + 2^2 = 16$, and $12 + 2^3 = 20$ bits, respectively.

In general, the lossless *bit-mask* compression is beneficial for datasets that have many values around the middle value, so that their MSBs have more identical bits to compress.

The algorithm for our lossy bitwise compression is presented in Algorithm 1. The encoded bits are concatenated to form a bit stream that will be transferred from the sender side. Figure 7 illustrates an example of the encoded bits generated by our compression algorithm, assuming that the number of flag bits is $f = 2$. Different compression types use differential prefix bits in the bit stream, which are identified for smooth decompression on the receiver side.

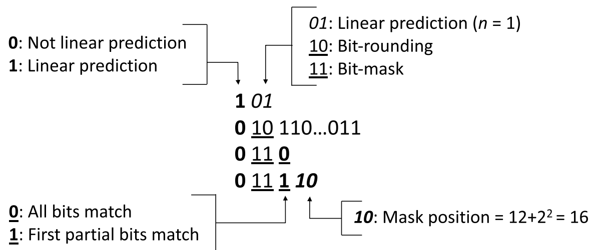


FIGURE 7. An example of the encoded bits by our compression approach (the number of *bit-mask* flag bits $f = 2$).

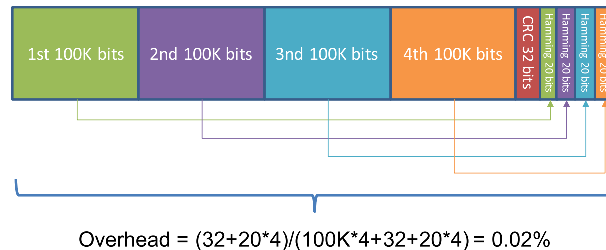


FIGURE 8. Bit-flip check and correction for a piece of 400K-bit compressed data.

Algorithm 1 The Lossy Bitwise Floating-Point Compression

Input:

1-D floating-point (FP) array $D[1 \ldots m]$, error bound E

Output:

bit-stream D_{bit} ▷ encapsulated in a byte array

1: convert $D[1 \ldots m]$ into non-negative FP dataset $d[1 \ldots m]$
2: $D_{bit} \leftarrow NULL$
3: **for** $i = 1 \rightarrow m$ **do**
4:   **if** $i \leq 3$ **then**
5:     $bs \leftarrow$ bit-cut(d_i, E)
6:     $bs \leftarrow$ bit-mask(bs)
7:     append bs to D_{bit}
8:   **else**
9:     $bestfit = \arg\min(|p_i^0 - d_i|, |p_i^1 - d_i|, |p_i^2 - d_i|, |p_i^3 - d_i|)$
10:     **if** $p_i^{bestfit} \leq E$ **then**
11:       **switch** $bestfit$ **do**
12:         **case** 0: append $(100)_2$ to D_{bit}
13:         **case** 1: append $(101)_2$ to D_{bit}
14:         **case** 2: append $(110)_2$ to D_{bit}
15:         **case** 3: append $(111)_2$ to D_{bit}
16:     **else**
17:       $bs \leftarrow$ bit-cut(d_i, E)
18:       $bs \leftarrow$ bit-mask(bs)
19:       append bs to D_{bit}
20:     **end if**
21:   **end if**
22: **end for**

C. BIT-FLIP FAULT TOLERANCE

In modern high-bandwidth interconnection networks, multi-level modulations such as pulse amplitude modulation (PAM) and quadrature amplitude modulation (QAM) are typically required. Due to their narrow design margin, bit flips often occur with a non-negligible probability. This necessitates a costly forward error correction (FEC) logic to ensure error-free communication.

In this study, we aim to develop fault-tolerant countermeasures against bit flips during transmission on high-performance interconnection networks. We utilize CRC-32 to detect bit-flip errors in the communication data and provide effective protection. In addition, we employ Hamming code to correct single bit errors on a conditional basis. To account for the higher probability of bit flips in larger bit streams during transmission, we divide the entire stream adaptively into multiple data blocks (DBs) and apply Hamming code to each DB accordingly. The data block size is determined by the bit error rate (BER) on the target high-performance interconnection network and must be small enough to tolerate a high BER. A Hamming code is typically defined as $(2^n-1, 2^n-n-1)$, where n is the number of overhead bits. All Hamming codes can detect two errors and correct one error. For instance, if the BER is 10^{-5}, the data block size is set to 10^5 bits.

In case a single bit flip is detected in a DB, the receiver can directly correct the error. However, if there are multiple bit flips, such as burst errors with the burst length ≤ 31 bits, detected in a DB, the sender is required to resend the DB. The length of the overhead bits is insignificant compared to that of the data themselves.

To illustrate, a 400K-bit communication stream using CRC-32 and Hamming code is depicted in Fig. 8, assuming a BER of 10^{-5}. CRC-32 requires 32 overhead bits for bit-flip check, while Hamming code necessitates 20 overhead bits per 100K-bit DB for bit-flip correction, requiring a total of 80 overhead bits for bit-flip correction of the entire 400K-bit communication data. In this example, the total bit-flip check and correction overhead is $32 + 20 \times 4 = 112$ bits, representing only about 0.02% of the entire communication bit stream. As the BER decreases below 10^{-5}, the data block size can be increased accordingly, leading to an even lower overhead rate.

D. MPI IMPLEMENTATION

Our lossy bitwise floating-point compression scheme is highly portable across various MPI implementations, as it employs basic MPI functions and data types. The sender only needs to transmit the size of the compressed data, constructed with a single MPI_INT, before transferring the compressed data, which is constructed with MPI_UNSIGNED_CHAR arrays. Additionally, the sender transmits the difference information, constructed with a single MPI_DOUBLE, generated during the difference preprocessing step.

For point-to-point communication, we use MPI_Isend, MPI_Irecv, and MPI_Waitall between the corresponding

processes, while for collective communication, such as broadcast, we use MPI_Bcast among the involved processes. Importantly, all the compressed data bits are concatenated seamlessly at the sender side, and the prefix bits guide smooth decompression at the receiver side, eliminating the need for any additional overhead for bit displacement information.

We utilize CRC verification at the receiver side as a bit-flip error check approach. The compressed data is decompressed only if the verification is passed. At the receiver side, Hamming code is conditionally applied to correct single-bit errors, resulting in successful decompression of the compressed data after error correction. If multiple bit flips in a single DB are detected at the receiver side, the corresponding DB must be resent by the sender.

Figure 9 illustrates the enhanced lossy bitwise floating-point compression process with the bit-flip check and correction countermeasures.

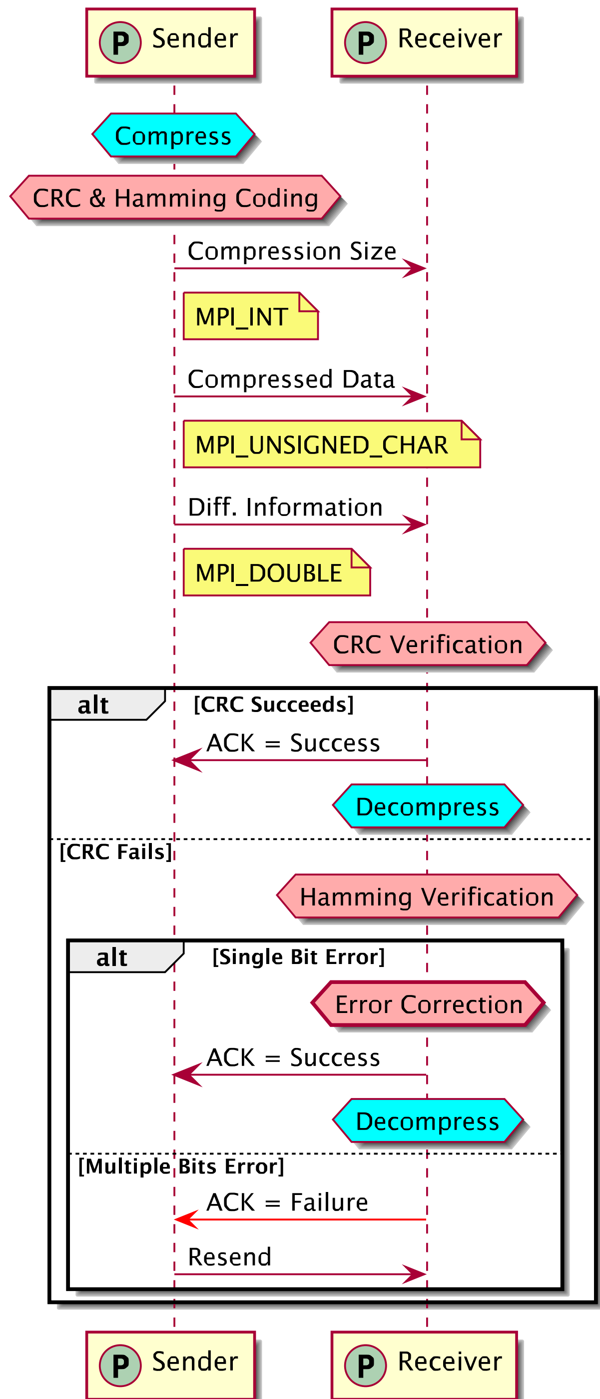


FIGURE 9. Diagram of the lossy bitwise floating-point compression enhanced by bit-flip error check and correction.

IV. EVALUATION

Firstly, we assess the efficacy and efficiency of our lossy bitwise floating-point compression algorithm. Next, we evaluate the algorithm's performance on interconnection networks, utilizing synthetic traffic patterns and representative MPI applications to determine the improvement gained from our compression technique. Lastly, we conduct an overall evaluation of the error recovery schemes to address bit-flip errors occurring during compressed data transmission on interconnection networks.

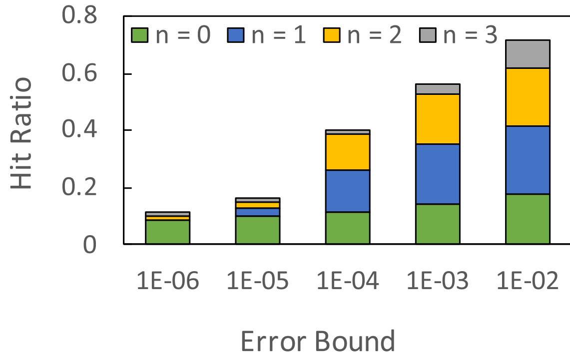
A. EFFECTIVENESS AND EFFICIENCY

We conduct the evaluation using two compute nodes equipped with an Intel Xeon Processor X5690, featuring a 3.47 GHz 12-core processor, and a GbE network interface using Broadcom NetXtreme II BCM5709 1000Base-T. Inter-process communication is facilitated by OpenMPI v3.1.3, and the nodes run on Linux Kernel 4.9.0-8-amd64.

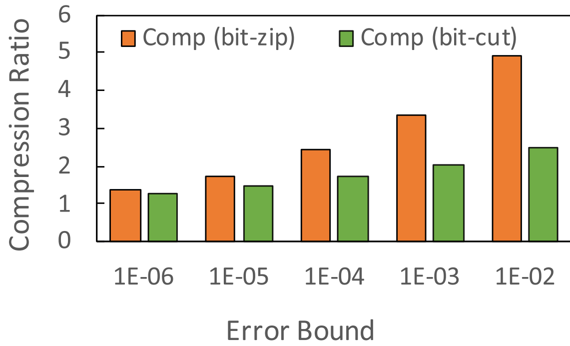
We first describe our lossy bitwise compression algorithm applied to a *Ping-pong* MPI program, which consists of two processes continuously sending messages to each other. The program uses a *ping_pong_count* initiated at zero and incremented by the sending process at each ping-pong step. The processes take turns being the sender and receiver while incrementing the *ping_pong_count*, and the program stops sending and receiving after a predetermined limit is reached (10,000 in this case). The input dataset is 65,536 samples of double-precision (64-bit) floating-point data from Blast2 [31].

Figure 10(a) displays the hit ratios of the first four linear predictions ($n = 0, 1, 2,$ and 3) in the lossy bitwise compression algorithm. The total hit ratio increases from 11.1% to 71.9% as the *error bound* is relaxed from 10^{-6} to 10^{-2}, reflecting the acceptable precision quality of the compressed data. The first three linear predictions ($n = 0, 1,$ and 2) contribute significantly to the increase in compression ratio. We use these four predictions in the lossy bitwise compression algorithm because they compress the predictable floating-point data to only three bits.

For comparison, we apply the *bit-cut* compression algorithm, which rounds bits for both predictable and unpredictable data. The algorithm does not include linear predictions even for predictable data. Figure 10(b) compares the compression ratios of the *bit-zip* and *bit-cut* compression algorithms. The compression ratio of *bit-cut* is lower than that



(a) Breakdown of hit ratios for linear predictions $n < 4$



(b) Compression ratio with and without linear predictions

FIGURE 10. Effect of linear predictions in our lossy bitwise compression algorithm.

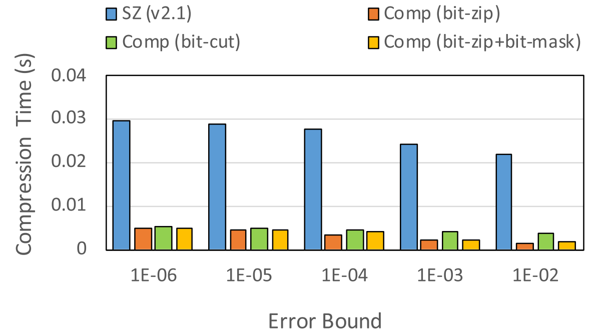
TABLE 1. Compound errors for different predefined *error bounds*.

Error Bound	Compound Error
1E-6	4.49×10^{-7}
1E-5	1.40×10^{-6}
1E-4	8.46×10^{-5}
1E-3	6.97×10^{-4}
1E-2	6.65×10^{-3}

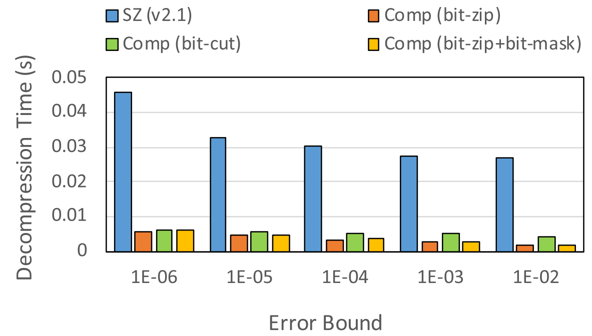
of *bit-zip*, and the gap widens for more relaxed (larger) *error bounds*. This demonstrates that the linear prediction plays a crucial role in our lossy bitwise compression algorithm by providing an extremely high compression ratio for predictable data.

We ensure that individual floating-point data values meet the specified *error bound* in our compression algorithm. However, compounding errors may occur due to the compression of consecutive values. The results of the eventual compound errors under different predefined *error bounds* are provided in Table 1. The table indicates that the eventual compound errors remain within the corresponding specified *error bounds*.

We turn our attention to the time cost of compression and decompression in our lossy bitwise compression algorithm, as depicted in Fig. 11. To provide a reference, we also evaluate the SZ (v2.1) algorithm [7], which is not directly applicable to inter-process communication on interconnection networks.



(a) Compression time



(b) Decompression time

FIGURE 11. Time cost of lossy compression algorithms.

Our results show that our lossy bitwise compression algorithm outperforms SZ in terms of speed, with a compression speedup of 8.5x and a decompression speedup of 9.1x when the *error bound* is set to 10^{-4}. Moreover, we observe that the *bit-cut* compression algorithm takes longer to compress and decompress when compared to the *bit-zip* based compression algorithms. This is because *bit-cut* applies bit rounding to each floating-point data value without using linear predictions. Therefore, the linear prediction approach provides advantages in both compression ratio and speed for our lossy bitwise compression algorithm.

B. SYNTHETIC TRAFFIC PATTERNS

We evaluate the performance of parallel application benchmarks using SimGrid (v3.21) [32], a discrete-event simulation framework. SimGrid is configured such that each switch has a 100ns delay, switches and compute nodes are interconnected via links with 200Gbps bandwidth each, and each compute node has a computation power of 5TFlops. The built-in version of the MVAPICH2 [33] implementation is used for MPI collective communications. The simulations are based on a 3-D torus [34] interconnection network with minimal routing using the Dijkstra algorithm [35]. The 3-D torus is a representative topology for interconnection networks in parallel computer systems because it provides a scalable, low-latency, high-bandwidth, fault-tolerant, and programmable network structure that is well-suited for HPC and other data-intensive applications. As an end-to-end compression framework for floating-point communication data, our compression algorithm remains unaffected by any

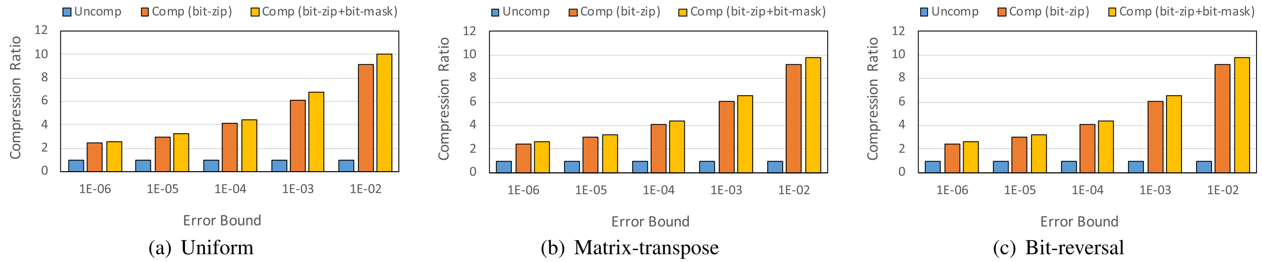


FIGURE 12. Compression ratio for synthetic traffic patterns.

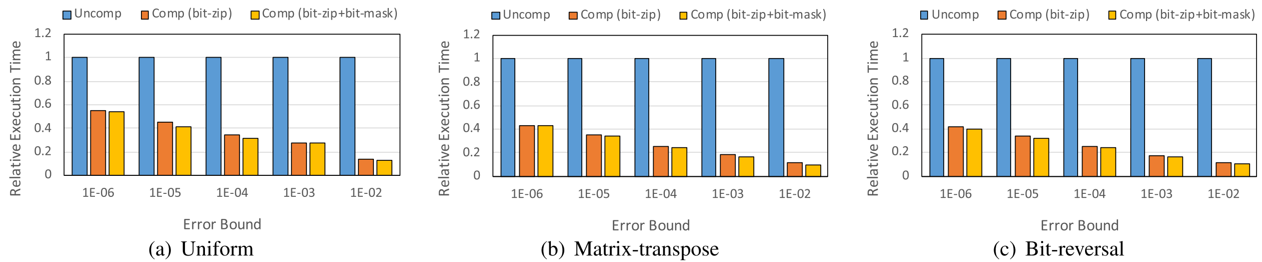


FIGURE 13. Relative execution time for synthetic traffic patterns.

particular interconnection network topology, as long as the network latency remains identical. Therefore, we exclude the discussion of its application to other network topologies such as traditional mesh and fat trees, as well as emerging circulant, ring-like, and HyperX topologies.

Synthetic traffic patterns are simulated to determine the communication node pairs, including *Uniform*, *Matrix-transpose*, and *Bit-reversal*. These traffic patterns are commonly used for measuring the performance of interconnection networks as described in [36]. Data packets are assumed to be injected into the interconnection networks independently by each node. Each process exchanges the same dataset as used in the previous section.

In this evaluation, we simulate the executions of both unmodified and modified versions of MPI parallel applications using the lossy bitwise compression algorithm. Figure 12 illustrates the compression ratio, and Figure 13 shows the execution time, using *bit-zip* compression and *bit-zip+bit-mask* compression for comparison. As the *error bound* becomes more relaxed, we observe a high improvement ratio of the compressed communication by our approaches over the original uncompressed communication for both metrics. Additionally, the *bit-mask* compression algorithm further improves the compression ratio without sacrificing execution time for different traffic patterns.

C. MPI APPLICATIONS

In this section, we evaluate the lossy bitwise compression algorithm on several communication-intensive MPI applications as described below. The datasets used in each application consist of IEEE 754 double-precision floating-point values.

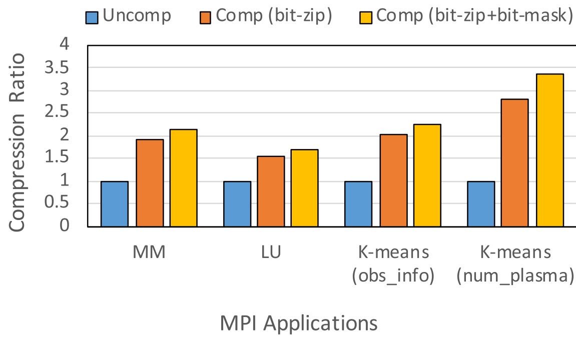
MM [37]: The *MM* (Matrix Multiplication) application aims to multiply two matrices together. Both matrices used in this evaluation have $1,024$ columns and $1,024$ lines, resulting in two $1,024 \times 1,024$ matrices.

LU [38]: The *LU* decomposition application is to decompose a square matrix A ($n \times n$) into a lower triangular matrix (L) and an upper triangular matrix (U). We assume that A is a 256×256 matrix.

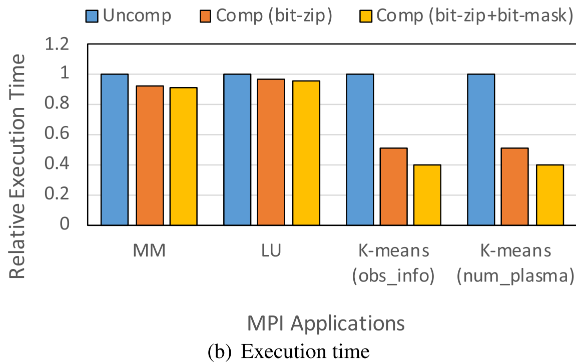
K-means [39]: The *K-means* clustering program partitions an input dataset into multiple subsets called clusters. Similar elements are grouped into the same cluster, calculated based on distance metrics such as euclidean distance or hamming distance. We assume 100 clusters and set the maximum calculation iteration to 1,000. We use the following two input datasets [40]:

- *obs_info*: the measurement from scientific instruments which comprises the latitude and the longitude information of the observation points of a weather satellite (2,366,316 values and 0.3% are unique)
- *num_plasma*: the result of numeric simulations which simulate the plasma temperature evolution of a wire array z-pinch experiment (4,386,200 values and 23.9% are unique)

The compression ratio and relative execution time of the *bit-zip* and *bit-zip+bit-mask* methods are presented in Fig. 14 for the above MPI applications, with the *error bound* set to 10^{-6}. The *bit-zip+bit-mask* method generally achieves higher compression ratios than the *bit-zip* method, resulting in faster execution times for the target applications. Specifically, for *K-means*, the *bit-zip+bit-mask* method improves the compression ratio by up to 3.4x and speeds up the execution time by around 59.7% compared to the uncompressed version.



(a) Compression ratio



(b) Execution time

FIGURE 14. Effect of integrating the lossy bitwise compression algorithm into MPI applications ($E = 10^{-6}$).

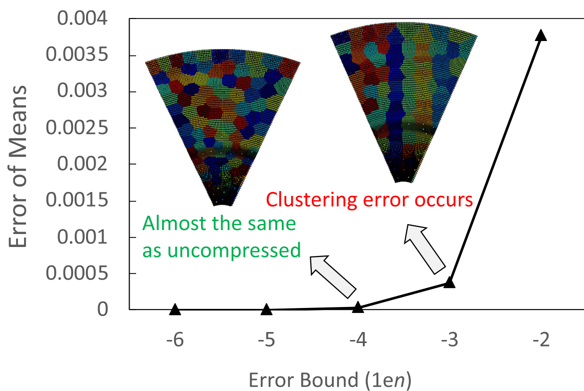


FIGURE 15. Error of *K-means* clustering (*num_plasma*) using the lossy bitwise compression for inter-process communication.

The impact of the *bit-zip+bit-mask* compression on the clustering result for the *num_plasma* dataset is presented in Fig. 15. The plot displays the average error of means in the clusters, with each cluster represented by the same color in the output. We observe that the clustering result obtained with our compression algorithm remains almost identical to the uncompressed version when the *error bound* is relaxed to 10^{-4}. However, the clustering error increases as the *error bound* exceeds 10^{-3}.

D. BIT-FLIP CHECK AND CORRECTION

1) TIME COST

We conduct an evaluation of the time cost of the bit-flip check and correction techniques, namely CRC and Hamming code. To create various data sizes, we have utilized the dataset consisting of double-precision (64-bit) floating-point numbers obtained from Blast2 [31], and split it accordingly.

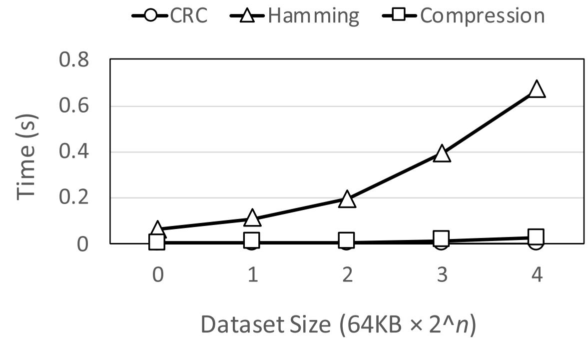


FIGURE 16. Time cost of bit-flip error check and correction.

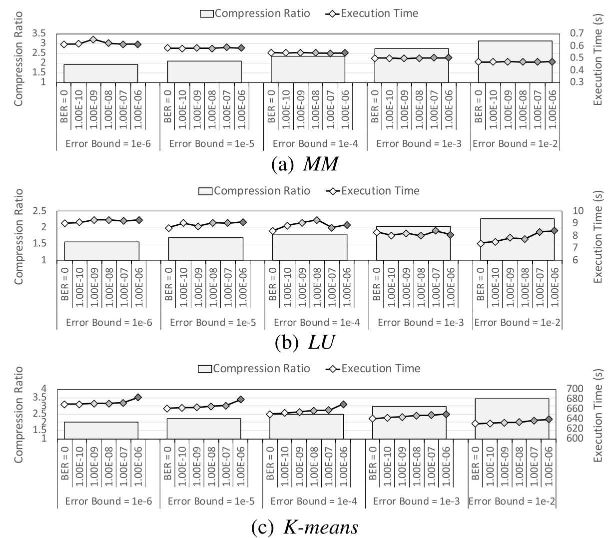


FIGURE 17. Effect of bit-flip error recovery on compressed MPI communication.

Figure 16 illustrates that, for various dataset sizes, CRC has a lower time cost than the lossy bitwise compression algorithm itself. On the other hand, Hamming code requires more time to perform bit-flip correction, particularly for larger dataset sizes. Such time cost is unacceptable for time-sensitive communication on high-performance interconnection networks. Therefore, we recommend using only CRC for bit-flip check when the dataset size is large. In this case, if any bit flip is detected on the receiver side, the errorous data block (DB) will need to be retransmitted from the sender side.

2) BENCHMARK PERFORMANCE

We have integrated bit-flip recovery techniques with the lossy bitwise compression algorithm into MPI applications such as *MM*, *LU*, and *K-means*. We use the same datasets as those utilized in Sec. IV-C. Due to the large size of the datasets used in these applications, we only rely on CRC and retransmission for bit-flip check and correction on the target interconnection network.

In Fig. 17, we present the total execution times of the MPI applications, including bit-flip check and recovery. We observe that the speedup obtained by the lossy bitwise compression algorithm gradually increases as the *error bound* becomes relaxed, which is consistent with our previous findings. Additionally, while a high BER typically compromises the speedup, the time overhead is negligible when compared to an interconnection network with $BER = 0$. This is because, in such cases, only the data blocks where bit flips occur need to be retransmitted, instead of the entire compressed data.

V. CONCLUSION

Optimizing data compression for approximate communication can enhance the effective network bandwidth on an interconnection network of parallel computers. In contrast to using hardware compression at network interfaces, we developed an application-level *error-bounded* lossy bitwise compression algorithm for floating-point communication to improve the performance of parallel applications. The compressed floating-point values are combined into a bit stream and encapsulated in a byte array, which corresponds to an MPI unsigned char type to ensure high portability. In addition, we explored error check and correction techniques to safeguard the compressed data against bit flips that may occur during transmission. The evaluation results show that our lossy bitwise compression algorithm is effective in enhancing the execution performance of parallel applications while preserving a specific *error bound* on high-performance interconnection networks.

REFERENCES

[1] A. Agrawal, J. Choi, K. Gopalakrishnan, S. Gupta, R. Nair, J. Oh, D. A. Prener, S. Shukla, V. Srinivasan, and Z. Sura, ''Approximate computing: Challenges and opportunities,'' in *Proc. IEEE Int. Conf. Rebooting Comput. (ICRC)*, Oct. 2016, pp. 1–8.

[2] S. Chunduri, S. Parker, P. Balaji, K. Harms, and K. Kumaran, ''Characterization of MPI usage on a production supercomputer,'' in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, Nov. 2018, pp. 386–400.

[3] Q. Fan, D. J. Lilja, and S. S. Sapatnekar, ''Using DCT-based approximate communication to improve MPI performance in parallel clusters,'' in *Proc. IEEE 38th Int. Perform. Comput. Commun. Conf. (IPCCC)*, Oct. 2019, pp. 1–10.

[4] T. Ueno, K. Sano, and S. Yamamoto, ''Bandwidth compression of floating-point numerical data streams for FPGA-based high-performance computing,'' *ACM Trans. Reconfigurable Technol. Syst.*, vol. 10, no. 3, pp. 1–22, May 2017.

[5] J. Tomkins, ''Interconnects: A buyers point of view,'' in *Proc. ACS Workshop*, Jun. 2007, pp. 1–12.

[6] *IEEE Standard for Floating-Point Arithmetic*, IEEE Standard 754-2019, 2019, pp. 1–84.

[7] X. Liang, S. Di, D. Tao, S. Li, S. Li, H. Guo, Z. Chen, and F. Cappello, ''Error-controlled lossy compression optimized for high compression ratios of scientific datasets,'' in *Proc. IEEE Int. Conf. Big Data (Big Data)*, Dec. 2018, pp. 438–447.

[8] *Cyclic Redundancy Checks*. Accessed: Feb. 18, 2023. [Online]. Available: https://www.mathpages.com/home/kmath458.htm

[9] D. K. Bhattacharyya and S. Nandi, ''An efficient class of SEC-DED-AUED codes,'' in *Proc. Int. Symp. Parallel Architectures, Algorithms Netw. (I-SPAN)*, 1997, pp. 410–416.

[10] M. Narasimha and A. Peterson, ''On the computation of the discrete cosine transform,'' *IEEE Trans. Commun.*, vol. C-26, no. 6, pp. 934–936, Jun. 1978.

[11] M. Heideman, D. Johnson, and C. Burrus, ''Gauss and the history of the fast Fourier transform,'' *IEEE ASSP Mag.*, vol. M-1, no. 4, pp. 14–21, Oct. 1984.

[12] A. A. Shinde and P. Kanjalkar, ''The comparison of different transform based methods for ECG data compression,'' in *Proc. Int. Conf. Signal Process., Commun., Comput. Netw. Technol.*, Jul. 2011, pp. 332–335.

[13] S. Lakshminarasimhan, N. Shah, S. Ethier, S.-H. Ku, C. Chang, S. Klasky, R. Latham, R. Ross, and N. Samatova, ''Isabela for effective in situ compression of scientific data,'' *Concurrency Comput., Pract. Exp.*, vol. 25, pp. 524–540, Feb. 2013.

[14] B. Dickov, M. Pericàs, P. M. Carpenter, N. Navarro, and E. Ayguadé, ''Analyzing performance improvements and energy savings in infiniband architecture using network compression,'' in *Proc. IEEE 26th Int. Symp. Comput. Archit. High Perform. Comput.*, Oct. 2014, pp. 73–80.

[15] C. C. Cutler, ''Differential quantization of communication signals,'' U.S. Patent 2 605 361, Jul. 29, 1952.

[16] L. Deng and D. O'Shaughnessy, *Speech Processing: A Dynamic and Optimization-Oriented Approach*. New York, NY, USA: Marcel Dekker, 2003, pp. 41–48.

[17] P. Lindstrom and M. Isenburg, ''Fast and efficient compression of floating-point data,'' *IEEE Trans. Vis. Comput. Graphics*, vol. 12, no. 5, pp. 1245–1250, Sep. 2006.

[18] P. Lindstrom, ''Fixed-rate compressed floating-point arrays,'' *IEEE Trans. Vis. Comput. Graphics*, vol. 20, no. 12, pp. 2674–2683, Dec. 2014.

[19] P. Ratanaworabhan, J. Ke, and M. Burtscher, ''Fast lossless compression of scientific floating-point data,'' in *Proc. Data Compress. Conf. (DCC)*, 2006, pp. 133–142.

[20] M. Burtscher and P. Ratanaworabhan, ''High throughput compression of double-precision floating-point data,'' in *Proc. Data Compress. Conf. (DCC)*, 2007, pp. 293–302.

[21] M. Burtscher and P. Ratanaworabhan, ''FPC: A high-speed compressor for double-precision floating-point data,'' *IEEE Trans. Comput.*, vol. 58, no. 1, pp. 18–31, Jan. 2009.

[22] J. Liu, S. Li, S. Di, X. Liang, K. Zhao, D. Tao, Z. Chen, and F. Cappello, ''Improving lossy compression for SZ by exploring the best-fit lossless compression techniques,'' in *Proc. IEEE Int. Conf. Big Data (Big Data)*, Dec. 2021, pp. 2986–2991.

[23] S. Li, S. Di, K. Zhao, X. Liang, Z. Chen, and F. Cappello, ''Resilient error-bounded lossy compressor for data transfer,'' in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, Nov. 2021, pp. 1–14.

[24] X. Liang, K. Zhao, S. Di, S. Li, R. Underwood, A. M. Gok, J. Tian, J. Deng, J. C. Calhoun, D. Tao, Z. Chen, and F. Cappello, ''SZ3: A modular framework for composing prediction-based error-bounded lossy compressors,'' *IEEE Trans. Big Data*, vol. 9, no. 2, pp. 485–498, Apr. 2023.

[25] X. Liang, S. Di, S. Li, D. Tao, B. Nicolae, Z. Chen, and F. Cappello, ''Significantly improving lossy compression quality based on an optimized hybrid prediction model,'' in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, Nov. 2019, p. 33.

[26] N. Sasaki, K. Sato, T. Endo, and S. Matsuoka, ''Exploration of lossy compression for application-level checkpoint/restart,'' in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, May 2015, pp. 914–922.

[27] Y. Hu and M. Koibuchi, ''The case for error-bounded lossy floating-point data compression on interconnection networks,'' in *Proc. Comput. Sci. Inf. Technol. (CS IT)*, May 2021, pp. 55–76.

[28] V. Engelson, D. Fritzson, and P. Fritzson, ''Lossless compression of high-volume numerical data from simulations,'' in *Proc. Data Compress. Conf.*, 2000, pp. 574–586.

[29] S.-W. Seong and P. Mishra, ''Bitmask-based code compression for embedded systems,'' *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 27, no. 4, pp. 673–685, Apr. 2008.

[30] L. A. B. Gomez and F. Cappello, ''Improving floating point compression through binary masks,'' in *Proc. IEEE Int. Conf. Big Data*, Oct. 2013, pp. 326–331.

[31] P. Colella and P. R. Woodward, ''The piecewise parabolic method (PPM) for gas-dynamical simulations,'' *J. Comput. Phys.*, vol. 54, no. 1, pp. 174–201, Apr. 1984.

[32] H. Casanova, A. Giersch, A. Legrand, M. Quinson, and F. Suter, ''Versatile, scalable, and accurate simulation of distributed applications and platforms,'' *J. Parallel Distrib. Comput.*, vol. 74, no. 10, pp. 2899–2917, Oct. 2014.
[33] D. K. Panda, H. Subramoni, C.-H. Chu, and M. Bayatpour, ''The MVAPICH project: Transforming research into high-performance MPI library for HPC community,'' *J. Comput. Sci.*, vol. 52, May 2021, Art. no. 101208.
[34] C. Albing, N. Troullier, S. Whalen, R. Olson, J. Glenski, H. Pritchard, and H. Mills, ''Scalable node allocation for improved performance in regular and anisotropic 3D torus supercomputers,'' in *Recent Advances in the Message Passing Interface*, Y. Cotronis, A. Danalis, D. S. Nikolopoulos, and J. Dongarra, Eds. Berlin, Germany: Springer, 2011, pp. 61–70.
[35] *Dijkstra's Algorithm*. Accessed: Feb. 18, 2023. [Online]. Available: https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm
[36] J. Duato, S. Yalamanchili, and L. Ni, *Interconnection Networks: An Engineering Approach*. San Mateo, CA, USA: Morgan Kaufmann, 2002.
[37] *Matrix Multiplication*. Accessed: Feb. 18, 2023. [Online]. Available: https://en.wikipedia.org/wiki/Matrix_multiplication
[38] *Lu Decomposition*. Accessed: Feb. 18, 2023. [Online]. Available: https://en.wikipedia.org/wiki/LU_decomposition
[39] *K-Means Clustering: A Distributed MPI Implementation*. Accessed: Feb. 18, 2023. [Online]. Available: https://github.com/dzdao/k-means-clustering-mpi
[40] *Scientific IEEE 754 64-Bit Double-Precision Floating-Point Datasets*. Accessed: Feb. 18, 2023. [Online]. Available: https://userweb.cs.txstate.edu/burtscher/research/datasets/FPdouble/

YAO HU (Member, IEEE) received the M.S. degree from the Beijing University of Posts and Telecommunications, China, in 2009, and the Ph.D. degree from the Department of Computer Science and Engineering, Waseda University, Tokyo, Japan, in 2015. He is currently an Assistant Professor with Keio University, Yokohama, Kanagawa, Japan. His main research interests include high-performance computing and graph computation.