SC24: International Conference for High Performance Computing, Networking, Storage and Analysis | 979-8-3503-5291-7/24/$31.00 ©2024 IEEE | DOI: 10.1109/SC41406.2024.00110

*h*ZCCL: Accelerating Collective Communication with Co-Designed Homomorphic Compression

Jiajun Huang [*], Sheng Di [°], Xiaodong Yu [▷], Yujia Zhai [*], Jinyang Liu [*], Zizhe Jian [*], Xin Liang [◇], Kai Zhao [§], Xiaoyi Lu [‖], Zizhong Chen [*], Franck Cappello [°], Yanfei Guo [°], Rajeev Thakur [°]

[*] University of California, Riverside, CA, USA;
{ jhuan380, yzhai015, jliu447, zjian106 }@ucr.edu, chen@cs.ucr.edu

[°] Argonne National Laboratory, Lemont, IL, USA;
{ sdi1, yguo, thakur }@anl.gov, cappello@mcs.anl.gov

[▷] Stevens Institute of Technology, Hoboken, NJ, USA; xyu38@stevens.edu

[◇] University of Kentucky, Lexington, KY, USA; xliang@uky.edu

[§] Florida State University, Tallahassee, FL, USA; kzhao@cs.fsu.edu

[‖] University of California, Merced, CA, USA; xiaoyi.lu@ucmerced.edu

***Abstract*—As network bandwidth struggles to keep up with rapidly growing computing capabilities, the efficiency of collective communication has become a critical challenge for exa-scale distributed and parallel applications. Traditional approaches directly utilize error-bounded lossy compression to accelerate collective computation operations, exposing unsatisfying performance due to the expensive decompression-operation-compression (DOC) workflow. To address this issue, we present a *first-ever* homomorphic compression-communication co-design, *h*ZCCL, which enables operations to be performed directly on compressed data, saving the cost of time-consuming decompression and recompression. In addition to the co-design framework, we build a light-weight compressor, optimized specifically for multi-core CPU platforms. We also present a homomorphic compressor with a run-time heuristic to dynamically select efficient compression pipelines for reducing the cost of DOC handling. We evaluate *h*ZCCL with up to 512 nodes and across five application datasets. The experimental results demonstrate that our homomorphic compressor achieves a CPU throughput of up to 379.08 GB/s, surpassing the conventional DOC workflow by up to 36.53×. Moreover, our *h*ZCCL-accelerated collectives outperform two state-of-the-art baselines, delivering speedups of up to 2.12× and 6.77× compared to original MPI collectives in single-thread and multi-thread modes, respectively, while maintaining data accuracy.**

***Index Terms*—Collective Communication, Homomorphic Compression, Distributed Computing, Parallel Algorithm**

I. Introduction

In the era of exascale computing, optimizing collective communications for large messages becomes a pivotal aspect of enhancing the performance of high-performance computing clusters. This necessity is especially pronounced in scientific applications like molecular dynamics simulations [1] and seismic modeling [2], alongside data analysis, visualization applications [3], and deep learning tasks [4]. These fields, characterized by intensive data processing and exchange, underscore the importance of refining collective communication strategies to transfer large messages efficiently [5], [6].

The internode communications, limited by the network bandwidth is always the major concern for the efficiency of collective communications. For the large message collectives, the networks are easily saturated, making it especially challenging to get a high overall collective performance. Prior researchers are actively working on minimizing the overall communication volume to mitigate the network saturation [7], [8], [9]. Recently, with the development of high-speed error-bounded lossy compression techniques [10], [11], [12], it becomes possible to utilize error-bounded lossy compression for significantly decreasing the message sizes and thus mitigating the performance issue while maintaining the data quality.

The current error-bounded lossy compression-integrated collective frameworks have achieved significant speedups with well-controlled error propagation compared with the previous approaches without compression integrated. Specifically, the state-of-the-art C-Coll framework that realizes high-performance with bounded errors for all collective operations outperforms the original MPI collectives by 1.8–2.7× [13]. However, it is subjected to a traditional decompression-operation-compression (DOC) workflow, in which each node has to fully decompress the compressed data before applying operations, and then recompress the operated data into the compressed format. This process causes an inevitable huge cost in the compression-accelerated collective communication and thereby affecting the overall collective performance.

In order to further improve the overall reduction throughput in the compression-accelerated collective communications, we need to address a series of challenges. (1) How to develop an ultra-fast error-bounded lossy compressor for CPU architectures to achieve both high compression throughput and quality? (2) How to design a new workflow to deal with the DOC more efficiently in the collective communication scenario? (3) How to co-design and implement a general communication framework to effectively utilize the new DOC-handling workflow?

To address the aforementioned challenges, in this paper, we present *h*ZCCL, a high-performance homomorphic compression-accelerated collective communication library, which enables performing operations *directly* on compressed datasets without requiring the costly decompression and recompression process. To the best of our knowledge, *h*ZCCL is the first-ever high-performance co-design for homomorphic

Corresponding authors: Sheng Di, Jiajun Huang.

compression and collective communication. More specifically, our contributions include:

- To address challenge (1), we present an optimized block partitioning scheme. This tiling strategy enables a more efficient memory footprint and lesser outlier storage space for each CPU thread. We also employ a bit-shifting fixed-length encoding scheme to further squeeze the performance from compressing pipeline. The STREAM benchmark validates our optimized compressor, *fZ*-light, achieves up to 94.5% of the peak memory throughput for decompressing and compressing operations.
- To address challenge (2), we design and implement our communication-targeted homomorphic compressor *hZ*-dynamic based on *fZ*-light. *hZ*-dynamic features a dynamic homomorphic compression pipeline, which is able to select the most light-weight compression pipeline based on the properties of compressed data inputs to directly operate on compressed data.
- To tackle challenge (3), we carefully co-design a homomorphic compression-accelerated collective framework for collective computation operations, by leveraging the unique advantages of our designed homomorphic compressor–*hZ*-dynamic. For all the collective computation operations, we redesign the compression-enabled communication workflow to significantly improve the compression and computation runtime with homomorphic compression. We further optimize the performance for the widely-used Allreduce operation in particular, by eliminating the decompression step in its Reduce_scatter stage and the compression step in the Allgather stage.
- We evaluate our optimized *fZ*-light and *hZ*-dynamic with five application datasets. Compared with the *ompSZp* (multi-threaded CPU version of cuSZp [14]), our *fZ*-light achieves the compression ratio improvement of up to 37.65. Regarding performance, *fZ*-light brings up to 9.71× and 28.33× speedups in compression and decompression, respectively. Notably, our dynamic homomorphic compression pipeline of *hZ*-dynamic reaches the overall compression throughput of up to 379.08 GB/s, which is 36.53× faster than the 10.38 GB/s of traditional DOC workflow.
- We utilize up to 512 Intel Broadwell nodes to evaluate both the performance and accuracy of our *h*ZCCL-accelerated collectives. Experiments demonstrate that our *h*ZCCL-accelerated Reduce_scatter achieves up to 1.9× and 5.85× performance improvements and our *h*ZCCL-accelerated Allreduce reaches up to 2.12× and 6.77× speedups compared with the original MPI without compression in the single-thread and multi-thread modes, respectively. We also employ a practical use case – image stacking analysis, to showcase the real-world efficacy of the *h*ZCCL-accelerated Allreduce. Our *h*ZCCL outperforms C-Coll and realizes 1.81× and 5.02× performance enhancements compared with MPI with satisfying operational results in both statistical and visual analyses.

The rest of this paper is structured as follows: Section II provides an overview of background and related work. Our design and optimization strategies are elaborated in Section III. The evaluation findings are disclosed in Section IV. We provide a conclusion and future work in Section V.

II. Background and Related Work

Error-bounded lossy compression can achieve a much higher compression ratio than lossless compression while the difference between original data and decompressed data is strictly bounded by the user-required error-bound [15], [16], [17]. Thus, it has been effective in various fields, from climate science, exemplified by the CESM project at NCAR [18], to Quantum Monte Carlo with QMCPACK at US National labs [19], and geophysical explorations with RTM used by Saudi Aramco [2]. Many researchers have focused on designing high-speed lossy compressors [12], [11], [20]. Among these, SZx [11] stands out as the fastest CPU compressor. However, its constant block design may severely degrade data reconstruction quality, limiting its usability across several domains, as demonstrated in [14]. In contrast, the GPU-based compressor cuSZp showcases significantly improved data quality over SZx, yet its parallelism strategies fall short when applied to CPU architectures. In comparison, our *fZ*-light achieves a remarkably high throughput (e.g., a 28.33× speedup over ompSZp in decompression) while providing slightly better reconstructed data quality. This high throughput is attributed to the very high memory bandwidth efficiency in our design, validated by STREAM benchmarks [21].

The integration of compression to speed up communication within high-performance computing clusters has garnered considerable attention in recent research [22], [23], [24], [25], [26], [13], [27]. Among them, the error-bounded lossy compression-accelerated approaches [13], [27] are particularly valued for their ability to precisely control error propagation, as proved by [13] through both theoretical & experimental analyses. While the C-Coll framework stands as a leading solution for CPU-centric collective operations, it is hindered by the inefficiencies associated with the traditional decompression-operation-compression workflow, which has to decompress the compressed data before operating on them [13]. In contrast, our *h*ZCCL effectively co-designs collective communication with our optimized homomorphic compression method, reaching a significant performance improvement over C-Coll, to be shown in Sections IV-C and IV-D.

III. *h*ZCCL Design and Optimization

This section details the design and optimization of *h*ZCCL, illustrated in Figure 1 with newly designed modules in green, cyan, and pink. Central to our approach is the dynamic homomorphic compressor, *hZ*-dynamic, described in Section III-B. We then co-design the homomorphic compression-accelerated-collectives with *hZ*-dynamic, as detailed in Section III-C.

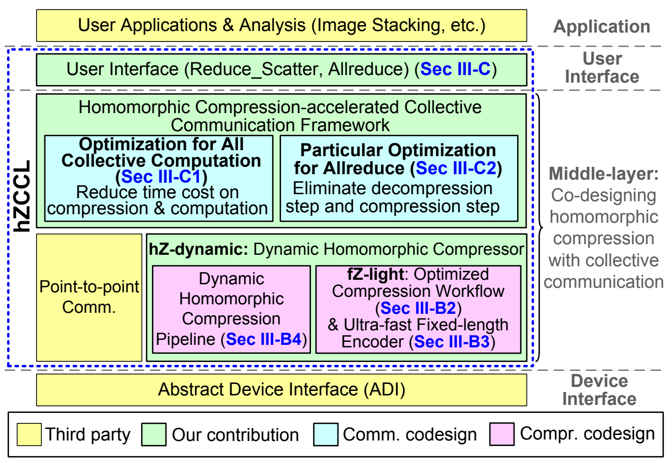

User Applications & Analysis (Image Stacking, etc.)
Application
User Interface (Reduce_Scatter, Allreduce) (Sec III-C)
User Interface
hZCCL
Homomorphic Compression-accelerated Collective Communication Framework
Optimization for All Collective Computation (Sec III-C1) Reduce time cost on compression & computation
Particular Optimization for Allreduce (Sec III-C2) Eliminate decompression step and compression step
Middle-layer: Co-designing homomorphic compression with collective communication
Point-to-point Comm.
hZ-dynamic: Dynamic Homomorphic Compressor
Dynamic Homomorphic Compression Pipeline (Sec III-B4)
fZ-light: Optimized Compression Workflow (Sec III-B2) & Ultra-fast Fixed-length Encoder (Sec III-B3)
Abstract Device Interface (ADI)
Device Interface
Third party
Our contribution
Comm. codesign
Compr. codesign

Fig. 1: *h*ZCCL design architecture.

A. Analysis of existing compression-accelerated collectives

In this section, we examine the shortcomings of previous solutions and offer a thorough analysis of performance to pinpoint potential bottlenecks. The existing error-bounded lossy compression-accelerated collectives [13], [27] have achieved significant speedups compared with traditional no-compression collectives but their costly decompression-operation-compression workflow is still a significant bottleneck. DOC is a critical step in the compression-accelerated collective communications, including the state-of-the-art C-Coll framework [13]. Generally, C-Coll pre-compresses the original data and transfers the compressed bytes in the collective data movement operations, and overlaps the compression with communication to reduce the overall collective runtime. Specifically, in its collective computation framework, each node needs to decompress the received data before operating on them and then compress the operated data before sending them, which is referred to as the DOC workflow. This design can achieve prominent performance gains over the CPR-P2P method [25], while its entire runtime is still substantially limited to the DOC workflow.

We demonstrate the significant DOC bottleneck of the traditional design in Figure 2, which presents a meticulous performance breakdown for the C-Coll-accelerated ring-based Allreduce. The widely-used ring-based Allreduce [28], [8] contains both collective data movement (Allgather) and collective computation (Reduce_scatter), which have been optimized in C-Coll [13]. Our experiments are conducted on 16 Intel Broadwell nodes interconnected via Intel Omni-Path Architecture. In this context, DPR+CPT+CPR denotes the time spent for decompression, compression and computation, MPI signifies the communication time, and OTHER encompasses all additional runtime consumed. Observations reveal that in the single-thread mode of C-Coll, DPR+CPT+CPR dominates the runtime, accounting for 78.18%, while MPI communication constitutes only 21.56%. This trend persists in the multi-thread mode of C-Coll, with DPR+CPT+CPR remaining the primary time consumer at 52.26%, followed by MPI at 47.02%. The

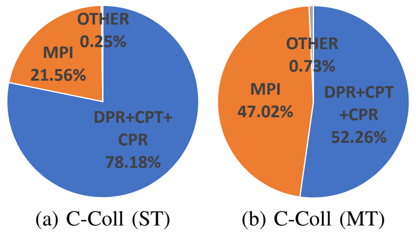


Fig. 2: Performance breakdown of Allreduce using the single-thread (ST) mode and multi-thread (MT) mode of C-Coll.

compression-related cost is high because the traditional lossy compressor has to fully decompress the compressed bytes before any calculations and then recompress the operated data to compressed format.

To address the DOC bottleneck, we need to design a new DOC-handling workflow that is able to avoid the full compression and decompression. Besides, the existing optimization strategies and frameworks in C-Coll are specifically designed for the traditional DOC workflow, which prevents the integration of other DOC-handling workflows that can result in better overall performance. Thus, we also need to redesign the compression-accelerated collective framework to effectively utilize the newly designed DOC-handling method.

B. Designing a high-performance homomorphic compressor for collective communication

As previously mentioned, the traditional DOC workflow utilized in the C-Coll framework leads to sub-optimal performance in collective computation operations. Consequently, in this section, we describe our approach to designing a high-performance dynamic homomorphic compressor, aimed at significantly enhancing the efficiency of compression-enabled collective communication.

1) Analysis of existing high-speed compression pipelines

To develop a high-performance homomorphic compressor, identifying the optimal high-speed compression pipeline is the initial step. Various pipelines offer rapid compression on CPUs, including SZx, ZFP, and SZ3 [11], [12], [29]. SZx stands out for its superior compression speed and favorable compression ratio on CPUs [11], [13]. Nonetheless, its use of a constant block design, which reduces data points in smooth regions to a constant value, can significantly degrade the quality of reconstructed data [14]. Another promising pipeline, cuSZp, boasts potential for high compression throughput and compression quality but is primarily optimized for GPU usage. Its current parallelism strategy may not deliver optimal performance on CPUs. Therefore, it is imperative to revise the design of cuSZp to enhance its efficiency on CPU architectures.

2) Optimizing compression workflow for multithreading in CPU

The first primary design distinction between *fZ*-light and cuSZp emerges in the stage of block partitioning. As depicted in the left part of Figure 3, cuSZp utilizes single-layer block partitioning to segment input data into smaller blocks. Then, each block undergoes block-wise quantization and prediction. Notably, the first quantized value of a block is stored as the outlier in the compressed bytes.

In contrast, our optimized methodology within *fZ*-light segments the input data through a multi-layered partitioning approach, as depicted in the right portion of Figure 3. Initially, the data is divided into several large chunks, with each thread processing one chunk where the chunk length is D/N, D being the total length of the input data and N the number of threads. The last $D\%N$ data points are managed by the $(N-1)$-th thread. Each large threadblock is then subdivided into smaller blocks as per the user-defined block length. Within each threadblock, the fused quantization and prediction process converts floating-point data into integer prediction values, with only the first quantized value or the outlier of each threadblock being preserved in the compressed data. Consequently, unlike cuSZp, where threads hop between distant small blocks, threads in our *fZ*-light consistently work on contiguous memory segments, enhancing memory access efficiency. Moreover, our fused quantization and prediction reduce the number of memory accesses compared to the unfused version, thereby further enhancing performance. It is worth noting that *fZ*-light stores merely a single outlier (four bytes) for each large threadblock, whereas cuSZp stores one outlier for every small block. Thus, *fZ*-light can reach higher compression ratios than cuSZp.

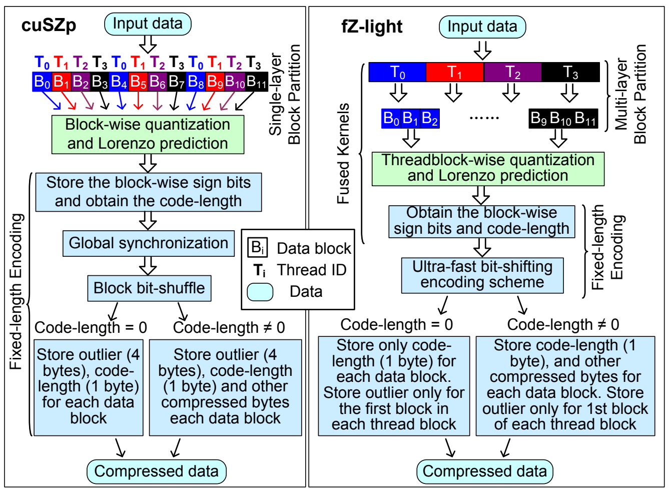

cuSZp
Input data
T_0 T_1 T_2 T_3 T_0 T_1 T_2 T_3 T_0 T_1 T_2 T_3
B_0 B_1 B_2 B_3 B_4 B_5 B_6 B_7 B_8 B_9 B_{10} B_{11}
Single-layer Block Partition
Block-wise quantization and Lorenzo prediction
Fixed-length Encoding
Store the block-wise sign bits and obtain the code-length
Global synchronization
Block bit-shuffle
B_i Data block
T_i Thread ID
Data
Code-length = 0
Code-length ≠ 0
Store outlier (4 bytes), code-length (1 byte) for each data block
Store outlier (4 bytes), code-length (1 byte) and other compressed bytes each data block
Compressed data
fZ-light
Input data
T_0 T_1 T_2 T_3
Multi-layer Block Partition
B_0 B_1 B_2 ...... B_9 B_{10} B_{11}
Fused Kernels
Threadblock-wise quantization and Lorenzo prediction
Obtain the block-wise sign bits and code-length
Fixed-length Encoding
Ultra-fast bit-shifting encoding scheme
Code-length = 0
Code-length ≠ 0
Store only code-length (1 byte) for each data block. Store outlier only for the first block in each thread block
Store code-length (1 byte), and other compressed bytes for each data block. Store outlier only for 1st block of each thread block
Compressed data

Fig. 3: Compare compression workflows of *fZ*-light & cuSZp.

3) *Ultra-fast fixed-length encoding scheme*

Another major design difference between *fZ*-light and cuSZp is evident in the fixed-length encoding phase. cuSZp first stores the block-wise sign bits and obtain the code-length, followed by a global synchronization. Subsequently, it employs a bit-shuffle technique to rearrange the compressed, fixed-length encoded integers and store them based on the indices derived from global synchronization.

Contrastingly, our approach integrates the acquisition of sign bits and the determination of the code length for each small data block with the process of threadblock-wise quantization and prediction, resulting in a more efficient pattern of memory access. Subsequently, we employ an ultra-fast bit-shifting technique to encode both the sign bits and the fixed-length bits of the integer block into a byte array. The core concept involves initially storing the complete bytes of the input unsigned integer array, followed by the residual bits. Specifically, the algorithm calculates the count of complete bytes (`byte_count`) for the given code length and then determines the remaining bit count (`remainder_bit`), excluding the full bytes. If the `byte_count` > 0, the full bytes of the unsigned integer array are stored into a byte array utilizing the ultra-fast bit-shifting method. Moreover, if the `remainder_bit` > 0 and the `byte_count` > 0, each element of the unsigned integer array undergoes a left shift by $32 -$ `remainder_bit` and then a right shift by the same amount to eliminate all bits to the left of the residual bits. This process allows these remaining bits to be accurately stored in the output byte array using the designated `ultra_fast_bit_shifting_x` function for varying counts of residual bits, denoted by `x` (with `x` spanning from 1 to 7). By adopting this lightweight bit-shifting methodology, we achieve substantial enhancements in the efficiency of fixed-length encoding on CPU architectures.

4) *Dynamic homomorphic compressor–hZ-dynamic*

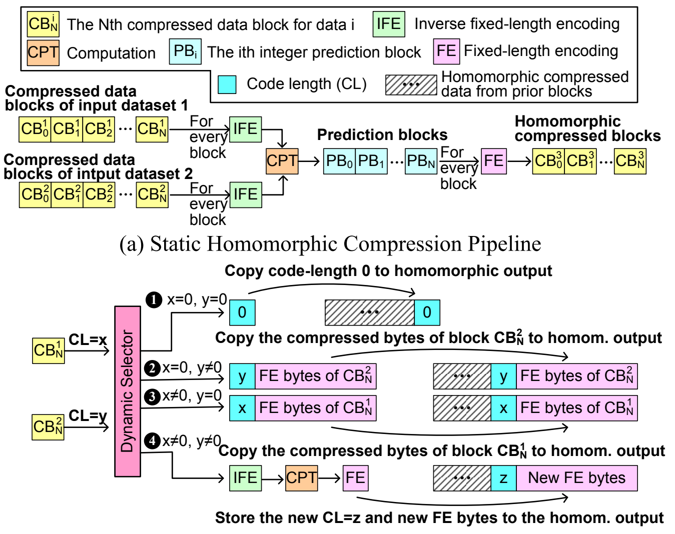
In this section, we elaborate on the design of the dynamic homomorphic compressor–*hZ*-dynamic. Building upon our optimized *fZ*-light, *hZ*-dynamic dynamically selects the most light-weight compression pipeline to perform reduction operations directly on compressed data inputs. This dynamic approach offers dual advantages: (1) It avoids full decompression and recompression in the DOC workflow, thereby saving time. (2) It dynamically selects the most light-weight pipeline based on the properties of compressed data inputs, offering a much more lightweight solution than the static homomorphic compression pipeline, which always requires ‘partial’ data decompression and recompression [30].

We use the most widely used ‘sum’ operation as an illustrative example of our homomorphic designs, though the principles are applicable to other reduction operations as well. Figure 4 contrasts our dynamic homomorphic compression pipeline with the static approach. In the latter, each data block of the two compressed data inputs undergoes inverse fixed-length encoding (IFE) to revert to integer prediction data, a process considered as ‘partial’ decompression. Subsequently, a specific reduction operation (e.g., sum) operates on these integer blocks in the computation (CPT) phase. The integer array output of CPT, containing all integer prediction blocks, undergoes ‘partial’ recompression through fixed-length encoding (FE) to form homomorphic compressed blocks.

Unlike the fixed approach of static homomorphic compression that consistently requires ‘partial’ data decompression and recompression, our dynamic pipeline dynamically selects the most lightweight compression pipeline based on the nature of the compressed data blocks to optimize throughput. Initially, the process involves adding the outliers from two large compressed threadblocks and storing the resultant outlier in the homomorphic output. The thread then sequentially processes corresponding blocks from the two compressed data inputs.

As illustrated in Figure 4, for two compressed blocks CB_N^1 and CB_N^2 with code-lengths x and y, the process varies: ❶ If both x and y are zero, indicating 'constant' blocks, only the '0' code-length is recorded in the homomorphic output, simplifying the process significantly compared to static methods. ❷ If x=0 but y≠0, meaning CB_N^1 is 'constant' and CB_N^2 is 'non-constant', only the bytes of CB_N^2 are copied to the output, reflecting its non-zero code-length and encoded bits. This method is similarly lightweight. ❸ A reversal of ❷, where CB_N^1 is 'non-constant' and CB_N^2 is 'constant', necessitates copying only the bytes of CB_N^1 for similar reasons. ❹ If both x and y are non-zero, indicating 'non-constant' blocks, both compressed blocks are partially decoded into integer prediction blocks via inverse fixed-length encoding (IFE). These prediction integers are then summed and re-encoded with fixed-length encoding (FE) to produce a new code-length z and encoding bits for the homomorphic compressed output. This procedure resembles the static homomorphic compression pipeline but improves memory efficiency by eliminating the need for allocating a large integer prediction array required by the static approach.

As discussed in Section III-B2, the quantization stage is the sole source of bounded compression error. Thus, our *h*Z-dynamic does not introduce additional errors beyond those inherent to the original compression process, as it does not involve a quantization phase.



CB$_N^i$ The Nth compressed data block for data i
IFE Inverse fixed-length encoding
CPT Computation
PB$_i$ The ith integer prediction block
FE Fixed-length encoding
Code length (CL)
Homomorphic compressed data from prior blocks
Compressed data blocks of input dataset 1
CB$_0^1$ CB$_1^1$ CB$_2^1$ ··· CB$_N^1$
For every block
IFE
Prediction blocks
PB$_0$ PB$_1$ ··· PB$_N$
Homomorphic compressed blocks
CB$_0^3$ CB$_1^3$ ··· CB$_N^3$
CPT
FE
Compressed data blocks of intput dataset 2
CB$_0^2$ CB$_1^2$ CB$_2^2$ ··· CB$_N^2$
Copy code-length 0 to homomorphic output
❶ x=0, y=0
0
Copy the compressed bytes of block CB$_N^2$ to homom. output
CB$_N^1$ CL=x
Dynamic Selector
❷ x=0, y≠0
y FE bytes of CB$_N^2$
❸ x≠0, y=0
x FE bytes of CB$_N^1$
CB$_N^2$ CL=y
❹ x≠0, y≠0
Copy the compressed bytes of block CB$_N^1$ to homom. output
IFE → CPT → FE
z New FE bytes
Store the new CL=z and new FE bytes to the homom. output

(a) Static Homomorphic Compression Pipeline

(b) Our Dynamic Homomorphic Compression Pipeline

Fig. 4: Compare our dynamic homomorphic compression pipeline with the static homomorphic compression pipeline.

C. *Co-designing the homomorphic compression-accelerated collective communication framework*

To optimize performance in collective communication using homomorphic compression, it is essential to redesign the compression-accelerated collective framework, as the current one is designed for traditional DOC workflow and can not be used with homomorphic compression [13]. We utilize two widely used collective computation operations–Reduce_scatter and Allreduce–to showcase our co-design strategies.

1) ***Reduce_scatter***

In this section, we detail how *h*ZCCL substantially enhances the efficiency of Reduce_scatter operation by reducing compression and computation costs with homomorphic compression. Figure 5 contrasts the C-Coll framework with our *h*ZCCL framework with the Reduce_scatter operation. In Round j of C-Coll, Node i compresses its designated data block $\sum_{n=i-j+1}^{i} B_n^{i+j-1}$ and sends the compressed data block to the next node, then decompresses the received compressed data block $(\sum_{n=i-j+2}^{i+1} B_n^{i+j})_C$ from the previous node before reducing it with its corresponding block B_{i-j+1}^{i+j}. This process repeats for every round across all nodes, incurring three DOC-related costs per round: compression cost (CPR), decompression cost (DPR), and computation cost (CPT) for a single data block, leading to a total cost for Reduce_scatter in C-Coll of $T_{\textit{C-Coll}}^{RS}$=$(N-1)CPR$+$(N-1)DPR$+$(N-1)CPT$, where N refers to the total node count.

Conversely, in the *h*ZCCL framework, Node i compresses all N data blocks $\{B_i^n | 1 \leq n \leq N\}$ only once in the Round 1, then sends one compressed block $\sum_{n=i-j+1}^{i}(B_n^{i+j-1})_C$ to the next node. Upon receiving a compressed block $\sum_{n=i-j+2}^{i+1}(B_n^{i+j})_C$, Node i applies the homomorphic compression to directly reduce two compressed blocks without decompressing. The cost of processing one compressed block homomorphically is noted as HPR. This makes the total compression and computation costs of the first round $N \times CPR$+DPR. For subsequent Round j, the compressed data blocks are transferred between nodes in intensive communications, and homomorphic compression is utilized to reduce the compressed data blocks as in the first round. Consequently, the compression and computation costs for each of these rounds are quantified as $1 \times HPR$. In the final Round N-1, each node employs homomorphic compression to produce the final compressed block, which is then decompressed to restore the final reduced data block. Thus, the compression and computation costs for this concluding round amount to $1 \times HPR$+$1 \times DPR$. The total cost for Reduce_scatter in *h*ZCCL becomes $T_{\textit{hZCCL}}^{RS}$=$(N)CPR$+$(1)DPR$+$(N-1)HPR$, making the difference between C-Coll and *h*ZCCL $T_{\textit{C-Coll}}^{RS} - T_{\textit{hZCCL}}^{RS} = (N-1)(DPR + CPT - HPR) - 1CPR - 1DPR$. Given that our homomorphic compressor–*h*Z-dynamic–is significantly faster than the conventional DOC workflow due to its lightweight design, the DPR+CPT greatly exceeds HPR, and this performance difference is even magnified by $(N-1)$ times. Therefore, *h*ZCCL soundly diminishes the runtime associated with compression and computation compared to C-Coll, elevating overall efficiency.

2) ***Allreduce***

This section focuses on optimizing the Allreduce operation, highlighting strategies that further boost its performance beyond the improvements made in the Reduce_scatter stage. Unlike Reduce_scatter, which solely focuses on collective computation, the ring-based Allreduce operation in-

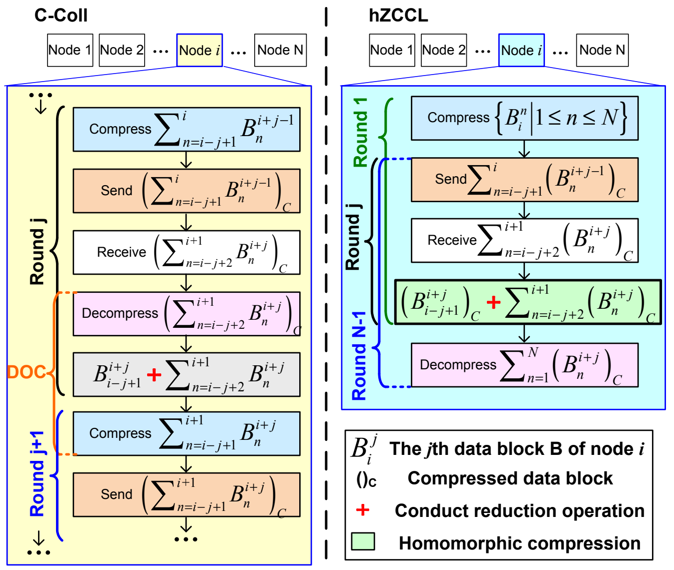

C-Coll
Node 1
Node 2
Node i
Node N
Compress $\sum_{n=i-j+1}^{i} B_n^{i+j-1}$
Send $\left(\sum_{n=i-j+1}^{i} B_n^{i+j-1}\right)_C$
Receive $\left(\sum_{n=i-j+2}^{i+1} B_n^{i+j}\right)_C$
Decompress $\left(\sum_{n=i-j+2}^{i+1} B_n^{i+j}\right)_C$
$B_{i-j+1}^{i+j} + \sum_{n=i-j+2}^{i+1} B_n^{i+j}$
Compress $\sum_{n=i-j+1}^{i+1} B_n^{i+j}$
Send $\left(\sum_{n=i-j+1}^{i+1} B_n^{i+j}\right)_C$
Round j
DOC
Round j+1
hZCCL
Node 1
Node 2
Node i
Node N
Compress $\{B_i^n \mid 1 \le n \le N\}$
Send $\sum_{n=i-j+1}^{i} \left(B_n^{i+j-1}\right)_C$
Receive $\sum_{n=i-j+2}^{i+1} \left(B_n^{i+j}\right)_C$
$\left(B_{i-j+1}^{i+j}\right)_C + \sum_{n=i-j+2}^{i+1} \left(B_n^{i+j}\right)_C$
Decompress $\sum_{n=1}^{N} \left(B_n^{i+j}\right)_C$
Round 1
Round j
Round N-1
B_i^j The jth data block B of node i
$()_C$ Compressed data block
+ Conduct reduction operation
Homomorphic compression

Fig. 5: High-level design of our *h*ZCCL in the ring-based Reduce_scatter algorithm. This algorithm completes in $N{-}1$ rounds, where N is the number of processes.

volves both collective data movement (Allgather) and computation (Reduce_scatter). The original Allgather design in the C-Coll framework requires each node to compress its data chunk and synchronize the compressed data size with others. Following synchronization, the compressed data are communicated among the nodes in a ring pattern. After $N{-}1$ communication rounds, each node decompresses the received compressed data chunks to obtain the gathered data, resulting in a total compression and computation cost of $T^{AG}_{\textit{C-Coll}}{=}CPR{+}(N{-}1)DPR$. This design notably diminishes the compression cost compared with the compression-enabled point-to-point communication method [25], [13], however, it is limited to the conventional DOC workflow and suffers from the suboptimal performance as the Reduce_scatter case detailed in the last section.

To further optimize the Allreduce operation, *h*ZCCL framework leverages the direct operation capability of homomorphic compression on compressed data. This approach eliminates the decompression step in the last round of the Reduce_scatter algorithm. Instead of decompressing the data for Allgather, the compressed data and their sizes are directly fed into the redesigned Allgather stage. This results in a compression and computation cost of $T^{RS}_{\textit{hZCCL}}{=}(N)CPR{+}(N{-}1)HPR$ for the Reduce_scatter stage. In the Allgather stage, the nodes do not compress the input data since it is already compressed. The nodes proceed to directly synchronize the sizes of these compressed data chunks with one another, ensuring all ensuing data exchanges occur in this compressed format. The final decompression of the $N{-}1$ compressed data chunks in the last round leads to a compression cost of $T^{AG}_{\textit{hZCCL}}{=}(N{-}1)DPR$. Accordingly, the total cost for the Allreduce operation in the *h*ZCCL framework is $T^{AR}_{\textit{hZCCL}} = T^{RS}_{\textit{hZCCL}} + T^{AG}_{\textit{hZCCL}} = (N)CPR + (N-1)DPR + (N-1)HPR$. In comparison, the C-Coll framework incurs a total cost of $T^{AR}_{\textit{C-Coll}} = T^{RS}_{\textit{C-Coll}} + T^{AG}_{\textit{C-Coll}} = (N)CPR + 2(N{-}1)DPR + (N{-}1)CPT$. The time difference between the two frameworks in executing the Allreduce operation, $T^{AR}_{\textit{C-Coll}} - T^{AR}_{\textit{hZCCL}} = (N-1)(DPR - HPR){+}(N{-}1)CPT$, highlights the efficiency of the *h*ZCCL framework which co-designs homomorphic compression with collective communication for reducing both compression and computation costs significantly.

IV. Experimental Evaluation

We present and discuss the evaluation results as follows.

A. Experimental Setup

For our experiments, we leverage a 512-node cluster, assigning one process per node, as inter-node communication is the major bottleneck in collectives. Each node is equipped with two Intel Xeon E5-2695v4 Broadwell processors and we utilize only one socket (18 threads) for compression in all collective experiments. These nodes are connected via Intel Omni-Path Architecture, providing 100 Gbps bandwidth. The absolute error bound for compression is set at 1E-4 by default.

Five real-world scientific applications from different domains are utilized in the evaluation as shown in Table I. They are: two distinct RTM application datasets [2], which are generated under two different simulation settings of the real-world 3D SEG/EAGE Overthrust model, the NYX application dataset, which is from a cosmological hydrodynamics simulation based on adaptive mesh [31], the CESM-ATM application dataset, which is from the atmosphere model of a climate simulation package CESM [18], and the Hurricane application dataset, which is from the simulation of hurricane Isabel from the National Center for Atmospheric Research [32]. The last three application datasets can be downloaded from SDRBench [33]. We summarize all the compression and collective communication solutions evaluated in Table II.

TABLE I: Information of the application datasets in evaluation.

Application	# fields	Dims per field	Total Size	Domain
Sim. Set. 1	3601	449x449x235	635.5 GB	Seismic Wave
Sim. Set. 2	151	849x849x235	95.3 GB	Seismic Wave
NYX	6	512x512x512	3.1 GB	Cosmology
CESM-ATM	79	1800x3600	2.0 GB	Climate Simu.
Hurricane	13	100x500x500	1.3 GB	Weather Simu.

TABLE II: Compression and collective communication solutions.

Solution	Description
ompSZp	CPU version of cuSZp's parallelism strategy (baseline)
***fZ*-light**	Our brand new ultra-fast error-bounded lossy compressor
***hZ*-dynamic**	*fZ*-light with dynamic homomorphic compression pipeline
Original Collectives (MPI)	No compression involved (baseline)
C-Coll (single-thread)	Single-thread mode of C-Coll (baseline)
C-Coll (multi-thread)	Multi-thread mode of C-Coll (baseline)
***h*ZCCL (single-thread)**	Single-thread mode of *h*ZCCL
***h*ZCCL (multi-thread)**	Multi-thread mode of *h*ZCCL

B. Evaluating the Homomorphic Compressor–hZ-dynamic

The *hZ*-dynamic homomorphic compressor consists of two key components: our *fZ*-light, a brand new optimized lossy compressor for CPU architectures that we made from scratch, and a dynamic homomorphic compression pipeline that adapts to data in real-time for efficient processing. The *hZ*-dynamic is specifically designed to boost both the speed and ratio of compression, making it exceptionally suited for distributed computing scenarios. To ensure consistency across various tests, the number of threads utilized in compression is fixed at 36 for all subsequent experiments.

1) Optimized compression workflow

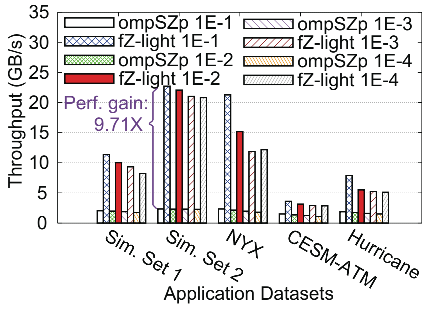
In our evaluation, we compare the compression ratio and quality of *fZ*-light against ompSZp (CPU version of cuSZp) across different application datasets with various relative error bounds to highlight the advantages of our optimized compression workflow. The comparison, detailed in Table III, reveals that *fZ*-light consistently outperforms ompSZp in terms of compression ratio across various relative error bounds (REL) and simulation settings. Notably, *fZ*-light achieves the maximum compression ratio improvements of up to 37.65 in CESM-ATM application dataset. This increase is attributed to the efficient outlier storage management in the workflow of *fZ*-light. Furthermore, the minimal improvement is achieved in the Simulation Setting 1 application dataset, where *fZ*-light reaches a 5.94 compression ratio improvement with the 1E-1 REL and is even 1.20 worse than ompSZp at the 1E-2 REL. This is because ompSZp benefits from its design to omit the 'zero' blocks in the original dataset and Simulation Setting 1 contains a considerable proportion of such blocks. Additionally, *fZ*-light exhibits a slightly better NRMSE (normalized root mean square error) than ompSZp in all cases, indicating that our workflow enhances compression ratios without compromising quality.

2) Ultra-fast fixed-length encoding scheme

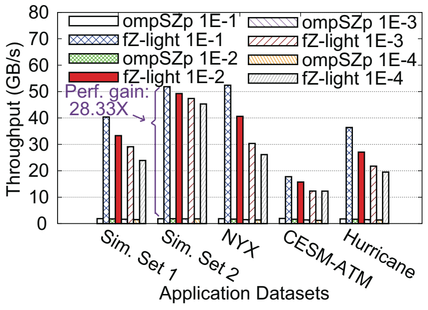
Figure 6 showcases a speed comparison between *fZ*-light and ompSZp (CPU version of cuSZp), focusing on both compression and decompression throughput across various relative error bounds and application datasets. The data reveals that *fZ*-light markedly surpasses ompSZp in speed, achieving up to 9.71× and 28.33× speedups in compression and decompression, respectively, for the simulation setting 2. The smallest though still significant speedups are achieved in the CESM-ATM application dataset, reaching 2.62× for compression and 10.09× for decompression. This substantial performance improvement stems primarily from the ultra-fast fixed-length encoding scheme of *fZ*-light, which employs efficient bit-shifting techniques for encoding integer prediction values into compressed bytes. The enhanced throughput of *fZ*-light significantly reduces compression-related runtime in compression-accelerated collectives, demonstrating its effectiveness in improving overall system performance.

3) Memory bandwidth efficiency evaluation

Our *fZ*-light features an optimized compression workflow and an ultra-fast fixed-length encoding scheme, outperforming ompSZp regarding its GPU-centric parallelism strategies. To underscore the efficiency of our optimizations, we showcase the average memory bandwidth efficiency as percentages in Table IV. According to this table, *fZ*-light achieves up to



(a) Compression



(b) Decompression

Fig. 6: Compression/Decompression Throughput (GB/s).

94.5% of the maximum memory bandwidth with a relative error bound of 1E-4 in the NYX application dataset, marking a 17.82× efficiency improvement over ompSZp. This highlights the effectiveness of our parallelism strategies in *fZ*-light. Notably, we determine the system peak memory throughput using the STREAM benchmark suite [21], selecting the highest throughput among the four provided by STREAM to calculate our memory utilization efficiency.

4) Dynamic homomorphic compression pipeline

In Table VI, we compare the average overall compression throughput, quality, and ratio of *hZ*-dynamic with *fZ*-light when conducting the reduce operation on two compressed data inputs. Unlike *fZ*-light, which necessitates decompressing data before the reduce operation and recompressing afterwards, *hZ*-dynamic directly operates on compressed data through an adaptive homomorphic compression pipeline, significantly enhancing speed. We can see that our *hZ*-dynamic outperforms our *fZ*-light in terms of performance in every application dataset. Specifically, *hZ*-dynamic achieves up to 379.08 GB/s of overall throughput in NYX, making a 36.53× improvement over the throughput (10.38 GB/s) of *fZ*-light. In simulation setting 2, the speedup is up to 31.43×. The smallest but also significant speedup is witnessed in the CESM-ATM, where *hZ*-dynamic is up to 11.03× faster than *fZ*-light. This significant performance boost stems from the ability of *hZ*-dynamic to (1) operate on compressed data directly, bypassing the decompression and recompression in the traditional DOC workflow of *fZ*-light, and (2) dynamically select the most lightweight homomorphic compression pipeline for data processing, outperforming static homomorphic compres-

TABLE III: Compression quality (NRMSE) with standard deviation (STD) and compression ratio (Ratio). The higher compression ratios and lower NRMSEs are underlined.

		Simulation Setting 1			Simulation Setting 2			NYX			CESM-ATM			Hurricane		
	REL	**Ratio**	**NRMSE**	**STD**	**Ratio**	**NRMSE**	**STD**	**Ratio**	**NRMSE**	**STD**	**Ratio**	**NRMSE**	**STD**	**Ratio**	**NRMSE**	**STD**
***fZ*-light**	1E-1	111.82	1.01E-02	8E-03	129.64	4.62E-03	3E-03	107.83	2.17E-02	2E-02	69.45	3.74E-02	2E-02	73.74	2.29E-02	2E-02
	1E-2	40.79	2.13E-03	2E-03	107.06	6.41E-04	5E-04	27.00	3.20E-03	3E-03	21.76	4.88E-03	1E-03	25.76	3.07E-03	2E-03
	1E-3	20.29	2.93E-04	2E-04	81.04	8.12E-05	7E-05	14.97	4.03E-04	3E-04	12.61	5.30E-04	9E-05	13.65	3.43E-04	2E-04
	1E-4	10.83	3.65E-05	2E-05	61.51	8.85E-06	7E-06	7.81	4.69E-05	2E-05	7.18	5.36E-05	8E-06	8.12	3.57E-05	2E-05
ompSZp	1E-1	105.89	1.09E-02	8E-03	118.36	4.87E-03	4E-03	85.27	2.38E-02	3E-02	31.80	4.00E-02	2E-02	46.94	2.45E-02	2E-02
	1E-2	41.99	2.25E-03	2E-03	100.20	6.60E-04	5E-04	20.25	3.27E-03	3E-03	11.33	5.12E-03	1E-03	17.91	3.15E-03	2E-03
	1E-3	19.98	3.01E-04	2E-04	77.75	8.39E-05	7E-05	9.59	4.18E-04	3E-04	6.10	5.32E-04	8E-05	9.73	3.47E-04	2E-04
	1E-4	10.71	3.74E-05	2E-05	60.21	8.94E-06	7E-06	5.69	4.73E-05	2E-05	3.98	5.37E-05	8E-06	6.34	3.58E-05	2E-05

TABLE IV: Mem. bandwidth efficiency of *fZ*-light & ompSZp.

		ompSZp		*fZ*-light	
	REL	**Compr.**	**Decom.**	**Compr.**	**Decom.**
Sim. Set. 2	1E-3	6.64%	3.55%	59.33%	91.13%
	1E-4	6.63%	3.57%	59.12%	87.51%
NYX	1E-3	6.58%	5.08%	44.83%	94.45%
	1E-4	6.72%	5.30%	52.66%	94.50%

TABLE V: Performance (GB/s) and pipeline selection percentages of dynamic homomorphic compression pipeline in *hZ*-dynamic. The speedups are based on *fZ*-light.

	Speedups	***hZ*-dynamic Throu.**	**Pipeline 1**	**Pipeline 2**	**Pipeline 3**	**Pipeline 4**
NYX	50.01	537.41	99.36%	0.05%	0.40%	0.19%
Sim. Set. 1	25.95	156.36	53.84%	0.00%	46.16%	0.00%
Hurricane	20.58	79.49	0.75%	0.00%	99.25%	0.00%
Sim. Set. 2	8.87	71.56	84.46%	11.14%	1.53%	2.86%
CESM-ATM	2.62	9.00	1.20%	5.10%	5.06%	88.64%

sion methods which often involve partial data decompression and recompression.

To highlight the benefits of the dynamic homomorphic compression pipeline in *hZ*-dynamic, we detail the utilization percentages of four different pipelines when homomorphically compressing two fields/snapshots from various application datasets with a 1E-3 relative error bound in Table V. Notably, in the NYX application dataset, *hZ*-dynamic achieves a remarkable 537.41 GB/s throughput and a 50.01× speedup over *fZ*-light, primarily because 99.36% of the compression tasks leverage pipeline 1. This pipeline is exceptionally efficient, since it only copies the '0' code-length into the homomorphically compressed data. In Simulation Setting 2, performance slightly decreases to 156.36 GB/s, with a 25.95× speedup compared with *fZ*-light, as the percentage of pipeline 1 drops to 53.84% and the proportion of pipelines 2 and 3 increases to 46.16%. This demonstrates that although pipelines 2 and 3 are efficient, they are more expensive than pipeline 1 because they require copying all compressed bytes of a block to the homomorphically compressed data. For the CESM-ATM application dataset, the predominance of pipeline 4 at 88.64% correlates with a decrease in performance to 9 GB/s, though it remains 2.62× faster than *fZ*-light. This underscores the higher cost of pipeline 4, necessitating partial decompression and recompression of compressed data blocks, unlike the more lightweight pipelines 1-3. This comparison illustrates the significant time savings offered by the dynamic pipeline of *hZ*-dynamic over the static homomorphic compression methods, which are similar to pipeline 4.

Moreover, *hZ*-dynamic slightly surpasses *fZ*-light in PSNR and NRMSE, attributed to the data accuracy loss of *fZ*-light during recompression–a step *hZ*-dynamic circumvents with its efficient design. This design difference also causes a minor compression ratio difference between *hZ*-dynamic and *fZ*-light since lost accuracy results in less compressed bytes.

C. Evaluating the co-designed homomorphic compression-accelerated collective algorithms

In this section, we evaluate the performance of our *h*ZCCL collective operations against C-Coll, utilizing 64 Intel Broadwell nodes.

1) Reduce_scatter

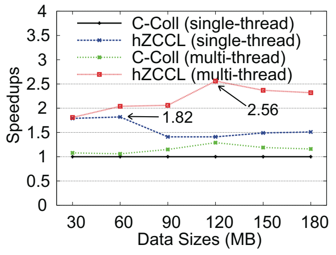
In Figure 7, we detail the performance comparison of our *h*ZCCL against C-Coll across two scientific application datasets using the Reduce_scatter operation. In Simulation Setting 1, *h*ZCCL (multi-thread) outperforms all counterparts, including *h*ZCCL in single-thread mode, with up to 2.01× speedup over C-Coll (multi-thread). Similarly, *h*ZCCL (single-thread) reaches a 1.82× speedup compared to C-Coll (single-thread). These performance trends hold in Simulation Setting 2, where *h*ZCCL in multi-thread and single-thread modes achieve performance enhancements of 1.64× and 1.31×, respectively. The notable speedups stem from co-designing our homomorphic compressor, *hZ*-dynamic, with the compression-accelerated Reduce_scatter algorithm, effectively reducing the DOC-handling time. Additionally, the improvement tends to increase with data size, highlighting the advantages of *h*ZCCL's efficiency in handling larger datasets by optimizing compression and computation runtime.

2) Allreduce

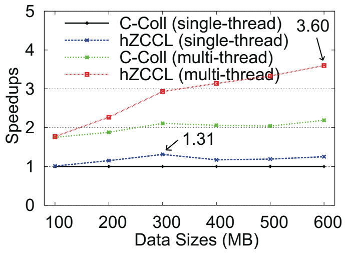
Figure 8 demonstrates that our *h*ZCCL (single-thread) significantly outperforms C-Coll in (single-thread), with speedups of 1.78× in Simulation Setting 1 and 1.55× in Simulation Setting 2. Furthermore, *h*ZCCL in multi-thread mode achieves even greater speedups of 2.10× and 2.00× over C-Coll in multi-thread mode for Simulation Settings 1 and 2, respectively. Allreduce operation consists of both collective data movement (Allgather) and collective computation (Reduce_scatter). The remarkable improvement in performance is attributed not only to the gains in the Reduce_scatter operation

TABLE VI: Overall compression performance (GB/s). The higher performance is underlined.

		Simulation Setting 1				Simulation Setting 2				NYX				CESM-ATM				Hurricane			
	REL	Throu.	Ratio	NRMSE	STD	Throu.	Ratio	NRMSE	STD	Throu.	Ratio	NRMSE	STD	Throu.	Ratio	NRMSE	STD	Throu.	Ratio	NRMSE	STD
***h*Z-dynamic**	1E-1	210.04	102.67	1.0E-02	8E-03	331.32	124.71	4.6E-03	3E-03	379.08	97.99	2.2E-02	2E-02	55.19	57.56	3.7E-02	2E-02	99.03	62.01	2.3E-02	2E-02
	1E-2	98.66	32.22	2.1E-03	2E-03	254.66	99.19	6.4E-04	5E-04	248.48	20.43	3.2E-03	3E-03	15.67	16.31	4.9E-03	1E-03	21.08	20.13	3.1E-03	2E-03
	1E-3	69.43	16.51	2.9E-04	2E-04	207.11	73.93	8.1E-05	7E-05	109.39	11.54	4.0E-04	3E-04	9.42	9.69	5.3E-04	9E-05	13.43	11.21	3.4E-04	2E-04
	1E-4	64.30	9.21	3.7E-05	2E-05	174.79	56.62	8.9E-06	7E-06	32.50	6.57	4.7E-05	2E-05	7.90	6.01	5.4E-05	8E-06	10.71	7.13	3.6E-05	2E-05
***f*Z-light (DOC)**	1E-1	8.35	107.40	1.1E-02	8E-03	10.54	126.20	5.6E-03	4E-03	10.38	99.69	2.2E-02	2E-02	5.00	60.84	4.3E-02	2E-02	6.69	65.70	2.8E-02	3E-02
	1E-2	7.02	33.12	2.6E-03	2E-03	10.03	99.70	8.4E-04	7E-04	8.44	20.82	4.1E-03	4E-03	4.14	16.77	7.2E-03	2E-03	5.49	20.92	4.4E-03	3E-03
	1E-3	6.20	16.76	3.9E-04	3E-04	9.57	74.69	1.2E-04	1E-04	6.71	11.77	6.2E-04	4E-04	3.40	9.78	7.3E-04	2E-04	4.37	11.32	5.0E-04	3E-04
	1E-4	5.50	9.26	4.7E-05	3E-05	9.13	56.72	1.2E-05	1E-05	5.63	6.66	6.6E-05	3E-05	3.22	6.03	7.8E-05	2E-05	4.21	7.14	5.6E-05	4E-05



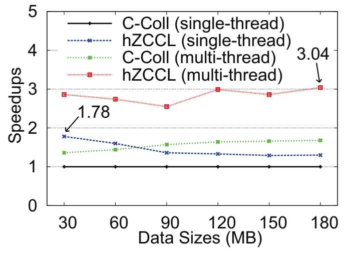
(a) Simulation Setting 1



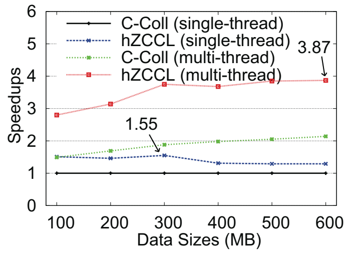
(b) Simulation Setting 2

Fig. 7: Evaluation of *h*ZCCL with Reduce_scatter.

discussed previously but also to our tailored optimization that removes the need for decompression in the Reduce_scatter phase and compression in the Allgather phase, further elevating the efficiency of Allreduce operation.



(a) Simulation Setting 1



(b) Simulation Setting 2

Fig. 8: Evaluation of *h*ZCCL using Allreduce.

D. *Comparisons of* hZCCL *with other collective libraries*

In this section, we present the performance evaluation of our *h*ZCCL-accelerated collective operations, utilizing up to 512 Intel Broadwell nodes. We compare our results against two baselines: the compression-accelerated collectives from C-Coll [13] and the traditional collectives offered by MPICH [28], as outlined in Table II.

1) Reduce_scatter

In this section, we assess the performance of our *h*ZCCL-accelerated Reduce_scatter operation across various data sizes and node counts.

Evaluation with different message sizes.

In Figure 9, we evaluate the performance of *h*ZCCL-accelerated Reduce_scatter operation across various data sizes, employing 64 Intel Broadwell nodes. The results demonstrate that both single-thread and multi-thread versions of *h*ZCCL significantly outperform the corresponding versions of C-Coll, achieving speedups of up to 1.58× and 4.04× respectively, when compared to the original MPI Reduce_scatter operation (MPI). This notable improvement is primarily attributed to the reduced communication sizes after compression and the decreased compression and computation expenses in *h*ZCCL. Our approach co-designs collective communication with homomorphic compression, showcasing superior DOC-handling throughput than the traditional DOC workflow, as detailed in Section IV-B4. Moreover, the performance enhancement observed tends to grow with increasing data sizes. This is because the larger the data size is, the more congested the network is, which benefits more from the decreased communication volume in our *h*ZCCL.

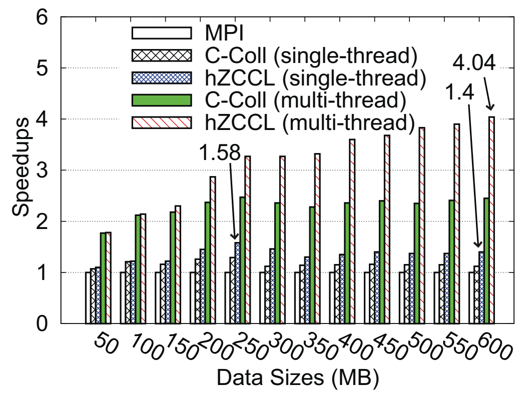


Fig. 9: Performance evaluation of *h*ZCCL-accelerated Reduce_scatter against MPI and C-Coll in different data sizes.

Evaluation with different node counts.

Figure 10 showcases the scalability evaluation of our *h*ZCCL-accelerated Reduce_scatter operation using the complete RTM application dataset of 646 MB, tested on up to 512 Intel Broadwell nodes. Consistent with previous findings, both single-thread and multi-thread modes of *h*ZCCL outperform their C-Coll counterparts, achieving speedups of up to 1.9× and 5.85×, respectively, over traditional MPI implementations. A notable trend observed is that performance improvement initially increases as the node count rises, but eventually decreases and stabilizes. This initial performance boost occurs as network congestion grows with more nodes participating in collective communication. Our *h*ZCCL effectively reduces the message size, thereby mitigating the impact of network congestion. However, the nature of Reduce_scatter as a 'Scatter'-like operation leads to a diminishing output data size with an increasing number of nodes, elevating the frequency of communication and compression while only marginally growing the overall communication volume. Consequently, the advantage of reduced communication volume is somewhat offset by the increased compression latency. Even so, with 512 nodes, *h*ZCCL in both single-thread and multi-thread modes still secures substantial performance gains of 1.46× and 4.12×, respectively, compared to the MPI baseline.

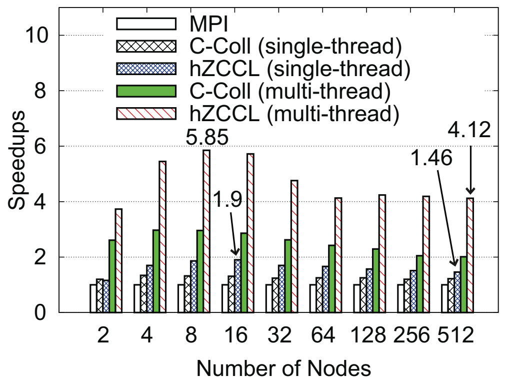


Fig. 10: Scalability evaluation of *h*ZCCL-accelerated Reduce_scatter against MPI and C-Coll in different node counts.

2) Allreduce

In this section, we evaluate our *h*ZCCL-accelerated Allreduce operation across a variety of data sizes and node counts.

Evaluation with different message sizes.

We assess the performance of our *h*ZCCL-accelerated Allreduce operation for data sizes up to 600 MB using 64 Intel Broadwell nodes. Figure 11 demonstrates that *h*ZCCL consistently surpasses both C-Coll and MPI for all data sizes. It is worth noting that, as the data size expands, the speedup of *h*ZCCL increases, reaching up to 1.96× and 5.35× improvements over MPI. This underscores the efficiency of *h*ZCCL in reducing message sizes and decreasing compression runtime, enhancing collective performance significantly.

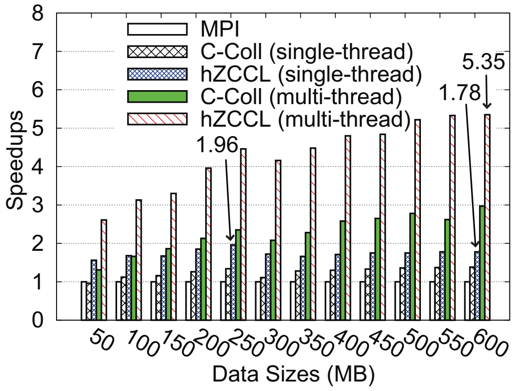


Fig. 11: Performance evaluation of *h*ZCCL-accelerated Allreduce against MPI and C-Coll in different data sizes.

Evaluation with different node counts.

We evaluate the scalability of our *h*ZCCL-accelerated Allreduce operation using the full RTM application dataset of 646 MB across up to 512 Intel Broadwell nodes. As depicted in Figure 12, both the single-thread and multi-thread modes of *h*ZCCL consistently surpass C-Coll and MPI, achieving maximum speedups of 2.12× and 6.77× over MPI, respectively. This improvement stems from the reduced communication volume and lower compression/computation costs in our homomorphic compression-accelerated Allreduce operation. Similar to the Reduce_scatter operation, the speedup achieved by the *h*ZCCL-accelerated Allreduce initially increases with the node count but eventually experiences a slight decrease before stabilizing. The reason is similar to the Reduce_scatter case discussed previously in Section IV-D1. However, the Allreduce operation differs since it does not 'scatter' the data in the same manner as Reduce_scatter, and the final output size is always the same as the original data input. This characteristic helps to maintain the benefits brought from the decreased communication volume of *h*ZCCL, leading to significant performance enhancements of 1.88× and 5.58× compared to MPI, even with 512 nodes.

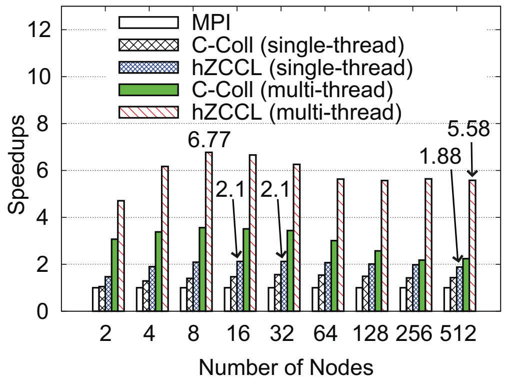


Fig. 12: Scalability evaluation of *h*ZCCL-accelerated Allreduce against MPI and C-Coll in different node counts.

E. Image Stacking Performance and Accuracy Analysis

In this section, we use the image stacking application to assess the performance and accuracy of our *h*ZCCL. Image stacking is a method extensively utilized in diverse scientific domains, including atmospheric science and geology, to create high-resolution images by stacking multiple single images. This procedure inherently performs an Allreduce operation. As highlighted by Gurhem in [34], researchers leverage MPI to amalgamate these singular images into final images.

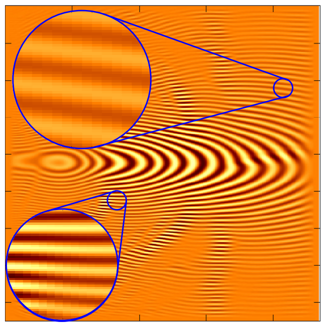
Table VII showcases that our *h*ZCCL-accelerated Allreduce effectively surpasses C-Coll in both single-thread and multi-thread modes, achieving speedups of 1.81× and 5.02× over MPI with an absolute error bound of 1E-4. This notable performance boost is attributed to reduced message sizes and low DOC-related expenses in *h*ZCCL. In comparison to C-Coll, *h*ZCCL soundly cuts down the compression and computation (CPR+CPT) runtime, with a percentage decrease from 81.95% to 77.96% in single-thread scenario and a significant drop from 59.04% to 38.61% in multi-thread context. These improvements result from our strategic co-design, integrating

TABLE VII: Image stacking performance analysis (The speedups are based on MPI. The last three columns are performance breakdowns).

	Speedups	CPR+CPT	MPI	Others
***h*ZCCL (single-thread)**	1.81	77.96%	21.76%	0.28%
C-Coll (single-thread)	1.45	81.95%	17.76%	0.29%
***h*ZCCL (multi-thread)**	5.02	38.61%	60.42%	0.96%
C-Coll (multi-thread)	3.34	59.04%	40.31%	0.65%

homomorphic compression with collective communication to enable direct operations on compressed data, thus achieving a more efficient DOC-handling throughput.

Beyond performance metrics, we also present both numerical (PSNR and NRMSE) and visual analyses to showcase the controlled error propagation achieved by our *h*ZCCL. With an absolute error bound of 1E-4, *h*ZCCL achieves a satisfactory PSNR of 62.00 and a sound NRMSE of 8.0E-4. Figure 13 displays visual comparisons of stacking images using *h*ZCCL and the original un-compressed MPI method. The comparison reveals no visual difference between the two images, indicating that *h*ZCCL preserves excellent image quality. This combination of high numerical scores and visual quality underlines the effectiveness of *h*ZCCL in delivering enhanced performance while maintaining precise control over error propagation.


(a) MPI (lossless)

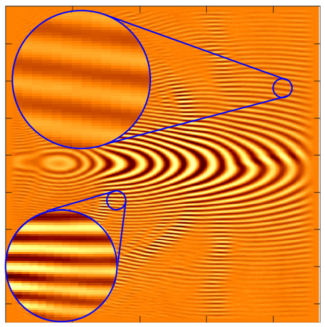

(b) *h*ZCCL

Fig. 13: Visualization of final stacking image.

V. Conclusion and Future Work

This paper introduces *h*ZCCL, a novel co-design of homomorphic compression and collective communication. Through evaluation on up to 512 nodes and across five application datasets, our dynamic homomorphic compressor, *h*Z-dynamic, has demonstrated impressive compression throughput, reaching as high as 379.08 GB/s, which is 36.53× faster than traditional DOC workflow. Additionally, collectives accelerated by *h*ZCCL significantly surpass baselines, achieving performance gains of up to 2.12× and 6.77× over MPI in single-thread and multi-thread modes, respectively.

In the future, we plan to integrate *h*ZCCL or its homomorphic compression design into applications with communication bottlenecks to enhance their performance. Additionally, we will tailor homomorphic compression algorithms to the specific data characteristics of various applications and use cases. Furthermore, we will expand *h*ZCCL's applicability to a broader range of hardware platforms.

VI. Acknowledgment

This material was supported by the U.S. Dept. of Energy, Office of Science, Advanced Scientific Computing Research (ASCR), under contracts DE-AC02-06CH11357 and DE-SC0024207. The material was also supported by the National Science Foundation under Grants OAC-2003709, OAC-2104023, OAC-2311875, OAC-2340982, OAC-2348465, and OIA-2327266. The experimental resource for this paper was provided by the Laboratory Computing Resource Center on the Bebop cluster at Argonne National Laboratory.

References

[1] Y. Zhao, A. De Nicola, T. Kawakatsu, and G. Milano, "Hybrid particle-field molecular dynamics simulations: Parallelization and benchmarks," Journal of computational chemistry, vol. 33, pp. 868–80, 03 2012.

[2] S. Kayum et al., "GeoDRIVE – a high performance computing flexible platform for seismic applications," First Break, vol. 38, no. 2, pp. 97–100, 2020.

[3] W. He, H. Guo, T. Peterka, S. Di, F. Cappello, and H.-W. Shen, "Parallel partial reduction for large-scale data analysis and visualization," in 2018 IEEE 8th Symposium on Large Data Analysis and Visualization (LDAV), 2018, pp. 45–55.

[4] A. M. Abdelmoniem, A. Elzanaty, M.-S. Alouini, and M. Canini, "An efficient statistical-based gradient compression technique for distributed training systems," 2021.

[5] S. Chunduri, S. Parker, P. Balaji, K. Harms, and K. Kumaran, "Characterization of mpi usage on a production supercomputer," in SC18: International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE, 2018, pp. 386–400.

[6] M. Bayatpour and M. A. Hashmi, "Salar: Scalable and adaptive designs for large message reduction collectives," in 2018 IEEE International Conference on Cluster Computing (CLUSTER), 2018, pp. 1–10.

[7] G. Almási, P. Heidelberger, C. J. Archer, X. Martorell, C. C. Erway, J. E. Moreira, B. Steinmacher-Burow, and Y. Zheng, "Optimization of mpi collective communication on bluegene/l systems," in Proceedings of the 19th Annual International Conference on Supercomputing, 2005.

[8] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in mpich," The International Journal of High Performance Computing Applications, vol. 19, no. 1, pp. 49–66, 2005.

[9] P. Patarasuk and X. Yuan, "Bandwidth optimal all-reduce algorithms for clusters of workstations," Journal of Parallel and Distributed Computing, vol. 69, no. 2, pp. 117–124, 2009.

[10] S. Di and F. Cappello, "Fast error-bounded lossy hpc data compression with sz," in 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS). IEEE, 2016, pp. 730–739.

[11] X. Yu, S. Di, K. Zhao, J. Tian, D. Tao, X. Liang, and F. Cappello, "Ultrafast error-bounded lossy compression for scientific datasets," in Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing, 2022.

[12] P. Lindstrom, "Fixed-rate compressed floating-point arrays," IEEE Transactions on Visualization and Computer Graphics, vol. 20, pp. 2674–2683, 2014.

[13] J. Huang, S. Di, X. Yu, Y. Zhai, Z. Zhang, J. Liu, X. Lu, K. Raffenetti, H. Zhou, K. Zhao, Z. Chen, F. Cappello, Y. Guo, and R. Thakur, "An optimized error-controlled mpi collective framework integrated with lossy compression," in 2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2024, pp. 752–764.

[14] Y. Huang, S. Di, X. Yu, G. Li, and F. Cappello, "cuszp: An ultra-fast gpu error-bounded lossy compression framework with optimized end-to-end performance," in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2023.

[15] K. Zhao, S. Di, M. Dmitriev, T.-L. D. Tonellot, Z. Chen, and F. Cappello, "Optimizing error-bounded lossy compression for scientific data by dynamic spline interpolation," in 2021 IEEE 37th International Conference on Data Engineering (ICDE), 2021, pp. 1643–1654.

[16] J. Huang, J. Liu, S. Di, Y. Zhai, Z. Jian, S. Wu, K. Zhao, Z. Chen, Y. Guo, and F. Cappello, "Exploring wavelet transform usages for error-bounded scientific data compression," in 2023 IEEE International Conference on Big Data (BigData), 2023, pp. 4233–4239.

[17] 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, ser. SC '19. New York, NY, USA: Association for Computing Machinery, 2019.

[18] J. E. Kay and et al., "The Community Earth System Model (CESM) large ensemble project: A community resource for studying climate change in the presence of internal climate variability," Bulletin of the American Meteorological Society, vol. 96, no. 8, pp. 1333–1349, 2015.

[19] J. Kim et al., "QMCPACK: an open source ab initio quantum monte carlo package for the electronic structure of atoms, molecules and solids," Journal of Physics: Condensed Matter, vol. 30, no. 19, p. 195901, 2018.

[20] A. N. Laboratory, "cuszp-a lossy error-bounded compression library for compression of floating-point data in nvidia gpu." https://github.com/szcompressor/cuSZp, 2023.

[21] J. McCalpin, "Memory bandwidth and machine balance in high performance computers," IEEE Technical Committee on Computer Architecture Newsletter, pp. 19–25, 12 1995.

[22] W. Wen, C. Xu, F. Yan, C. Wu, Y. Wang, Y. Chen, and H. Li, "Terngrad: ternary gradients to reduce communication in distributed deep learning," in Proceedings of the 31st International Conference on Neural Information Processing Systems, 2017.

[23] Y. Lin, S. Han, H. Mao, Y. Wang, and W. J. Dally, "Deep Gradient Compression: Reducing the communication bandwidth for distributed training," in The International Conference on Learning Representations, 2018.

[24] M. Li, R. B. Basat, S. Vargaftik, C. Lao, K. Xu, M. Mitzenmacher, and M. Yu, "THC: Accelerating distributed deep learning using tensor homomorphic compression," in 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24), 2024.

[25] Q. Zhou, C. Chu, N. S. Kumar, P. Kousha, S. M. Ghazimirsaeed, H. Subramoni, and D. K. Panda, "Designing high-performance mpi libraries with on-the-fly compression for modern gpu clusters," in 2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2021.

[26] Q. Zhou, P. Kousha, Q. Anthony, K. Shafie Khorassani, A. Shafi, H. Subramoni, and D. K. Panda, "Accelerating mpi all-to-all communication with online compression on modern gpu clusters," in High Performance Computing: 37th International Conference, ISC High Performance 2022, Hamburg, Germany, May 29 – June 2, 2022, Proceedings, 2022.

[27] J. Huang, S. Di, X. Yu, Y. Zhai, J. Liu, Y. Huang, K. Raffenetti, H. Zhou, K. Zhao, X. Lu, Z. Chen, F. Cappello, Y. Guo, and R. Thakur, "gzccl: Compression-accelerated collective communication framework for gpu clusters," in Proceedings of the 38th ACM International Conference on Supercomputing, ser. ICS '24, 2024, p. 437–448. [Online]. Available: https://doi.org/10.1145/3650200.3656636

[28] A. N. Laboratory, "Mpich – a high-performance and widely portable implementation of the mpi-4.0 standard." https://www.mpich.org, 2023.

[29] K. Zhao, S. Di, M. Dmitriev, T.-L. D. Tonellot, Z. Chen, and F. Cappello, "Optimizing error-bounded lossy compression for scientific data by dynamic spline interpolation," in 2021 IEEE 37th International Conference on Data Engineering (ICDE), 2021, pp. 1643–1654.

[30] T. Agarwal, S. Di, J. Huang, Y. Huang, G. Gopalakrishnan, R. Underwood, K. Zhao, X. Liang, G. Li, and F. Cappello, "Hoszp: An efficient homomorphic error-bounded lossy compressor for scientific data," 2024.

[31] NYX simulation, https://amrex-astro.github.io/Nyx, 2019, online.

[32] Hurricane ISABEL simulation data, http://vis.computer.org/vis2004contest/data.html, 2004, online.

[33] K. Zhao, S. Di, X. Lian, S. Li, D. Tao, J. Bessac, Z. Chen, and F. Cappello, "SDRBench: Scientific data reduction benchmark for lossy compressors," in 2020 IEEE International Conference on Big Data (Big Data), 2020, pp. 2716–2724.

[34] J. Gurhem, H. Calandra, and S. G. Petiton, "Parallel and distributed task-based kirchhoff seismic pre-stack depth migration application," in 2021 20th International Symposium on Parallel and Distributed Computing (ISPDC), 2021, pp. 65–72.

Appendix: Artifact Description/Artifact Evaluation

Artifact Description/Evaluation (AD/AE)

I. Overview of Contributions and Artifacts

A. Paper's Main Contributions

- C_1 We present a homomorphic compressor with a runtime heuristic to dynamically select efficient compression pipelines for reducing the cost of decompression-operation-compression (DOC) handling.
- C_2 We present a homomorphic compression-communication co-design, *hZCCL*, which enables operations to be performed directly on compressed data, saving the cost of time-consuming decompression and recompression.
- C_3 We evaluate *hZCCL* with up to 512 nodes and across five application datasets. The experimental results demonstrate that our homomorphic compressor achieves a single-socket CPU throughput of up to 379.08 GB/s, surpassing the conventional DOC workflow by up to 36.53×.
- C_4 Moreover, our *hZCCL*-accelerated collectives outperform two state-of-the-art baselines, delivering speedups of up to 2.12× and 6.77× compared to original MPI collectives in single-thread and multi-thread modes, respectively, while maintaining the data accuracy.

B. Computational Artifacts

hZCCL-artifact (A_1) is available at the following link:

A_1 DOI: `10.5281/zenodo.13317638`.

The Table below illustrates how the contributions of *hZCCL* relate to the experimental results presented in the paper.

Artifact ID	Contributions Supported	Related Paper Elements
A_1	C_1&3	Table III
A_1	C_2&4	Figure 11-12

II. Artifact Identification

A. Computational Artifact A_1

Relation To Contributions

The artifact enables the execution of key experiments described in our paper: (1) a comparison of compression throughput and quality between our proposed dynamic homomorphic compressor—referred to as *hZ-dynamic*—and the conventional DOC workflow (see Table III); (2) an evaluation of the performance of the *hZCCL*-accelerated Allreduce operation in comparison to other collective libraries, considering various message sizes and node counts (refer to Figures 11 and 12).

Expected Results

- E_1 *hZ-dynamic* significantly outperforms *fZ-light* (DOC) across all datasets, not only in terms of speed but also achieving slightly better compression quality.
- E_2 *hZCCL* consistently exceeds the performance of both C-Coll and the original MPI for all tested data sizes and node counts.

Expected Reproduction Time (in Minutes)

Setting up the artifact may take between 10 to 30 minutes, depending on your network speed and hardware availability. The expected execution time of this artifact on Intel Broadwell nodes is approximately 30 minutes. Analyzing the artifact may require an additional 10 minutes.

Artifact Setup (incl. Inputs)

Hardware: For our experiments, we utilize a cluster consisting of 512 nodes. Each node is equipped with two Intel Xeon E5-2695v4 Broadwell processors. The nodes are interconnected using Intel Omni-Path Architecture, which provides a bandwidth of 100 Gbps.

Software: Computational artifact: hZCCL-artifact. Software: MPICH 4.1.3, available for download from the official website at https://www.mpich.org.

Datasets / Inputs:

- D_1 NYX dataset, which can be downloaded from the SDRBench at: Link: `NYX-dataset`.
- D_2 Hurricane dataset, which can be downloaded from the SDRBench at: Link: `Hurricane-dataset`.
- D_3 CESM-ATM dataset, which can be downloaded from the SDRBench at: Link: `CESM-ATM-dataset`.
- D_4 The RTM dataset, derived from proprietary simulations, is not publicly accessible. Therefore, it is excluded from this artifact.

Installation and Deployment: In this section, we outline the necessary steps to build and prepare the computational artifact for use.

1) Install an MPI implementation:
   - We recommend using MPICH 4.1.3, which is available for download at the official MPICH website: https://www.mpich.org.
   - Comprehensive configuration and building instructions are provided on the website.
2) Environment setup:
   - After installation, add MPICH to your system's PATH using the following command: `export PATH=mpich/install/bin:$PATH`
   - Verify the installation by checking the location of the MPICH compiler with: `which mpicc`
   - Note: Building MPICH from scratch may take about 30 minutes due to its extensive library components.
3) Download datasets and artifact:
   - Download the hZCCL-artifact (computational artifact) and the NYX dataset from the provided links.
   - Ensure there is at least 30 GB of available space on your directory for the datasets.

- The download process should take less than 5 minutes with a fast internet connection.
- Optionally, the Hurricane and the CESM-ATM datasets are also available for download, depending on your time and space constraints.

4) Configure and build the hZCCL-artifact:

```
    # Decompress the downloaded files
    # Change to the artifact directory and set
up the installation directory:
    cd hZCCL-artifact
    mkdir install

    # Run the configuration script:
    ./autogen.sh
    ./configure --prefix=$(pwd)/install --
enable-openmp CC=mpicc

    # Compile the artifact using multiple
threads:
    make -j

    # Install the compiled artifact:
    make install

    # Change to the jobs directory to proceed
with running the artifact:
    cd jobs
```

By following these instructions, you will set up your system with the necessary software and datasets to run the computational artifact efficiently. Make sure all paths and dependencies are correctly configured before proceeding with the experiments.

Artifact Execution

Evaluate dynamic homomorphic compressor–hZ-dynamic and the traditional DOC workflow: Please adhere to the instructions below to evaluate the compression throughput and quality of our dynamic homomorphic compressor, *hZ-dynamic*, compared to the conventional DOC workflow.

- Modify the `evaluate_compressor_NYX.sh` shell script to include the paths for your NYX dataset and the installation directory. Also, update your Slurm configuration settings as needed. Look for the contents marked with `$XXX` to make these changes.
- Submit the updated `evaluate_compressor_NYX.sh` script and wait for the job to complete.
- Optionally, you may alter the script to test additional datasets.

Evaluate hZCCL with other baselines using different data sizes: Please follow the instructions below to compare *hZCCL* with other baseline methods (C-Coll and MPI) across various data sizes:

- Please update the `different_sizes.sh` shell script with the size of your chosen dataset, the path to your dataset, and the installation directory. Additionally, configure your own Slurm settings as needed. All segments requiring your input are marked with `$XXX`.
- Submit the revised `different_sizes.sh` script and wait for the processing to finish.

Evaluate hZCCL with other baselines using various node counts: Please adhere to the instructions below to evaluate *hZCCL* alongside other baselines (C-Coll and MPI) using different node counts:

- Please modify the `different_nodes.sh` shell script to update the dataset path and dataset sizes according to your chosen dataset. You will also need to update the installation directory and configure your Slurm settings. All segments requiring modifications are highlighted with `$XXX`.
- Submit the updated `different_nodes.sh` script and wait for the job to complete.

Artifact Analysis (incl. Outputs)

Evaluate dynamic homomorphic compressor–hZ-dynamic and the traditional DOC workflow: The expected outputs:

```
Evaluate hZ-dynamic with traditional DOC workflow
Error bound is 1E-1
Dataset: $YOUR_DIR/SDRBENCH-EXASKY-NYX-512x512x512/
baryon_density.f32

hZ-dynamic performance: time: 0.000899 s, throughput
: 597.186776 GBps
Traditional DOC workflow (decompression+operation+
compression) performance: time: 0.048888 s,
throughput: 10.981650 GBps
hZ-dynamic speedup: 54.38X

Traditional DOC workflow quality:
Min=0.11599393188953399658, Max=231724.578125, range
=231724.46875
Max absolute error = 33856.3203125000
Max relative error = 0.146106
Max pw relative error = 1.000000
PSNR = 79.254, NRMSE= 0.00010897
Compression Ratio = 143.965410

hZ-dynamic quality:
...

Error bound is 1E-2
...

Error bound is 1E-3
...

Error bound is 1E-4
...
```

The most important information of the outputs is the `hZ-dynamic speedup`, which demonstrates how fast is the *hZ-dynamic* compared with the traditional DOC workflow. Additionally, the `Traditioanl DOC workflow quality` and `hZ-dynamic quality` assess the compression quality and ratios of the two methods.

Evaluate hZCCL with other baselines using different data sizes: The expected outputs:

```
Running compression-accelerated allreduce with
different data sizes
NNODES: 64, DATADIR: $YOUR_DIR, BLOCKSIZE: 36,
ERRORBOUND: 1E-4, KERNELMAX: 4, KERNELMIN: 0,
START_SIZE: 52428800, NUMTHREADS: 18

Kernel 0 (52428800,677552940,52428800 or 50MB)
```

```
Compression-accelerated Kernel 0 For datasize:
52428800 bytes, the avg_time is xxx us, the max_time
 is xxx us, the min_time is xxx us
Compression-accelerated Kernel 0 For datasize:
104857600 bytes, the avg_time is xxx us, the
max_time is xxx us, the min_time is xxx us
...

Kernel 1 (52428800,677552940,52428800 or 50MB)
...

Kernel 2 (52428800,677552940,52428800 or 50MB)
...

Kernel 3 (52428800,677552940,52428800 or 50MB)
...

Kernel 4 (52428800,677552940,52428800 or 50MB)
...
```

The different kernels used in our experiments include the following:

1) Kernel 0: The original `MPI_Allreduce`.
2) Kernel 1: The multi-thread mode of C-Coll.
3) Kernel 2: The multi-thread mode of *hZCCL*.
4) Kernel 3: The single-thread mode of C-Coll.
5) Kernel 4: The single-thread mode of *hZCCL*.

You are expected to observe that *hZCCL* outperforms both C-Coll and MPI across all data sizes.

Evaluate hZCCL with other baselines using various node counts: The expected outputs:

```
Running compression-accelerated allreduce with
different node counts
NNODES: 512, BLOCKSIZE: 36, ERRORBOUND: 1E-4,
KERNELMAX: 4, KERNELMIN: 0
SIZEMAX = 677552940, DATADIR: $YOUR_DIR, NUMTHREADS:
 18

Kernel 0 (2,512,x2)
SCALE = 2
Compression-accelerated Kernel 0 For datasize:
677552940 bytes, the avg_time is xxx us, the
max_time is xxx us, the min_time is xxx us
SCALE = 4
Compression-accelerated Kernel 0 For datasize:
677552940 bytes, the avg_time is xxx us, the
max_time is xxx us, the min_time is xxx us
...

Kernel 1 (2,512,x2)
...

Kernel 2 (2,512,x2)
...

Kernel 3 (2,512,x2)
...

Kernel 4 (2,512,x2)
...
```

The kernels utilized in this experiment are consistent with those used in the previous experiment, and similar performance improvements are observed.