THE CASE FOR ERROR-BOUNDED LOSSY FLOATING-POINT DATA COMPRESSION ON INTERCONNECTION NETWORKS

Yao Hu and Michihiro Koibuchi

National Institute of Informatics, 2-1-2, Hitotsubashi, Chiyoda-ku, Tokyo, Japan

ABSTRACT

Data compression virtually increases the effective network bandwidth on an interconnection network of parallel computers. Although a floating-point dataset is frequently exchanged between compute nodes in parallel applications, its compression ratio often becomes low when using simple lossless compression algorithms. In this study, we aggressively introduce a lossy compression algorithm for floating-point values on interconnection networks. We take an application-level compression for providing high portability: a source process compresses communication datasets at an MPI parallel program, and a destination process decompresses them. Since recent interconnection networks are latency-sensitive, sophisticated lossy compression techniques that introduce large compression overhead are not suitable for compressing communication data. In this context, we apply a linear predictor with the user-defined error bound to the compression of communication datasets. We design, implement, and evaluate the compression technique for the floating-point communication datasets generated in MPI parallel programs, i.e., Ping Pong, Himeno, K-means Clustering, and Fast Fourier Transform (FFT). The proposed compression technique achieves 2.4x, 6.6x, 4.3x and 2.7x compression ratio for Ping Pong, Himeno, K-means and FFT at the cost of the moderate decrease of quality of results (error bound is 10^{-4}), thus achieving 2.1x, 1.7x, 2.0x and 2.4x speedup of the execution time, respectively. More generally, our cycle-accurate network simulation shows that a high compression ratio provides comparably low communication latency, and significantly improves effective network throughput on typical synthetic traffic patterns when compared to no data compression on a conventional interconnection network.

KEYWORDS

Interconnection Network, Lossy Compression, Floating-point Number, Linear Predictor, High-performance Computing (HPC).

1. INTRODUCTION

The network bandwidth becomes one of the primary concerns on recent parallel computers, because its annual improvement is moderate compared to that of computation power in compute nodes. A way to virtually increase the network bandwidth is the reduction of redundancy of communication data themselves. Some parallel scientific applications repeatedly generate similar communication data [1]. For this kind of communication, each compute node has a chance to reduce traffic by compressing communication data, e.g., by sending only the information of difference from the prior data.

A floating-point dataset is frequently exchanged between compute nodes in parallel applications. However, its compression ratio generally becomes low when using a simple lossless compression algorithm. A complicated lossless compression algorithm would have a high compression ratio.

David C. Wyld et al. (Eds): CCSIT, SIPP, PDCTA, AISC, NLPCL, BIGML, NCWMC - 2021
pp. 55-76, 2021. CS & IT - CSCP 2021 DOI: 10.5121/csit.2021.110706

However, it has a large latency overhead to compress communication data. It is not suitable for interconnection networks, because HPC interconnection networks are latency-sensitive, e.g., less than one microsecond for inter-process communication [2].

In this study, we aggressively apply a fast lossy compression algorithm to IEEE 754 floating-point communication data on interconnection networks. Generally, a lossy compression achieves a higher compression ratio than that by a counterpart lossless compression for a given dataset.

We take an application-level compression for providing high portability. In parallel programs, floating-point values are compressed before they are sent at the source side, and exchanged to relevant processes. They are then decompressed at the receiver side. The compression algorithm relies on value predictions for floating-point values, like SZ [3]. We provide byte- and bit-wise compression techniques to MPI (message passing interface) implementation. If the value prediction succeeds in the byte-wise compression, the floating-point value is converted to a single byte expression, corresponding to an MPI char type. Otherwise, the original value is not compressed, and it is transferred as it is. Although the byte-wise compression has low compression-operation latency, its upper bound of the compression ratio is not high, i.e., obviously four and eight for single-precision and double-precision floating-point values, respectively. By contrast, the bit-wise compression generates a bitstream encapsulated in a byte array, corresponding to an MPI char type as well. If the value prediction succeeds at a source, the floating-point value is converted to three bits. Even if the value prediction fails, the least significant bits (LSBs) are discarded from the IEEE 754 floating-point expression of the value to obtain a relatively high compression ratio, while maintaining a given error bound.

In this study, the loss during data compression is restricted within a specified absolute error bound to have acceptable execution results for target MPI applications. Both approaches work well in cases where subsequent data correlate reasonably with earlier data, which is often the case for the intermediate and final floating-point results produced by some scientific programs. In this case, approximate communication can be applied to error-resilient applications for speedup while maintaining the accuracy of the output at an acceptable level. In our study, several representative MPI-based error-resilient applications are used for the evaluation of the proposed lossy compression algorithm. Their computing tasks either do not aim at an exact numerical answer or they have inherent resilience to output error.

Our main contributions in this work are as follows:

- We designed, implemented, and evaluated byte- and bit-wise lossy application-level compression techniques based on several predication models for floating-point communication values in MPI parallel applications.
- We obtained 2.4x, 6.6x, 4.3x and 2.7x compression ratio (error bound is 10^{-4}) for Ping Pong, Himeno, K-means clustering and FFT applications, thus speeding up the execution time by 2.1x, 1.7x, 2.0x and 2.4x, respectively.
- From the network point of view, our cycle-accurate network simulation shows that, when the compression ratios become 1.5, 3.0 and 6.0, the effective network throughput is improved by 133%, 176% and 260%, respectively.

The rest of this work is organized as follows. Background information and related work are discussed in Section 2. Section 3 presents the byte- and bit-wise compression techniques. Section 4 shows evaluation methodology and results. Section 5 concludes with a summary of our findings in this work.

2. BACK GROUND AND RELATED WORK

2.1. Lossy Compression

Lossy compression or irreversible compression is a class of data encoding methods, which uses inexact approximation and partial data discarding to represent the content. Lossy compression is opposed to lossless compression which does not degrade the data precision or quality. The compression ratio of lossy compression is usually higher than that of lossless compression due to the data precision loss to a limited extent.

A widely used lossy compression algorithm is the discrete cosine transform (DCT) [4], which is most commonly used to compress multimedia data such as audio, video and images. Another famous lossy compression algorithm of the transform type is fast Fourier transform (FFT) [5], which converts a signal from its original domain (often time or space) to a representation in the frequency domain and vice versa.

Lossy compression of the predictive type was applied to differential pulse-code modulation (DPCM) [6], which is used as a signal encoder that uses the baseline of pulse-code modulation (PCM) but adds some functionalities based on the prediction of the samples of the signal. Another proposed lossy compression algorithm called linear predictive coding (LPC) [7] is widely used in speech coding and speech synthesis, using the information of a linear predictive model to represent the spectral envelope of a digital signal of speech in compressed form in audio signal processing and speech processing. FPZIP [8] was primarily designed for lossless compression but also has provision for lossy compression to achieve a high compression ratio. It predicts the data by a subset of encoded data and maps the difference to an integer number. For lossy compression, ZFP [9] often outperforms FPZIP by dividing 3-D floating-point arrays into small and fixed-size blocks of dimensions to achieve a higher compression ratio. A more complicated lossy compression algorithm called ISABELA [10] can achieve a higher compression ratio by transforming data layout such as sorting, cubic B-splines fitting and window splitting.

The prediction techniques based on the preceding data elements have received much attention for both lossless compression and lossy compression in recent years. For instance, the works [11] [12] [13] extend predictive coding to floating-point data and compress floating-point values without loss. The predicted and the actual floating-point values are broken up into sign, exponent and mantissa, and their corrections are compressed separately with context-based arithmetic coding. FPC [14] [15] [16] is a high-speed compressor for double-precision floating-point data. In FPC, the data are predicted by using FCM/DFCM (differential-finite-context-method predictor) and selecting closer values to the true ones. The bit-wise XOR operations are then performed between the predicted values and true values. Finally, the leading zeros in the result are compressed to less (e.g., 4) bits. SZ [3] [17] [18] is proposed for lossy compression by predicting data using three curve-fitting models, i.e., preceding-neighbor fitting model, linear-curve fitting model and quadratic-curve fitting model. The essential idea of SZ is to use linear predictive coding for predictable data and to perform complicated binary analysis for unpredictable data. Another work [1] presented a similar idea of floating-point data compression for FPGA-based high-performance computing by using a one-dimensional polynomial predictor.
The above lossy compression algorithms usually have a tradeoff between the compression latency overhead and the compression ratio. Most of them are historically optimized for the purpose of storing compact data in storage. Even for recent HPC and cloud purposes, the main target is still to compress data on storage [19], e.g., storing checkpoint images at the cost of trivial decrease on quality of results [20]. Our target, the interconnection network, is radically different

from that assumed in most of existing compression algorithms. Compressing data for inter-process communication in parallel programs is latency-sensitive when compared to that for storage. A recent work [34] proposed a DCT-based approximate communication scheme to reduce communication overhead without a considerable quality loss of the result. It obtains a good balance between compression speed and compression ratio. However, its limitation is that it is only useful for non-random message patterns. To the best of our knowledge, this study is the first challenge to apply a prediction-based lossy data compression algorithm at a program level to the inter-process communication datasets generated in MPI parallel applications.

2.2. Data Compression on Interconnection Networks

Besides off-chip interconnection networks, there are some prior works on data compression techniques targeting interconnection networks on a chip. Lossless frequent-pattern compression (FPC) is a significance-based compression scheme for on-chip communication to cache [21]. It compresses only some data patterns, such as leading 0s and 1s in data streams on the network-interface hardware. FPC introduces a variable-length cache line, and its essential design is to handle variable cache line sizes [22]. It enables low compression overhead, e.g., one or two clock cycles. FPC is efficient for fixed-point values, e.g., integer numbers. However, it does not work well for IEEE 754 floating-point values. This is because a floating-point value includes many mantissa bits that hardly include locality. The work [23] on lossy compression of floating-point data provides a fast lossy compression scheme which simply truncates the 16, 24 or 32 least-significant bits to save total link energy. However, the error bound is not guaranteed during compression especially for near-zero small floating-point values, which is distinct from our error-bounded lossy compression techniques.

3. ERROR-BOUND LOSSY FLOATING-POINT COMPRESSION TECHNIQUES

In this section, we present byte- and bit-wise prediction-based compression techniques that have different tradeoffs between the compression latency overhead and the compression ratio.

3.1. Linear-predictive Byte-wise Compression

A compression algorithm often encodes the differences or XORs between the given input data and the one predicted by using previous data. In this study, we map the input data to predefined symbols if the given data can be predicted by their previous data within the error bound.

3.1.1. Design

Based on how prediction is actually performed, prediction-based algorithms are classified into two groups: arithmetic-based algorithms [11] [12] [13] [8] and context-based algorithms [14] [15] [16]. Arithmetic-based algorithms use an arithmetic predictor to obtain prediction via calculations, whereas context-based algorithms use hash tables to look up data that appear after the same input phrase to predict the next input. We choose the arithmetic-based algorithmic approach because our target is IEEE 754 (single- and double-precision) floating-point formats with almost no bit-level locality for expressing two similar values. Although the arithmetic-based prediction requires buffer memory to retain previous inputs, the size of the buffer can be reduced by employing one-dimensional polynomial predictions [1].

We use a versatile linear predictor for lossy compression [24] [3] [1]. For a one-dimensional numerical dataset $d = \{d_1, d_2, \ldots, d_M\}$, the linear predictor predicts each element d_i by its $n + 1$ preceding elements $\{d_{i-(n+1)}, \ldots, d_{i-1}\}$ within the given error bound. We use typical polynomial

predictions, p_i, according to varying n. The predictions for the first four values of n are as follows:

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

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

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

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

The byte-wise compression is illustrated in Algorithm 1. The basic idea is borrowed from the bestfit curve-fitting algorithm in [3]. In Line 3, the conversion is performed to handle bit strings of floating-point values such that each element of d is not less than zero, i.e., $d_i \geq 0$. This conversion ensures that the sign bits of negative floating-point values are flipped to zero, which improves the performance of the linear prediction.

Algorithm 1 The byte-wise compression.

Input:
1-D floating-point array D, and user-defined error bound E
Output:
1-D byte array D_{cmp}, 1-D floating-point array D_{uncmp}, and 1-D integer array D_{index}

```
1:  for i = 1 → M do
2:      /* Difference Preprocessing */
3:      convert D_i into non-negative floating-point d_i
4:      bestfit = arg min (|p_i^0 − d_i|, |p_i^1 − d_i|, |p_i^2 − d_i|, |p_i^3 − d_i|)
5:      if p_i^bestfit ≤ E then
6:          switch bestfit do
7:              case 0: append (00)_16 to D_cmp /* 1 byte */
8:              case 1: append (01)_16 to D_cmp
9:              case 2: append (02)_16 to D_cmp
10:             case 3: append (03)_16 to D_cmp
11:         append i to D_index
12:     else
13:         append d_i to D_uncmp /* 4 or 8 bytes */
14:     end if
15: end for
```

The character symbol can be coded in a single byte, e.g., ASCII code, for the linear prediction of an input original floating-point value (4 bytes for single precision and 8 bytes for double precision). We can assign different character symbols to express $2^8 = 256$ predictions at maximum in Line 6. However, since the hit ratios for $n = 0, 1, 2, 3$ are relatively high, we simply attempt the first four predictions described in Equations 1 - 4 (the detail is evaluated in Section 4.1.2.).

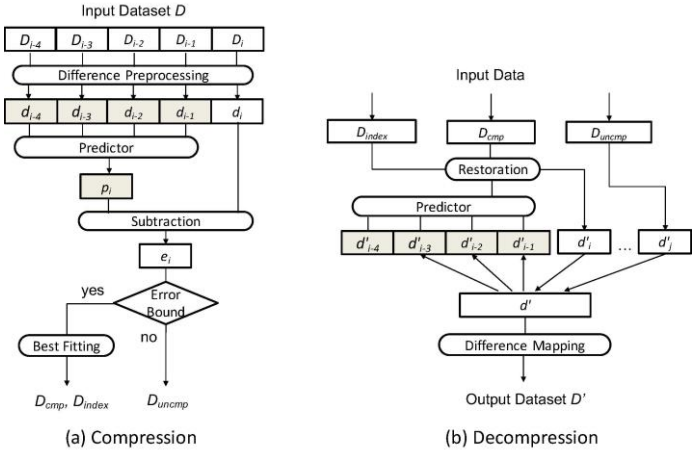


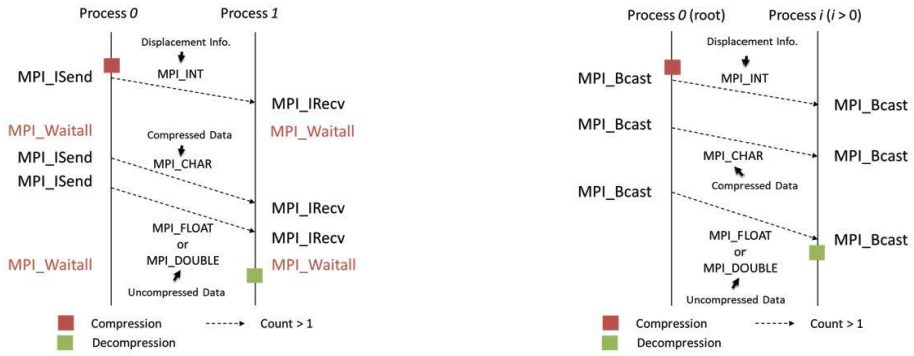
Figure 1. Block diagram of the byte-wise (de)compression for 1-D floating-point array.

Notice that D_{index} represents the displacement information of the predicted data. To identify d_i is stored in D_{cmp} or D_{uncmp}, the displacement information is needed for its smooth decompression at the receiver side. The block diagram of the byte-wise (de)compression is illustrated in Fig. 1. Since the decompression operation is performed as opposed to the compression operation, we omit to describe the decompression algorithm like Algorithm 1.

3.1.2. Implementation

We describe the implementation of the byte-wise compression for two basic MPI communication mechanisms. The first kind of communication is the point-to-point communication. The second kind of communication is the collective communication established amongst a group of processes.

MPI is a standard upon which many implementations exist. The byte-wise compression uses basic MPI functions such as MPI_Isend, MPI_Irecv and MPI_Waitall, and basic MPI data types such as MPI_INT, MPI_FLOAT/DOUBLE and MPI_UNSIGNED_CHAR, thus providing high portability to various MPI implementations.



(a) Point-to-point (b) Collective (broadcast)

Figure 2. The byte-wise compression to MPI messages.

Fig. 2(a) shows the diagram of adding the byte-wise compression to MPI point-to-point messages. This example assumes that process 0 sends messages to process 1. Because the compressed data and the uncompressed data are transferred separately, the receiver process should be aware of their respective counts to provide the corresponding receive buffers in advance. For this reason, after the sender completes data compression, it first sends the displacement information (constructed with MPI_INT) of the compressed data, which is essential for data decompression at the receiver side. The sender then transfers the compressed data (constructed with MPI_UNSIGNED_CHAR) and the uncompressed data (constructed with MPI_FLOAT or MPI_DOUBLE). After receiving the data, the receiver performs the decompression by using the displacement information. Fig. 2(b) shows the example of the implementation of adding the byte-wise compression to MPI collective (broadcast) messages. This example assumes that process 0 is the root process and it simultaneously sends messages to other processes. The flow is similar to that in the point-to-point communication.

3.2. Linear-predictive Bit-wise Compression

In this subsection, we improve the compression ratio by a bit-wise compression technique at the cost of a moderate increase of compression complexity.

3.2.1. Approximation of IEEE 754 Floating-point Values

We describe the background information on rounding IEEE 754 floating-point values. IEEE 754 standard specifies a single-precision floating-point format for a 32-bit value, which consists of 1 sign bit, 8 exponent bits and 23 mantissa bits, and a double-precision floating-point format for a 64-bit value, which consists of 1 sign bit, 11 exponent bits and 52 mantissa bits [25]. A floating-point value is computed by $sign \times mantissa \times 2^{exponent}$, where the sign is 1 or -1 if the leading bit is 0 or 1, respectively. The mantissa expresses a real value between 1.0 and 2.0, with a fractional part represented in binary format. For the single-precision format, the exponent equals the 8 bits in the middle minus 127; for the double-precision format, the exponent equals the 11 bits in the middle minus 1,023. For example, the single-precision value (0, 10000000, $10010010000111111010000)_2$ expresses $(-1)^0 \times 1.570795\ldots \times 2^{128-127} \approx (3.14159)_{10}$.

The least significant bits (LSBs) in the mantissa have little impact on the value of a floating-point number. In the bit-wise compression, we only retain the necessary b bits in the mantissa while maintaining the user-defined error bound. Thus, we aggressively discard the last $23 - b$ bits for the single-precision format, $52 - b$ bits for the double-precision format, as neglectable LSBs in the mantissa. In the above example, discarding the last 10 bits, e.g., setting the last 10 bits to 0s, will make the value 3.1413574 with 0.0074% error. If the error bound is set to 10^{-3}, then this inexact value is acceptable in terms of data precision.

It is obvious that the number of necessary bits in the mantissa, i.e., b, depends on the error bound. We figure out the value of b of the floating-point number d_i according to the predefined error bound E. First, we determine the integer value of n where $2^{-n} \leq E < 2^{-n+1}$ ($n > 0$). Then, we calculate $b = m + n$ where $2^m \leq d_i < 2^{m+1}$. Here, if $b < 0$, we set $b = 0$. The least necessary number of bits for the error-bounded compression is $1 + 8 + b = 9 + b$ for a single-precision floating-point value, and $1 + 11 + b = 12 + b$ for a double-precision floating-point value. Therefore, we can get the compression ratio, $32/(9 + b)$ for a single-precision floating-point value, and $64/(12 + b)$ for a double-precision floating-point value. Obviously, the double precision floating-point data benefits more from the lossy bit-wise compression due to a larger compression ratio.

3.2.2. Design

Algorithm 2 The bit-wise compression.

Input:
1-D floating-point array D, and user-defined error bound E
Output:
bitstream D_{bit} (encapsulated in a byte array)

```
1:  for i = 1 → M do
2:      convert D_i into non-negative floating-point d_i
3:      bestfit = arg min (|p_i^0 − d_i|, |p_i^1 − d_i|, |p_i^2 − d_i|, |p_i^3 − d_i|)
4:      if p_i^bestfit ≤ E then
5:          switch bestfit do
6:              case 0: append (100)_2 to D_bit
7:              case 1: append (101)_2 to D_bit
8:              case 2: append (110)_2 to D_bit
9:              case 3: append (111)_2 to D_bit
10:     else
11:         cut the LSBs, and append the remaining bits to D_bit
12:     end if
13: end for
```

We describe the bit-wise compression in Algorithm 2. The concept of using the linear prediction is the same as the byte-wise compression. For the smooth decompression, we convert the input floating-point values to non-negative values in Line 2 to ensure that the sign bits of negative

floating-point values are flipped to zero like in the byte-wise compression. In Line 11, we discard the maximum length of LSBs under the condition that the user-defined error bound is satisfied.

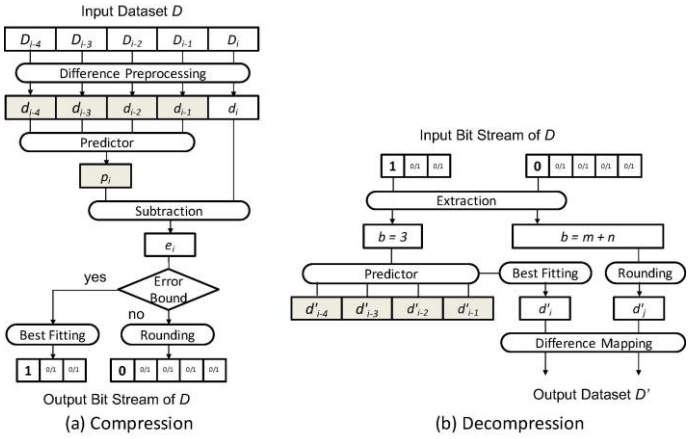


Figure 3. Block diagram of the bit-wise (de)compression for 1-D floating-point array.

Figure 3 illustrates the block diagram of the bit-wise data compression and decompression, respectively. This procedure of the linear prediction is similar to that in the byte-wise compression described in Section III-A. If the prediction fails ($p_i^{bestfit} > E$), the original value is compressed to a variable bit length, i.e., $9 + b$ bits for a single-precision floating-point value and $12 + b$ bits for a double-precision floating-point value.

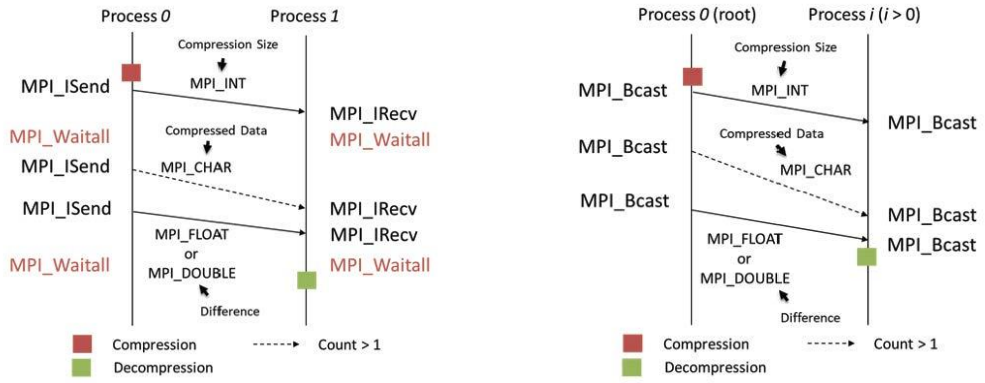
All these bits are concatenated to a continuous output bitstream. Note that the encoded data bits can be organized in the output bitstream in accordance with their original order in the input dataset. In other words, the bit-wise compression technique does not require any displacement information like in the byte-wise compression, and thus reduces the communication overhead.

For the bit-wise decompression, we can easily reconstruct the linearly predicted data and the bit-wise compressed data extracted from the received bitstream. First, we recognize the leading bit of each data piece, 1 or 0. To decode the data piece, it is also indispensable to know the bit length of the data piece, i.e., how many bits follow the leading bit. If the leading bit is 1, the data piece belongs to the linearly predicted data, thus the bit length is always 3. Then we decode the linearly predicted data by applying the calculation similar to Equations 1 - 4. Otherwise, if the leading bit is 0, the data piece belongs to the bit-wise compressed data, thus the bit length is $9 + b$ for single-precision floating-point data and $12 + b$ for double-precision floating-point data. In this case, the value of b is the key to calculate the total bit length of the data piece for its decoding. Remember that the value of b depends on the value of the exponent, thus we can get $b = m + n$, where $2^{-n} \leq E < 2^{-n+1}$ ($n > 0$) and the value of m can be obtained from the exponent bits. The lost LSBs in the data piece are padded with $(1000\ldots)_2$. For single-precision floating-point data, the number of supplemental bits is $23 - b$; for double-precision floating-point data, the number of supplemental bits is $52 - b$.

After the decoding phase, we perform a simple difference mapping procedure to convert the decoded dataset to the final decompressed dataset, which is the inverse operation of the difference preprocessing in the compression phase.

3.2.3. Implementation

The bit-wise compression uses basic MPI functions such as MPI_Isend, MPI_Irecv, and MPI_Waitall, and basic MPI data types such as MPI_INT, MPI_FLOAT/DOUBLE and MPI_UNSIGNED_CHAR for high portability to various MPI implementations.



(a) Point-to-point (b) Collective (broadcast)

Figure 4. The bit-wise compression to MPI messages.

Figure 4(a) shows the implementation of adding the bit-wise compression to MPI point-to-point messages. For the bit-wise compression, the sender only needs to send the size of the compressed data (constructed with a single MPI_INT) before transferring the compressed data (constructed with MPI_UNSIGNED_CHAR). Afterwards, the sender transfers the difference information (constructed with a single MPI_FLOAT or MPI_DOUBLE) generated at the difference preprocessing phase for decompression at the receiver side. Obviously, the bit-wise compression exchanges smaller amount of MPI communication messages when compared to the byte-wise compression. Figure 4(b) shows the implementation of adding the bit-wise compression to MPI collective (broadcast) messages.

4. EVALUATION

We firstly perform the byte- and bit-wise compression techniques on a real machine consisting of two compute nodes. We secondly perform them on an event-driven system simulator assuming a modern 64-node system. We finally investigate the impact of the compression ratio on the network throughput and communication latency using a cycle-accurate network simulator.

4.1. Parallel Application Performance on A Real Machine

4.1.1. Condition

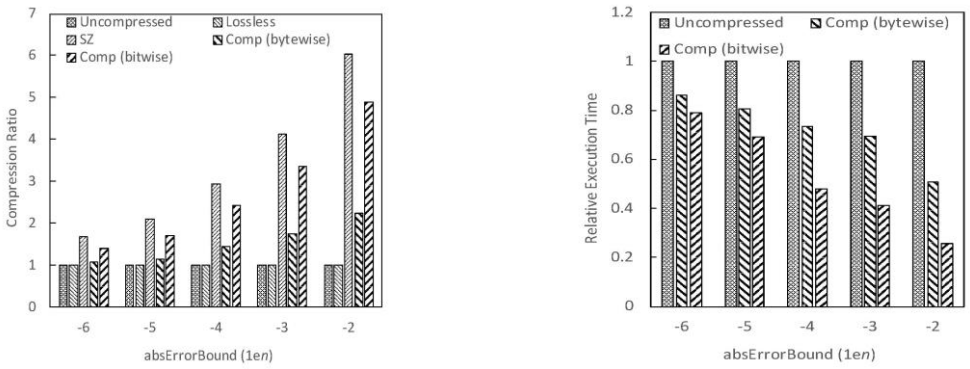
We perform the evaluation using the MPI applications running on two compute nodes in which Intel Xeon Processor X5690 is equipped with a 3.47 GHz 12-core processor. The two compute nodes have a GbE network interface, Broadcom NetXtreme II BCM5709 1000Base-T. We use OpenMPI v3.1.3 for inter-process communication on Linux Kernel 4.9.0-8-amd64. Table 1 describes the communication data types generated in the evaluated MPI applications.

Table 1. MPI parallel applications used in the evaluation.

Application	Data Type	Message Count
Ping Pong	MPI_FLOAT	8,192
Himeno	MPI_FLOAT	16,384
K-means Clustering	MPI_DOUBLE	2,366,316 (obs_info) 4,386,200 (num_plasma)
FFT	MPI_DOUBLE	2,101,248

4.1.2. Ping Pong

We introduce our floating-point data compression to a ping-pong MPI program. Two processes use MPI_Send and MPI_Recv to continually bounce messages off of each other until they decide to stop. A ping_pong_count is initiated to zero, and it is incremented at each Ping Pong step by the sending process. As the ping_pong_count is incremented, the processes take turns being the sender and receiver. Finally, after the limit is reached (10,000 in this study), the processes stop sending and receiving. We use 8,192 samplings (at time step 100) single-precision (32 bits) floating-point data from Blast2 [26] as the input dataset.



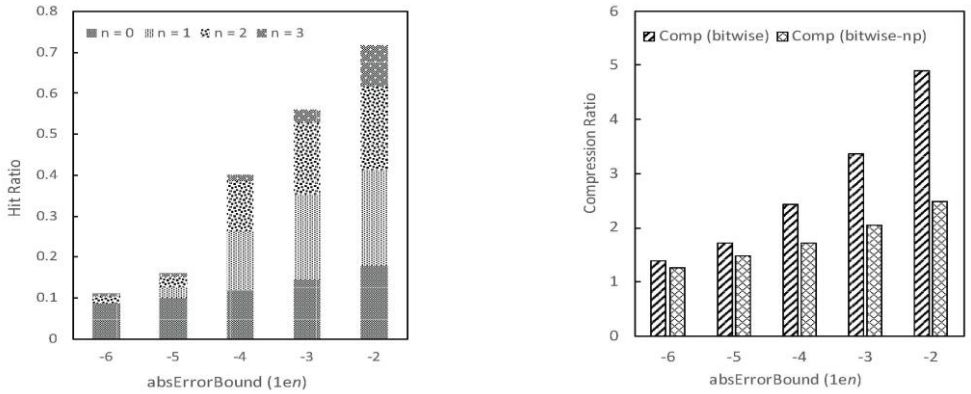
(a) Compression ratio (b) Application execution time

Figure 5. Evaluation of Ping Pong on two nodes.

Figure 5(a) shows the compression ratio with different given error bounds. Since the target data are single-precision floating-point numbers, we set the error bound E from 10^{-6} to 10^{-2}. We select Lossless [1] and SZ (v1.0) [3] as our competitors. Lossless is one of state-of-the-art lossless compression techniques. It uses a single one-dimensional polynomial predictor, subtracts from the original value, and compresses the successive zeros in their difference. SZ is one of the state-of-the-art lossy compression algorithms, as described in Section 2.1.. Unsurprisingly, the lossy compression algorithms obtain higher compression ratios than the lossless ones while maintaining within the defined error bounds. As the error bound relaxes, the compression ratio becomes larger for SZ and our compression techniques. Comparatively, the bitwise compression technique maintains a higher compression ratio than the byte-wise compression technique, and it gets comparable performance to SZ. For instance, when the error bound is 10^{-4}, the bit-wise compression algorithm gets 4.9x compression ratio, which reaches 82.6% compression ratio compared to SZ.

Figure 5(b) depicts the execution time of the Ping Pong MPI application by compressing floating-point communication data using the byte-wise and bit-wise compression techniques. We measure the execution time using the MPI_Wtime function that is inserted just before and after the target MPI functions, thus the execution time does not include the MPI initialization and finalization. Notice that SZ is not directly applicable to MPI inter-process communication on interconnection networks. We thus omit it in Fig. 5(b). The lossless compression is not included in the evaluation because its compression ratio is almost 1.0, as illustrated in Fig. 5(a).

According to the comparison with the original version of the MPI application, we found that both the byte-wise and bit-wise compression techniques obtain better performance in terms of execution time. Due to a higher compression ratio of MPI messages transferred between processes, the bit-wise compression technique performs better than the byte-wise one. For instance, when the error bound is 10^{-4}, compared to the uncompressed version, the byte-wise and bit-wise compression techniques reduce the execution time by 40.1% and 74.4%, respectively.



(a) Hit ratio of different linear predictions (b) Compression ratio w/ and w/o linear predictions

Figure 6. Effect of linear predictions on Ping Pong.

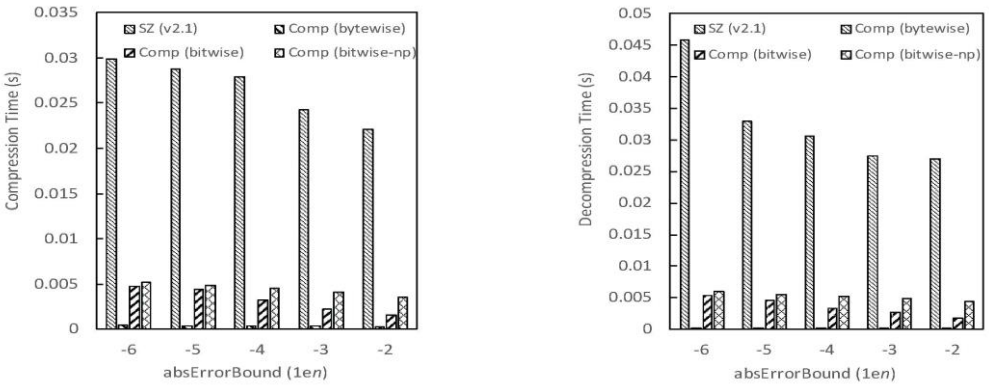
To understand the behavior of the compression techniques, Figure 6(a) illustrates the breakdown of hit ratios of the first four linear predictions (n = 0, 1, 2, 3) of the proposed compression techniques in the Ping Pong application. Notice that both the byte-wise and bit-wise compression techniques have the same hit ratio. The total hit ratio increases from 11.1% to 71.9% as the error bound relaxes from 10^{-6} to 10^{-2}. The first three linear predictions contribute much to an increase of the compression ratio. Figure 6(b) shows the comparison of compression ratios of the bit-wise compression techniques including and excluding linear predictions in the Ping Pong application. The latter (bitwise-np) applies the bit rounding to both predictable data and unpredictable data. In this case, its compression ratio is lower than the bit-wise compression technique which uses the linear predictions to compress each predictable data to only three bits. This proves that the linear prediction plays an important role in the bitwise compression technique due to its high compression ratio.

Table 2. Compound error based on error bound (bit-wise).

absErrorBound (1en)	-6	-5	-4	-3	-2
Compound Error	0	0.000001	0.000085	0.000697	0.006649

We maintain the error bound for each individual data prediction, but error can compound via compression of consecutive data pieces. Table 2 provides the results for using the bit-wise

compression technique in the Ping Pong application, which give the final compound error after the processes stop sending and receiving. It can be found that the final compound error can be still maintained within the different predefined error bounds.



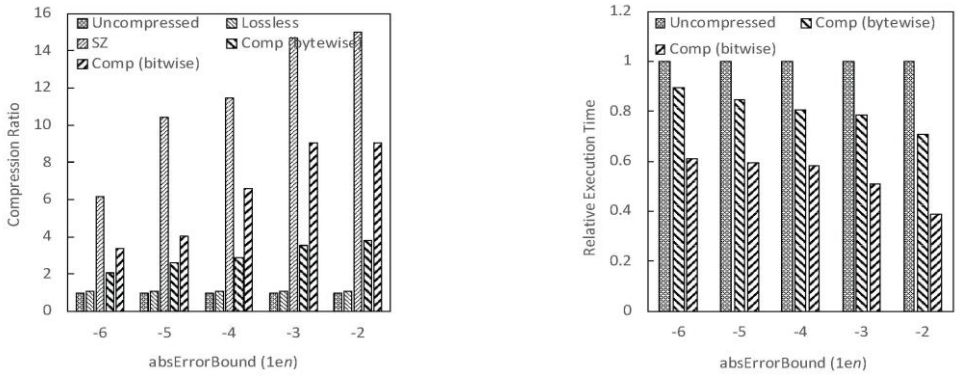
(a) Compression time (b) Decompression time

Figure 7. Compression/decompression time on Ping Pong.

We shift to compare the time cost of compression and decompression with SZ (v2.1), as shown in Fig. 7. Since SZ is not directly applicable to inter-process communication on interconnection networks, here, SZ is evaluated as a reference. Simply relying on the linear prediction, the byte-wise compression technique significantly outperforms SZ as well as the bit-wise compression technique in terms of both the compression time and decompression time. The bit-wise compression technique maintains high superiority to SZ, e.g., 8.5x speedup for compression and 9.1x speedup for decompression when the error bound is 10^{-4}, although it is inferior to the byte-wise compression technique. The bit-wise compression technique without linear predictions (bitwise-np) takes larger compression time and decompression time when compared to both the byte-wise and bit-wise compression techniques. Considering its worse compression ratio than the bitwise compression technique, the bitwise-np compression is omitted in the following evaluation.

4.1.3. Himeno

The Himeno benchmark program is developed to take measurements to proceed with major loops in solving the Poisson's equation solution using the Jacobi iteration method [27]. We modified the Himeno benchmark program for the purpose of the compression evaluation. In the Himeno benchmark program, the most used MPI functions are MPI_Isend and MPI_Irecv. The calculation size we use in the evaluation is *m* (256 × 128 × 128).



(a) Compression ratio (b) Application execution time

Figure 8. Evaluation of Himeno benchmark on two nodes.

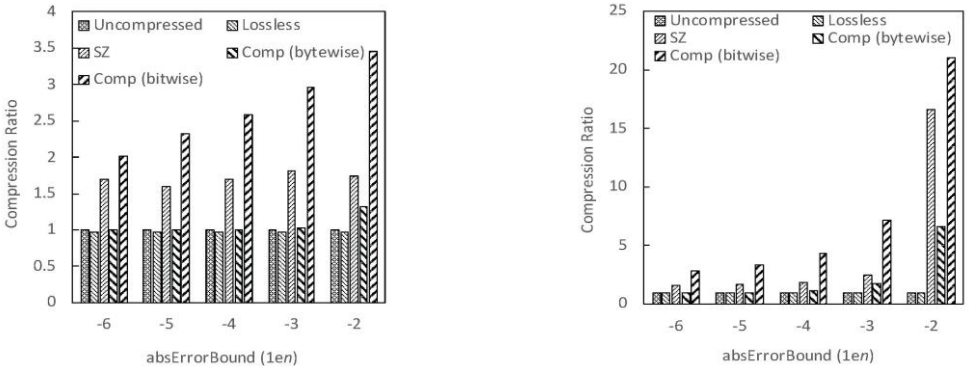
Figures 8(a) and 8(b) show the compression ratio and the application execution time, respectively. They both show a similar tendency to those in the Ping Pong application. In the case of the error bound 10^{-4}, the bit-wise compression upgrades the compression ratio by x6.6 and improves the execution time by x1.7 when compared to the uncompressed version.

4.1.4. K-means Clustering

The program of K-means clustering [28] partitions input dataset into subsets called clusters. The similar elements are placed in the same cluster. The similarity is calculated based on distance metrics, such as euclidean distance or hamming distance. In the evaluation, we assume 100 clusters and set the maximum calculation iteration to 1,000.

We employ the following two input datasets in our evaluation.

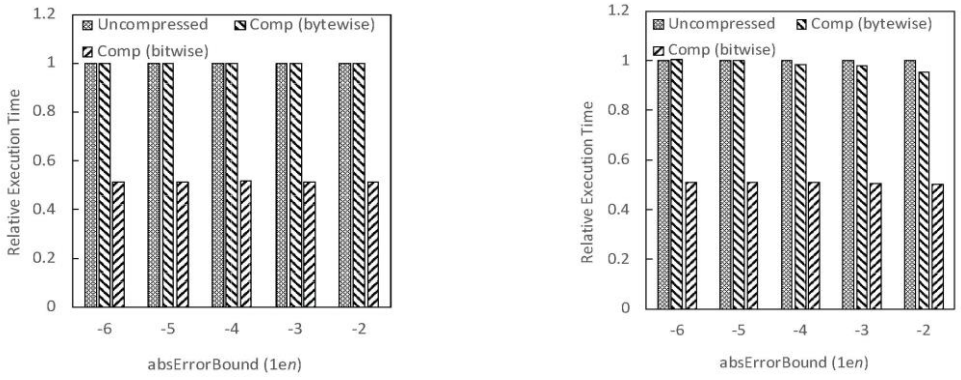
- obs_info: measurement from scientific instruments which comprises latitude and longitude information of the observation points of a weather satellite (0.3% are unique values)
- num_plasma: the result of numeric simulations which simulate plasma temperature evolution of a wire array z-pinch experiment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



(a) obs_info (b) num_plasma

Figure 9. Compression ratio of K-means clustering on two nodes.

The evaluation results of the compression ratio are depicted in Fig. 9. A finding is that the bit-wise compression technique outperforms SZ for any error bound between 10^{-6} and 10^{-2}, because the difference preprocessing converts the original floating-point data to small values that increase the compression ratio for the bit-wise compression technique. The use of a more linear prediction (n = 3) than SZ also helps to further improve the compression ratio for the bit-wise compression technique. Besides, we found that the bit-wise compression achieves a high compression ratio as the error bound relaxes to 10^{-2}, which is very close to the theoretically maximum value for the compression of double-precision floating-point data. This implies that the linear prediction almost contributes the entire data compression, which compresses a 64-bit value to 3 bits. On the other hand, the clustering result would become more flexible when the error bound relaxes to 10^{-2}.



(a) obs_info (b) num_plasma

Figure 10. Application execution time of K-means clustering on two nodes.

As shown in Fig. 10, the bit-wise compression technique reduces the execution time by half for either obs_info or num_plasma. However, the byte-wise compression technique presents a tiny advantage when compared to the uncompressed communication data in terms of application execution time. This is because the root node repeatedly broadcasts the k-means arrays, which aggravate the frequent operations of the compression and decompression. The compression benefit for the data communication is thus sacrificed by the increased (de)compression time overhead.

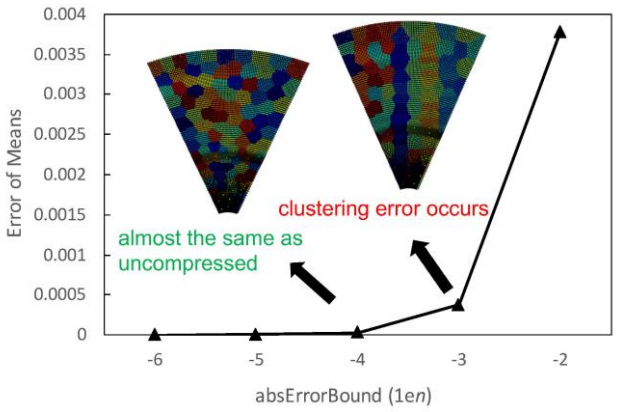
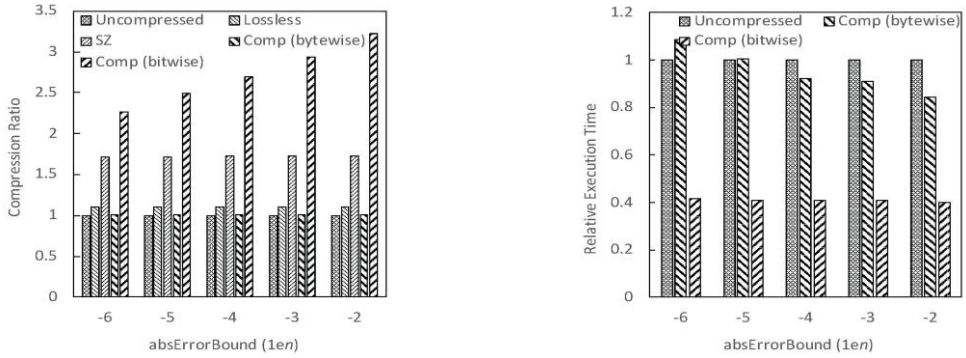


Figure 11. Error of k-means clustering (num plasma).

Figure 11 shows the average error of means on the clusters for the dataset num_plasma when using the bit-wise compression technique, together with its impact on the final clustering result. The same color indicates the same cluster in the clustering output. When the error bound relaxes to 10^{-4}, the bit-wise compression still generates almost the same clustering result as the uncompressed version. The clustering error occurs when the error bound jumps over 10^{-3}.

4.1.5. FFT

The program of FFTSS [29] is an open source library for computing the Fast Fourier Transform (FFT). The FFTSS library includes various FFT kernel routines. We modify FFTSS v3.0 to implement the floating-point data compression techniques. In the evaluation, we create and execute a plan for computing the double-precision data with two-dimensional transforms for processors with MPI library.



(a) Compression ratio (b) Application execution time

Figure 12. Evaluation of FFT on two nodes.

The evaluation results of compression ratio and execution time are depicted in Fig. 12(a) and 12(b), respectively. The byte-wise compression seems not beneficial for both the measurements because it simply relies on the linear prediction, which seems not to work well in this case. The bit-wise compression significantly outperforms the byte-wise compression as the error bound relaxes. It keeps a higher compression ratio than SZ within the error bounds from 10^{-6} to 10^{-2} and saves the execution time by around 60% compared to the uncompressed version.

4.2. Parallel Application Performance on 64 Compute Nodes

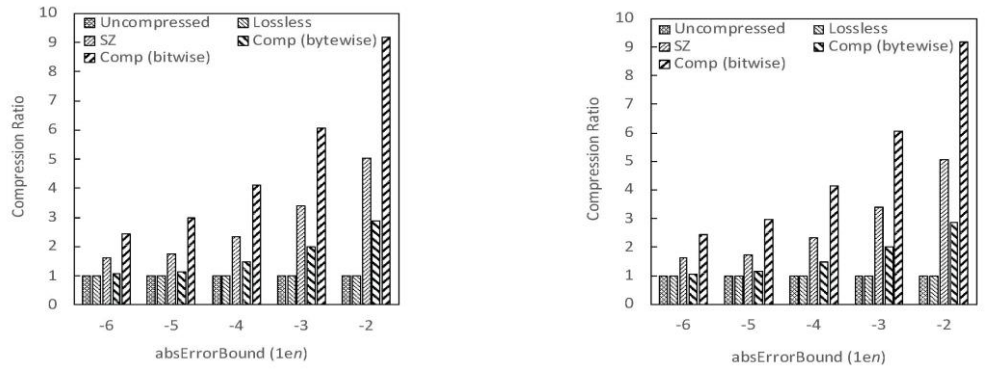
4.2.1. Condition

Since we do not have a large-scale real machine, instead, we use a discrete-event simulation to evaluate the performance of parallel application benchmarks. To this end, we use the SimGrid (v3.21) simulation framework [30]. It simulates the executions of the unmodified MPI parallel applications [31] and the modified versions using the byte- and bit-wise compression techniques.

In the simulation, we assume the minimal routing using the Dijkstra algorithm on a 4×4×4 3-D torus interconnection network. We configure SimGrid so that each switch has a 100ns delay, switches and compute nodes are interconnected via the links with 200Gbps bandwidth each, and each compute node has a 5TFlops computation power. The parameters of each application are the same as those in the previous subsection.

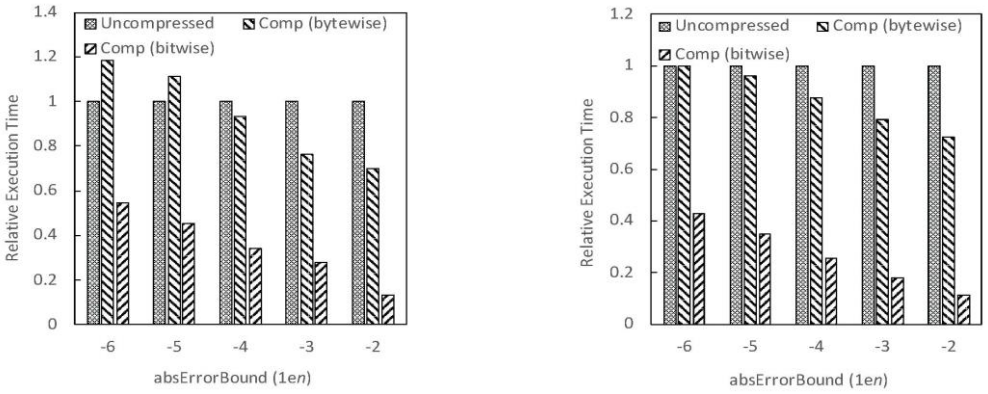
4.2.2. Ping Pong

We simulate the synthetic traffic patterns that determine each source-and-destination communication node pair: random uniform and matrix-transpose. Each process exchanges the same dataset as that used in the previous subsection, according to the synthetic access patterns. These traffic patterns are commonly used for measuring the performance of interconnection networks, as described in [32]. A node injects data packets into the interconnection networks independently of each other.



(a) Random uniform (b) Matrix-transpose

Figure 13. Compression ratio of Ping Pong on 64 nodes.

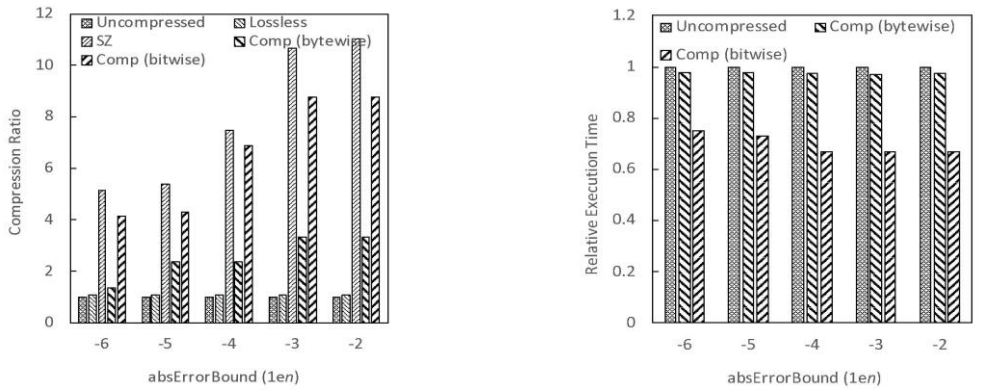


(a) Random uniform (b) Matrix-transpose

Figure 14. Relative execution time of Ping Pong on 64 nodes.

Figure 13 illustrates the comparison of the compression ratio, and Figure 14 is the execution time relative to the non-compression communication on the interconnection network. These results are consistent with the results in Fig. 5(a) and 5(b). We observe that the improvement ratios over the original uncompressed interconnection network become high as the acceptable quality turns low.

4.2.3. Himeno

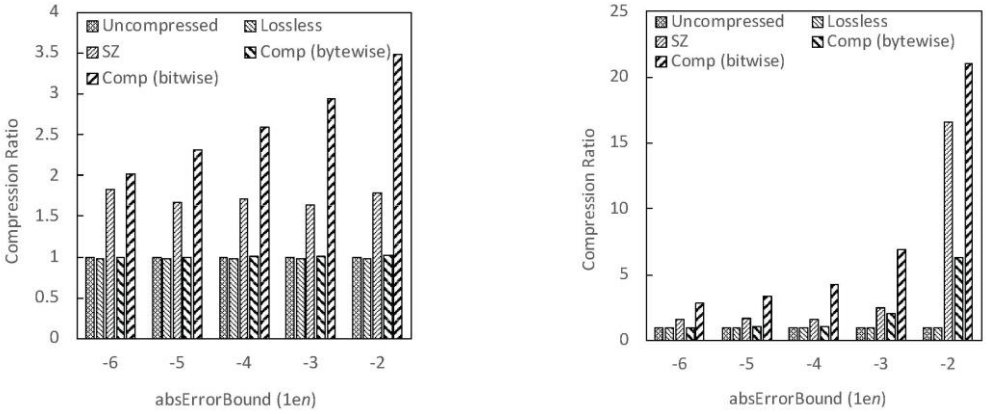


(a) Compression ratio (b) Application execution time

Figure 15. Evaluation of Himeno benchmark on 64 nodes.

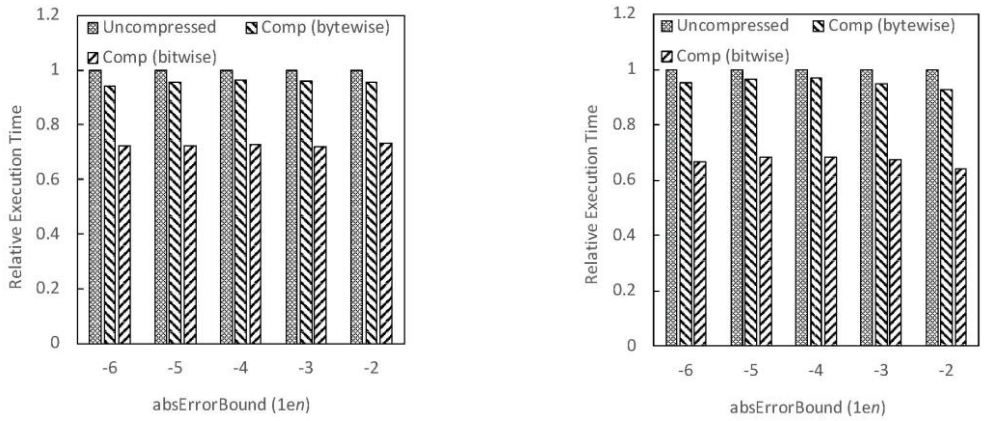
We show the simulation results for the Himeno benchmark in Fig. 15(a) and 15(b). We set the calculation size as *s* (128 × 64 × 64) and tuned the iteration times according to the simulated CPU speed, which do not impact the comparison of our data compression algorithms. In this case, the byte-wise compression technique seems to bring a limited improvement in terms of either compression ratio or execution time. This indicates that the linear prediction plays an insignificant role in the dataset for the byte-wise compression technique. By contrast, the bit-wise compression technique obtains a comparable compression ratio to SZ, and reduces the execution time by up to 33.2% compared to the original uncompressed version.

4.2.4. K-means Clustering



(a) obs_info (b) num_plasma

Figure 16. Compression ratio of K-means clustering on 64 nodes.



(a) obs_info (b) num_plasma

Figure 17. Application execution time of K-means clustering on 64 nodes.

As depicted in Fig. 16 and 17, we perform the evaluation of k-means clustering using the datasets obs_info and num_plasma. The settings of other parameters are the same as those in the real-machine evaluation. Similarly, the trends indicate the superiority of the bit-wise compression technique using both the datasets. More specifically, due to the benefit of small values obtained by the difference preprocessing phase, the bit-wise compression technique presents a higher compression ratio than SZ especially as the error bound relaxes, and it reduces the execution time by around 30%. Similarly to the evaluation on a real machine, the bit-wise compression technique achieves a high compression ratio close to the theoretically maximum value when the error bound relaxes to 10^{-2}.

4.3. Throughput and Latency

We illustrated the performance of MPI applications in the previous subsection. In this subsection, we generalize and investigate the effective network performance obtained by different compression ratios using a cycle-accurate network simulation.

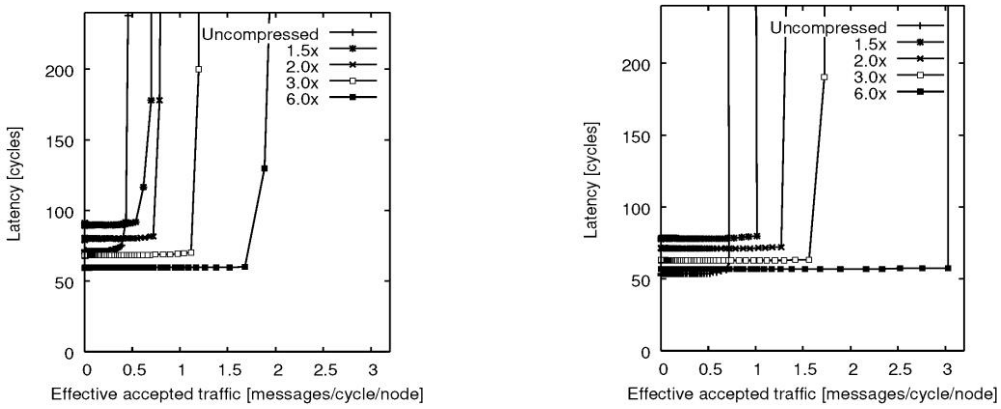
4.3.1. Condition

We use a cycle-accurate network simulator written in C++ [33]. A router model consists of channel buffers, a crossbar and a link controller, and the control circuits are used to simulate the switching fabric. On a conventional packet router, a header flit transfer requires four cycles that include routing, virtual-channel allocation, switch allocation and flit transfer from an input channel to an output channel through a crossbar. We use the minimal adaptive routing via two virtual channels on a 4×4×4 3-D torus interconnection network. Each node includes a router with a local processor. The packet length is set to 32 flits when no data compression is applied. We simulate the same synthetic traffic patterns as those in the previous subsection: random uniform and matrix-transpose.

We investigate the impact of the compression ratio of data packets on the communication latency and the effective network performance. Although the compression ratio depends on the data type and compression algorithm, we parameterize the compression ratio for illustrating the network performance in this simulation. For the consistency with the previous subsection, the compression ratio is set to 1.5, 2.0, 3.0 and 6.0. We set 40 cycles as the (de)compression overhead at each source and destination pair.

Our results show two important metrics: communication latency and effective throughput. The communication latency is the elapsed time between the generation of a packet at a source host and its delivery at a destination processing element. We measure the communication latency in simulation cycles. The effective throughput is defined as the maximum accepted traffic represented by the flit delivery rate. The flit delivery rate is computed as $C \times R$, where C and R are the compression rate and the average number of received flits at a processing element within a cycle.

4.3.2. Results



(a) Random uniform (b) Matrix-transpose

Figure 18. Communication latency vs. accepted traffic for (non-)compression communications.

Figure 18 illustrates the relationship between the communication latency and the accepted traffic rate for non-compression and compression communication datasets. The plot of "2.0x" represents the case where all packets are compressed with the compression ratio of 2.0.

As the compression ratio increases, the packet length becomes shorter, thus efficiently reducing the network injection latency. Since the communication latency includes the network injection latency, we observe that a higher compression ratio leads to a better communication latency at the low traffic load. For example, the 1.5 and 2.0 compression ratios bring the higher communication latency than no compression on the interconnection network at the low traffic load. By contrast, the 3.0 and 6.0 compression ratios improve the communication latency even at the low traffic load with the random uniform traffic.

A higher compression ratio also leads to a better effective network throughput, since shortening the packet length by the data compression decreases the network load. For example, when the compression ratio is 3.0, it improves the effective network throughput by 176% and 140% with the traffic patterns of random uniform and matrix transpose, respectively. Similarly, the 1.5 and 6.0 compression ratios improve the effective network throughput by up to 133% and 260%, respectively.

Through the simulation results, we observe that a high compression ratio improves both the communication latency at a high traffic load and the effective network throughput with both the evaluated traffic patterns.

5. CONCLUSION

Data compression increases the effective network bandwidth on an interconnection network of parallel computers. Generally, a lossy compression achieves a higher compression ratio than that by a counterpart lossless compression. In this study, we introduce a lossy compression algorithm to floating-point communication data on interconnection networks.

Since recent interconnection networks are latency-sensitive, a simple lossy compression technique that has small compression overhead is preferred. In this context, we apply a linear predictor with the user-defined error bound to floating-point communication data compression. We provide byte- and bit-wise compression techniques. In the byte-wise compression, if the value prediction succeeds, a floating-point value is converted to a single byte expression, corresponding to an MPI char type. Otherwise, the original value is not compressed. Although the byte-wise compression has low compression-operation latency, its upper bound of the compression ratio is not high, i.e., obviously four and eight for single-precision and double-precision floating-point values, respectively. By contrast, the bit-wise compression generates a bitstream encapsulated in a byte array, corresponding to an MPI char type. If the value prediction succeeds at a source node, the floating-point value is converted to three bits. Even if the value prediction fails, the least significant bits (LSBs) are discarded from the IEEE 754 floating-point expression of the value in order to obtain a relatively high compression ratio, while maintaining a given error bound.

We implemented and evaluated the byte- and bit-wise compression techniques for floating-point communication data generated in the MPI parallel programs of Ping Pong, Himeno, K-means Clustering and FFT. The bit-wise compression technique achieves 2.4x, 6.6x, 4.3x and 2.7x compression ratio for Ping Pong, Himeno, K-means and FFT at the cost of a moderate decrease of quality of results (error bound is 10^{-4}), thus achieving 2.1x, 1.7x, 2.0x and 2.4x speedup of the execution time, respectively. More generally, from the network point of view, the cycle-accurate network simulation shows that, when the compression ratio becomes 1.5, 3.0 and 6.0, the interconnection network improves the effective network throughput by up to 133%, 176% and 260%, respectively. Through this observation, we highly recommend using the lossy application-level compression, i.e., the bit-wise compression, on interconnection networks.

ACKNOWLEDGEMENTS

This work was supported by JSPS KAKENHI Grant Number 19H01106 and JST PRESTO JPMJPR19M1.

REFERENCES

[1] T. Ueno, K. Sano, and S. Yamamoto, “Bandwidth compression of floating-point numerical data streams for fpga-based high-performance computing,” ACM Transactions on Reconfigurable Technology and Systems, vol. 10, pp. 1–22, 05 2017.
[2] J. Tomkins, “Interconnects: A Buyers Point of View,” ACS Workshop, 2007.
[3] S. Di and F. Cappello, “Fast error-bounded lossy hpc data compression with sz,” in IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2016, pp. 730–739.
[4] M. Narasimha and A. Peterson, “On the computation of the discrete cosine transform,” IEEE Transactions on Communications, vol. 26, no. 6, pp. 934–936, June 1978.
[5] D. H. J. Michael T. Heideman and C. S. Burrus, “Gauss and the history of the fast fourier transform,” IEEE ASSP Magazine, vol. 1, no. 4, pp. 14–21, 1984.
[6] C. C. Cutler, “Differential quantization of communication signals, u.s. patent 2605361,” July 1952.
[7] L. Deng and D. O’Shaughnessy, “Speech processing: a dynamic and optimization-oriented approach,” in Marcel Dekker, 2003, pp. 41–48.

[8] P. Lindstrom and M. Isenburg, "Fast and efficient compression of floating-point data," IEEE transactions on visualization and computer graphics, vol. 12, pp. 1245–50, 09 2006.
[9] P. Lindstrom, "Fixed-rate compressed floating-point arrays," IEEE Transactions on Visualization and Computer Graphics, vol. 20, 08 2014.
[10] 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 and Computation: Practice and Experience, vol. 25, 02 2013.
[11] M. Isenburg, P. Lindstrom, and J. Snoeyink, "Lossless compression of floating-point geometry," in Computer-Aided Design and Applications, vol. 1, 04 2004.
[12] M. Isenburg, P. Lindstrom, and J. Snoeyink, "Lossless compression of predicted floating-point geometry," Computer-Aided Design, pp. 869–877, 07 2005.
[13] 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.
[14] P. Ratanaworabhan, J. Ke, and M. Burtscher, "Fast lossless compression of scientific floating-point data," in Data Compression Conference Proceedings, 04 2006, pp. 133– 142.
[15] M. Burtscher and P. Ratanaworabhan, "High throughput compression of double-precision floating-point data," in Data Compression Conference (DCC), 2007, pp. 293–302.
[16] M. Burtscher and P. Ratanaworabhan, "Fpc: A high-speed compressor for double-precision floating-point data," Computers, IEEE Transactions on, vol. 58, pp. 18 – 31, 02 2009.
[17] D. Tao, S. Di, Z. Chen, and F. Cappello, "Significantly improving lossy compression for scientific data sets based on multidimensional prediction and error-controlled quantization," in IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2017, pp. 1129–1139.
[18] 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 IEEE International Conference on Big Data (Big Data), 2018, pp. 438–447.
[19] 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 Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC, 2019, pp. 33:1–33:26.
[20] N. Sasaki, K. Sato, T. Endo, and S. Matsuoka, "Exploration of lossy compression for application-level checkpoint/restart," in IEEE International Parallel and Distributed Processing Symposium, 2015, pp. 914–922.
[21] A. R. Alameldeen and D. A. Wood, "Frequent pattern compression: A significance-based compression scheme f or l2 caches," in Technical Report 1500, Computer Sciences Dept. UW-Madison, Apr. 2004.
[22] R. Das, A. K. Mishra, C. Nicopoulos, D. Park, V. Narayanan, R. R. Iyer, M. S. Yousif, and C. R. Das, "Performance and power optimization through data compression in network-on-chip architectures," in International Conference on High-Performance Computer Architecture (HPCA), 2008, pp. 215–225.
[23] B. Dickov, M. Peric`as, P. M. Carpenter, N. Navarro, and E. Ayguad´e, "Analyzing performance improvements and energy savings in infiniband architecture using network compression," in 2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing, Oct 2014, pp. 73–80.
[24] V. Engelson, D. Fritzson, and P. Fritzson, "Lossless compression of high-volume numerical data from simulations," in Proceedings of the Data Compression Conference, 02 2000, pp. 574–586.
[25] "Ieee standard for floating-point arithmetic," IEEE Std 754-2019 (Revision of IEEE 754-2008), pp. 1–84, 2019.
[26] P. Colella and P. R. Woodward, "The piecewise parabolic method (ppm) for gas-dynamical simulations," Journal of Computational Physics, vol. 54, no. 1, pp. 174 – 201, 1984. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0021999184901438
[27] "Himeno benchmark," http://i.riken.jp/en/supercom/documents/himenobmt/.
[28] "k-means clustering: A distributed mpi implementation," https://github.com/dzdao/k-means-clustering-mpi.
[29] A. Nukada, "Fftss: A high performance fast fourier transform library," in IEEE International Conference on Acoustics Speech and Signal Processing Proceedings, vol. 3, 2006, pp. III–III.
[30] "Simgrid: Versatile simulation of distributed systems," http://simgrid.gforge.inria.fr/.

[31] H. Casanova, A. Giersch, A. Legrand, M. Quinson, and F. Suter, “Versatile, Scalable, and Accurate Simulation of Distributed Applications and Platforms,” Journal of Parallel and Distributed Computing, vol. 74, no. 10, pp. 2899–2917, 2014.
[32] J. Duato, S. Yalamanchili, and L. Ni, Interconnection Networks: an engineering approach. Morgan Kaufmann, 2002.
[33] A. Jouraku, M. Koibuchi, and H. Amano, “An Effective Design of Deadlock-Free Routing Algorithms Based on 2-D Turn Model for Irregular Networks,” IEEE Transactions on Parallel and Distributed Systems, vol. 18, no. 3, pp. 320–333, Mar. 2007.
[34] Q. Fan, D. J. Lilja and S. S. Sapatnekar, "Using DCT-based Approximate Communication to Improve MPI Performance in Parallel Clusters," 2019 IEEE 38th International Performance Computing and Communications Conference (IPCCC), London, United Kingdom, 2019, pp. 1-10.

AUTHORS

Yao Hu received the M.S. degree from Beijing University of Posts and Telecommunications, China, in 2009, and received the PhD degree from the Department of Computer Science and Engineering, Waseda University, Tokyo, Japan, in 2015. He is currently working as a project researcher in the National Institute of Informatics, Tokyo, Japan. His main research interests include the area of high-performance computing.

Michihiro Koibuchi received the BE, ME and PhD degrees from Keio University, Yokohama, Japan, in 2000, 2002 and 2003, respectively. Currently, he is an associate professor in the Information Systems Architecture Research Division, National Institute of Informatics and the Graduate University of Advanced Studies, Tokyo, Japan. His research interests include the areas of high-performance computing and interconnection networks. He is a senior member of the IEICE, IEEE and IPSJ.