December 2021

Approachable Error Bounded Lossy Compression

Robert Raymond Underwood
Clemson University, rr.underwood94@gmail.com

Approachable Error Bounded Lossy Compression

A Dissertation
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
Computer Science

---

by
Robert Underwood
December 2021

Accepted by:
Dr. Amy Apon, Committee Chair
Dr. Jon C. Calhoun
Dr. Franck Cappello
Dr. Rong Ge
Dr. Jacob Sorber

Abstract

Compression is commonly used in HPC applications to move and store data. Traditional lossless compression, however, does not provide adequate compression of floating point data often found in scientific codes. Recently, researchers and scientists have turned to lossy compression techniques that approximate the original data rather than reproduce it in order to achieve desired levels of compression. Typical lossy compressors do not bound the errors introduced into the data, leading to the development of error bounded lossy compressors (EBLC). These tools provide the desired levels of compression as mathematical guarantees on the errors introduced. However, the current state of EBLC leaves much to be desired. The existing EBLC all have different interfaces requiring codes to be changed to adopt new techniques; EBLC have many more configuration options than their predecessors, making them more difficult to use; and EBLC typically bound quantities like point wise errors rather than higher level metrics such as spectra, p-values, or test statistics that scientists typically use. My dissertation aims to provide a uniform interface to compression and to develop tools to allow application scientists to understand and apply EBLC. This dissertation proposal presents three groups of work: LibPressio, a standard interface for compression and analysis; FRaZ/LibPressio-Opt frameworks for the automated configuration of compressors using LibPressio; and work on tools for analyzing errors in particular domains.

Dedication

I dedicate this work to Aspen my beloved wife who helps me be strong, to my friends and family who supported me thus far, to my advisers who showed me the way, and to the shoulders of the giants on which I stand.

But above all, *Soli Deo Gloria*.

Acknowledgments

I would like to thank my advisers:

Dr. Amy Apon who encouraged me in my studies and taught me to be a scientist. Dr. Jon Calhoun who introduced me to lossy compression and offers his frequent insights and support for my work. Dr. Franck Cappello whose cast a vision for what my work can be.

I would like to give special thanks to Sheng Di for his help on so many projects. I would like to give thanks to my other collaborators including Alison Baker, Amy Burton, Dorrit Hammerling, Justin Sybrant, Justin Wozniak, Jian Tian, and Xin Liang.

I also appreciate the loving support of my wife Aspen is strong in my weaknesses. I also thank my family who have supported me while to the point that I could even write a dissertation.

Funding Acknowledgments

This research was supported by the Exascale Computing Project (ECP),Project Number: 17-SC-20-SC, a collaborative effort of two DOE organizations - the Office of Science and the National Nuclear Security Administration, responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering and early testbed platforms, to support the nation's exascale computing imperative.

The material was supported by the U.S. Department of Energy, Office of Science, under contract DE-AC02-06CH11357, and supported by the National Science Foundation under Grant No. 1619253 and 1910197.

We acknowledge the computing resources provided on Bebop, which is operated by the Laboratory Computing Resource Center at Argonne National Laboratory.

This material is also based upon work supported by the U.S. Department of Energy, Office

of Science, Office of Workforce Development for Teachers and Scientists, Office of Science Graduate Student Research (SCGSR) program.The SCGSR program is administered by the Oak Ridge Institute for Scienceand Education (ORISE) for the DOE. ORISE is managed by ORAU undercontract number DE-SC0014664. All opinions expressed in this paper are theauthors and do not necessarily reflect the policies and views of DOE, ORAU, or ORISE.

Publication Acknowledgments

I acknowledge that part of this thesis was previously published in the following venues:

- Portions of Chapter 3 are in submission [1] to IEEE Transactions on Parallel and Distributed Systems
- Portions of Chapter 4 were previously published [2] in the International Parallel and Distributed Processing Symposium 2020 under the title: FRaZ: A Generic High-Fidelity Fixed-Ratio Lossy Compression Framework for Scientific Floating-point Data
- Portions of Chapter 5 are in submission [3] to IEEE Transactions on Parallel and Distributed Systems
- Portions of Chapter 6 are in preparation [4] the International Parallel and Distributed Processing Symposium.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Compression transforms data to a more compact representation than the original. When this transformation does not lose information, we call it lossless (i.e. zip files). While these kinds of transformations can be used almost anywhere, their ability to reduce data size is limited, especially for floating point data. When the transformation instead makes a "close" approximation of the original, we call this lossy compression (i.e. JPEG images). However, most lossy compression compressors do not make guarantees about where or how much information is lost. Error-Bounded Lossy Compressors (EBLC) such as SZ [5], ZFP[6], and MGARD[7] make these kinds of guarantees and have better compression ratios than lossless methods making them ideal for scientific applications.

However, the current state of the art in EBLC features many challenges which make it is not very approachable. The compressors have completely different interfaces and calling conventions (such as C/Fortran data order) requiring components of applications to be re-written to adopt new techniques. Additionally, Some compressors, like SZ, have over 27 factors that impact compression or like MGARD require a substantial background in mathematics to understand. Finally, the bounds that these compressors enforce differ greatly from what application scientists use to assess quality so often need help understanding how it will affect their analysis. Instead, application scientists want a consistent interface that bounds the metrics that they are interested in with tools that help them figure out how to do correctly and effectively employ compression.

My dissertation endeavors to address these and other challenges to help application scientists take advantage of these tools to improve the performance of the applications and understand the trade-offs of lossy compression. It is divided into 3 parts: a standard, parallel and efficient

interface for compression; automated configuration tools for compression; and tools and techniques for understanding compression. The organization of these topics can be see in Figure 1.1 The remainder of the dissertation is organized as follows: Chapter 2 provides an overview of compression, error bounded lossy compression specifically, and the applications of lossy compression in scientific applications and instruments. Chapter 3 addresses the first task: providing a standard interface for compression. It describes LibPressio a fundamental library which I developed to abstract away the differences between various compressors and used in all the remaining sections of my dissertation. I describe the implementation of an automated parallelization scheme that improves the performance of compression on CPUs. Chapter 4 begins the first of two chapters on the automated configuration of compressors. It looks at the preliminary problem of tuning compression for a fixed compression ratio. Chapter 5 continues the theme of the automated tuning of compression to look at single buffer user metrics. Lastly, Chapter 6 describes work which looks at tooling for analyzing the effects of compression on machine learning and artificial intelligence applications.

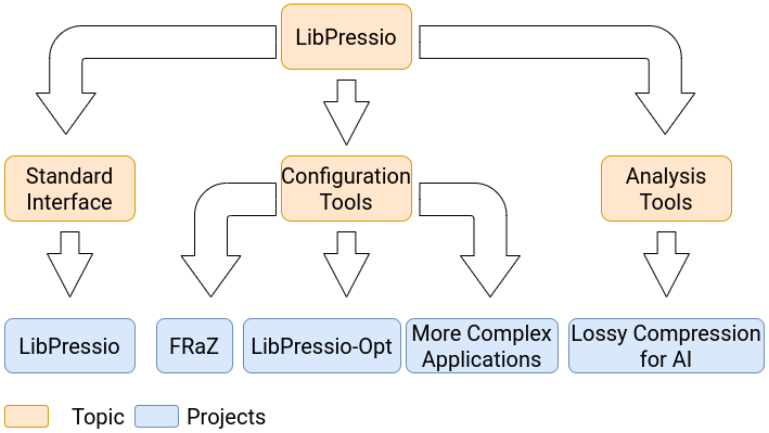


Figure 1.1: Dissertation Topics

Chapter 2

Background

2.1 Data Reduction Techniques

Data Reduction fundamentally is a class of techniques used to reduce the volume of data while it is begin stored, transported, or computed upon. These techniques are useful because they allow us to move, store, or compute upon less information to improve the performance of these processes. Han et al in [8], propose a tripartite distinction of data reduction techniques: dimensionality reduction, numerosity reduction, and data compression. Dimensionality reduction exploits correlations or patterns between different facets or features of the data to only store the most important facets or features. Techniques in this category include principle component analysis or wavelet transform methods. Numerosity reduction reduces the number of instances or observations of data either parametrically or non-parametrically. Parametric numerosity reduction replaces the data points with a model of them such that only the model parameters need be stored. Non-Parametric numerosity reduction either uses a non-parametric statistical model such as a histogram or sampling based approach. Data Compression focuses on techniques that attempt to represent the data in different encodings which are more compact. Data Compression techniques are the focus of this work[1].

Data compression methods consists of two processes: compression and decompression. The

[1]This distinction is not perfect. It is possible to view both dimensionality and numerosity reductions as forms of lossy data compression. However, the distinction is still helpful because in data compression, the emphasis is on the compression and decompression operations, whereas dimensionality and numerosity reductions focus simply on the reduction step rather than the re-constructive decompression step.

compression process takes the original or uncompressed data and produces a compressed representation of the data. The decompression process takes the compressed representation and produces a decompressed dataset. The compressed data is often smaller than the original or uncompressed dataset. Compression researchers often talk about the effectiveness of the size reduction using the compression ratio. Although few papers use the inverse, the compression ratio is reported as the ratio of the size of the uncompressed data divided by the size of the compressed data.

Data Compression is then traditionally divided into lossless and lossy methods. Lossless methods create a compressed representation which can be used to perfectly reconstruct the original data by identifying and storing common patterns amongst the data. This has the advantage of being usable in almost all applications since user's don't have to worry about data integrity. However, data with high entropy such as images, audio, or video which do not achieve high compression ratios led researchers to consider lossy compression. Lossy compression, like lossless, creates a smaller compressed representation, but unlike lossless, when decompressed, it produces an approximation of the original data rather than a prefect reconstruction. The benefit of this approach is that the compressed representation can be much smaller because it does not need to make a prefect reconstruction. Due to the limitations of lossless compression, this work will focus on lossy compression.

For an illustration of the power of lossy compression, consider the example in Figure 2.1. The image on the left was losslessly compressed and is about 4MB, the image in the middle is compressed with some loss, while the image on the right illustrates a greater amount of loss. The middle image strikes a nice balance being 10 times smaller, but with less lost than what a typical observer could discern the differences [9]. While the quality of the image on the right is much lower, it is 17 times smaller showing the kinds of trade-offs that exist in these techniques.

(a) Lake Hartwell Losslessly Compressed (b) Lake Hartwell with some loss Compressed (c) Lake Hartwell using high loss

Figure 2.1: Comparison of a Lossless and lossy Compressed Images. The figure in the middle is 10x smaller, The figure on the right is 17x smaller. Error bounded lossy compression offers a midpoint between these options

More recently, lossless compression continues to under-perform on scientific codes which make heavy use of floating point numbers. This is because of the high degree of entropy that

appears in the mantissa bits of IEEE floating point numbers. This has lead some researchers to consider lossy compression for scientific codes and to the introduction of error bounded loss compressors. However, for scientists concerned about data integrity, the possibly of a compression result like 2.1c is unacceptable. This inspired the development of error bounded lossy compressors which provide strong mathematical guarantees about the amount of error that can be introduced during the compression and decompression process. This work focuses most specifically on these techniques.

2.2 Big Data Applications and Use Cases

Over the last few decades, the capabilities of storage systems have been increasing. Table 2.1 shows the development and expected development of storage systems at the time of writing. However, the in values reported in Table 2.1 are the advertised peak values – what one might expect with the ideal bulk sequential parallel write using the entire machine. In practice [10] studies such as this one show that few applications get anywhere near peak IO performance. They found that over 75% of applications get less than 1% of the peak storage bandwidth, and 99% jobs on machines like Mira get approximately the peak performance of a USB stick. They also found that many HPC applications spend close to 60% of their total runtime performing IO – something one might expect to get worse as the gap between peak compute and storage performance widens.

System	Year	Disk Storage	Peak Bandwith	Memory
Palmetto	Various	.725PB	n/a	.168 PB
CRAY Jaguar	2008	10PB	0.24 TB/s	.36 PB
CRAY Blue Waters	2012	20PB	1.1 TB/s	≈1 PB
Mira	2012	24PB	≈0.7 TB/s	≈1 PB
Bebop	2017	< 2PB	n/a	0.3 PB
Theta	2016/2020†	11PB	n/a	≈1 PB
CRAY CORI	2017	30PB	1.7 TB/s	1.4 PB
Summit	2018	250PB	2.5 TB/s	10 PB
Aurora	2021 (expected)	230PB	25 TB/s	10 PB
Projected Exascale	2022 (expected)	500PB		

† Theta was expanded in 2020 with a CARES Act grant
‡ Aurora only achieves 25 TB/s using DAOS, and may not be portable to applications

Table 2.1: Reported System Storage on Major HPC Clusters

However, since 2015 when [10] studied the issue, the volumes of data that users want to write have continued to grow. Cosmology simulations of the Hardware/Hybrid Accelerated Cosmology Code (HACC) [11], for instance, may produce ∼20 petabytes of data when simulating 1 trillion

particles for 500 time steps, while one of the most powerful supercomputers—the Oak Ridge National Laboratory Summit [12]—provides only hundreds of terabytes of storage for each user; and hundreds of users share the limited total storage space (250 PB in total). In materials science, raw data can be produced with a very high acquisition rate. for example, 250 GB/s on LCLS-II [13]). In order to sustain the data production rate, storing the raw data without compression is impractical. The LCLS-II data system designers calculated that the expected data compression ratio required is 10 or higher to reduce the required storage bandwidth to 25 GB/s.

Between the tiny faction of peak IO that applications achieve, the growing demand for IO from applications, and the fundamental capabilities of the machines, there is a clear need for techniques to reduce the amount of data that needs to be written to or even read from disk. Compression provides this capability. Lossless compression often is ill-suited for many HPC codes for the reasons presented in the end of Section 2.1. This has lead researchers to explore areas where lossy compression can prove valuable [13]:

1. Visualization – Visualization is a common principally lossy piece of post-analysis performed for many applications. The human visual system is limited in its ability to perceive differences between images. This means that lossy compression can be used to reduce the volume of data destined for visualization when the differences are "diagnostically lossless" [9]. These kinds of techniques have been used for decades since JPEG compression is a lossy compression algorithm [14], but traditional lossy methods like JPEG have been shown to be insufficient for climate research because the results are not consistently diagnostically lossless. This has lead climate researchers to study the use of error bounded lossy compressors for this work [9], [15].

2. Reducing Streaming Data Intensity – Instruments like the LCLS-II and the APS (an X-ray based laser used in crystallography and other applications) transport huge volumes (>250GB/s) of information from detectors to local analysis clusters for visualization and post-processing. However, this process is bottle-necked by the transfer from the detector to the analysis cluster which has less bandwidth available. Researchers have determined that a 10X reduction in data volume would aleviate this bottleneck. Work is ongoing to meet this requirement.

3. Reducing Storage Footprint – Some applications such as HACC and CESM-LE can produce enormous volumes of data ($\approx$ 20PB and 200 TB respectively) that need to be stored for later analysis for *each* full size run of the applications. Because the largest machines are shared

and users do not have access to the full storage space of the machine. Scientists have been exploring the use of lossy compression to store especially large results from their experiments.

4. Reducing IO time – Work by [16] has shown that by first lossy compressing data in memory, it is possible to reduce the overall run time of the application by reducing the time spent performing IO for applications like NYX – a cosmology code.

5. Accelerating Checkpoint Restart – Many HPC applications use checkpoint restart to provide resilience to node and other kinds of job failures. However, as machines grow, so does the overhead of using checkpoint restart. There has been work investigating the use of lossy compressed checkpoint restart [17] to accelerate the IO bandwidth used in Partial Derivative Equation based applications. This particular class of applications is often naturally self-correcting to the kinds of errors introduced by lossy compression, making it an ideal use case. Initial evaluation showed that as the number of processes grow, using lossy compression can reduce checkpoint times by ≈ 60 % allowing more time to be used for the application.

6. Reducing Memory Footprint – A quantum circuit simulation application [18] uses lossy compression to store the quantum state in memory to enable the application to simulate larger circuits than would be possible without compression due to limited available system memory. By reducing in the in-memory footprint, to only the needed state at any particular moment, the application can simulate larger quantum circuits without a discernible impact on the quality of the results.

7. Accelerating Execution – The project PaSTRI [19] uses error bounded loss compression to a a "lossy cache" for the GAMESS application used in quantum chemistry. Applications like GAMESS do many repeated calculations that would be ideal for caching, but if stored in full, would not fit into memory. The lossy compressor PaSTRI provides an efficient method to store the data generated by GAMESS, which resulted in a 24% improvement in run time with no discernible effect on the quality of the result.

Readers are encouraged to read [13] for more details on these use cases.

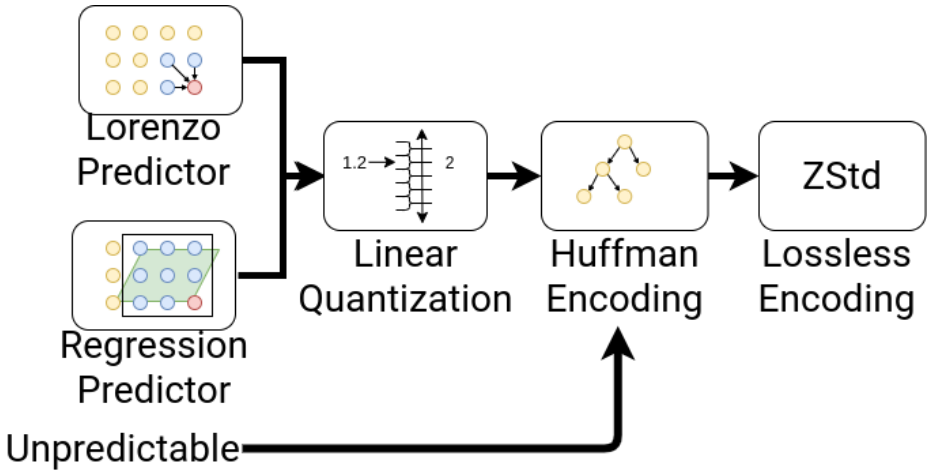


Figure 2.2: SZ Compression Principles

2.3 Error-Bounded Lossy Compression

Over the last decade, several error bounded lossy compressors have been developed. In this dissertation, I will focus on SZ, ZFP, and MGARD some of the current leading contenders. In this section, I will discuss the principles used to perform the compression and decompression operations.

There are three major lossy compressors – SZ, ZFP and MGARD.

2.3.1 SZ

SZ which was developed over the papers by [5], [20]–[22] has been widely evaluated in the scientific floating-point data compression community [9], [18], [23], [24] , showing that it is one of the best compressors in its class.

SZ splits each dataset into many consecutive non-overlapped blocks (such as 6×6×6 for a 3D dataset) and performs compression in each block. Compression then involves three key steps. (1) data prediction – each data point is predicted based on its nearby values in space; two major predictors are applied, Lorenzo [5] and linear regression [20]. Blocks that cannot be predicted are stored losslessly. (2) linear quantization – each data point is converted to an integer by applying a equal-bin-size quantization on the difference between its predicted value and real value; and (3) compression of the generated integer code arrays by a series of custom lossless compression methods including entropy encoding such as Huffman encoding and dictionary encoding, such as Zstd [25]. The compression principles of SZ are summarized in Figure 2.2.

SZ offers multiple error-controlling approaches, including absolute error bound (denoted

ϵ_{abs}), relative error bound (denoted ϵ_{rel}), and target PSNR (denoted by ϵ_{psnr}), which are formulated in the following three constraints, respectively:

$$|x_i - \tilde{x}_i| \leq \epsilon_{abs}, \forall x_i \in d_{f,t} \tag{2.1}$$

$$value\_range(d_{f,t})|x_i - \tilde{x}_i| \leq \epsilon_{rel} \cdot x_i, \forall x_i \in d_{f,t} \tag{2.2}$$

$$10 \log_{10} \frac{value\_range(d_{f,t})^2}{\sqrt{mse(d_{f,t}, \tilde{d}_{f,t})}} \leq \epsilon_{psnr}, \tag{2.3}$$

where $d_{f,t}$ refers to the raw data, $\tilde{d}_{f,t}$ refers to the decompressed data, $mse()$ is the mean squared error, and $value\_range()$ refers to the range of values in the dataset $d_{f,t}$ (i.e., $\max(d_{f,t}) - \min(d_{f,t})$).

2.3.2 ZFP

[6] is another outstanding error-bounded lossy compressor, which is broadly evaluated in many scientific research studies [2], [24]. Similar to SZ, ZFP supports different types of error controls, such as absolute error bound and precision. The precision mode allows users to set an integer number between 1 and the number of bits in the type to control the data distortion with an approximately relative error effect. The higher the precision number is, the lower the data distortion. A precision value of 32 causes 32-bit IEEE floating-point data to be losslessly compressed.

Unlike SZ, ZFP adopts a blockwise transform-based compression model, which includes four critical steps: (1) exponent alignment and fixed-point representation, which align the values in each block to a common exponent and performs fixed-point representation conversion; (2) transformation, which applies a near orthogonal transform to each block; and (3) embedded-coding, which orders the transform coefficients and encodes the coefficients one "bit plane" at a time. SZ and ZFP have different design principles, and neither always has the best compression quality on all types of data sets [2], [24]. The stages of this pipeline are summarized in Figure 2.3.

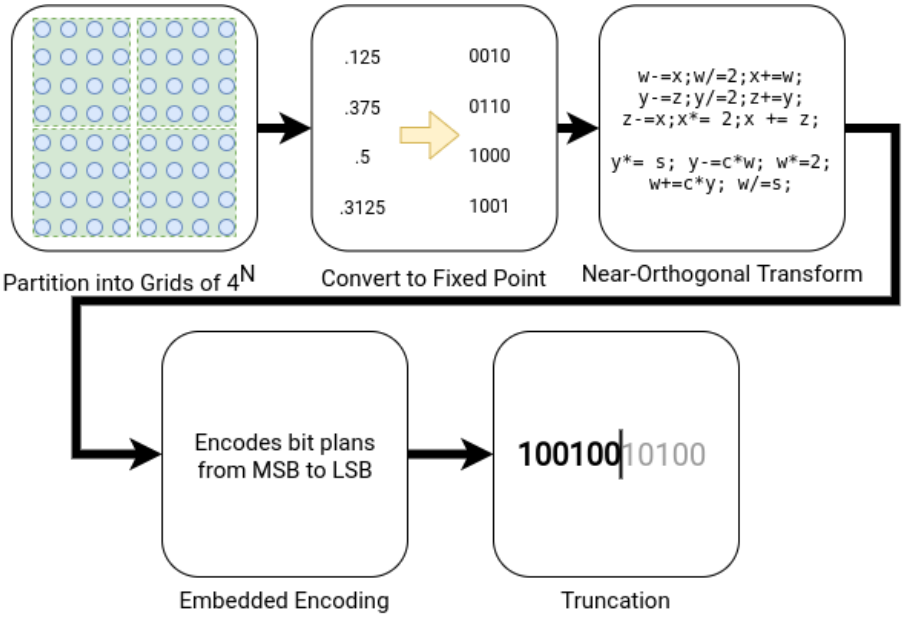


Figure 2.3: ZFP Compression Principles

2.3.3 MGARD

MultiGrid Adaptive Reduction of Data (MGARD) [7], [26], [27] is an error-controlled lossy compressor supporting multilevel lossy reduction of scientific floating-point data. MGARD was designed based on the theory of multigrid methods [28]. An important feature of MGARD is providing guaranteed, compute-able bounds on the loss incurred by the data reduction. MGARD provides different types of norms, such as infinity norm and L2 norm, to control the data distortion. The infinity norm is equivalent to the absolute error bound, and the L2 norm mode can be used to control the mean squared error (MSE) during the lossy compression. levels of partial decompression such that users reduce the dataset by either minimizing storage with a required data fidelity or minimizing the data distortion with a target compression ratio. The compression principles of MGARD are summarized in Figure 2.4.

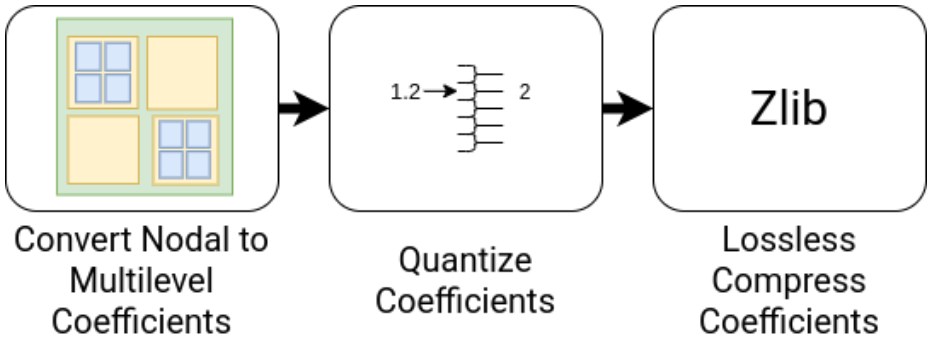


Figure 2.4: MGARD Compression Principles

Chapter 3

Productive, Performant, and Parallel Generic Lossy Data Compression with LibPressio

Portions this chapter are in submission to IEEE Transactions on Parallel and Distributed Systems the under the title: Productive, Performant, and Parallel Generic Lossy Data Compression with LibPressio

Abstract

In recent years, error bounded lossy compressors have been developed to cope with the ever increasing volume of scientific floating point data. However not all compression techniques are appropriate for all datasets, and determining which one to use can be time consuming requiring code modifications and trial and error. We present LibPressio a generic abstraction for the compression of dense tensors to minimize the code changes scientists need to make to take advantage of new and improved compression techniques. In this work we assess how using an abstraction like LibPressio can have on usability, overhead, and scalability of compression showing at least 50% reduction in the volume of client code for 8 different applications while preserving statistically insignificant overhead for compression and automatic parallelization capabilities which have comparable scalability

and less than 10% compression ratio overhead for moderate numbers of tasks relative to bespoke implementations enabling performance portability.

3.1 Introduction

In this paper, we present FRaZ: a generic fixed-ratio lossy compression framework respecting user-specified error constraints. The contribution is twofold. (1) We develop an efficient iterative approach to accurately determine the appropriate error settings for different lossy compressors based on target compression ratios. (2) We perform a thorough performance and accuracy evaluation for our proposed fixed-ratio compression framework with multiple state-of-the-art error-controlled lossy compressors, using several real-world scientific floating-point datasets from different domains. Experiments show that FRaZ effectively identifies the optimum error setting in the entire error setting space of any given lossy compressor.

In recent years, error bounded lossy compression methods such as SZ, ZFP, and MGARD have begun to revolutionize the way that application scientists transport data across the network, store data to persistent storage, and process data in memory. This is because Error Bounded Lossy Compression methods achieve much higher compression ratios than what can be typically achieved with lossless compression methods, but also provide a bound on the errors introduced making lossy compression possible for scientific applications where data integrity is key. These advances have enabled advances in checkpointing, cosmology codes, climate codes, physics simulations, and more [13].

However, to gain further adoption scientists who use these methods need a consistent, flexible, and high performance interface to use and understand the effects of compression regardless of what programming language they use. All of the existing lossy compressor methods suffer from a proliferation of interfaces and semantics making it difficult to perform comparisons between methods [2], [29], [30]. Users need one interface so they can focus on their science and allow compression researchers to develop their methods independently.

Additionally, many compressors are not equipped to take advantage of multi-core architectures. Many compressors have only a single mode [31] or operation [6] of the of the compressor that has been parallelized – if any modes or operations have CPU parallel versions at all [7] – resulting in an inefficient use of hardware. Creating parallel implementations for these compressors

can require significant engineering effort as evident from the volume and complexity of the code for SZ and ZFP parallel implementations and the lack of feature parity between the parallel and serial versions. User's could invoke compressors in parallel using a framework such as OpenMP or MPI if the compressor supports it, however this process requires attention to detail to do correctly and difference compressors may benefit from different discretion schemes and requires the scientist to further make changes to their code from adopting a serial version of a compressor. Finally, parallel implementations require careful performance tuning to obtain optimal performance on a particular platform which may change from compressor to compressor. Again this code can be written, but further explodes the volume of code required if it is to be done efficiently.

In this paper, we aim at developing a low-overhead interface LibPressio which provides one common, low overhead, productive interface which applications can use to automatically configure, perform in parallel, and analyze the results of compression. This work is challenging because: (1) the sheer number and variety of interfaces that compressors currently provide requires careful attention to detail to provide an interface that all compressors can implement efficiently while being easy to use, (2) developing scalable, high-performance parallel compressors is difficult for a single compression algorithm – let alone a generic version that could be used with many compressors, (3) parallel systems often require careful tuning and adaptable algorithms to best utilize available hardware and naive implementations have high overhead This paper presents the design, implementation, and some of the applications of LibPressio. Our contributions are:

- an extensive analysis of the API designs of 8 competing compression abstractions highlighting their strengths and weaknesses for use in high performance parallel and distributed computing.
- a detailed consideration of the design considerations made in LibPressio
- We demonstrate at a 50% - 90% reduction in the volume of client code as measured across 11 different applications implemented with at least feature parity and often additional features, while showing a 50% reduction in the complexity of the implementation of the command line interface for SZ and ZFP and decreased porting effort.
- We demonstrate that LibPressio has statistically insignifigant overhead across multiple datasets and error bounds.
- We demonstrate comparable run-time scalability and compression ratio to the bespoke parallel

implementations of SZ and ZFP for the modes that they implement. We further demonstrate scalability of decompression for ZFP and compression and decompression for MGARD which previously did not exist.

Here we outline the remainder of the paper: First in Section 3.2 we provide an overview of the compression principles, APIs, and uses of error bounded lossy compressors used in HPC. Then in Section 3.3 we compare our proposed solution to the extensive work in this area. Next in Section 3.4 we discuss the design trade-offs in the development of LibPressio. After that in Section 3.5 we demonstrate the generalizablity of LibPressio to an array of different applications, and evaluate its effectiveness by considering run-time overheads and code complexity changes. Finally, we draw conclusions about the design of LibPressio and offer a path forward in the architecture of new compression libraries.

3.2 Background

3.2.1 Compression Interfaces

Designing an effective interface requires a deep understanding of the kinds of operations that are performed in compression and decompression. For lossless compression where no information is lost, this would seem somewhat straight forward. However, due to differences in semantics, this can be complex.

The simplest lossless compressors – such run length encoding which removes repeated consecutive sequences in the binary representation of the data and replaces then with a special sequence that encodes the sequence and the number of times that sequence was repeated – can be viewed as isomorphisms that map between the "origional" and "compressed" representations. Additionally, many but not all of these compressors, accept any sequence of bytes as valid inputs to compression allowing them to be widely used. This property lends itself to a relatively small number of meaningfully different interface design. The interface often consists of two functions compress and decompress which are inverses of each other that perform compression and decompression on 1d arrays in input bytes to produce a 1d array of output bytes.

However even for lossless compressors, there are some minor differences in the interfaces relating to initialization of global structures, const-ness of the buffers accepted as arguments, whether

to accept inputs from the file system or from memory also, memory management for the underlying buffers, and the passing of additional configuration parameters. Additonally, as soon as it is possible for compression to fail such as with the ubiquitous zlib library – such as due to insufficient memory or space in the output buffer, or a malformed input to decompression – compression and decompression no longer form an semi-isomorphism let alone a isomorphism because once the compressor returns the failure state, both compression and decompression operations can no longer uniquely recover the original input prior to the compression using only the failure state.

Furthermore, some specialized lossless compressors like fpzip only accept IEEE arrays of floating point inputs. This means that a compressor interface that abstracts between different lossless compressors needs to pass type information about the data being compressed.

With lossy compression this problem is compounded. The decompression process for most lossy compressors preserves the "structure" of the output while the values are often not preserved but instead are approximations of the originals. The loss of losslessness requires users to conduct tests and experiment with different lossy compressors to ensure that their data is preserved sufficently for their applications. However, the lack of consistent implementations for this often leads to a proliferation of metrics interfaces and implementations as well. Additionally many of the leading lossy compressors for dense tensors take advantage of spatial information requiring a more sophisticated description input metadata to describe dimension ordering, type information for their inputs and the leading compressors again all differ on how to best do this.

3.3 Related Work

There have been several other related works that attempt to provide generic abstractions for compression. Many of these approaches are specialized for specific domains that are not suitable general for scientific computing. Table 3.1 provides an overview of the features of the various compressor abstraction libraries.

3.3.1 Points of Comparison

In this section we compare compression abstraction libraries using the following criteria: Does it support lossless compressors, Does it support lossy compressors, is it dimension aware, is it datatype aware, is it embed-able in-process, does it allow arbitrary pointers for configuration, can

library	purpose	lossless compression	lossy compression	n-d data aware	datatype-aware	embed-able design	arbitrary configuration	option introspection	third party extensions
ADIOS-2	data processing framework	✓	✓	✓	✓	✓	✗	✗	✗
ffmpeg	video	✓	✓	❑	✓	✓	✓	✓	✓
Foresight/CBench	analysis	✓	✓	✓	✗	❑	✓	✗	✗
HDF5	data processing framework	✓	✓	✓	✓	✓	✗	✓	✓
imagemagick	images	✓	✓	❑	✓	✓	✓	✗	✓
libarchive	file-backup	✓	✗	✗	✗	✓	✗	✓	✗
NumCodes	compression	✓	✓	✓	✓	✗	✗	✓	✓
SCIL	compression + analysis	✓	✓	✓	✓	✓	✗	✓	✗
Z-checker (0.7)	analysis	✓	✓	❑	✓	❑	✗	✗	✓
LibPressio	compression + analysis	✓	✓	✓	✓	✓	✓	✓	✓

Table 3.1: Feature Comparison Table

options of compressors be introspected, and does it allow 3rd party extensions?

The first two categories are relatively self explanatory. If a compressor abstraction provides any lossless compressors, it has a check for lossless compressors. If a compressor abstraction provides any lossy compressor, it has a check for lossy compressors.

If a compressor abstraction is n-d data aware, it accepts information about the organization of data in memory for N arbitrary dimensions. This is important because understanding the layout of the data allows you to take advantage of spatial features in the data when compressing. Most lossy and some lossless compressors use this information – such as some lossless image formats such as PNG – to improve compression ratios. Many compressors do not support arbitrary higher dimensional data structures; however, supporting 1-4d data is common [5]–[7]. It is important to note that most compressor which supports at least 1d can often still operate on higher dimensional information by treating contagious higher dimensions as a single dimension with a larger stride. This is not always true, MGARD requires at least 3 rows in each dimension or it returns an error rather than compressing the data [7]. Likewise while compression may not fail, passing incorrect information about dimensions can produce inefficient compression. For example, with ZFP, passing a dimension smaller than the blocksize results in inefficient compression due to required zero padding for the algorithm. Therefore passing the dimensions accurately is important.

If a compressor abstraction is data-type aware, it accepts information about the data type and allows at least two datatypes. This is important because this information is key to correctly preserving data using lossy compression. You cannot preserve a data type to a non-zero error tolerance if you don't know how the data is stored. Lossless compressors can allow multiple data types without begin data-type aware because they treat all input types as a stream of bytes regardless of the underlying structure of the data. However, they also typically do not except information about type information [1].

If a compressor abstraction is embeddable, it can be embedded into a application written in native languages such as C or C++ without the use of the `exec` or loading an interpreter. This is important because many HPC environments and frameworks (such as MPI) limit the use of `exec` to start other processes, and running interpreters can be expensive overhead for running an application [33].

If a compressor abstraction supports arbitrary configuration, it can accept arguments of arbitrary type. This is essential to support compressors which can be configured with non-serializable native types such as `MPI_Comm` or `cudaStream_t` to control the degree and placement of parallel resources [34], [35]. This can also be important for compressors such as SZ which require structs be passed for configuration of certain modes which may not have native serialization representations. For this reason, compressor abstractions which are string-ly typed (use strings to store configuration [29], [36], and parse the string to the appropriate type at runtime) or JSON typed are not appropriate for existing lossy compressors.

If a compressor abstraction is introspectiable, it allows users to ask what options a compressor supports and what types are used to represent these options. This becomes more and more important the more compressor plugins that an abstraction provides. Users need to have some common options way of enumerating the options supported by a compressor in order to programmatically configure them. Compressors abstractions which allow this for all non-arbitrary types are introspectiable.

If a compressor abstraction supports third party extensions, it allows additional compressors or other facilities to be added to the abstractions interface without modifying the code for the abstraction. This becomes important because it allows developers to create their plugins out of tree so that they can be used experimentally before being upstreamed. If the source code has to be

[1] For two exceptions consider [32] and ZFP's reversible mode [6]

modified, this becomes more difficult.

3.3.2 Existing Libraries

One example of this is libarchive – the library that underlies many modern implementations of the Linux/UNIX `tar` command. It supports a number of what it calls filters which are lossless compressors such as gzip, lz4, lzma, sx, zstd, and others. It also supports a number of formats such as cip, zip, tar, rar and other container formats which encapsulate separate files stored within the buffer. One of its key portability features is that it uses callback functions so that archives can be read or written from sockets, files, memory, or other custom resources. Lastly libarchive expects a "record" organized layout – that an archive file consists of one or more "named" records. However in scientific codes, this is not always true, but share some similarities with HDF5 which is commonly used in HPC. The weaknesses of libarchive is that it only supports lossless formats, has no concept of the underlying type or structure of the data being stored, and does not allow third party filters.

There are two examples of libraries dedicated to compressing/decompressing lossy artifacts: `imagemagick` and `ffmpeg`. These compress/decompress images and video respectively. However not all scientific data can be neatly characterized either as 2d image or even a video. So while these interfaces could be used, they do not necessarily represent a universal interface for lossy compression, but rather specialized interfaces for specific domains. ImageMagick further supports a variety of features that generally feel out of place in scientific computing such as image transforms or color mapping. The same is true for the various libraries such as the one in VLC called `libcompression`. In addition to these libraries there are also specialized libraries for specific algorithms such as `libjpeg-turbo`. These libraries use still different interfaces which optimize for there usecase by providing a "by-scanline" interface which allows reading images by row for ease of rendering. This falls short of the various random and parallel strided access patterns used by HPC codes.

In Python there are tools that are closer to LibPressio, for example, NumCodecs is a python library that provides a set of "codecs" which implement various lossless and loss compressors. First of all NumCodecs is a python library, and a vast majority of HPC codes are not written in python. The translation between python and C++, even with the python buffer protocol introduced in Python 3.2 and improved in 3.3 is still moderately expensive. The further restrictions of the global interpreter lock for multi-threading in C Python make it unlikely that applications will embed Python to perform compression in a multi-threaded C or Fortran application. However, the interface requirements of

NumCodecs are overly strict. For example, at time of writing, SZ one of the leading error bounded lossy compressors cannot implement its interface correctly due to its use of global memory to store configuration parameters. Additionally, NumCodecs does not support all of the kinds of options modern lossy compressors in HPC require. For example, it doesn't support passing structures or opaque pointers that can't be serialized as JSON such as `MPI_Comm` or `sycl::queue` structures used by some compressors to control parallelism. Finally, NumCodecs doesn't have a uniform way to query certain kinds of important information about compressors such as their thread safety level.

Analysis tools such as Z-checker and Foresight also provide there own lossy compression abstractions. Older version of Z-Checker as well as current versions of CBench – the compression library behind Foresight – provide their own compressor abstraction layers. Both of these interfaces are limited in that they aim to adapt only a subset of compressor options. Z-checker is notable in that it provides adapters to convert the native bounds kept by error bounded lossy compressors those supported by SZ using mathmatical relationships between the bounds. Both of these tools use "stringly typed" configuration parameters. This shares the weakness of NumCodecs, but then aggravates the problem by using string based encodings of floating point numbers which further loose precision. Neither of these libraries provide runtime information such as thread safety that can be used to safely parallelize workloads. Futhermore, only parts of Zchecker[37] which are designed to collect and store metrics are designed to be embedded into other applications. While CBench the actual compressor abstraction can be embedded, Foresight was designed as a standalone application. Because of these weaknesses, current versions of Z-Checker are in the process of adopting LibPressio for interfacing with compressors.

3.4 Design Overview

The architecture of LibPressio was developed after evaluating all of the strengths and weaknesses of the existing compressor libraries. The goal was to develop a library which would fit the specific needs of users in the HPC community who have some of the most complex needs of any compression user.

LibPressio has six major components. `pressio` structures are used to create references to, enumerate, and handle errors while creating compressors, metrics and io modules. `pressio_data` structures are an abstraction for handling memory management, and different shapes and types of

data buffers. `pressio_compressor` structures are used to compress and decompress data. `pressio_options` structures hold introspectiable configuration for compressors, metrics, and io objects. `pressio_io` structures provide convenience functions for reading data from various sources into our out of `pressio_data` buffers. `pressio_metrics` structures provide functions for measuring the performance of compression and the quality of compression. These relationships are summarized in figure 3.1.

For a high level overview of the usage of the library, A complete example of the LibPressio API can be found in Appendix A. In the remainder of this section, we discuss some of the key architectural decisions in LibPressio.

3.4.1 C api with a C++ implementation

Compressors that can be embedded within an application can be used in far more contexts than those that must be used from a separate process. This is because some frameworks such as MPI limit access to the `fork` system call (or `CreateProcess` on Windows). Because you can't rely on libraries that make out of process calls, an embedded design is prefered.

Then comes the question of what language to choose for the task. One could either choose a native or a byte-code/interpreted interface. Likewise, there can be dramatic performance and dependency management implications for using an interpreter such as Python or Java's JVM. Many of these runtimes don't support running multiple independent, concurrent runtimes, a necessity for parallel codes and have some an initialization routine that can be called at most once. WebAssembly offers a promising option, but currently doesn't support all of the various types used in HPC codes and doesn't currently have wide adoption. Thus a compressor abstraction should prefer a native interface.

In order to save engineering time and effort a single language for the interface and plugins should be chosen. For ease of porting that language should have numerous foreign function interfaces. The language that has the most foreign functions interface implementations is C because all programs need to make system calls, and system call interfaces are almost always written in C. However, C does not provide commonly used higher level abstractions that would aid in implementation of plugins – or support compressors written natively in C++ such as MGARD. Thus, C++ was chosen for the implementation language because of its C compatible ABI and higher level abstractions such as templates which greatly simplify code.

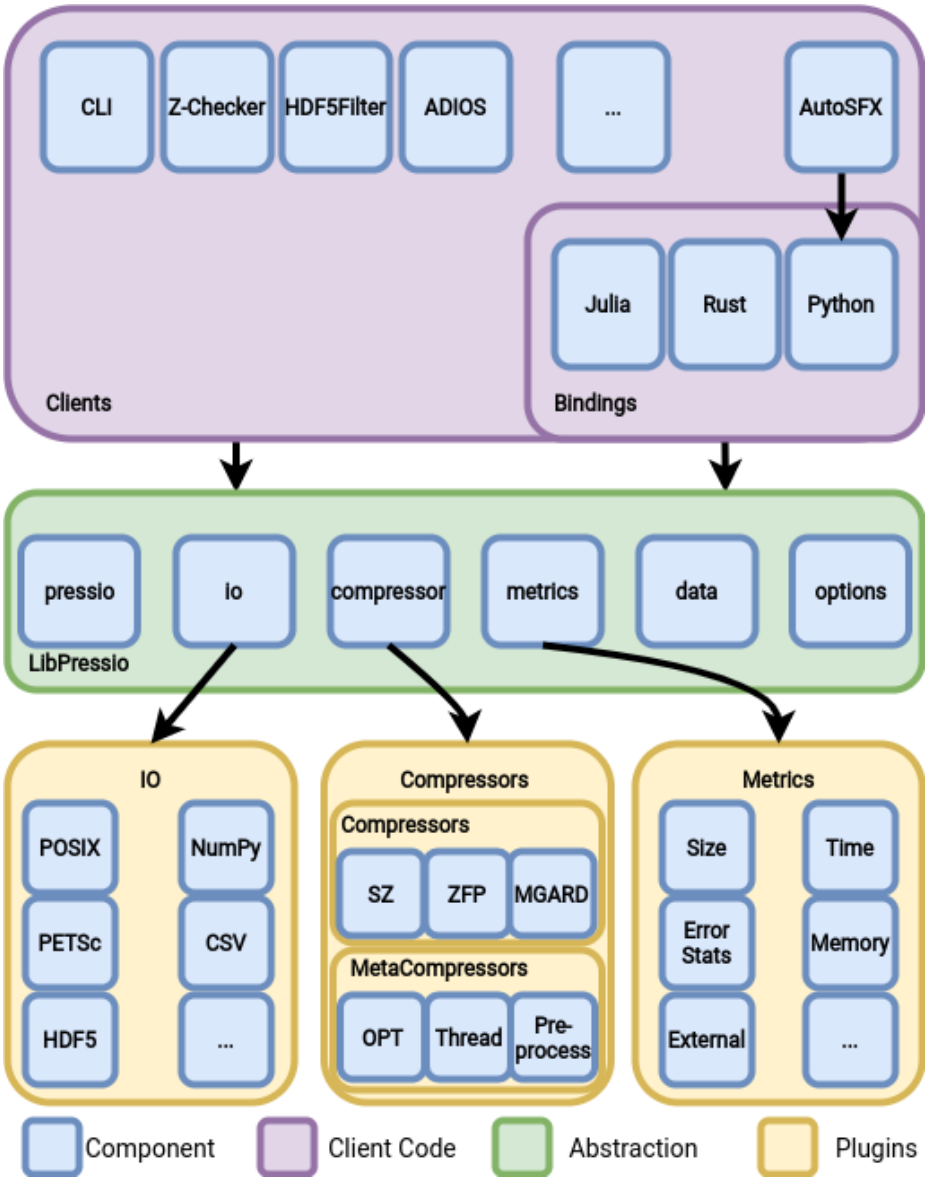


Figure 3.1: Major Components of LibPressio and how they interact with plugins and client code

3.4.2 Data Abstraction

A compressor abstraction library ought to have an abstraction to describe the type and layout of data. This is because every compressor library that we studied has a different understanding of what it should be passed. This also allows the underlying implementation to use the dimension and type information if it can be used, and ignore it if it cannot be.

Memory management is another key aspect of the data abstraction. The compression library might want provided memory to compress/decompress into, it may allocate the compressed or decompressed buffer on compression and decompression respectively, it may need to run on a GPU or on persistent memory [34]. If the library didn't take responsibility for memory management, users would not know how to pass memory to or from each application and it would leak through the abstraction.

The most flexible design is essentially a pointer, with an array to store the dimension information, and an enum to store the datatype, a function pointer to a deleter method, and an optional void pointer to state for the deleter method. As an alternative to an enum the address of a fully specialized templates function could be used; this design is used by many implementations of `std::any` to allow it to be used without RTTI. The advantage of this design is that it trivially supports new types. The disadvantage of this design is that this address could differ from compiler to compiler, and could foil network serialization in a heterogeneous environment. The design allows users to use persistent or GPU unified memory memory function with API's like `mmap` or `sycl::malloc_device` easily. The deleter can be a static function to `memunmap` or `sycl::free` respectively. Likewise, this design allow for shallow copies where the deleter function is a noop.

3.4.3 Compress and Decompress

Compressor interfaces themselves differ in a surprising number of ways. A good abstraction needs to handle all of these cases.

For example, ZFP and image and video libraries tend to use a Fortran based dimension ordering whereas SZ and MGARD expect a C based dimension ordering. Passing the wrong dimension ordering can result in poor compression quality because incorrect strides are used to represent the data. For sake of simplicity, the user should only have to learn and write one ordering and have the abstraction provide the compressor with the correct underlying ordering where required.

Compressors have different construction methods. SZ has a single shared configuration store which is created by `SZ_Init` and deallocted by `SZ_Finalize`. ZFP can have multiple independent configuration stores. This has impacts on thread safety since a thread can only call SZ_Finalize if they are confident no other thread (possibly in a different library) is still using SZ. The safest approach is reference count instances of compressors and to provide an interface to indicate if the instance returned is a shared instance or not – allowing use of multi-threading if it is not shared.

Compressors may or may not encode the compressor configuration into the compressed byte stream. The abstraction should require that users provide the configuration outside of the use of the compressor, but provide interfaces to use the stream encoded metadata if it exists. This is the same solution choosen by NumCodecs.

Compressors may or may not clobber the buffers passed to them by the user. Some versions of MGARD treat the input data as mutable to save on memory during compression. However clobbering the input data is unusual amongst compressors, and could surprise the user. Enforcing constness of the input data is preferable from a consistency perspective, so compressors that clobber the input data should generally make a copy and compress on the copy.

Error bounded compressors don't all have the same notions of error bounds or options. SZ for example has 27+ different configuration parameters, where as some lossless compressors have either zero or one parameter. The compressor abstraction should allow compressors to have arbitrarily many options, while at the same time providing a list of "common" options understood by one or more compressors. When the compressor abstraction also provides introspection, users can select the compressors that meet their specific needs programmatically.

3.4.4 Options Abstractions

Now that we have discuss the ways in which compressors can differ, we consider the abstractions for representing these configuration options.

When compressors can use different types for the same option, it becomes important to provide conversions between these options. However, implicit casting between options has the potential to be surprising, and depending on the conversion to lead to a loss of precision thus only should be enabled when desired. LibPressio chooses to allow safe conversions with the `as_set` routine, but also has a `cast_set` routine which allows a safely level parameter for increasing more surprising conversions from implicit, to explicit, to special conversions (such as `atoi`, or `to_string`).

Introspection of types is key to an interpretable interface between compressors. Users need to know what type the compressor expects in order to supply arguments of the correct type. In Libpressio, each option can report its type as one of 9 options: signed and unsigned integers of size 8, 16, 32, and 64, IEEE 32bit single precision floating point, IEEE 64bit double precision floating point, string, array of string, data, user data, and unset. The first 5 are self explanatory and store a scalar of the specified type. The array of string option allows a dynamically sized list of string to be passed to an option. It can be used for compressors which support multple error bounds at a time. The data option allows a full data buffer to be passed to an option. It can be used for compressors which need a mask such as SZ's ExaFEL mode [38]. The user data mode allow the user to pass a void pointer to a compressor. It is most commonly used to pass opaque types that represent parallel resources such as MPI_Comm. The unset type is used to indicate an invalid error state and does not actually contain data.

Another key design decision is to allow hierarchical options setting. With the introduction of a clean API for interacting with compressors, developers can create meta-compressors which provide helpful functionality for other compressors. A few good examples of this from LibPressio are the `many_independent` and `many_dependent` modules which parallelize a series of independent or dependent compression tasks in parallel using MPI. Other good examples are a compression auto-configuration plugin which tunes the compressor the users desires to meet some objective. This could be through of as a hierarchy of compressors. With a hierarchy of compressors, users need a way to configure the various levels of the hierarchy both efficiently, but also independently. LibPressio sets an option for all compressors that support the same option lower on the hierarchy unless it has a different setting explicitly set for it.

3.4.5 Metrics Abstractions

A good compressor abstraction should also provide at least a basic debugging and metrics interface. Being able to inspect the uncompressed, compressed, and decompressed buffers in transit can save users a great deal of effort in analyzing their code. This allows all compressors to share the same metrics gathering code for things like compression ratio where are common to all compressors. This doesn't mean that compressors can't provide their own specialized metrics, SZ for example reports the number of blocks that were predicted with the lorenzo predictor vs the regression predictor. What is key is to provide these interfaces as a series of callbacks. The most important callbacks

are pre/post compression and decompression, but as identified above, a callback to get compressor specific metrics can also be important[31].

3.5 Example Applications and Effectiveness

In this section, we highlight a number of sample applications and plugins that we developed that we developed to demonstrate the generalness and flexiblity of the LibPressio api.

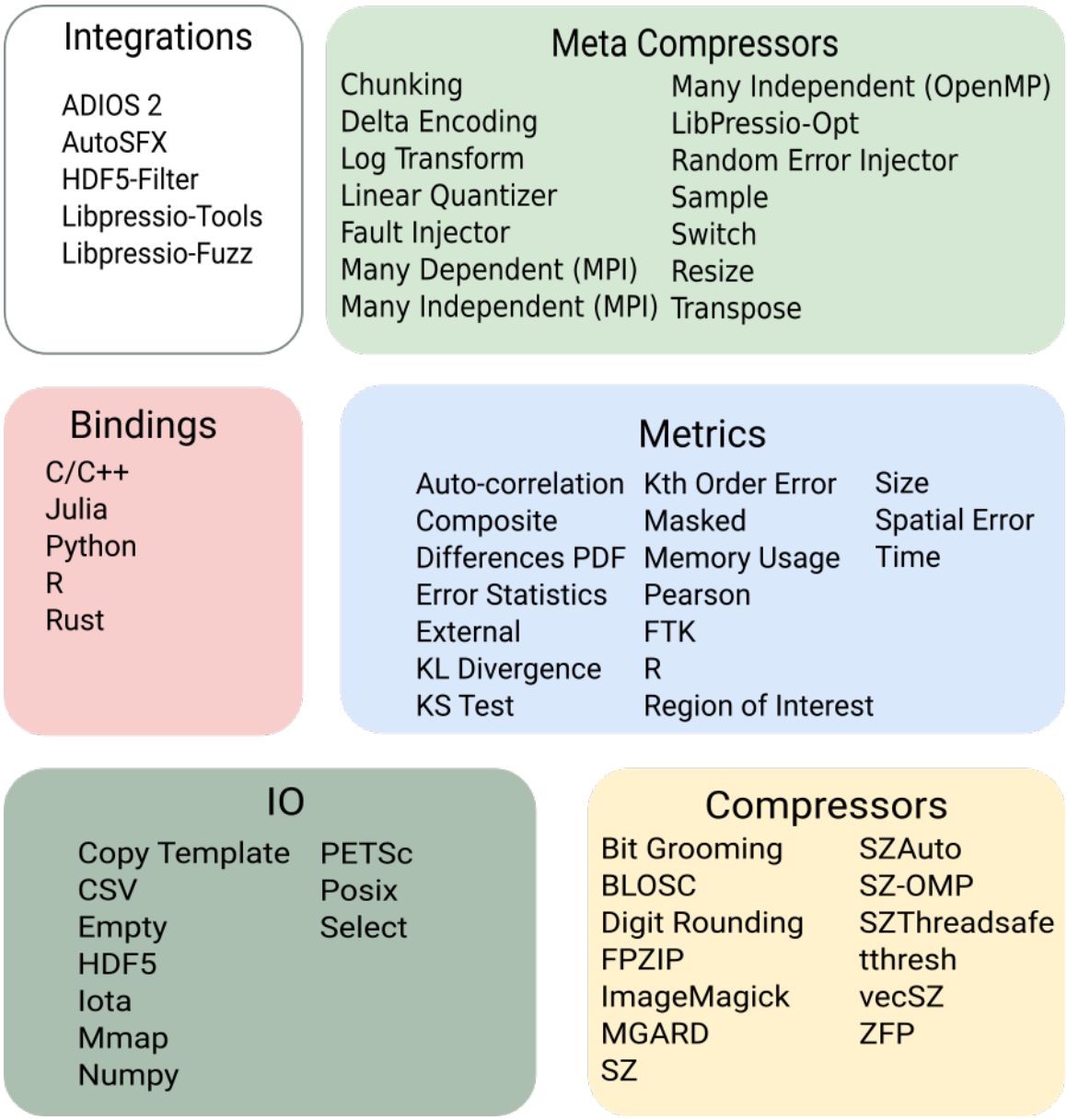
3.5.1 Plugins

At time of writing, LibPressio currently has over 54 public first-party plugins that were developed in connection with researchers and scientists at 6 different institutions. These plugins include all of the leading error bounded lossy compressors as well as common lossless and image compressors. LibPressio also supports some of the most commonly used IO formats used in scientific computing such as flat binary files, HDF5 files, and character-delimited files such as CSV. These plugins enabled researchers to quickly adopt LibPressio into their existing work flows. Figure 3.2 gives an overview of the available plugins. Descriptions of these can be found in the glossary.

LibPressio also supports a number of meta compressors. As alluded to in section 3.4.4, meta-compressors are not compressors themselves, but compressor interface compatible tools that boost productivity of those using and developing compression techniques. Some of these plugins allow performing common pre/post processing steps such as transposition, resizing, and linear quantization. Others provide more robust capabilities such as fault or statistical error injection, auto tuning, and automatic task-based parallelization. By allowing and encouraging these kinds of services to conform to the compressor interface and using the same data abstraction for uncompressed, compressed, decompressed data compression and decompression operations become endomorphisms allowing arbitrary composition of compressors and these services. This allows users or even compressor developers to experiment with different compressor designs out of their consistent functional parts such as quantization, transform, prediction, and encoding stages. This even allow new approaches to compressor architecture by allowing compressors to provide only small components of the compressor pipeline that can be chained with the most efficient stages developed by other developers.

Beyond meta compressors, unlike prior approaches for developing tools for compression which were tied to specific compressors, meta compressors and external tools built upon LibPressio

Figure 3.2: List of Plugins, Applications and Language Bindings using LibPressio. Descriptions can be found in the Glossary. A up to date list and description can be found at `https://github.com/robertu94/libpressio`



benefit the entire compression community. Abstractions such as ZFP's inline array's, python bindings for a compressor, HDF5 plugins no longer have to be developed for a single compressor, but all of them simultaneously. In the remainder of this section, we discuss some particular tools that we have or are developing using LibPressio.

3.5.2 Productivity

In this section, we evaluate productivity improvements from adopting LibPressio. We measure productivity improvements in three ways: First, we consider the number of lines of normalized client code which has long been used as an estimate of effort for developing new applications and maintenance effort required for an implementation and show a 50% to 90% reduction in lines of client code. Next, we consider the complexity of the client code using the established McCabe Cyclomatic Complexity which estimates the complexity of client code by the number of possible paths through the code and show a 50% improvement for SZ and ZFP. Lastly, we evaluate porting costs when switching between compressors by using both the lines of changed code and the Levenshtein's distance and demonstrate a lower switching cost by all measures considered.

First we look at the effort to develop or maintain a code base supporting multiple compressors using lines of code. We started with a number of use cases which were supported by at least one of the leading lossy compressors: ADIOS2, Julia bindings, Python bindings, Rust bindings, command line interfaces (CLI), HDF5 filters, a configuration optimizer, and Z-Checker. We added to our list a few use cases that were requested by our collaborators: R bindings, an experimental test harness written in C++ distributed with MPI, a fuzzer which provides random inputs to the compressor to identify implementation flaws in the compressors. We then implemented each of these facilities in LibPressio to at least feature parity with the native tool. In some cases such as the LibPressio CLI, the LibPressio version implements many more features – for example, the libpressio CLI can compress and decompress HDF5 datasets where as the SZ, ZFP, and MGARD cannot. Additionally, in some cases the LibPressio bindings use the compressors in a more correct way such as passing dimensionality information correctly. Finally, in there of these cases – the CLI, python bidings and HDF5 filter – the implementation do not have competing a multi-compressor implementation: in these cases we simply sum the lines of code in each implementation. While this will over-count some code, like command line argument parsing code, it is often less than the code required to implement a correct abstraction. For clarity, we mark these entries in our table.

Task	Compressors	Lines Native	Lines LibPressio	Improvement	Relative Improvement
ADIOS2 [36]	3	744	367	377	50.67%
BindingJulia [39]	1	299	25	274	91.64%
BindingPython [38], [40] †	2	768	363	405	52.73%
BindingR	-	-	793	-	-
BindingRust [41]	1	112	34	78	69.64%
CLI [6], [7], [38] †	3	1649	756	893	54.15%
Configuration Optimizer [42]	1	4683	1869	2814	60.09%
DistributedExperiment	-	-	613	-	-
Fuzzer	-	-	24	-	-
HDF5 filter [6], [38] †	2	1469	438	1031	70.18%
Z-Checker [37]	7	3052	405	2647	86.73%

Table 3.2: Lines of Client Code for Various Usages, † indicates no native multi-compressor implementation exists

To account for differences in formatting/style, clang-format was applied to all files, and we then measured the number of lines of code example applications utilities using the `cloc` utility. In the case of larger libraries like ADIOS2 [36] or Z-checker ([37], we include only the files that directly include compressor libraries headers.

Table 3.2 provides a summary of our results. We consistently find that LibPressio decreases the number of lines of code required between 50-90%.

We also computed the McCabe complexity of the command line tools [43]. MGARD again is the smallest at 14, followed by libpressio at 55, and SZ and ZFP clock in at 128 and 137 respectively. Most of the complexity is due to parsing of the command line arguments, but some of this is responsible for handling different reading in different types of data or different handling of some modes. Note for libpressio, this is a fully mpi-parallelized implementation that supports 6+ input formats as well as 27+ compressors and meta compressors whereas the others are a serial C version for a single compressor and input type. Figure 3.3 summarizes these results.

We computed the difference between switching from SZ to ZFP both with and without LibPressio for a basic use with a hard-coded error bound. We computed the Levenshtein's distance for both configurations [44]. We report these values in Table 3.3. While for small examples like

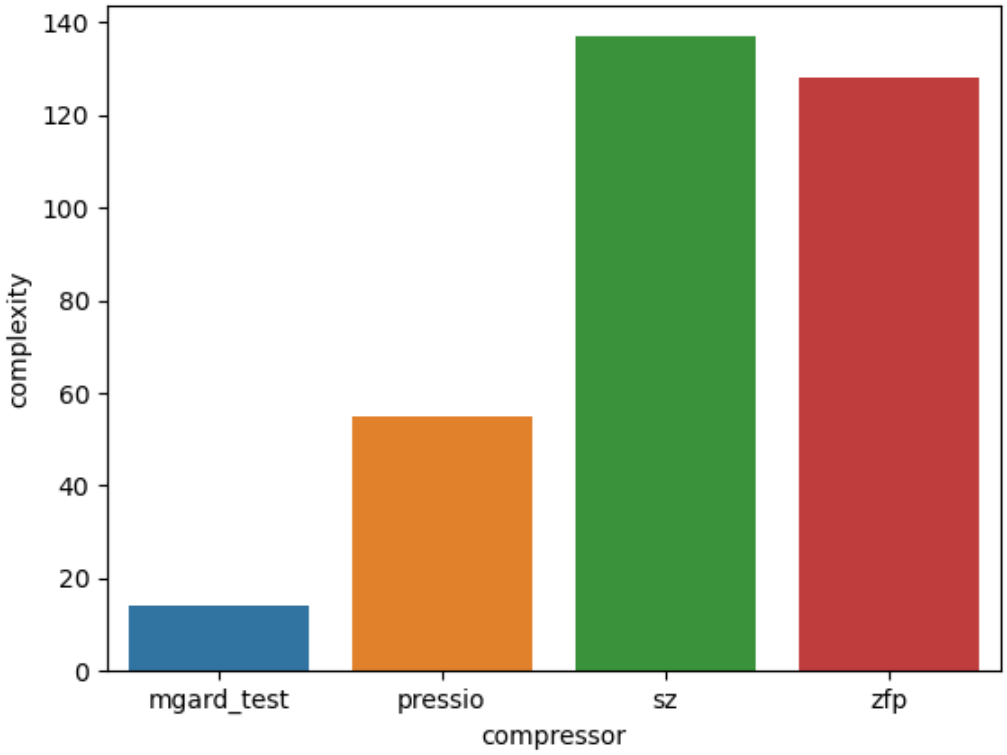


Figure 3.3: McCabe Cyclomatic Complexity for the Command Line Interfaces; MGARD has fewer robustness checks and can crash if used incorrectly

Method	Source Characters	Source Lines	Config Lines
Native	768	27	0
LibPressio	90	3	0
LibPressio+ConfigFile	0	0	3

Table 3.3: Minimum Changes to switch from SZ → ZFP in C

Component	Description	Component	Version
CPU	Intel Xeon 6148G (40 Cores)	Compiler	GCC 8.3.1
RAM	372 GB	OS	CentOS 8
Interconnect	100 GB/s HDR Infiniband	MPI	OpenMPI 3.1.6
MGARD	v0.1.0	SZ	v2.1.12
ZFP	v0.5.5	LibPressio	0.70.4

Table 3.4: Hardware and Software Details

this one the cost of switching is relatively minor, they do not account for the added complexities of learning new APIs or the benefits of performance portability to be gained by adopting LibPressio.

3.6 Quantifying Overhead

First we measure the runtime overhead of using LibPressio relative to using the native APIs for the compressors such as SZ, ZFP, and MGARD. For this portion, we consider only the serial CPU version of these functions and leave the evaluation of the parallel functions to Section 3.7.

The hardware we used for our experiments in summarized in Table 3.4. We selected these particular nodes because they were the largest nodes on the cluster that were available in sufficiently large quantifies to run the experiments in a reasonable time. Generally we selected the newest "prefered" version of the software available in Spack's develop branch at the time of submission and the compiler was the default system compiler [45]. All software was compiled using default optimizations provided by Spack (generally `-O2`).

For native APIs, we modify the command line interfaces to introduce timing calls directly around the calls to `SZ_Compress`, `zfp_compress`, and `mgard::compress` using the POSIX `clock_gettime` api with `CLOCK_MONOTONIC`. For LibPressio, we used the metrics interface `time` module which uses libstdcxx's `std::chrono::steady_clock::now()` before and after the call to the compressor's compress function which in turn uses the same POSIX api. We confirm with GDB that the same arguments are passed to the calls to the underlying compression functions. We further

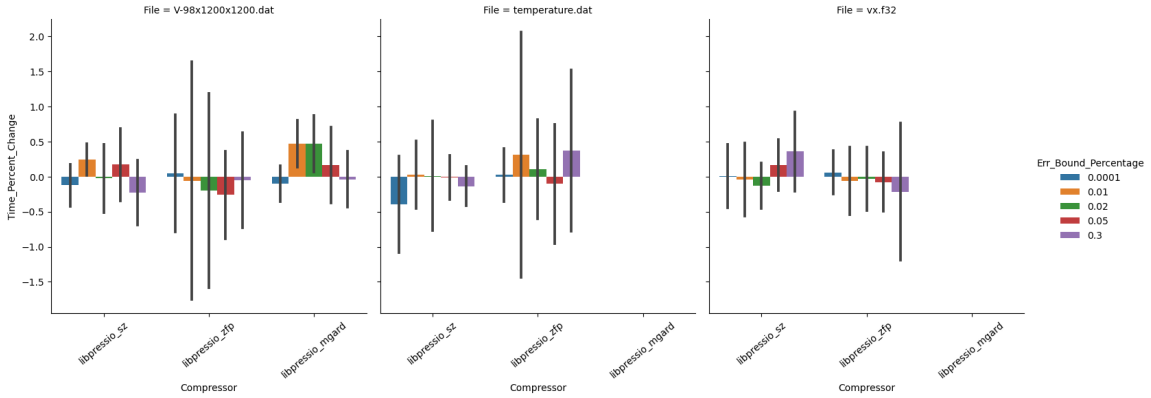


Figure 3.4: Compression Overhead for SZ/ZFP using LibPressio

confirm that both the native and libpressio implementations link the the same shared libraries that provide the compression functions to ensure that equivalent code generation occurs.

We conduct our experiments for several value range error bounds on various files from the SDRBench repositories to show that our results are robust to the choice of file and error bound for multiple compressors [46].

As we can see in Figure 3.4, the overhead measured for compression and decompression is for the overwhelming number of cases not statistically significant with the absolute overhead within the 95% confidence interval for the test and always less than 0.7% runtime overhead. While not dis-positive, there is insufficient evidence to conclude that there is a difference in run-time between these configurations.

3.7 Performance Portability

A general abstraction like LibPressio can enable a variety of possible features that can be used between compressors. One such features is the automated cpu-based parallelization of compression for thread-safe compressors. In this section, we evaluate the development effort, and features, compression performance, and run-time performance of LibPressio's automatic parallel compression against the bespoke parallel versions of SZ and ZFP to demonstrate how it can enable a scalable implementation as well as for MGARD which at time of writing has no bespoke parallel implementation.

We begin by evaluating the features of LibPressio's parallel implementation against SZ and ZFP custom implementations. Table 3.5 summarizes the difference between the implementations.

Implementation	Lines	n-d data	datatypes
SZ-OMP	830	1 (3d)	1
ZFP-OMP	299	4 (1-4d)	4
LibPressio	356	arbitrary	arbitrary

Table 3.5: Comparison of Parallel Implementations

SZ's parallel implementation runs in parallel 3 portions of the SZ algorithm: the prediction of points, the copying of unpredictable data, and the encoding of the saved data using a Huffman encoding. The algorithm it uses it requires that the number of threads used be a multiple of 4 and is only implemented for data for 32 bit floating point 3-d data bounded by the absolute error bound.

ZFP's error bounded parallel CPU implementation of compression in ZFP but not decompression [2]. ZFP's compression implementation supports all data types and dimensions supported by ZFP. ZFP implementation operates by splitting the data into 4^n chunks as it does in the serial case, but instead the compressing each of the chunks in parallel, and then concatenating the results. When the data is smaller than a 4^n chunk, it is padded to the nearest multiple of size 4^n.

However, many compressors such as MGARD do not have a parallel CPU implementation. We include MGARD in this section as an evaluation for a compressor one would like to parallelize, but may not have the time or expertise to warrant a bespoke parallelization.

LibPressio's implementation is in principle similar to ZFP's but with a few important differences. First, it has been generalized to operate with any compressor rather than only ZFP by the use of libpressio's compressor abstraction. This means that LibPressio's implementation has lower compression ratios that what a bespoke parallel implementation can offer since it doesn't have sufficient information to output common header information – such as SZ's huffman tree which could be shared – and because of the built-in index described next. It also means that LibPressio's implementation will support the same dimensions and data-types as the underlying compressor. The exact difference in compression ratios differs depending on the exact compressor, dataset, and chunksize, but can be quite comparable as shown in Table 3.6 However because LibPressio's implementation does not make any assumptions about the underlying compressor outside of thread safety, compressors like MGARD just work with LibPressio's implementation. Second, LibPressio writes a header containing the sizes of each compressed chunk to the beginning of the compressed output. This en-

[2]There is also a parallel GPU implementation of its non-error bounded mode which does implement parallel decompression, but is excluded from consideration here for a closer comparison between methods and because non-error-bounded modes may not be appropriate for scientific codes

ables the parallel decompression feature not offered by ZFP. During decompression, the compressed buffer is partitioned according to the header and then decompressed in parallel. Third, the current implementation in LibPressio allows arbitrary block sizes that contiguous in memory. In order to share the serial decompression implementation, ZFP uses the blocks of size 4^n used by the serial version, however, because LibPressio has the header it can support re-assembling arbitrary chunks. Additionally, the custom chunk size effects performance as we show shortly. It is planned that a future version of LibPressio will lift this restriction and allow padding like ZFP solution to allow truly arbitrary block sizes or other possible alternative solutions.

We conducted a series of scalablity and compression ratio experiments using SZ, ZFP, and MGARD using their custom parallel implementations and using a automatic parallelization system using LibPressio we developed.

We used the same hardware and software as described in Table 3.4 These nodes have 2 sockets with 20 cores each therefore we choose several chunk sizes: 1, 2, 4, 8, 10, 20, 22, 24, 26, 28, 30, and 40. SZ's parallel implementation requires either 1, 2 or thread multiples of 4, and will not use additional threads unless they complete a full multiple of 4. LibPressio and ZFP's implementation places no restriction on the number of threads. For sake of space, we report up to size 24; results after 24 threads are similar. For these experiments, we repeat each experiment 30 times to account for variance between runs.

For our experiments, we choose the volume field from the SCALE-LETKF dataset on SDRBench. Other data-sets yielded similar results. We choose this dataset as one the largest 3d floating point data sets included in SDRBench with dimension $1200 \times 1200 \times 98$. Large data-sets sizes provide parallel compression implementations provide sufficient data to warrant a threaded implementation. The current implementation of LibPressio's parallel implementation requires that the chunk sizes be contiguous and equal sized. Therefore we considered chunk sizes of 7 chunks of $1200 \times 1200 \times 14$, 14 chunks of $1200 \times 1200 \times 7$, and 49 chunks of $1200 \times 1200 \times 2$. We excluded 2 chunks of $1200 \times 1200 \times 49$ as this would limit the amount of parallelism that could be expressed. SZ does not have a concept of chunk-size that is exposed. ZFP also has a chunk-size parameter, by default it creates a number of chunks equal to the number of threads which we found to be the best configuration. For this dataset, ZFP has 2,250,000 fixed size blocks of size $4 \times 4 \times 4$ of which 90,000 of them are padded to $2\times$ of their original size which are divided as evenly as possible between the available chunks resulting in an effective input data size of that is 2% larger.

configuration	chunks	compression_ratio
libpressiothread MGARD	7	1.389971
	14	1.435876
	49	n/a
libpressiothread sz	7	2.245248
	14	2.197810
	49	1.887099
libpressiothread zfp	7	1.427047
	14	1.377799
	49	0.830169
szthreads sz	1	2.256079
zfpthreads zfp	1	1.589013

Table 3.6: Compression Ratio of different parallel implementations, error_bound=1e-6, 7 chunks, SCALE-LETKF V dataset, chunksize=49 not supported for MGARD due to minimum data size requirements in MGARD

We first consider the compression ratio results in Table 3.6. For LibPressio threads, the compression ratio generally worsens as the number of chunks increases. For small chunk sizes, a compression ratio overhead of between .04 and 10% Excluding ZFP with 49 chunks, For larger chunk sizes, a compression ratio overhead up to 16% is observed. This is because the libpressio threaded versions have duplicate header information that a custom implementation could discard, less spatial information to use to find trends in the data which can be exploited for compression, and similar meta-data structures that could be combined such how the Huffman tree in SZ which SZ's openmp implementation uses a common tree. The results of ZFP with 49 chunks is a pathological case. Recall that ZFP effectively pads blocks that are not 4^n. Where as ZFP parallel implementation padded only 4% of the blocks, the libpressio implementation with 49 chunks forces ZFP to pad all 2250000 blocks because it sees each input as $1200 \times 1200 \times 2$. If the LibPressio implementation were modified was able to use a chunk size of $1200 \times 1200 \times 4$ much of this overhead would be eliminated for this dataset. Likewise the MGARD with chunksize=49 is omitted because the compressor does not support chunks with any dimension less than 3.

Next we consider the parallel runtime for compression in figure 3.5. We see at small thread counts, each of the methods is relatively similar regardless of what methods are used. Generally, greater chunk counts allow a higher maximum performance at the optimal number of threads. This is because there is more work that can be done in parallel allowing it to be completed more quickly. However even for 49 chunks, performance maxes out after about 20 threads for both LibPressio and

native methods. This is likely due to these systems have 20 cores per socket and with more than 20 threads, the tasks must be spread across a NUMA boundary and participate in cross-socket cache invalidation. One exception is that LibPressio with 49 chunks shows a performance degradation after 22 threads that gets worse as thread counts increase that is not seen with LibPressio with using ZFP. One explanation for this behavior is that SZ performs both a Huffman encoding and a lossless ZSTD encoding which are both heavily dependent on the CPU cache to have high performance. As the thread count rises, there is increased contention for these cache resources. The reason that SZ's native implementation does not suffer the same performance degradation as the number of threads grows is because it shares a common Huffman tree allowing for better cache behavior and the lossless encoding is done serially. MGARD which has no other parallel CPU implementation also scales nicely.

Next we consider the parallel runtime for decompression in figure 3.6. We see similar results to the parallel speedup for compression, however there are a few notable differences. First, we don't see the performance degradation in ZFP for SZ above 22 threads that we saw for compression with 49 chunks. This is likely because during decompression there aren't any writes to the memory containing the Huffman trees after they are loaded requiring cache invalidation between NUMA nodes resulting in a more efficient usage pattern. Next, we observe that ZFP natively doesn't parallelize decompression and thus does not provide a performance speedup meaning that LibPressio is the only currently the error bounded implementation for parallel decompression using ZFP. We also observe that decompression scales better than compression achieving a greater speedup than what compression achieves. This is likely due to better cache behavior for decompression than for compression which has been long observed in lossless compression algorithms [34].

When compression and decompression results are viewed in concert with the compression ratio results, a chunk count of 14 could be recommended. However, these parameters can very from machine to machine. LibPressio provides a meta-compressor – LibPressio-Opt – that can optimize the configuration of a compressor on given data-sets so the user does not have to conduct this analysis themselves [2]. This initial validation could be performed on the first compression operation to enable easy to use performance portability across machines.

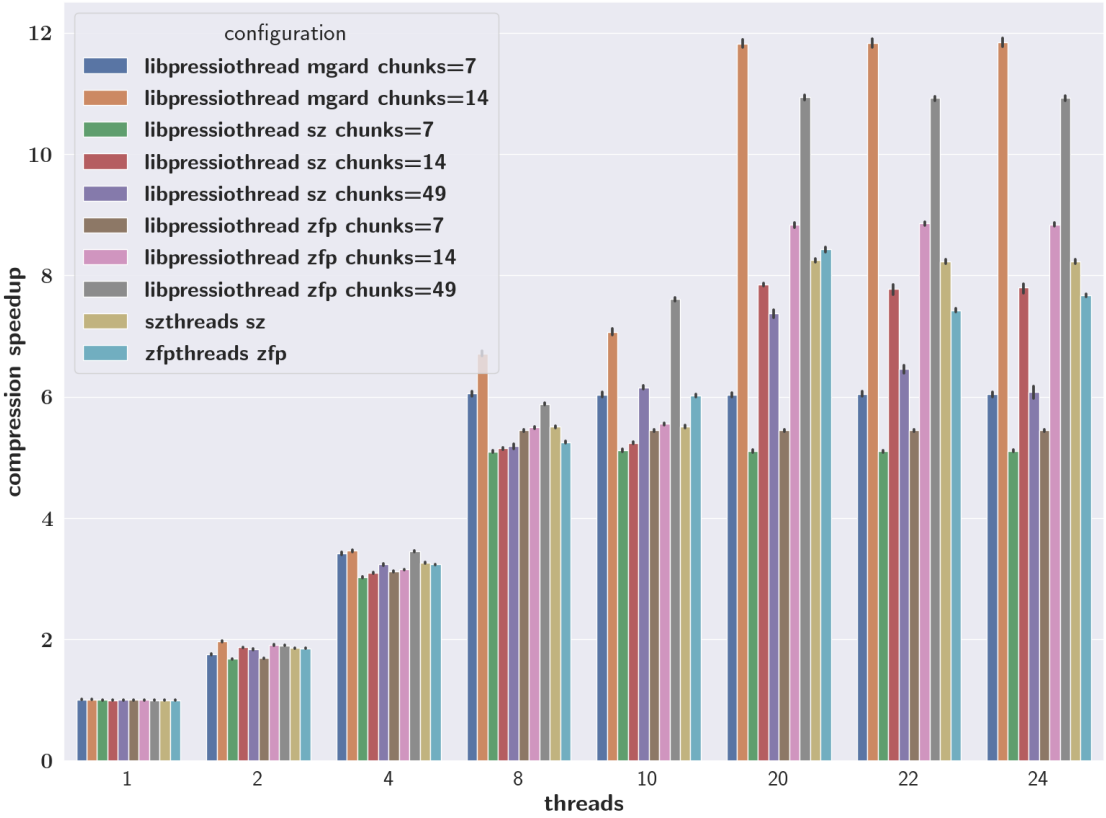


Figure 3.5: Compression Speedup for SZ/ZFP using LibPressio

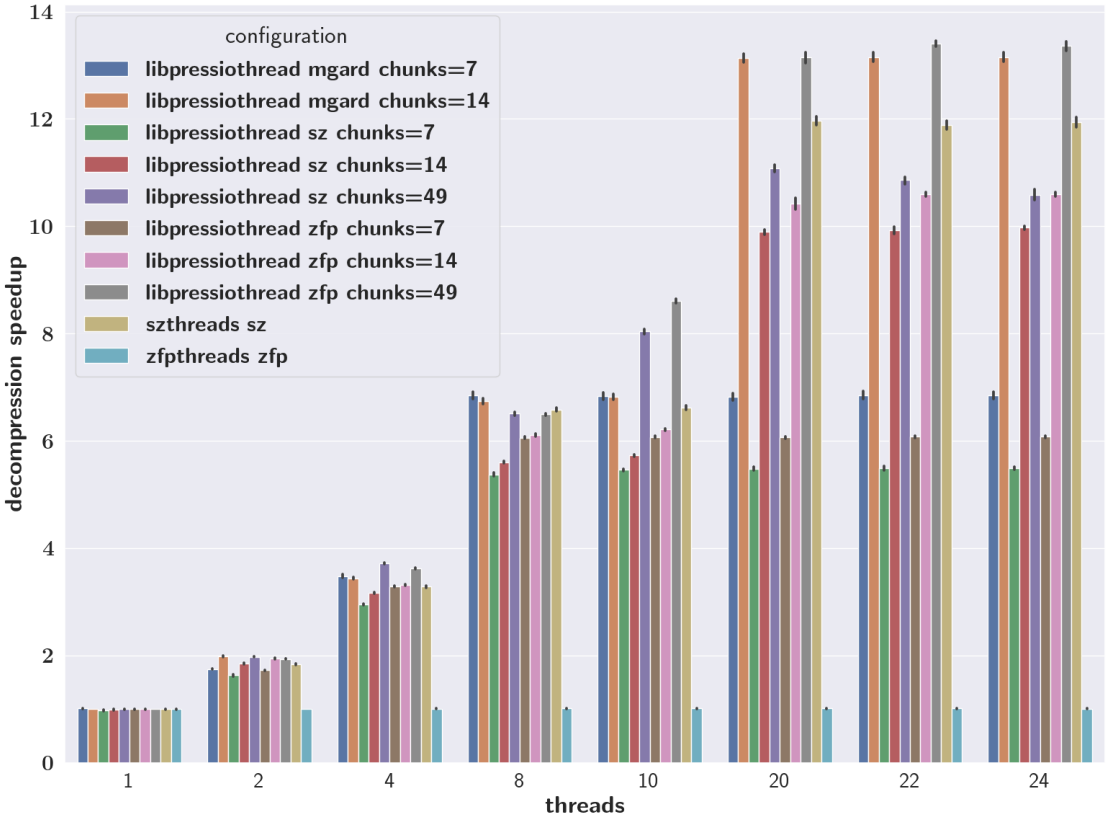


Figure 3.6: Decompression Speedup for SZ/ZFP using LibPressio

3.8 Conclusions and Future Work

In this paper, we highlighted several key aspects of LibPressio and how they can be used to enable a wide array to tools to advance the state of compression usage and research by simplifying existing work flows. We further demonstrate These various improvements brought many new features to existing compressors and new compressors to new languages and tools while simplifying the existing compressor work flows and reducing redundant work.

For future work on the abstraction of compressors, we plan to extend LibPressio to account for the following use cases. Each of these cases require research to answer questions regarding how to most efficiently perform this these improvements.

1. Better support for Accelerators – with the apparent demise of Moore's law, computing architectures are becoming increasing heterogeneous requiring libraries to become aware of this heterogeneity to make maximal use of the hardware. For libraries like LibPressio this means three things: 1) a notion of data locality for compression tasks to enable efficient migration of data to and from memory pools on accelerators such as the GPUs, FPGAs, and specialized memory resources on board with the CPU. The abstaction should provide means for indicating which memory types the compressor supports to reduce or eliminate redundant copies to and from these different memory pools, and potentially allow for better non-uniform memory access patterns on mulit-core CPUs. 2) a consistent means of compressors reporting and requesting which parallel resources compression libraries require, can use if are available, are allocated for the task, and a means to share common parallelism resources such as GPU devices devices between other libraries that may use them to avoid contention and allow for efficient scheduling of compression tasks to available accelerators. 3) Support in the compressor plugins to use the parallel implementations where available.

2. Better support for Asynchrony and Streaming Compression – this is another key feature for supporting accelerators, but deserves its own heading because it has applications beyond accelerators. A common challenge with compressors is having sufficient data to compress in order to achieve a desirable bandwidth through the compressor. One way to achieve this is to batch multiple requests for compression to a single underlying call the compression library. Beyond accelerators, some applications such as LCLS-II produce data in a streaming fashion. In such a case, the user does not know how large the incoming stream of data will be. Given

the volumes of data involved with LCLS-II, buffering the entire dataset in memory is infeasible due to the shear volume of memory involved with a single stream. Instead compressors that support it need a way to provide asynchrony between calls to the compressor and the compression stream. Some of this design could be abstracted to a meta compressor to enable these features for existing compressors.

3. Better support for Sparse data Compression – Another growing problem in HPC is the problem of sparse datasets. Sparse datasets have many redundant zeros, but that doesn't mean they are small. Even sparse datasets can be many GB, thus inviting the use of compression. These datasets present different challenges for lossy compressors since traditional prediction, transform, multi-grid, and encoding compressors were designed for dense data representations. To be a compression library that supports all compression problems, sparse problems cannot be ignored. From a compression abstraction library perspective, supporting sparse problem means providing a means of using spare data volumes to compressors in a way to which they can be converted to appropriate representations for the compressors is key.

Chapter 4

FRaZ: A Generic High-Fidelity Fixed-Ratio Lossy Compression Framework for Scientific Floating-point Data

Portions this chapter were previously published [2] in the International Parallel and Distributed Processing Symposium 2020 under the title: FRaZ: A Generic High-Fidelity Fixed-Ratio Lossy Compression Framework for Scientific Floating-point Data

Abstract

With ever-increasing volumes of scientific floating-point data being produced by high-performance computing applications, significantly reducing scientific floating-point data size is critical, and error-controlled lossy compressors have been developed for years. None of the existing scientific floating-point lossy data compressors, however, support effective fixed-ratio lossy compression. Yet fixed-ratio lossy compression for scientific floating-point data not only compresses to the requested ratio but also respects a user-specified error bound with higher fidelity. In this paper, we present FRaZ:

a generic fixed-ratio lossy compression framework respecting user-specified error constraints. The contribution is twofold. (1) We develop an efficient iterative approach to accurately determine the appropriate error settings for different lossy compressors based on target compression ratios. (2) We perform a thorough performance and accuracy evaluation for our proposed fixed-ratio compression framework with multiple state-of-the-art error-controlled lossy compressors, using several real-world scientific floating-point datasets from different domains. Experiments show that FRaZ effectively identifies the optimum error setting in the entire error setting space of any given lossy compressor.

4.1 Introduction

Today's scientific research applications produce volumes of data too large to be stored, transferred, and analyzed efficiently because of limited storage space and potential bottlenecks in I/O systems. Cosmological simulations [11], [47], for example, may generate more than 20 PB of data when simulating 1 trillion particles over hundreds of snapshots per run. Climate simulations, such as the Community Earth Simulation Model (CESM) [48], may produce hundreds of terabytes of data [49] for each run.

Effective data compression methods have been studied extensively. Since the major scientific floating-point datasets are composed of floating-point values, however, lossless compressors [32], [50], [51]cannot effectively compress such datasets because of high entropy of the mantissa bits. Therefore, error-bounded lossy compressors have been widely studied because they not only significantly reduce the data size but also prevent data distortion according to a user's specified error bound. Most existing lossy compressors consider the error bound preeminent and endeavor to improve the compression ratio and performance as much as possible subject to the error bound.

However, many scientific application users have requirements for the compression ratio. These requirements are determined by multiple factors such as the capacity of the assigned storage space, I/O bandwidth, or desired I/O performance. Hence, these users desire to perform fixed-ratio lossy compression—that is, compressing data based on the required compression ratio instead of only strictly respecting user's error bound. In this case, the lossy compressor needs to adjust the error bound to respect the target user-specified compression ratio, while minimizing the data distortion. The user can also provide additional constraints regarding the data distortion (such as maximum error bound) to guarantee the validity of the results from reconstructed data. While fixed-ratio

compression can be obtained by simply truncating the mantissa of the floating-point numbers, this approach may not respect the user's diverse error constraints. With such additional constraints, the lossy compressor should make the compression ratio approach the expected level as closely as possible, while strictly respecting the data distortion constraints.

In this paper, we propose a generic, efficient fixed-ratio lossy compression framework, FRaZ, that is used to determine the error settings accurately for various error-controlled lossy compressors, given the particular target compression ratio with a specific scientific floating-point dataset. Our design involves two critical optimization strategies. First, we develop a global optimum searching method by leveraging Davis King's global minimum finding algorithm [52] to determine the most appropriate error setting based on the given compression ratio and dataset in parallel. Second, our parallel algorithm optimizes the parameter searching performance by splitting the search range into distinct regions, parallelizing on file, and—in the offline case—by time-step.

Constructing a generic, high-fidelity framework for fixed-ratio lossy compression poses many research challenges. First, as we elaborate in Section 4.5, the relationship between error bounds and compression ratios is not always monotonic because of the use of dictionary encoder phases in some compressors such as SZ. Second, since we aim to create a generic framework such that more compressors can be included in the future, we cannot utilize properties of the specific compressors we use to optimize performance as has been done in prior work [21], [53] and must instead treat the compression algorithm as a black box. This means that our algorithm cannot take advantage of properties induced by block size or expected behavior induced for a particular data distribution. Third, since we treat the compressors as a black box, we must carefully study how to modify existing algorithms to minimize the calls to the underlying compressors and orchestrate the search in parallel, in order to have a tool that is useful to users while working around the limitations of the various current and potential future compressors.

We perform the evaluation for our framework based on the latest versions of state-of-the-art lossy compressors (including SZ [5], [20], [31], ZFP [6], and MGARD [7]), using well-known real-world scientific floating-point datasets from the public Scientific Data Reduction Benchmark (SDRBench) [46]. We perform the parallel performance evaluation on Argonne's Bebop supercomputer [54] with up to 416 cores. Experiments show that our framework can determine the error setting accurately within the user-tolerable errors based on the target compression ratios, with very limited time overhead in real-world cases.

The remainder of the paper is organized as follows. In Section 4.2, we introduce the background of this research regarding various state-of-the-art lossy compressors, and we present several examples about user's requirement on specific compression ratios. In Section 4.3, we compare our work with the related work from fixed-rate compression, image processing, and signal processing. In Section 4.4, we present a formal problem formulation to clarify our research objective. In Section 4.5, we describe our design and our performance optimization strategies. In Section 4.6, we present the evaluation results. In Section 4.7, we present or conclusions and end with a vision of future work.

4.2 Background

In this section, we describe the research background, including the existing state-of-the-art error-controlled lossy compressors and fixed-ratio use cases.

Although SZ, ZFP, and MGARD provide advanced features to control the distortion of lossy compression, none of them provide high-fidelity fixed-ratio compression.

Accuracy of EBLC is dictated at compression time by selection of an error bound and error bounding type — e.g., absolute, relative, number of bits and are selected to minimize impact on quantities of interest in scientific simulations. For use cases that preform data analytics on lossy compressed data, trial-and-error is often used to identify acceptable compression tolerances [15]. The trial-and-error is often done offline to ensure that the selected error bound is robust for multiple time-steps and does diminish the quality of the analysis. However, if the lossy compressed data is used to advance the simulation the simulation trial-and-error is possible [55], but recent works have explored the relation of compression error to numerical errors present in the simulation and provide strategies on error tolerance selection [17], [24], [56].

4.2.1 Fixed-Ratio Use Cases

In this subsection, we describe several fixed-ratio use cases to demonstrate the practical demands on the fixed-ratio compression by real-world application users.

The first use case is significant reduction of storage footprint. On the ORNL Summit system, for example, the capacity of the storage space is limited to 50 TB for each project by default. Many scientific floating-point simulations (such as the CESM climate simulation and HACC cosmological

simulation) may produce hundreds of terabytes of data in each run (or even over 1 PB of data), such that the compression ratio has to be 10:1 or higher to avoid execution crash due to no space being left on storage. Even if a larger storage allocation is awarded or purchased at considerable financial cost, projects generating extreme volumes often face the need to reduce their storage footprint in order to make room for their next executions. Fixed-rate compression provides the ability to store multiple simulations given a fixed amount of storage but suffers from large inaccuracies in the data which high compression ratios are required (see Figure 4.10).

The second practical use case explores best-fit lossy compression solutions based on the user's post-analysis requirement (such as visual quality or specific analysis property) by at fixed compressed sizes. None of the existing error-controlled lossy compressors provide the fixed-ratio compression mode, however, and therefore users have to seek the best-fit choice by conducting inefficient trial-and-error strategies with different error settings for each compressor to achieve a target compression ratio. Furthermore, there is no universal model that accurately predicts compression ratio based on compressor configuration for a variety of input data.

The third practical use case involves the matching of I/O bandwidth constraints and accelerating the I/O performance. Advanced light-source instruments, such as the Advanced Photon Source and Linac Coherent Light Source (LCLS-II), may generate image data at an extremely high acquisition rate, such that the raw data cannot be stored efficiently for post-analysis because of limited I/O bandwidth. Specifically, LCLS-II is producing instrument data with up to 250 GB/s while the corresponding storage bandwidth is only 25 GB/s. Thus the designers of LCLS-II expect to reduce the data size with a compression ratio of 10 or higher [13]. Spring8 [57] researchers also indicate that their data could be generated with 2 TB/s, which is expected to be reduced to 200 GB/s after data compression.

We note that users often require random-access decompression across time steps, which means that they prefer to be able to decompress the data individually at each time-step because decompressing the whole dataset with all time-steps requires a significant amount of time or is impossible because of memory allocation limits.

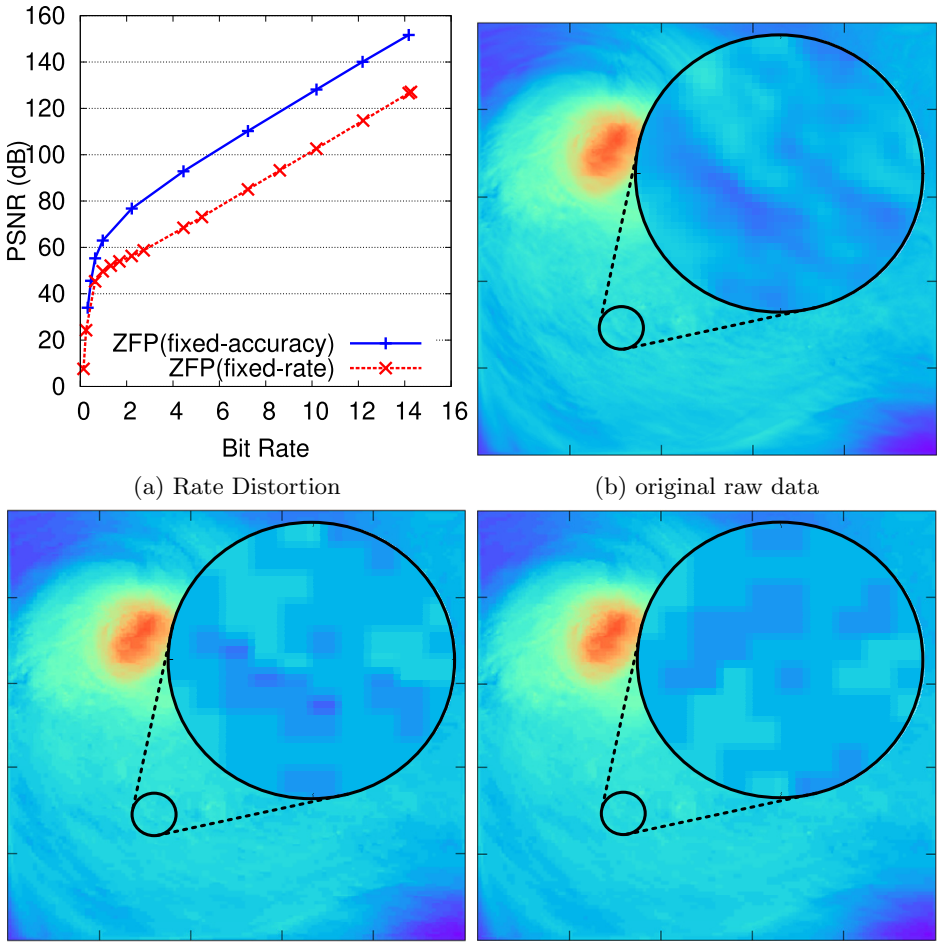
4.3 Related Work

In the compression community, the similar type of compression is called "fixed-rate compression," where the *rate* here refers to the bit rate, which is defined as the number of bits used to represent one symbol (or data point) on average after compression. The lower the bit rate, the higher the compression ratio. Hence, fixing the bit rate means fixing the compression ratio. In the remainder of this section, we compare our work with prior work in these areas.

In addition to the fixed-accuracy modes (i.e., accuracy and precision error-bounding modes), ZFP offers a fixed-rate mode [6]. The fixed-rate mode of ZFP offers precise control over the number of bits per symbol in the input data. It operates by transforming the data into a highly compressible domain and truncating each symbol to reach the appropriate rate. However, the fixed rate mode of ZFP is not error bounded, and it suffers from significantly lower compression quality than does the fixed-accuracy mode of ZFP. Figure 4.1 (b) demonstrates the compression quality of ZFP using fixed-accuracy mode and fixed-rate mode, respectively. One can clearly see that the latter exhibits much worse rate distortion than does the former (up to 30 dB difference with the same bit-rate in most cases). Rate distortion is an important indicator to assess the compression quality. Its detailed definition can be found in Section 4.6.2 (4). figures 4.1 (c) and (d) clearly show that the fixed-rate mode results in much lower visual quality (e.g., more data loss) than the fixed-accuracy mode with the same compression ratio, 50:1 in this example. In absolute terms, the fixed-accuracy mode leads to much higher peak signal-to-noise ratios (PSNR) and lower auto-correlation of compression errors (ACF(error)), which means better compression quality than the fixed-rate mode. In ZFP's website and user guide, the developer of ZFP also points out that the fixed rate mode is not recommended unless one needs "to bound the compressed size or need random access to blocks" [6].

In contrast, not only does our framework fix the compression ratio but it also achieves higher compression quality for different compressors (such as SZ, ZFP, and MGARD) based on their error-bounding mode. Additionally, it can provide random access to the same level as can ZFP's fixed rate mode when supported by the underlying compressor. Since our framework utilizes a control loop to bound the compression ratio, it may suffer a lower bandwidth than ZFP's fixed-rate mode to a certain extent. The tradeoff for this lower bandwidth is compressed data of far higher quality for the same compression ratio, which we demonstrate in detail in the evaluation section.

The literature also includes studies investigating the use of fixed-ratio compressors for im-



(a) Rate Distortion

(b) original raw data

(c) ZFP (accuracy) PSNR=55.3, max error=4.2, SSIM=0.94, ACF(error)=0.67

(d) ZFP(fixed-rate) PSNR=45.4, max error=33.7, SSIM=0.94, ACF(error)=0.72)

Figure 4.1: Data distortion of ZFP in fixed-accuracy mode and fixed-rate mode (Hurricane TCf field) with CR=50:1

ages. One such work [58] is JPEG-LS, a fixed-ratio compressor for images. This work adopts a combination of a prediction system for data values and two runs of Golumb-Rice encoding to encode RGB values for images. The first run of the Golumb-Rice encoding is used to estimate the quantization level used in the the second run of the encoder. Golumb-Rice assumes integer inputs, whereas our work is applicable on all numeric inputs.

Some work also has been done on fixed-ratio compression in digital signal processing [59] In this domain, adaptive sampling techniques are used to maintain a budget for how many points to transmit. When a point is determined to provide new information (using a predictor, interpolation scheme, or some other method), it is transmitted and the budget is expended as long as there is

remaining budget. If the budget is spent, then no points are transmitted until the budget is refilled. Over time, the budget is increased to keep the rate constant. In contrast, our work does not rely on a control loop to maintain the error budget, so can look at the data holistically to decide where to place the loss in our signal, allowing for more accurate reconstructions.

4.4 Problem Formulation

In this section, we formulate the research problem, by clarifying the inputs, constraints, and the target of our research.

Before describing the problem formulation, however, we introduce some related notations as follows. Given a specific field f at time-step t of an application, we denote the dataset by $D_{f,t}$ $= \{d_1, d_2, \ldots, d_n\}$, where d_i refers to the original value of data point i in the dataset and n is the number of elements. We denote its corresponding decompressed dataset by $D'_{f,t}= \{d'_1, d'_2, \cdots, d'_n\}$, where d'_i refers to the reconstructed value after the decompression. We denote the original data size and compressed data size by $s(D_{f,t})$ and $s(D'_{f,t})$, respectively. The compression ratio (denoted by ρ) then can be written as $\rho(D_{f,t}) = \frac{s(D_{f,t})}{s(D'_{f,t})}$. Moreover, we denote the target compression ratio specified by the users as $\rho_t(D_{f,t})$, and the real compression ratio after the compression as $\rho_r(D_{f,t}, e)$.

The fixed-ratio lossy compression problem is formulated as follows, based on whether it is subject to an error-control constraint or not.

- *Nonconstrained fixed-ratio compression*: The objective of the nonconstrained fixed-ratio lossy compression is to confine the real compression ratio to be around the target compression ratio within a user-specified tolerable error (denoted by ϵ), as shown below.

$$\rho_t(D_{f,t}) - \epsilon \leq \rho_r(D_{f,t}) \leq \rho_t(D_{f,t}) + \epsilon \tag{4.1}$$

- *Error-control-based fixed-ratio compression*: The objective of the error-control-based fixed-ratio compression is to tune the compression ratio to be within the acceptable range $[\rho_t(D)-\epsilon, \rho_t(D)+\epsilon]$, while respecting the user-specified error bound (denoted by e), as shown below.

$$\begin{aligned} &\rho_t(D_{f,t}) - \epsilon \leq \rho_r(D_{f,t}, e) \leq \rho_t(D_{f,t}) + \epsilon \\ &s.t.\ \Pi(D_{f,t}, D'_{f,t}) \leq e, \end{aligned} \tag{4.2}$$

where $\Pi(D_{f,t}, D'_{f,t})$ is a function of error control. For instance, $\Pi(D_{f,t}, D'_{f,t})$=$\max_i |d_i - d'_i|$ for the absolute error bound, and $\Pi(D_{f,t}, D'_{f,t})$=$\sum_{d_i \in D_{f,t}, d'_i \in D'_{f,t}} (d_i - d'_i)^2$ for the mean squared error bound.

We summarize the key notation in Table 4.1.

Table 4.1: Table of Key Notation

Notation	**Description**
D	original data set for all time-steps and fields
D_f	original data set for all time-steps of a particular field
$D_{f,t}$	original data set for a particular field and time-step
$D'_{f,t}$	decompressed data set for a particular field and time-step
ρ_t	target compression ratio
ρ_r	real compression ratio
ϵ	acceptable error for ρ_t
e	error bound for compression
γ	maximum value of the loss function
θ	fixed parameters of the compressor
N	the number of dimensions
n	the total number of data points
T	the number of time-steps
α	the degree overlap between error-bound search ranges
U	maximum allowed compression error

4.5 Design and Optimization

In this section, we present the design of our fixed-ratio lossy compression framework and optimization strategies.

4.5.1 Design Overview

Figure 4.2 shows the design overview with highlighted boxes indicating our major contributions in this paper and the relationship among different modules in the framework. As shown in the figure, our FraZ framework is composed of five modules, and the optimizing autotuner and parallel orchestrator are the core modules. They are, respectively, in charge of (1) searching for the optimal error setting based on the target compression ratio with few iterations and (2) parallelizing the overall tuning job involving different searching spaces for each field and different time-steps and across various fields. We develop an easy-to-use library (called Libpressio [60]) to build a middle layer for abstracting the discrepancies of the APIs of different compressors.

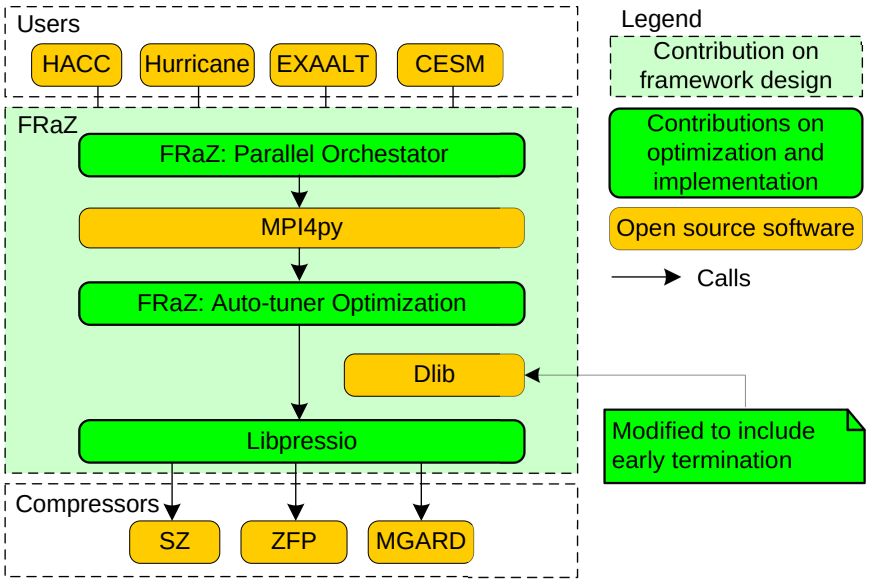


Figure 4.2: Design overview and summary of our contributions

We list our major contributions as follows:

1. Formulated fixed-ratio compression as an optimization problem in a way that converges quickly without resorting to multiobjective optimization

2. Evaluated several different optimization algorithms to find one that works on all of our test cases, and then modified it to improve performance for our FRaZ

3. Implemented and ran parallel search to improve the throughput of the technique

4.5.2 Autotuning Optimization

In this subsection, we describe our autotuning solution in detail, which includes three critical parts: (1) exploration of the initial optimization methods, (2) construction of a loss function, and (3) improvements to the optimization algorithm that involves how to deal with infeasible target compression ratio requirement and determines the exact error-bound setting.

4.5.2.1 Exploration of Initial Optimization Methods

In this subsection we describe how we choose which optimization method to use as a starting point for later refinement.

Before detailing FRaZ's optimizing autotuning method, we first analyze why the straightforward binary search is not suitable for our case. On the one hand, the application datasets may exhibit a non-monotonic compression ratio increase with error bounds. We present a typical example in Figure 4.3, which uses SZ to compress the QCLOUDf field of the hurricane simulation dataset. We can clearly see that the compression ratios may decrease significantly with larger error bounds in some cases. We also observe the spiky changes in the compression ratios with increasing error bounds on other datasets (not presented here due to space limit). The reason is that SZ needs to use decompressed data to do the prediction during the compression, which may cause unstable prediction accuracy. Moreover, SZ's fourth stage (dictionary encoder) may find various repeated occurrences of bytes based on output of the third stage, because a tiny change to the error bound may largely affect the Huffman tree constructed in the third phase of SZ. By comparison, our autotuning search algorithm is a general-purpose optimizer and takes into account the irregular relationship between compression ratios and error bounds. On the other hand, even on the datasets where monotonicity holds, binary search may still be slower than FRaZ's optimizing autotuner. For example, when searching for the target compression ratio 8:1 at the 48th time-step on the Hurriane-CLOUD field, our method requires only 6 iterations to converge to an acceptable solution, whereas binary search needs 39 iterations. The reason is that binary search may spend substantial time searching small error bounds, which would not result in an acceptable solution because it climbs from the minimum possible error bound to the user-specified upper limit.

When developing FRaZ's optimizing autotuner, we considered a number of different techniques to perform the tuning. Since we are developing a generic method, we cannot construct a general derivative function that relates the change in error bound to the change in compression ratio. Therefore, we need to decide between methods that use numerical derivatives and derivative-free optimization because the derivative of the compression ratio with respect to the error bound is unknown. The methods using numerical derivatives approximate the slope of the objective function by sampling nearby points. Some methods that fall in this category are gradient descent (i.e., Newton-like methods such as [61] and ADAM [62]). However, when evaluating an error bound to determine the compression ratio, we must run the compressor since we are using the compressors as black boxes, which may take a substantial amount of compared with the optimization problem. In this sense, numerical derivative-based methods are too slow.

We therefore turned our consideration to derivative-free optimization. We considered meth-

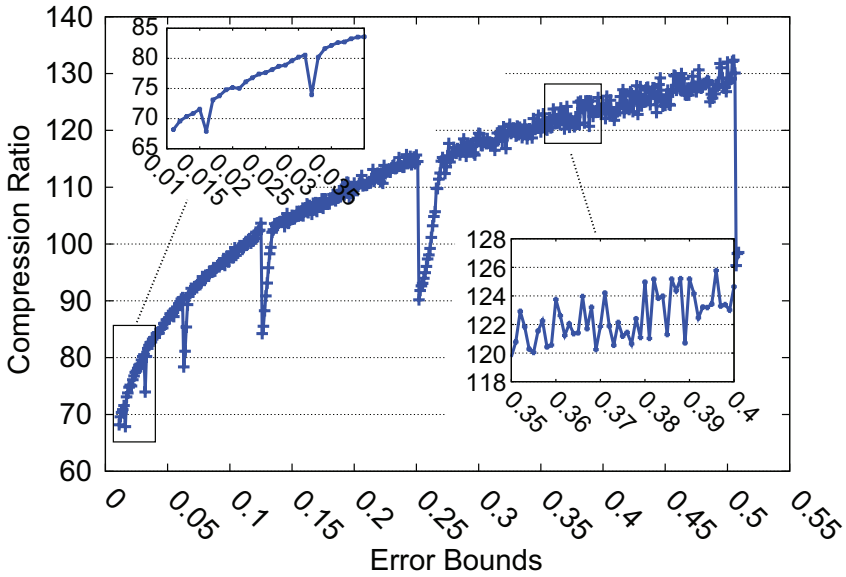


Figure 4.3: Example based on the hurricane simulation dataset (field: QCLOUDf.log10) showing that the relationship between error bounds and compression ratios is not always monotonic

ods such as BOBYQA [63], but they do not handle a large number of local optimums. This ability is essential for developing a robust tuning framework for lossy compression because many of the functions that relate error bounds to compression ratios look like the plot on the left of Figure 4.4: a step-like function with perhaps a slight upward slope on each step. In practice, we noted that it is easily able to escape the local optima in these functions.

We also took into account a variety of implementations of these algorithms, such as the ones in [52], [64], [65]. We decided between these libraries using three criteria: (1) correctness of the result, (2) time to solution, and (3) modifiability and readability of code. Ultimately we started with a black-box optimization function called `find_global_min` from the commonly used Dlib library from which we make our modifications [52]. The global-minimum-finding algorithm designed by Davis King that combines the works of [66] and [67]. It requires a deterministic function that maps from a vector to a scalar, a vector of lower bounds, and a vector of upper bounds as inputs. At a high level the algorithm works as follows. It begins with a randomly chosen point between the upper and lower bounds. Then, it alternates between a point chosen by the model in [66], which approximates the function by using a series of piecewise linear functions and chooses the global minimum of this function, and the model in [67], which does a quadratic refinement of the lowest valley in the model. According to [52], this method performs well on functions with a large number of local optimums,

and this performance was confirmed by our experience.

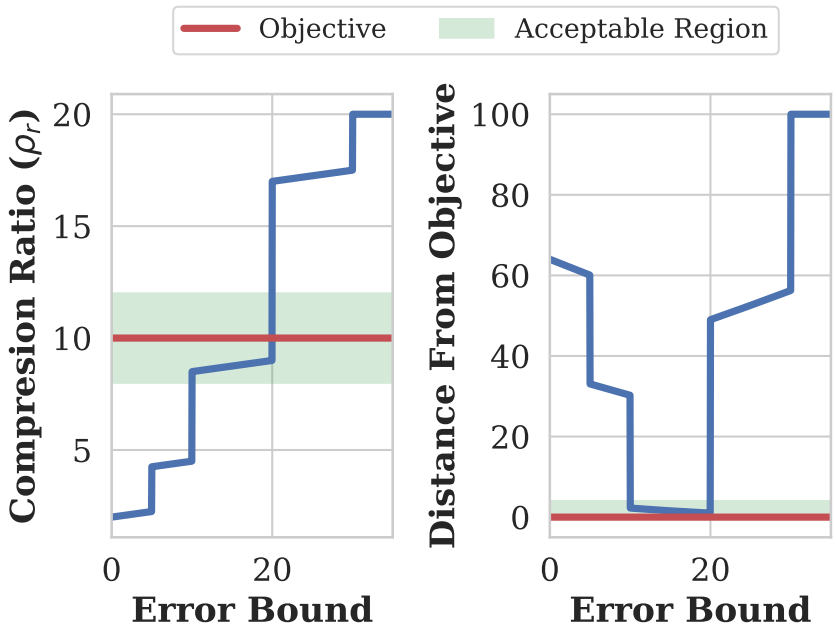


Figure 4.4: Illustration of autotuning optimization function: On the left is a hypothetical relationship between an error-bound level and compression ratio for some compressor and dataset. The target compression ratio is marked as a red line, and the acceptable region is colored green. On the right is the corresponding loss function using our method. The green area above the target compression ratio refers to the acceptable region. In this case, where there are blue points in the acceptable region, we call the result feasible. If the acceptable region was below the blue points, we would call it infeasible.

4.5.2.2 Construction of Loss Function

Now that we have an optimizer framework, we need to construct a loss function. First, we created a closure for each compressor, $\rho_r(D_{f,t}, e)$ that transformed its interface including a dataset D and parameters θ in a function accepting only the error bound e. To create the closure, we developed libpressio [60]—a generic interface for lossy compressors that abstracts between their differences so that we could write one implementation of the framework for SZ, ZFP, and MGRAD.

To convert this to a loss function, we chose the distance between the measured compression ratio and target compression ratio $\rho_r(D_{f,t}, e) - \rho_t(D_{f,t})$. Now, the function that relates an error bound to a compression ratio is an arbitrary function that may or may not have a global or local optimum. Therefore, we transformed the function by applying a clamped square function (i.e. $\min\left(x^2, \gamma\right)$, where γ is equal to 80% of the maximum representable double using IEEE 754 floating-point notation). This maps the possible range of the input function from the range $(-\infty, \infty)$ to the

range $[0, \gamma]$. The benefits of this are twofold. First, the function now has a lowest possible global minimum we can optimize for. Second, the function now has a highest possible value that avoids a bug in the Dlib `find_global_min` function that causes a segmentation fault. We also considered the function $\min(|x|, \gamma)$, but found that the quadratic version converged faster. This leaves us with the final optimization function $l(e) = \min\left((\rho_r(D_{f,t}, e) - \rho_t(D_{f,t})))^2, \gamma\right)$.

4.5.2.3 Development of Worker Task Algorithm

Our next insight was that often the exact match of the compression ratio is not always feasible and is neither desired nor required. It may not always be feasible because for some compressors, for example ZFP's accuracy mode, the function that maps from the error bound to the compression ratio is a step function, such that not all compression ratios are feasible. In addition, it may not always be desired or required because the user might accept a range of compression ratios and prefer finding a match quickly rather than waiting for a more precise match.

Looking again at Figure 4.4, we see a typical relationship between an error bound and the compression ratio. If the user asks for a compression ratio of 15, no error bound would satisfy that request using this compressor. In contrast, FRaZ will return the closest point that it observes to the target; in the case of Figure 4.4 it would report an error bound that results in a compression ratio near 17.5. Depending on the user's global error tolerance, this value near 17.5 may or may not be within the user's acceptable region, meaning it may or may not be a feasible solution.

Another case that the solution may be infeasible is when needed error bound required to meet the objective is above the user's specified upper error bound, U. In this case, FRaZ will report the error bound that resulted in the closest that it observed to the target compression ratio, and the user can run FRaZ again with the default upper bound, which is equal to the maximum allowed level of an error bound by the compressor. If FRaZ identifies a solution in this case, the user can evaluate whether to relax the perhaps overly strict error tolerance to meet the objective or decide that the fidelity of the results is more important and that the bound cannot be relaxed. Alternatively, the user can try a different compressor backend that implements the same error bound.

In fact, determining the exact error bound that produces a specified compression ratio may not be desired or required. The reason is that a large number of iterations may be needed in order to converge to an error bound, and the user would rather trade time for accuracy. Therefore, we implemented a version of Dlib's `find_global_min` that implements a global cutoff parameter

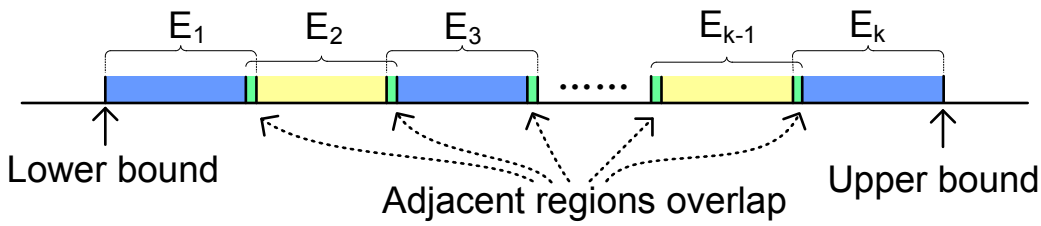


Figure 4.5: Illustration of error bound ranges. We divide the range from lower bound to upper bound into K slightly overlapping regions $(E_1, E_2, \ldots E_K)$. The overlap is a small fixed percentage of the width of the regions (i.e., 10%). Each region (E_1, E_2, ..., E_K) is then passed from the parallel orchestrator to the autotuning optimizer. Note that the ends E_1 and E_K are slightly smaller to preserve the error bound range.

$\epsilon \in [0, 1]$. Specifically, we allow the algorithm to terminate if the result of the optimization function results in a value in the range: $[0, \epsilon^2 \rho_t(D_{f,t})^2]$. This has a substantial impact on the performance on the typical case.

We combine these insights into our worker task algorithm, as shown in Algorithm 1.

Input: target ratio ($\rho_t(D_{f,t})$), acceptable error ϵ, dataset $D_{f,t}$, prediction p, region's lower bound l, region's upper bound u

Output: real compression ratio $\rho_r(D_{f,t}, e)$, recommended error bound setting e

1: **if** $p \neq 0$ **then**
2: $\quad \rho_r(D_{f,t}, e) \leftarrow compress(D_{f,t}, p)$ /*If a prediction was provided, try it first.*/
3: **end if**
4: **if** $(1 - \epsilon) * \rho_t(D_{f,t}) \leq \rho_r(D_{f,t}, e)) \leq (1 + \epsilon) * \rho_t(D_{f,t})$ **then**
5: $\quad$ **return** $\rho_r(D_{f,t}), p$ /*terminate if $\rho_r(D_{f,t}, e)$ meets requirement.*/
6: **end if**
7: $\rho_r(D_{f,t}, e), e \leftarrow train\_with\_cutoff(D_{f,t}, l, u, \rho_t(D_{f,t}), \epsilon)$
8: **return** $\rho_r(D_{f,t}, e), e$

Algorithm 1: Worker Task

4.5.3 Parallelism Scheme

After optimizing the serial performance via the design of the optimization algorithm, we develop a parallel optimization method using Dlib's built-in multithreaded optimization mode. Some compressors (such as SZ and MGARD) do not support being run with different settings in a multithreaded context because of the use of global variables. In this situation, we can only treat each compression as a non-multithreaded task because we are developing a generic framework.

We use multiple processes based on MPI to parallelize the search by error bound range. Figure 4.5 provides an overview of our method. Rather than a serial search over the entire lower to upper bound range, we divide the range into k overlapping regions. We then give each of the k regions to separate MPI processes, and use Algorithm 2 to process them. As the processes complete,

we test whether we have satisfied our objective subject to our global threshold ϵ (line 7–9). If so, we terminate all tasks that have not yet begun to execute (line 10–14). If a particular task finishes and we have not satisfied our objective, we do nothing. If all the tasks finish and we still have not met our objective, we conclude that the requested compression ratio is infeasible (line 18–25).

So why do we overlap the error bound regions? Overlapping the regions avoids extremely-long worst-case search time in the optimization algorithm. Since we terminate early once a solution is found, FRaZ's runtime depends on the region containing the target. Without small overlapping, if the target error-bound coincides a border, its MPI_rank iterates longer lacking stationary-points for quadratic refinement.

Input: target compression ratio $\rho_t(D_{f,t})$, acceptable error ϵ, dataset D_t, max allowed compression error U
Output: real compression ratio $\rho_r(D_{f,t}, e)$, recommended error bound setting e

1: $tasks[N]$
2: $done \leftarrow false$
3: **for** $(i, (l, u)) \in make\_error\_bounds(U)$ **do**
4: $\quad tasks[i] \leftarrow launch\_task(D_t, l, u, \rho_t(D_{f,t}), \epsilon, h)$
5: **end for**
6: **while** $notdone$ **do**
7: $\quad last\_task \leftarrow next\_completed(tasks)$
8: $\quad candiate \leftarrow compression\_ratio(last\_task)$
9: $\quad$ **if** $\rho_t(D_{f,t})(1-\epsilon) \leq candidate \leq \rho_t(D_{f,t})(1+\epsilon)$ **then**
10: $\quad\quad done \leftarrow true$
11: $\quad\quad$ **for** $task \in tasks$ **do**
12: $\quad\quad\quad cancel\_if\_not\_finished(task)$
13: $\quad\quad$ **end for**
14: $\quad$ **end if**
15: $\quad done \leftarrow has\_next(completed)$
16: **end while**
17: $\rho_r(D_{f,t}, e) = \infty$
18: **for** $task \in tasks$ **do**
19: $\quad$ **if** finished(task) **then**
20: $\quad\quad \rho \leftarrow compression\_ratio(task)$
21: $\quad\quad$ **if** $(\rho_r - \rho)^2 < (\rho_t - \rho)^2$ **then**
22: $\quad\quad\quad \rho_r = \rho$
23: $\quad\quad$ **end if**
24: $\quad$ **end if**
25: **end for**
26: **return** $\rho_r(D_{f,t}, e), error\_bound(task)$

Algorithm 2: Training

To limit the effects of waiting on wrong guesses, we constrain the number of iterations to a maximum value. We considered limiting by time instead, but we were unable to find a heuristic that worked well across multiple datasets, fields, and time-steps. This is because the compression

time is a function of the dataset size, the entropy of the data contained within, and properties of each compressor.

Limiting the amount of wasted computational resources is desirable. Since we are dividing on error-bound range, a small number of the searches (typically one) are expected to return successfully if the requested ratio is feasible. Additionally, there seems to be a floor for how many iterations are required to converge for a particular mode of a compressors. Hence, there is limited benefit to splitting into more than a few ranges, and cores could perhaps be more efficiently used for other fields. Preliminary experiments found that 12 tasks per a particular field and time-step dataset offered an ideal tradeoff between efficiency and runtime, and we set it as the default. The user can choose to use more tasks, however.

One can also perform additional optimization of multiple time-step data. Often, subsequent iterations in a large simulation do not differ substantially and have similar compression properties. Therefore we ran the first time-step as before, but then we assumed that the error bound found by the previous iteration was correct for the next full dataset. If our assumption proved correct, we continued on and skipped training. Otherwise, we reran the training and adopted the new trained solution for the next step. We then repeated this process over the remaining datasets. In practice, we retrained only a small percentage of the time. On the hurricane dataset, for example, we retrained only 4 times on the CLOUD field.

We also take advantage of the embarrassingly parallel nature of parallelizing by fields, as shown in Algorithm 3. The results show some additional speedup.

Input: target ratio $\rho_t(D_{f,t})$, ϵ, dataset D, max allowed compression error U
Output: real compression ratio $\rho_r(D_{f,t}, e)$, recommended error bound setting e

1: **for** $D_f \in D$ /*in parallel*/ **do**
2: $\quad p \leftarrow 0$
3: $\quad$ **for** $D_{f,t} \in D_f$ **do**
4: $\quad\quad \rho_r(D_{f,t}, e), e \leftarrow parallel\_error\_bound(D_t, \epsilon, U)$
5: $\quad\quad$ **if** $(1-\epsilon) * \rho_t(D_{f,t}) \leq \rho_r(D_{f,t}, e) \leq (1+\epsilon) * \rho_t(D_{f,t})$ **then**
6: $\quad\quad\quad p \leftarrow e$
7: $\quad\quad$ **end if**
8: $\quad$ **end for**
9: **end for**

Algorithm 3: Parallel by Field

4.6 Performance and Quality Evaluation

In this section, we first describe our experimental setup, including hardware, software, and datasets. We then describe our evaluation metrics and results using five real-world scientific floating-point datasets on Argonne's Bebop supercomputer [54].

4.6.1 Experimental Setup

4.6.1.1 Hardware and Software Used for Evaluation

The hardware and software versions we used on the Bebop supercomputer [54] are given in Table 4.2.

Table 4.2: Hardware and Software Versions Used

Hardware	**Description**		
CPU	36 Core Intel Xeon E5-2695v4		
MEM	128GB DDR4 Ram		
NIC	Intel Omni-Path HFI Silicon 100 Series		
Software	**Description**	**Software**	**Description**
OS	CentOS 7	SZ	2.1.7
CC/CXX	gcc/g++ 8.3.1	ZFP	0.5.5
MPI	OpenMPI 2.1.1	MGARD	0.0.0.2
Dlib	2.28	Singularity	3.0.2

We have packaged our software as a Singularity container for reproducibility.

4.6.1.2 Datasets used for Experiments

In our experiments, we evaluated our designed fixed-ratio lossy compression framework based on all three state-of-the-art compressors described in Section 4.2, using five real-world scientific simulation datasets downloaded from scientific data reduction benchmark [46]. The raw data are all stored in the form of single-precision data type (32-bit floating point). We describe the five application datasets in Table 4.3.

We chose these datasets for a few reasons: First, they offer results over multiple time-steps, which matches well user's practical post-analysis with a certain simulation period. Second, the datasets use floating-point data which are often not served well by traditional lossless compressors. Third, the datasets are commensurate with the use cases of fixed-ratio compression described in Section 4.2.

Table 4.3: Dataset Descriptions

Name	Domain	# Time-steps	Dim.	# Fields	Total size
Hurricane	Meteorology	48	3	13	59 GB
HACC	Cosmology	101	1	6	11 GB
CESM	Climate	62	2	6*	48 GB
Exaalt	Moledular Dyn.	82	1	3	1.1 GB
NYX	Cosmology	8	3	5	35 GB

* A limited number of fields had multi-time step data. Only fields for which multiple time step data were included.

In some cases, we are not able to use all the datasets with all compressors. We run all the experiments for all datasets and compressors when possible. MGARD supports only 2d and 3d data so it is not tested on the HACC and Exaalt datasets. We adopt 6 typical fields for CESM application because other fields exhibit similar results with one of them (CLDHGH CLDLOW, CLOUD, FLDSC, FREQSH, PHIS). We generally noted similar results for each dataset and compressor.

4.6.2 Experimental Results

Over the course of our experiments, we evaluated four properties of FRaZ using the datasets from SDRBench [46]:

1. How close do we get to the target compression ratio when it is feasible?
2. How long does it take to find the target compression ratio or determine that it is infeasible?
3. How does the runtime of the algorithm scale as the number of cores increase?
4. How does FraZ compare with existing fixed-rate methods in terms of rate distortion and visual quality?

4.6.2.1 How close do we get to the target compression ratio?

How close we get to the target compression ratio depends heavily on whether the requested compression ratio is feasible for the underlying compressor used. Figure 4.6 (a) and Figure 4.6 (b) show a bad case and a good case, respectively.

In Figure 4.6 (a), we see an example of where $\rho_t(D_{f,t})$ is infeasible for most time-steps for the CLOUD field. The early time-steps compress within the acceptable range, but by time-step ten the $\rho_t(D_{f,t}) = 15$ is no longer feasible. The reason is that as the time-steps progress, the properties

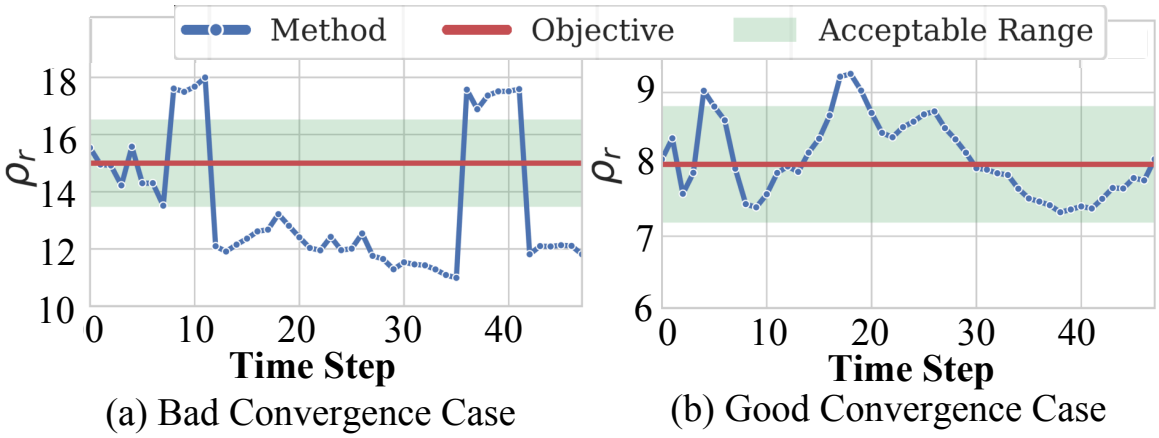


Figure 4.6: Demonstration of two types of convergence cases (Hurricane-CLOUD)

of the dataset change, affecting the ability of compressor to compresses it at this level. As a result, we oscillate between a compression ratio that is larger and a compression ratio that is smaller. However, a larger tolerance (i.e., $\epsilon = .2$) would have allowed even this case to converge for all time-steps.

In Figure 4.6 (b), we see an example of where the algorithm converges on over 90% of the time-steps. In this case, we quickly converge to the acceptable range and are able to often reuse the previous time steps error bound for future iterations. In this particular case, we have to retrain only four times over the course of the simulation on iterations: 0, 8, 15, 29. Thus, the algorithm can quickly process many time-steps.

4.6.2.2 How long does it take to reach the target compression ratio?

When evaluating the algorithm, we wanted to consider how long the algorithm takes to find the target compression ratio. This again depends greatly on whether $\rho_t(D_{f,t})$ is feasible or not. Therefore, we considered a large number of possible $\rho_t(D_{f,t})$'s for different datasets. The results of this search are shown in Figure 4.7. We can see that some compression ratios require far longer total times. figures 4.6 (a) and (b) show a zoomed in view of $\rho_t(D_{f,t}) = 8$ and $\rho_t(D_{f,t}) = 15$. The difference in runtime is explained by the difference in the number of time-steps that converge. In the case shown in Figure 4.6 (a) relatively few time-steps converged because the objective was infeasible with the specified compressor; in the case shown in Figure 4.6 (b) almost all time-steps converged because the objective was feasible. This resulted in about a 10x difference in performance between the two cases.

Why do low target compression ratios have long runtimes? Many of the lossy compressors

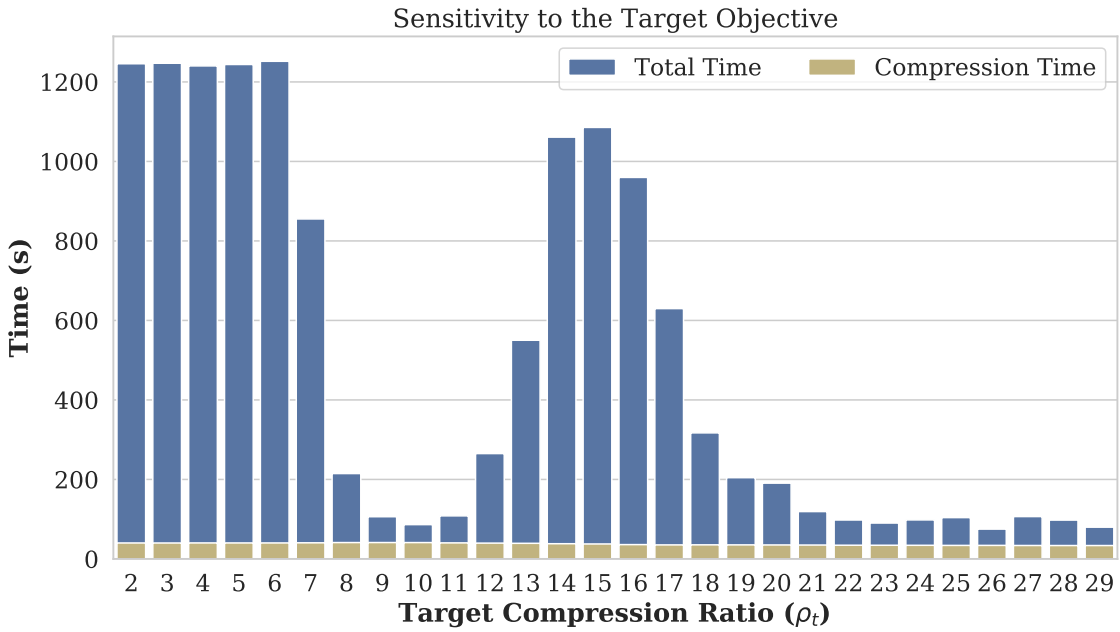


Figure 4.7: Sensitivity of FRaZ to the choice of $\rho_t(D_{f,t})$: This is because not all values of $\rho_t(D_{f,t})$ are elements of the co-domain of the function that relates ρ and e.

have an effective lower bound for the compression ratio. In Figure 4.7, it is about 7.5. This effective lower bound on the compression ratio, means that FRaZ will never meet its objective and spends the remainder of the time searching until it hits its timeout.

How does this change across datasets? In general, the more feasible compression ratios near the target, the better FRaZ preformed. Each dataset had a compressor which was able to more accurately compress and decompress the data.

How does this change between compressors? Generally SZ took less time than ZFP or MGARD even though ZFP may take less time for each compression. This is because ZFP typically had fewer viable compression ratios than SZ due to limitations of ZFP's transform based approach. As a result, FRaZ took more time-steps which took the maximum number of iterations lengthening the total runtime. The difference in runtime between SZ and ZFP for a representative dataset can be seen in Figure 4.8 below.

4.6.2.3 How does the algorithm scale?

To evaluate how well the algorithm scales, we considered the runtime of the algorithm as it scales over multiple cores on ANL Bebop [54].

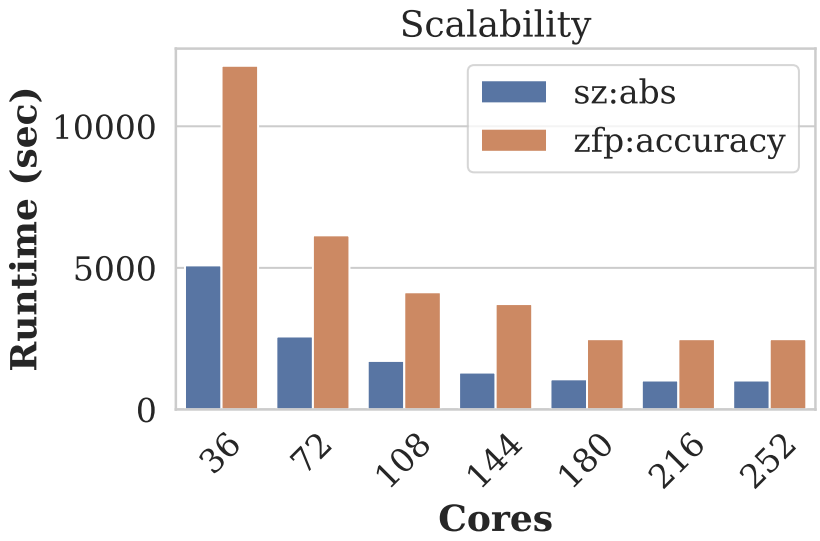
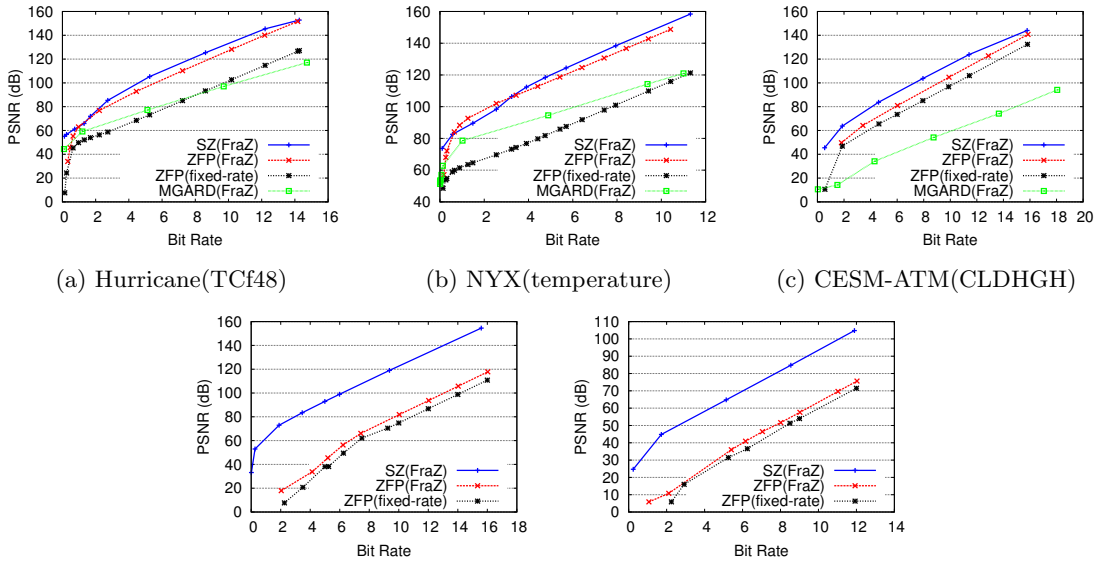


Figure 4.8: Scalability: our solution reaches the optimal performance at 180–216 cores, in that the total time is equal to the longest task's wallclock time at this scale.

Figure 4.8 shows the strong scalability of the algorithm. We see that the algorithm scales by time-step and field levels for the first 180–216 cores with steep decreases in runtime due to parallelism at early levels and then less additional parallelism after that. This is because the runtime of the algorithm is lower bounded by the longest running worker task. All of the datasets we tested has at least one fields that takes substantially longer to compress than others. And the scalability of this algorithm is limited by the longest of these. In the case of the Hurricane dataset using the error-bounded compressor SZ, the QCLOUD field took 1022 seconds to compress while the 75 percentile is less than 500 and the 50 percentile is less than 325.

What accounts for the substantial difference between the scalability of FRaZ using ZFP and SZ? Those familiar with ZFP likely know that it is typically faster than SZ, but this seems to contradict the result in Figure 4.8. This result is explained by considering the individual fields rather than the overall scalability. For the cases in which ZFP finds an error bound that satisfies the target compression ratio, it is much faster. However ZFP often expresses fewer compression ratios for the same error bound range, resulting in more infeasible compression ratios and thus increasing the runtime. ZFP expresses few compression ratios because it uses a flooring function in the minimum exponent calculation used in fixed-accuracy mode.

Fields may take longer for a variety of reasons: (1) the $\rho_t(D_{f,t})$ may not be feasible for one or more of the time-steps, (2) the dataset may have higher entropy resulting in a longer encoding



(a) Hurricane(TCf48) (b) NYX(temperature) (c) CESM-ATM(CLDHGH)

(d) HACC(x,y,z) (e) EXAALT(x,y,z)

Figure 4.9: Rate distortion of lossy compression (MGARD is missing in (d) and (e) because it does not support 1D dataset)

stage for algorithms such as SZ, or (3) the fields may be of different sizes, and larger fields take longer.

4.6.2.4 How does FRaZ compare with the existing fixed-rate compression methods in terms of rate distortion and visual quality?

We present the rate distortion in Figure 4.9, which shows the bit rate (the number of bits used per data point after the compression) versus the data distortion. Peak signal-to-noise ratio (PSNR) is a common indicator to assess the data distortion in the community. PSNR is defined as $20 \cdot log_{10}(\frac{d_{max}-d_{min}}{rmse})$ where $rmse = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(d_i - d'_i)^2}$, and d_{max} and d_{min} refer to the max and min value, respectively. In general, the higher the PSNR, the higher the quality of decompressed data.

In this figure, one can clearly see that ZFP (FRaZ) provides consistently better rate distortion than does ZFP (fixed-rate) across bit-rates (i.e., across compression ratios). Moreover, SZ (FRaZ) exhibits the best rate distortion in most cases, which is consistent with the high compression quality of SZ as presented in our prior work [31]. That being said, FRaZ can maintain the high fidelity of the data very well during the compression, by leveraging the error-bounded lossy compression mode for different compressors.

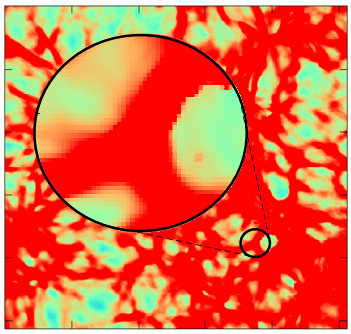
In addition to the rate distortion, we present in Figure 4.10 visualization images based on the same target compression ratio, to show fixed-ratio compression approach preserves visual quality. We wanted to set compression ratio of 100:1, but the closest fesible compression ratio for ZFP is ∼85:1 (see Section 4.6.2). Hence, we set the target compression ratio to be 85:1 for all compressors. Because of the space limit, we present the results only for NYX-temperature field; other fields/applications exhibit similar results. All the results are generated by FRaZ except for the ZFP(fixed-rate). ZFP(FRaZ) exhibits a much higher visual quality than does ZFP(fixed-rate) (see Figure 4.10 (b) vs. Figure 4.10 (c)), because FRaZ tunes the error bound based on fixed-accuracy mode, which has a higher compression quality than ZFP's built-in fixed-rate mode. ZFP(FraZ) exhibits higher PSNR than does ZFP(fixed-rate), which means higher visual quality. We also present the structural similarity index (SSIM)[68] for the slice images shown in the figure. SSIM indicates similarity in luminance, contrast, and structure between two images; the higher SSIM, the better. Our evaluation shows that ZFP(fixed-rate) has lower SSIM than ZFP(FRaZ) – i.e. better quality. From among all the compressors here, MGARD(FRaZ) leads to the lowest visual quality (as well as lowest PSNR and SSIM), because of inferior compression quality of MGARD on this dataset.

4.7 Conclusions and Future Work

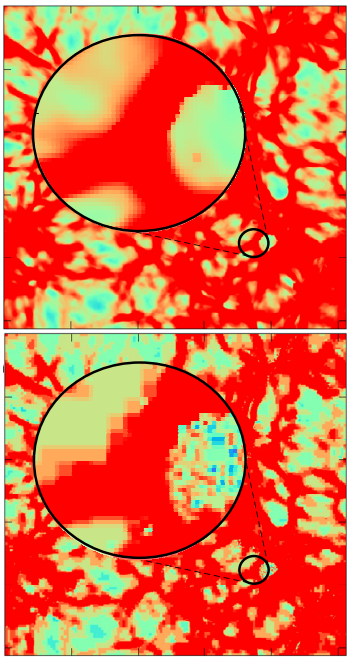
We have presented a functional, parallel, black-box autotuning framework that can produce fixed-ratio error-controlled lossy compression for scientific floating-point HPC datasets. Our work offers improvements over existing fixed-rate methods by better preserving the data quality for equivalent compression ratios. We showed that FRaZ works well for a variety of datasets and compressors. We discovered that FRaZ generally has lower runtime for dataset and compressor combinations that produce large numbers of feasible compression ratios.

A number of areas for potential improvement exist. First, we would like to consider arbitrary user error bounds. By user error bounds, we mean error bounds that correspond with the quality of a scientist's analysis result relative to that on noncompressed data, such as [15] which identifies a particular SSIM in lossy compressed data required for valid results in their field. Second, we would like to develop an online version of this algorithm to provide in situ fixed-ratio compression for simulation and instrument data. Third, we would like to further improve the convergence rate of our algorithm to make it applicable for more use cases. While fixed-ratio lossy compression is

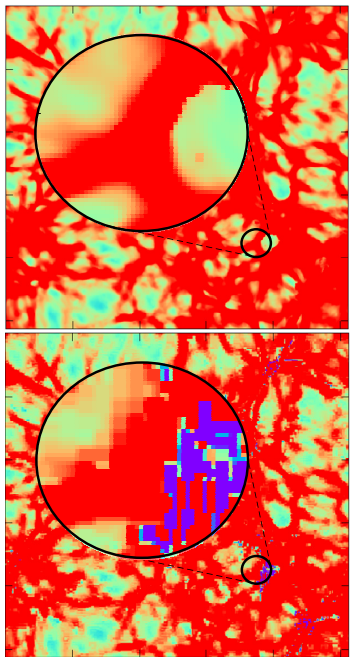
slower than fixed-error compression, it provides an important new lossy compression technique for users of very large scientific floating-point datasets.



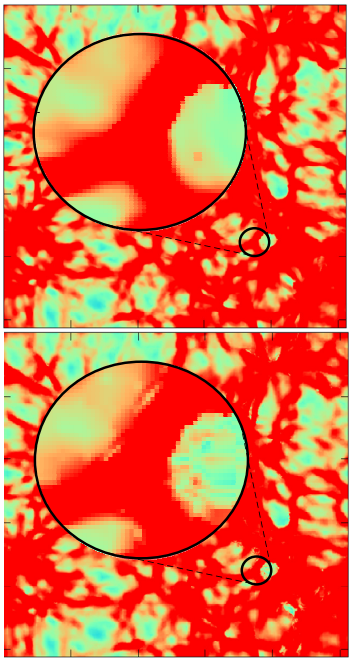
(a) original raw data



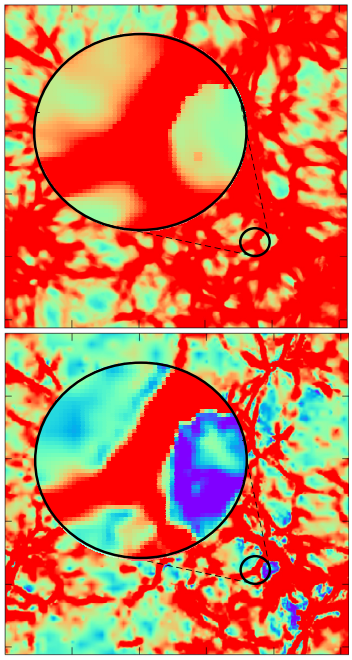
(b) ZFP (FRaZ) (PSNR=76, SSIM=0.997, ACF(error)=0.516), CR=85:1



(c) ZFP (fixed-rate) (PSNR=56, SSIM=0.986, ACF(error)=0.383), CR=85.3:1



(d) SZ (FraZ) (PSNR=80.4, SSIM=0.999, ACF(error)=0.344), CR=86:1



(e) MGARD (FraZ) (PSNR=70, SSIM=0.977, ACF(error)=0.92), CR=85.7:1

Figure 4.10: Visualization of NYX (temperature:slice 256) with CR $\approx$ 85:1

Chapter 5

OptZConfig: Fast Parallel Optimization of Lossy Compression Configuration

Portions of this chapter are in submission [3] to The IEEE Transactions on Parallel and Distributed Systems under the title: OptZConfig: Fast Parallel Optimization of Lossy Compression Configuration

Abstract

Lossless compressors have very low compression ratios that do not meet the needs of today's large-scale scientific applications that produce vast volumes of data. Error-bounded lossy compression (EBLC) is considered a critical technique for the success of scientific research. Although EBLC allows users to set an error bound for the compression, users have been unable to specify the requirements on the compression quality, limiting practical use. Our contributions are: (1) We formulate the problem of configuring EBLC to preserve a user-defined metric as an optimization problem. This allows many classes of new metrics to be preserved, which improves over current practices. (2) We present a framework, OptZConfig, that can adapt to improvements in the search algorithm, compressor, and metrics with minimal changes, enabling future advancements in this area. (3) We

demonstrate the advantages of our approach against the leading methods to configure compressors to preserve specific metrics. Our approach improves compression ratios against a specialized compressor by up to 3×, has a 56× speedup over FRaZ, 1000× speedup over MGARD-QOI post tuning, and 110× speedup over systematic approaches which had not been bounded by compressors before.

5.1 Introduction

the ever-increasing execution scale and problem size of today's scientific applications, volumes of simulation data are produced that cannot be stored and transferred efficiently because of the limited storage space and I/O or network bandwidth. Consequently, scientific data reduction techniques are studied. Since most scientific datasets are composed of floating-point numbers, traditional byte-stream-based lossless compressors (such as Gzip [69] and Zstd [25]) suffer from low compression ratios (generally ∼2× or lower). Although some lossless compressors such as FPC [70] are designed for floating-point data, the compression ratio is still far lower than the user-required level (often 10× or higher) because of the high entropy of the mantissa of floating-point numbers.

Error-bounded lossy compressors (EBLCs) address this issue by allowing users to control the data distortion by specifying point-wise error bounds. By leveraging the correlation of adjacent data points in either space or time, an EBLC can obtain a compression ratio of 100× or even higher while respecting user prescribed point-wise accuracy requirements on the decompressed data, significantly reducing the data storage and data movement burden. Nevertheless, a significant gap still exists between the EBLCs and the user's compression needs. Users must invest significant effort to understand the effectiveness of each lossy compressor on their specific scientific datasets and the impact of data distortion to their post-hoc analysis. To this end, users have to manually perform many tedious trial-and-error tests for the different compressors with their large datasets, and each compressor may involve setting many parameters. For example, SZ [5], [38] offers more than 25 parameters, including different types of error bounds (absolute error bound, relative error bound), number of quantization bins, block size, and lossless compression technique (e.g., Zlib [69] and Zstd [25]).

In addition to specialized compressors, there have been two major developments in this area: FRaZ [2] and MGARD-QOI mode [26]. The former uses numerical optimization techniques to find a fixed compression ratio. The latter uses general mathematical proprieties to bound compression

errors on certain specific and limited kinds of Quantities of Interest (QoIs). However, both share significant shortcomings, namely performance and applicability to a wide array of user defined metrics. In this paper, we propose *OptZConfig* and *Fixed Metric Fidelity Search* (FMFS), which help users select the best-fit lossy compressor with optimized parameter settings in a fully automatic way for more general user defined metrics by building upon the methodology in FRaZ. That is, provided a scientific dataset and the user's specific required compression quality (e.g., some analysis metric and/or target compression ratio), OptZConfig quickly selects the best-fit compressor and determines its optimal parameter setting.

Our major contributions are as follows:

1. We formulate the configuring of EBLCs to preserve a user-defined metrics as an optimization problem improving over current trial-and-error or scientist in-the-loop evaluations.

2. We present a framework, OptZConfig, that adapts to improvements in the search algorithm, compressor, and metrics with minimal changes, enabling future innovations. Prior approaches were strongly tied to at least one of these three.

3. We present and evaluate on real world data sets and metrics a novel parallel algorithm, Fixed Metrics Fidelity Search (FMFS), that improves over the compression ratio of specialized compressors [42] by up to 3× , is 56 × faster than prior black box searching methods [2], over 1000× faster post-tuning than MGARD-QOI post-turning [26], and 110× faster than systematic approaches to bound metrics that are not bound-able by any prior compressor or compressor framework.

5.2 Research Background and Related Work

This section presents the background and highlights the motivation for this research. First, we provide a discussion of several scientific applications with big data issues and the user's requirements. We discuss several use cases that require automatic lossy compression optimization strategies in practice. Then, we describe the state-of-the-art error-bounded lossy compression (EBLC) techniques fundamental to developing the OptZConfig framework.

5.2.1 Big Data Applications and Use Cases

Today's scientific applications are producing too much data to efficiently process or store at runtime. Cosmology simulations of the Hardware/Hybrid Accelerated Cosmology Code (HACC) [11], for instance, may produce 21.2 petabytes of data when simulating 2 trillion particles for 500 time-steps. Summit, one of the most powerful supercomputers, at the Oak Ridge National Laboratory [12] provides only hundreds of terabytes of storage for each user. Hundreds of users share the limited total storage space (250 PB in total).

Materials science can produce raw data with a very high rate (e.g. 250 GB/s with Light Coherent Light Source (LCLS-II) [13]). In order to sustain the data acquisition rate, storing the raw data without compression is impractical. The LCLS-II data system designers calculate that a data compression ratio of 10× or higher is needed to reduce the required storage bandwidth to 25 GB/s.

Each scientific application brings a unique use case for EBLC in the analysis of high-performance computing simulation data or instrument data. For example, cosmology researchers explore galaxy structures formed by particles coalescing into halos, and the bias of halo masses [71], [72] should be limited to within 3% based on the lossy reconstructed particle data.

A number of metrics can be used to understand the impact of compression errors. These metrics measure the quality of specific statistical outcomes, signal distortion (such as peak signal-to-noise ratio (PSNR)), and spatial error. However, not all metrics are supported by all lossy compressors. FPZIP and ZFP do not natively support PSNR or spatial-error-preserving modes requiring workarounds. SZ supports fixed PSNR but suffers from large errors especially for high-compression cases, which is attributed to the assumption of uniform error distribution. This may degrade compression ratios. We refer readers to the survey paper [13] for more use cases of EBLCs. We summarize some key user-analysis metrics and corresponding user-acceptable thresholds in Table 5.1.

Table 5.1: Summary of Important Lossy Compression User Analysis Metrics

Metric	Domain	Threshold	Range
Pearson Correlation (R value)	Climate	$\geq$ .99999 [9]	[-1, 1]
p value for KS Test	Climate	$\geq$ .05 [9]	[0,1]
Spatial Relative Error	Climate	$\leq$ .05, $\delta = 1e^{-4}$ [9]	[0, 1]
Peak Signal-to-Noise Ratio (PSNR)	Various	Various	[0, ∞)

Users want to guide the compression that they perform by metrics used within their communities [9], [15]. Some robust examples come from the climate community where research has gone

into determining thresholds acceptable to the climate community at large. The Pearson correlation coefficient is used to show the strength of a linear relationship between two datasets. When used between uncompressed and decompressed data, it measures how well the value in the uncompressed dataset represents the decompressed dataset. The Kolmogorov Smirnov (KS) test is a nonparametric hypothesis test that tests whether two samples are from the same distribution. The metric of interest is the p-value, which says how likely the observation is given that the hypothesis is true. It is calculated by computing an empirical cumulative distribution function for each sample and finding the largest difference between these functions. The probability of this distance occurring is computed by using methods from [73]. The spatial relative error is the percentage of points that exceed a specified relative error threshold indicating how widely spread a distortion is.

Another important set of metrics comes from the Scientific Data Reduction Benchmarks (SDRBench) [74]. These represent a collection of real-world problems from various domains paired with the metric(s) of interest. All datasets in this paper come from SDRBench.

5.2.2 Error-Bounded Lossy Compression

Here we describe the EBLC techniques used in the leading error-bounded lossy compressors SZ, ZFP, and MGARD.

SZ [5], [20]–[22] is an error-bounded lossy compressor offering multiple error-controlling approaches, including absolute error bound (denoted ϵ_{abs}) [5], value-range relative error bound (denoted ϵ_{rel}) [38], and target PSNR (denoted by ϵ_{psnr}) [22].

SZ adopts a blockwise prediction-based compression model, which involves three key steps: (1) data prediction – each data point is predicted based on its nearby values in space and two major predictors are applied, Lorenzo [5] and linear regression [20]; (2) linear-scale quantization – each data point is converted to an integer by applying a equal-bin-size quantization on the difference between its predicted value and real value; and (3) compression of the generated integer code arrays by a series of custom lossless compression methods including entropy encoding, such as Huffman encoding, and dictionary encoding, such as Zstd [25].

ZFP [6] is another outstanding error-bounded lossy compressor, which is broadly evaluated in many scientific research studies [2], [24], [75]. Similar to SZ, ZFP supports different types of error controls, such as absolute error bound and precision. The precision mode allows users to set an integer number to control the data distortion with an approximately relative error effect. The

higher the precision number is, the lower the data distortion.

Unlike SZ, ZFP adopts a blockwise transform-based compression model, which includes three critical steps: (1) exponent alignment and fixed-point representation, which align the values in each block to a common exponent and performs fixed-point representation conversion; (2) transformation, which applies a near orthogonal transform to each block; and (3) embedded-coding, which orders the transform coefficients and encodes the coefficients one "bit plane" at a time. SZ and ZFP have different design principles, and neither always has the best compression quality on all datasets [2], [24].

MGARD [7], [26], [27] is a state-of-the-art lossy compressor supporting multigrid adaptive reduction of data. The most important principle of MGARD is a hierarchical scheme that offers the flexibility to produce multiple levels of partial decompression such that users reduce the dataset by either minimizing storage with a required data fidelity or minimizing the data distortion with a target compression ratio. MGARD's Quantity of Interest mode bounds some limited kinds of metrics. We discuss it in detail in Section 5.7.3.

5.2.3 Numerical Optimization to Configure EBLC

The relationship between a metric and the compressor configuration can be multidimensional, non-monotonic, and non-convex. Therefore, naïve approaches such as binary search are not sufficient [2]. Rather, we require numerical optimization.

Analytic derivatives provide a closed form description of the relationship between the compressor settings. Also, metrics are difficult to construct and can change frequently depending on the implementation of the compressor, making them ill-suited to this task. Numerical methods that estimate the derivative by computing the slope of nearby points are too slow because of the time required to evaluate each point. That is, regularly computing a slope at each point is time prohibitive because it requires running the compressor and any associated metrics.

Derivative-free methods do not rely on having derivatives available during the search. Thus, they are a good candidate for use in this context since one does not have to wait to compute the derivative. A typical example is the FRaZ algorithm [2], which itself is based on [52], [66], [67]. FRaZ is hard-coded to adjust only one compressor setting using a specific kind of derivative-free optimization. It bounds compression ratios and does not require any call-backs to a user-defined metric. Its parallelism scheme is not thread-safe, however. The use of multi-threading in the search

function or compressor can cause incorrect results or failure. Most importantly, it has no protocol for communicating early termination between iterations, but only between grid cells. Since some sophisticated metrics that users care about could take hours to compute, one grid of the search might complete successfully, but the algorithm may continue to run for hours until each grid finishes its remaining iterations. This makes FRaZ feasible only for metrics that are quick to compute like compression ratios in offline use cases. Our two novel strategies (OptZConfig and FMFS), significantly outperform FraZ by addressing these and other issues, enabling an online use case. We discuss this in greater detail in Section 5.6.

5.2.4 White-Box/Trial-Based Approaches

At least two related works use white-box approaches to autotune compression [42], [53]. In contrast to a specialized compressor, white box methods allow a limited variety of metrics to be bounded on a limited number of compressors that share certain properties which are exploited for performance. They might use internal properties such as the exact methodology of the prediction and quantization scheme or the sampling based on the block size [53] in newer versions of SZ to speed up the search process to maximize bandwidth at a given error bound for a compressor [42]. Notably, these approaches have no means to invoke a user-defined metric. Furthermore, these approaches exploit properties of the compressor and how they are tied to the specific metric(s) they preserve and generally are not transferable to new problems without degrading performance relative to the problem they were designed for or require substantial rewrites if it is possible to adapt the method.

Another attempt was taken by SCIL[30]. SCIL shares properties of both OptZConfig and LibPressio. Like LibPressio, it attempts to abstract across compressors and also has the concept of a meta-compressor. However, it differs in that it has a fixed-function compression pipeline, which limits the chaining of arbitrary meta compressors and some of the more robust configurations supported by LibPressio. It also does not expose the metadata needed for safe multi-threading of compression nor does correctly pass dimensionality information onto the underlying compressor. SCIL also has features like OptZConfig. However, most importantly it only attempts to bound a specific list of compressor-centric measurements such as the relative tolerance or the number of significant digits preserved rather than arbitrary metrics provided by the users. It does so via previous runs which are then converted into an internal decision tree.

These approaches are complemented and extended by the approaches in OptZConfig. For

example, if the users know that they are employing SZ, they can use the search interface provided by OptZConfig to define a custom searcher based on the methods in these papers – allowing a white-box style usage when an algorithm is available. For example, one could adopt the approach by [42] to cache the $20 - 30$ best configurations and try the best ones at runtime, or use the approach by [53] and sample blocks at the compressor block size to choose between compressors.

5.3 Problem Formulation

In this section, we formalize our research problem. The overarching goal is to automatically select the best-fit lossy compressor with a tuned configuration based on user analysis metrics. This is a generalization and extension of past methods [2] that only considered a single objective with a single input parameter and a single output parameter. Constructing this problem in the general case enables entirely new classes of problems (see Section 5.7.4).

Let D be a set of data buffers that involve a set of fields, denoted by F_D, and a set of simulation time-steps, denoted by T_D. We refer to a specific data buffer as $d_{f,t}$ that holds a dataset to compress, where $t \in T_D$ and $f \in F_D$.

5.3.1 Lossy Compression-Based Parameter Space

State-of-the-art error-bounded lossy compressors use multiple configuration parameters (or error settings) to tune the compression quality and performance. We divide a lossy compressor's configuration parameters into two sets: a set of fixed parameters ($\vec{\theta_c}$) and a set of nonfixed parameters ($\vec{c}$). Fixed parameters' values are already specified by users and are not modified during tuning. Nonfixed parameters are to optimize performance or compression quality. For instance, SZ controls the compression quality and performance by tuning an error-bounding type (see Section 5.2.2), error bound value (a specific positive threshold value), and the number of quantization bins. Each of the three parameters is either set to be a fixed parameter as a constraint by the user, or set to be a nonfixed parameter to be optimized whose values are determined based on the feasible settings of corresponding compressors.

For simplicity of description, let U denote the whole set of the nonfixed parameters ($\vec{c}$), and let Ω denote the whole feasible parameter set (including both nonfixed parameters and fixed parameters). Without loss of generality, each $\vec{c}$ in U has a value constraint that is bounded by a

Table 5.2: Key Notations

Notation	Description
D	Set of all uncompressed buffers for all fields and time-steps
D'	Set of all decompressed buffers for all fields and time-steps
T_D	Set of all time-steps for dataset D
F_D	Set of all fields for dataset D
$d_{f,t}$	Buffer for field, f, and time-step, t, in uncompressed form
ϵ	Some threshold; Typically an error bound provided by a compressor
$\vec{c}$	Vector of nonfixed compressor parameters
$\vec{\theta_c}$	Vector of fixed compressor parameters
$\tilde{d_{f,t}}(\vec{c};\vec{\theta_c})$	Decompressed buffer for field, f, and time-step, t
$\vec{\theta_m}$	Vector of fixed parameters of the user-specified metrics function
U	Set of feasible nonfixed compressor parameters
Ω	Set of feasible fixed and nonfixed compressor parameters
$\mathcal{Q}\left(d_{f,t}, \tilde{d_{f,t}}\left(\vec{c};\vec{\theta_c}\right);\vec{\theta_m}\right)$	User-specified data fidelity metric function
$\mathcal{Q}_\tau(d_{f,t};\vec{\theta_m})$	Early termination threshold for user-specified metrics function

lower bound vector $\vec{l}$ and upper bound vector $\vec{u}$ such that $\vec{l_i} \leq \vec{c_i} \leq \vec{u_i}$, where $\vec{l_i}$, $\vec{u_i}$, and $\vec{c_i}$ are the ith element of $\vec{l}$, $\vec{u}$, and $\vec{c}$, respectively. We denote the reconstructed data buffer based on lossy compression by $\tilde{d_{f,t}}(\vec{c},\vec{\theta_c})$ and the corresponding set by D'.

5.3.2 User-Analysis-Based Parameter Space

Performing high-fidelity analysis on the decompressed buffers (D') for a particular time-step t and field f compared with the one with original uncompressed data may involve some fixed parameters, denoted by $\vec{\theta_m}$. We refer to the user-required fidelity comparison metric on the original uncompressed buffer $d_{f,t}$ and its decompressed buffer as $\mathcal{Q}(d_{f,t}, \tilde{d_{f,t}}(\vec{c};\vec{\theta_c});\vec{\theta_m})$. The user expresses a requirement by identifying some threshold for this fidelity comparison metric; we denote this threshold by $\mathcal{Q}_\tau(d_{f,t};\vec{\theta_m})$. We use an example to further explain the definition of $\mathcal{Q}$ and $\mathcal{Q}_\tau$ in the following text. Unlike the prior work [26] which requires $\mathcal{Q}$ be a bounded linear functional, we do not place any constraints on the fidelity requirement of $\mathcal{Q}$ (or $\mathcal{Q}_\tau$). We summarize all the key notations in Table 5.2.

5.3.3 Finding The Optimal Configuration

Based on the compression parameter space under some compressor and the analysis parameter space defined for some application, we formulate the task of finding the optimal configuration as the following optimization problem: Given $D, U, \vec{\theta_c}, \vec{\theta_m}, e$, $\forall\ d_{f,t} \in D$, optimize:

$\max_{\vec{c}\in U} \mathcal{Q}\left(d_{f,t}, \tilde{d_{f,t}}\left(\vec{c};\vec{\theta_c}\right);\vec{\theta_m}\right)$. If a threshold $\mathcal{Q}_\tau\left(d_{f,t};\vec{\theta_m}\right)$ is provided, the search may terminate early if this constraint is met: $\mathcal{Q}\left(d_{f,t}, \tilde{d_{f,t}}\left(\vec{c};\vec{\theta_c}\right);\vec{\theta_m}\right) \geq \mathcal{Q}_\tau\left(d_{f,t};\vec{\theta_m}\right)$

We further illustrate the research problem using the following example based on the target metric of the Pearson's correlation coefficent (R). This is a common measure for compression quality which no previous compressor bounds. Our FMFS helps find the best-fit parameter setting of the lossy compressor efficiently based on the user-defined target metric.

Here is an example using a real problem set to help illustrate the analysis process. Suppose, in our example, the user wants to use SZ's value-range-based error bound mode to compress as much as possible the Hurricane simulation dataset [74], such that R $\geq$ 0.99999. We tune SZ's relative error bound parameter (denoted by ϵ_{rel}) and the number of quantization bins (denoted by M) to get different quality and performance. The Hurricane simulation dataset has 13 fields with 48 time-steps, for a total of $13 \times 48 = 624$ buffers, each being a $d_{f,t}$. In this case, we set the nonfixed parameter vector $\vec{c}$ as $\{\epsilon,\ M\}$), with the first element representing the error bound parameter and the second element representing the number of quantization bins. We construct U by determining a vector of lower bounds and upper bounds. According to the SZ documentation [5], ϵ and M should be in the range of [0,1] and [1,65536], respectively. Thus, we have $\vec{l} = \{0, 1\}$ and $\vec{u} = \{1, 65536\}$.[1] This makes $\forall\vec{c}$ such that $\vec{l_i} \leq \vec{c_i} \leq \vec{u_i}, \vec{c} \in U$.

Without loss of generality, we set all of the remaining 25+ parameters to their defaults, forming the fixed-parameter vector ($\vec{\theta_c}$). With $d_{f,t}$ and $\vec{\theta_c}$, we compute $\tilde{d_{f,t}}(\vec{c};\vec{\theta_c})$ for any choice of $\vec{c}$ that the search specifies by setting the appropriate compressor settings and running the compressor on the buffer $d_{f,t}$. Now, we must define $\vec{\theta_m}$, $\mathcal{Q}$, and $\mathcal{Q}_\tau$. We let $\vec{\theta_m} = \{.99999\}$ corresponding to the R threshold of .99999 described above. In this case, $\mathcal{Q}$ is defined as follows:

$$\mathcal{Q}\left(d_{f,t}, \tilde{d_{f,t}}\left(\vec{c};\vec{\theta_c}\right);\vec{\theta_m}\right) = \begin{cases} \mathcal{CR}(d_{f,t}, \vec{c};\vec{\theta_c}) & \text{if} \quad \mathcal{R}(d_{f,t}, \tilde{d_{f,t}}(\vec{c};\vec{\theta_c})) \geq \vec{\theta_{m0}} \\ 0 & \text{otherwise,} \end{cases} \tag{5.1}$$

where $\mathcal{CR}$ is the compressor's compression ratio on a given buffer and compressor configuration, $\mathcal{R}$ is the Pearson's correlation coefficient between original datasets and decompressed datasets, and $\vec{\theta_{m0}}$ = .99999 in this example. Based on this formula, if we reach the target R value in a compression

[1] If more restrictive bounds are known or desired, specifying those makes OptZConfig's search task more efficient.

case $\{d_{f,t}, \tilde{d_{f,t}}(\vec{c})\}$, then our solution maximizes the compression ratio (CR) for the corresponding data buffer. $\mathcal{Q} = 0$ indicates that our solution skips all the configuration settings that do not meet the user-required R value threshold.

If the user specifies an acceptable value $\mathcal{Q}_\tau$ for all buffers in D, say a compression ratio 20, we terminate the search early if we find a value of $\mathcal{Q}$ that exceeds this value. We further distinguish between successful compression and unsuccessful compression tunings by checking if $\mathcal{Q}$ is 0. Without loss of generality, this formulation applies to the user's postanalysis metric that needs to be minimized (such as mean squared error or autocorrelation of errors).

It is possible that we are unable to find a configuration that satisfies the user's requirements either because insufficient resources were committed to the search or because the user requested an infeasible configuration. In these cases, we return to the user that the search has failed along with the best point found so far allowing them to determine if they wish to commit additional resources, widen their search area, or stop searching.

5.4 Overall System Design

At a high level, we implement OptZConfig as a metacompressor in the larger LibPressio ecosystem[2]. A metacompressor implements all the features of a compressor, allowing it to be used with little change to the user's code, and OptZConfig takes advantages of new compressors as they are developed which may have advantages for specific problems or data-sets [13]. OptZConfig, unlike prior work in [2], is completely embedded within an application to be used online. Figure 5.1 demonstrates where OptZConfig fits within this larger ecosystem and its major subcomponents.

The OptZConfig library consists of three sets of components: the `opt` metacompressor, the `pressio_search` interface, and many implementations of the search interface including FMFS and FRaZ's algorithm. The `opt` metacompressor and the `pressio_search` interface provide services to the search implementations such as: query if both the MPI implementation and the compressor supports threading, a callback to notify and check the early termination status, and a callback to invoke the compressor with a particular configuration and return a set of computed metrics for that configuration. The `opt` metacompressor is responsible for invoking the search implementation (such as FRaZ) which implements the `pressio_search` interface.

[2]LibPressio[76] is a library that provides an abstraction over common lossy and lossless compressors. It supports all of the compressors in this paper and more.

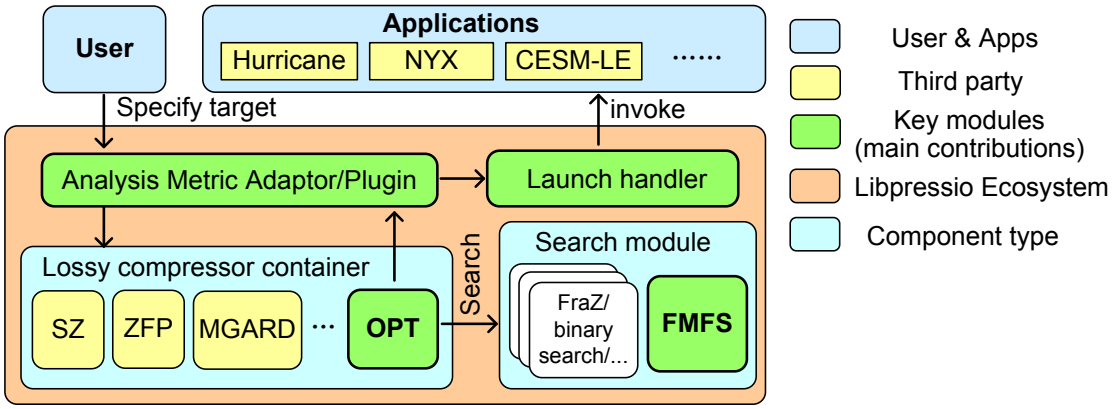


Figure 5.1: LibPressio [76] ecosystem with OptZConfig's metacompressor plugin (OPT), search interface (pressio_search_plugin), and several default search modules including FMFS, FRaZ algorithm [2], and binary search.

A key improvement we introduce in this paper is the Fixed Metric Fidelity Search (FMFS) search algorithm (see Algorithm 4). FMFS builds on the design of FRaZ's algorithm which builds on classical techniques in numerical optimization and parallelization [77], [78]. However, OptZConfig leverages additional information and cancellation callbacks to allow finer grained parallelism and cancellation, significantly improving performance and quality. The blue text represents new parallelism available (see Section 5.6.2). The red text represents significant changes to communication (see Section 5.6.3. The green evaluates a user defined metric. No metric in Table 5.1 or Section 5.7 are possible in FRaZ without OptZConfig and FMFS.

5.5 Computing Metrics

Another significant contribution of this work is a comprehensive framework to adapt user developed metrics for use with minimal effort to be used with any compressor. To this end, OptZConfig provides 47 commonly-used builtin metrics, 5 methods for using user provided metrics, and an interface for providing additional methods without recompiling OptZConfig; allowing users to mix and match for their needs.

We divide the Metrics types into in-core (dataset in RAM) and out-of-core (dataset communicated through the file-system or network). Figure 5.2 shows in-core metrics are direct implementations of the metrics interface. The out-of-core metrics are invoked from the "ExternalMetric" module which handles process management and serialization to/from the formats used by external metrics applications using one of many launch methods and a set of IO plugins.

```
begin search
    mpi-parallel for f ∈ F_D do
        last ← null;
        for t ∈ T_D do
            if last ≠ null then
                q ← Q(d_{f,t}, d̃_{f,t}(last.c⃗, θ⃗_c; θ⃗_m); //thread-parallel
                if good_enough(q) then
                    store(d_{f,t}, last.c⃗, q); continue;
                end
            end
            partitions ← make_partitions(U);
            mpi-parallel for p ∈ partitions; reduce(best : max) do
                s ← derivative_free_search_partition(p, last.c⃗);
                thread-parallel while ¬done(s) do
                    c⃗ ← next_point(s);
                    q ← Q(d_{f,t}, d̃_{f,t}(c⃗, θ⃗_c; θ⃗_m); //thread-parallel
                    best ← max(best, (q, c⃗));
                    if good_enough(q) then
                        global_notify_cancel(); break(2);
                    end
                end
            end
            store(d_{f,t}, best.c⃗, best.q); last ← best;
        end
    end
end
```

Algorithm 4: FMFS

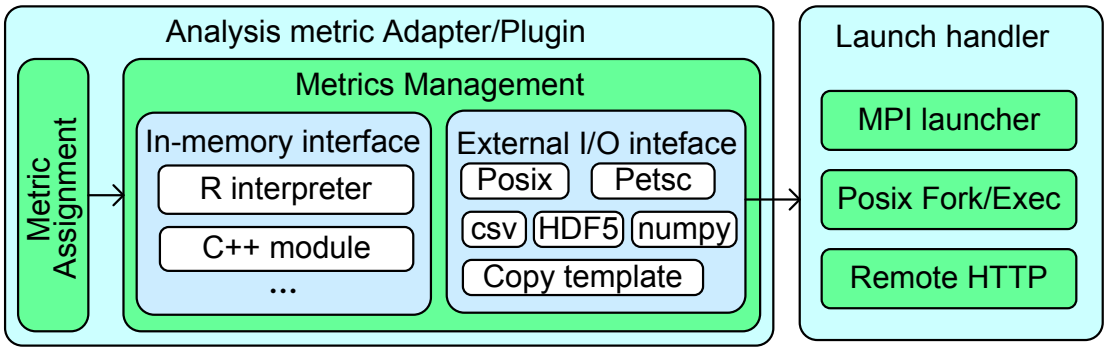


Figure 5.2: Components in metric adapter and launch handler.

In-core techniques are generally faster as they do not require reading/writing from external storage. Moreover, some environments prohibit the spawning of additional processes beyond the initial `mpiexec`. In these cases, in-core metrics modules may be all that are allowed by the system administrator. The primary disadvantage is that applications and/or libraries often require substantial rewrites to support embedding. OptZConfig provides two in-core methods: C++ shared library modules and embedded R scripts.

The primary advantage of out-of-core methods are that they require minimal changes to the user's application and avoid some of the threading limitations required for thread safety. OptZConfig currently provides three out-of-core-methods to maximize performance and meet specific environmental requirements: Fork+Exec, MPI_Comm_spawn, and HTTP endpoint. To use an out-of-core method, the user's application and OptZConfig need to agree on a protocol for communicating the decompressed data. To make this easy for users porting their applications, OptZConfig provides routines to write the data in many common formats including a binary blob, an HDF5 file, a numpy array, a PetSc Matrix, and CSV files. Additionally, we support dynamically loaded user defined serialization and de-serialization methods.

C++ shared libraries are the fundamental implementation of the metrics types. They allow the greatest degree of control over exactly how memory is used, allowing for a zero-copy usage for some metrics. They also provide more possible callbacks into LibPressio than other methods provide, enabling some classes of metrics which are currently not possible using the other methods, for example, timing the compression. They can be used in a multi-threaded context, but users can opt out of this behavior by setting a flag.

The R metrics module embeds R in OptZConfig [3] to efficiently access all of the modules in R. The R Metrics module takes an R script and a list of output variables as input to compute metrics. However, due to limitations in the R implementation, access to the R interpreter in any given process has to be protected by a shared lock which limits scalability.

The MPI_Comm_spawn method uses MPI-2 features to spawn child MPI programs. Some MPI implementations do not allow calling Fork+Exec explicitly and others do not support nested invocations of MPI programs, and because OptZConfig uses MPI, using Fork+Exec can cause undefined behavior. However, because the MPI-2 standard envisions this routine may talk to a batch

[3] a specialized programming language for statistical computing and has visualization and sophisticated error analysis tools not common in other languages

scheduler or other runtime system, some system administrators have disabled it, and some MPI implementations have never implemented it.

The alternative solution on systems where MPI_Comm_Spawn is not allowed is to use the HTTP endpoint module. It avoids the problem of nested MPI by calling out to a distinct process tree that does not use MPI. Thus, the the lifetime of the metrics program is extended beyond the life of the metrics invocation. This enables programs that are written in languages such as Julia or other systems, that have large startup times to be used for metrics.

We choose HTTP and not a more efficient protocol because of its ubiquity and wide support among many programming languages [4]. Additionally, if users want to use a more efficient protocol such as Mochi's Mango, Protocol Buffers, or Apache arrow, a launch plugin could be written and used.

5.6 Optimizating FMFS and OptZConfig

OptZConfig and FMFS make several important improvements over FRaZ, which is both an algorithm and an implementation [2]. First, the FRaZ implementation does not support being used online or embedded within an application as a metacompressor whereas OptZConfig does. Second, OptZConfig supports custom search methods. This allows OptZConfig to compete with advances in white-box-based methods (see Section 5.2.4) and adopt new searching techniques as they become available. Third, OptZConfig supports quality constraints on the objective — such as enforcing a specific PSNR or p-value on the KS test — and multidimensional searches allowing users to adjust multiple parameters of the compressor simultaneously to achieve higher compression ratios and higher quality than what is possible in FRaZ, which allows adjusting only the error bound. Finally, OptZConfig enables composeable distributed-memory and multithreaded parallel search algorithms and compressors with fine-grained resource allocation by allowing searches to delegate a subdomain of the search to other implementations of the `pressio_search` interface.

FMFS utilizes the additional context and features of OptZConfig to account for up to a 60× speedup over the FRaZ algorithm (see Section 5.7.2). We take a novel approach to parallelism for a black-box-based compressor auto tuner by allowing multithreading in the inner search algorithms, parallelism in the compressor implementation, and faster cancellation by moving the cancellation

[4] While this method was intended for HTTP(S), the implementation uses libcurl and supports any protocol supported by libcurl.

check between iterations of the underlying search algorithm rather than after all iterations. Sections 5.6.1, 5.6.2 and 5.6.3 describe the details.

5.6.1 Compose-able Parallelism

OptZConfig provides two types of search algorithms: concrete and metasearch algorithms. Concrete search algorithms implement a specific search algorithm. For example, `fmfs`, `fraz` and `binary_search` implement our new algorithm FMFS, the FRaZ algorithm from [2] and a simple binary search, respectively.

Metasearch algorithms (e.g., `guess_first`, `dist_gridsearch` modules) allow common search functionality to be implemented in a reusable way. The `guess_first` module implements the common functionality that attempts to test a prediction first before spawning an expensive search process. The `dist_gridsearch` takes a search request and spreads it out into a user-defined number of subgrids across the cluster, spawning a search on the search grid cell and enabling search methods that do not normally take advantage of distributed-memory parallelism to take advantage of it.

What makes this truly reusable, however, is the interface requirements for sharing resources. Numerous researchers find that when multiple frameworks attempt to control parallelism, parallelism suffers [79]–[82]. OptZConfig addresses this problem by requiring well-known options for expressing the allocation of parallel resources such as CPU processes, MPI communicators, and (in the future) GPUs and other accelerators. Thus, users can specify where they would like to allocate different scarce parallel resources, giving them fine-grained control over the degree of parallelism.

5.6.2 Safe Multithreading

Not all search algorithms or compressors are thread safe in all use cases. For example, SZ 1.X and 2.X safely uses multiple threads if the compression parameters are identical between the threads. But because SZ has global state parameters that are shared between threads, data races are possible if these are modified while another thread is invoking the compressor. However, checking if the compressor is thread safe alone is insufficient. The library must also check the MPI implementation, and ensure each invocation gets its own compressor handle. To support this, OptZConfig's callback to invoke the compressor clones the output buffer and compressor object explicitly for each thread, thereby implicitly cloning the metrics object held by the compressor. These clones ensure that threads do not clobber a compressed buffer produced by other threads. The remaining objects are

either stack allocated by each thread or are referenced in a read-only manner, preventing a data race.

Give the clones involved, it is important to consider the memory usage of this design. The outer-most loop of FMFS uses MPI parallelization giving each rank a seperate address space. In order to reduce memory usage at this level, a careful use of public, shared, virtual memory mappings ensures that each input is loaded exactly once and only on the rank that uses it. Additionally, the input buffers for each field and time-step that are not currently being processed can be paged out efficiently when they are not in use. In an HPC usage scenario, processes are typically allocated one thread per hardware core, and OptZConfig uses at most one input buffer per thread at any given time. The memory usage grows linearly with respect to the thread count and $O(1/x^2)$ with respect to the per-thread memory overhead. The largest single buffer used in our experiments is 539 MB. Assuming a worse case of a compression ratio of 1 for the compressed and output buffers, and the memory overhead from SZ (3.004 $\times$), ZFP (.6623 $\times$), or MGARD (4.412 $\times$)[5], the maximum memory usage per thread is at worst 4.0 GB per thread with fits well within the per-node memory limit.

5.6.3 Inter-Iteration Early Termination Support

The last major performance improvement is inter-iteration early termination support. Cancellation in OptZConfig is cooperative rather than preemptive. At the beginning of the search process, each of the worker ranks (excluding the master) begins an nonblocking collective broadcast with the master process. If the master wants to terminate, it completes the nonblocking broadcast, and the children test their handle for the broadcast to see whether it completes. If a worker wants to request a termination, it sends a message out of band to the master indicating the request. The master completes the broadcast indicating that a process requests termination.

5.7 Experimental Evaluation

We perform the evaluation as comprehensively as possible in four areas. First, in subsection 5.7.1, we evaluate OptZConfig against specialized compressors which are custom-built to

[5]measured via the peak resident set size for both compression and decompression, m, from `getrusage` for a data of input size i and compressed size c. Then computed as $(m - 2i - c) \div (i)$

Table 5.3: Hardware and Software Details

Component	**Description**	**Component**	**Version**
CPU	Intel Xeon 6148G (40 Cores)	Compiler	GCC 8.3.1
RAM	372 GB	OS	CentOS 8
Interconnect	100 GB/s HDR Infiniband	MPI	OpenMPI 3.1.6
MGARD	v0.1.0	SZ	v2.1.10
ZFP	v0.5.5	LibPressio	0.66.1

preserve a specific metric. In subsection 5.7.1, we show that OptZConfig achieves up to 3 × improvements in compression ratio on some data sets over a state-of-the-art specialized compressor while yielding a tighter bounding on the user's metric. Next, in subsection 5.7.2, we evaluate the performance improvement over the FRaZ algorithm. In subsection 5.7.2, we show how three performance optimizations we make over FRaZ contribute to a 56 × speedup. After that, in subsection 5.7.3, we conduct a comparison against MGARD's Quantity of Interest Mode (MGARD-QOI) which is the only other compressor which attempts to bound a very limited form of user-defined quantities. In subsection 5.7.3, we show a over a 1000× speedup when both MGARD-QOI OptZConfig are tuned. We show our technique reduces tuning time from 23.36 minutes for MGARD-QOI to at most 6 seconds for OptZConfig. We further demonstrate that OptZConfig without tuning is faster than the tuned version of MGARD-QOI 75% of the time and only up to 5× slower in the worst case. Finally, in subsection 5.7.4, we evaluate the runtime of OptZConfig on problems that are not solvable with any current other current compressor. In subsection 5.7.4, we show a speed up over the prior systematic approach from 24 node hours to 13 node minutes – a 110× speedup.

We conduct all of these evaluations on nodes of REDACTED cluster (see Table 5.3). We select these nodes to have a large number of CPUs since at the time of writing MGARD's GPU support is still maturing for all modes and does not include MGARD-QOI mode.

For our analysis we use a number of datasets from SDRBench[74]. These are summarized in Table 5.4. We choose these datasets because they represent regular-grids and have metrics of interests defined by their communities. Regular grids are the type of data structure that MGARD is designed to protect. SZ and ZFP work on other datasets as well, but we focus on regular grids for the purpose of comparison with MGARD. In particular we present only Cloudf48 from Hurricane, Prec from CESM-LE, pressure from miranda, baryon_density from NYX and volume from SCALE-LETKF, without loss of generality, in that other fields are similar based on our observation.

Table 5.4: Datasets

Dataset	Description	total size	buffer size
CESM-LE	Climate Earth Science Model Ensemble	17GB	643MB
Hurricane	Weather data from hurricane Isabelle	48GB	96MB
Miranda	Hydrodynamics turbulence simulation	1.87GB	288MB
NYX	Cosmology simulation	2.7GB	512MB
SCALE-LETKF	Local Ensemble Transform Kalman Filter	4.9GB	539MB

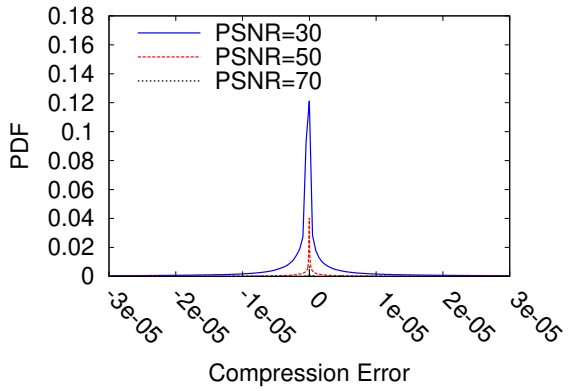
5.7.1 Comparison to Specialized Compressors

To evaluate OptZConfig against specialized compressors with built-in metric support, we use the PSNR metric. Among SZ, ZFP, and MGARD there are only a few specialized natively supported user-defined metrics: PSNR (supported only by SZ; used in image analysis, climate science, and other domains), metrics supported in MGARD's QOI-mode (which are discussed in Section 5.7.3) and the L_∞ norm (supported only by MGARD; used in mathematics and finite element methods); the other modes preserve some point-wise bound which are not widely adopted in the analysis used by users. We choose PSNR because it is commonly used in the literature, and SZ natively supports it. Additionally, both SZ and PSNR are significantly faster to compute presenting a worst case for our approach. We exclude ZFP and MGARD in this test because they do not have a native fixed-PSNR mode.

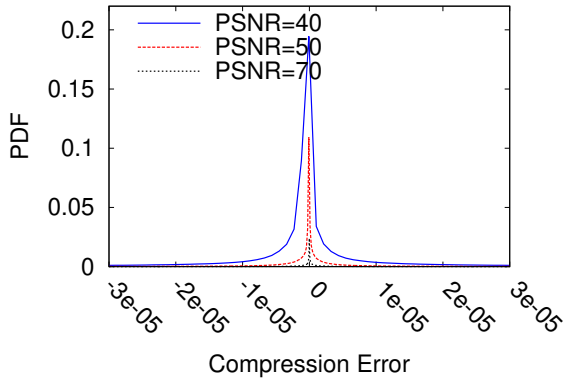
A straightforward idea of fixing PSNR within SZ is relating the absolute error bound to the PSNR, which was proposed by Tao et al. [22]: $20 \cdot log_{10}(value\_range(d_{f,t})/\epsilon_{abs}) + 10 \cdot log_{10}12$

In order to make it hold, there is a fairly strong assumption that the distribution of compression errors induced by the compressor must be uniform. However, much prior work has verified that compressors often have a non-uniform distribution [24]. Moreover, the error distribution can vary significantly with error bound [20], so that inevitably PSNR cannot be controlled accurately based on the above formula [22]. Based on our observation with various application datasets, distribution of errors are non-uniform in most cases, as exemplified in Figure 5.3, which also explains well why SZ's PSNR mode suffers inferior compression ratios, to be shown later.

We compare three different configurations: OptZConfig+sz (uses OptZConfig to set an absolute error bound for SZ); OptZConfig+zfp (uses OptZConfig with ZFP to set the ZFP accuracy parameter); and sz+psnr (specialized version of SZ which bounds PSNR). The goal of these configurations is to show how OptZConfig compares with the state of the art for bounding PSNR – SZ's



(a) SZ



(b) ZFP

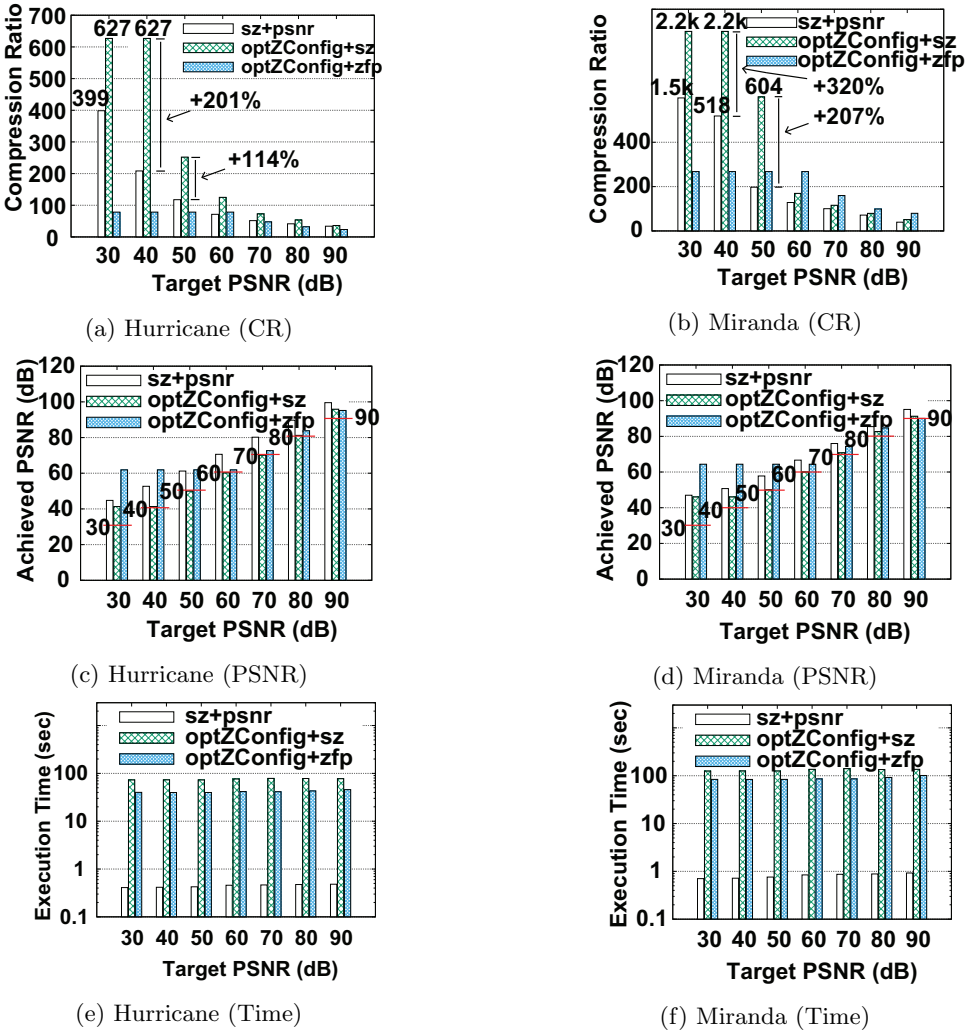
Figure 5.3: Distribution of Compressor Errors of SZ and ZFP for Miranda:pressure with different data distortion levels

PSNR mode.

We run the three configurations with several possible PSNR tolerances of 30 – 90 db as a set of plausible thresholds that a user might desire. The higher the PSNR threshold, the more challenging the problem is for OptZConfig to find a feasible solution because of the smaller number of nonfixed compressor settings that satisfy the bounds. During this evaluation, we find a solution for every configuration, slice, and tolerance desired. Additionally, we allow OptZConfig to terminate early if it finds a solution with a compression ratio such that most PSNR tolerances require searching – greater than 60× – and the quality thresholds are met.

On each run, we record the achieved compression ratio, compression time (including search time for OptZConfig-based methods), and the achieved PSNR. Compression time and compression ratio are two common measures to evaluate compressors. We consider the achieved PSNR to un-

derstand how much overpreservation occurs. For example, if asked to target a PSNR of 40dB, did the compressor get a higher PSNR like 80dB? Overpreservation is undesired because it will cause a much lower compression ratio and higher compression time than expected.



(a) Hurricane (CR)

(b) Miranda (CR)

(c) Hurricane (PSNR)

(d) Miranda (PSNR)

(e) Hurricane (Time)

(f) Miranda (Time)

Figure 5.4: Various methods attempting to maximize the compression ratio while maintaining a specified PSNR tolerance. We achieve up to 3.2 × improvement in compression ratio for the same target

We present our results in Figure 5.4. First, we consider compression ratios in Figure 5.4 (a) and (b). In general, OptZConfig achieves a higher compression ratio than what is possible using the state-of-the-art with the same underlying compressor, because the state-of-the-art SZ often overpreserve the PSNR by imposing a strict global bound rather than allowing individual points to vary given that the overall constraint is met. This overpreservation is observed in Figures 4(c) and 4(d). Additionally, OptZConfig tunes other compressor parameters simultaneously to achieve better

results.

After that, we look at compression time in Figure 5.4 (e) and (f). In the case of SZ, the state-of-the-art method is faster than using OptZConfig without tuning because the OptZConfig tuning process invokes the compressor multiple times. Even given multiple invocations, however, the runtimes are similar between SZ and OptZConfig because we minimize the number of search iterations; OptZConfig often has higher quality or/and compression ratios that may mitigate the differences in runtime in some use cases. We note, however, that this chart includes the tuning time – the time used to find a suitable configuration of the compressor – for methods using OptZConfig. Using the `guess_first` module, the user can specify the parameters as the results of the tuning as a prediction. When this prediction is correct, OptZConfig using SZ is the same as the runtime of SZ+PSNR since no tuning is performed. Additionally, prior work shows that one often can reuse predictions from prior time-steps or similar fields, resulting in even lower overhead [2].

5.7.2 Comparison versus FRaZ Algorithm

Allowing parallel compressors, parallel search techniques, and inter-iteration early termination improves the performance of the search over the techniques used in FRaZ algorithm. We first consider the impacts of each of these optimizations separately.

First, we evaluate using multi-threading in the compressor. At time of writing: ZFP is the compressor which has the greatest support for multi-threaded compression, MGARD has no support for multi-threading compression, and SZ does not have multi-threaded implementations of all of its modes. Therefore, we run ZFP with the same default configuration and error bound except to allow increasing numbers of threads. Since threading performance for ZFP is variable, we run each configuration of threads 30 times using defaults for all other parameters. Figure 5.5 shows the decrease in compression time for increasing numbers of threads in the ZFP compressor on the CLOUDf48 buffer from the Hurricane dataset. Similar results were seen for other datasets using ZFP. Enabling threading in the compressor decreases the time spent evaluating each point during the search progress and improves search performance on average by 2.07× for ZFP by going from 1 to 12 threads.

Next we consider only threading the search, we again use ZFP since it is the only EBLC compressor that supports being called from a multi-threaded context with different configurations. As we increase the threading in the search only from 1 to 4 threads performance improves on average

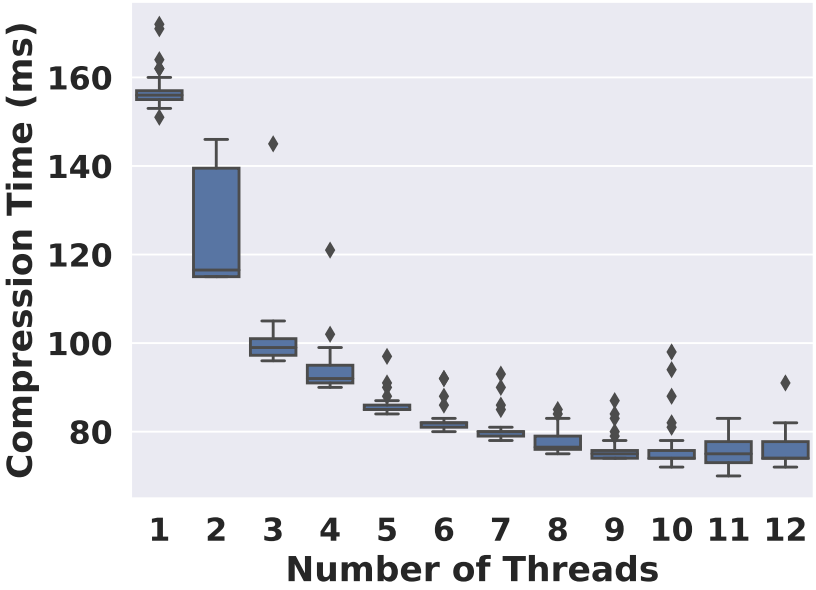


Figure 5.5: Improved search time by 2.07× on average (a speedup over the FRaZ algorithm) when using 12 threads within ZFP

Table 5.5: Improvement in performance by 2.03× on average when going from 1 to 4 search threads. Higher levels of threading have limited effect. This represents a speedup over the FRaZ algorithm.

Target PSNR	Speedup
40	1.88
50	1.97
60	2.24

by 2.03× for different PSNR targets ($40 \rightarrow 1.88\times, 50 \rightarrow 1.97\times, 60 \rightarrow 2.24\times$). We again show results using buffers from the Hurricane dataset, but other results from other datasets are similar. Allowing multithreading of the search itself also has impacts on the execution time—as much as an additional 2.03× improvement in our tests. These improvements are similar to what we observe by using a distributed grid search. However, in the multithreaded implementation, the implementation of the search algorithm shares knowledge of the points as they are searched by other threads each iteration. This allows a better guiding of the search process. Currently, only ZFP supports being called from a multithreaded context, but this situation is expected to change with improvements to SZ and MGARD.

The benefits of allowing inter-iteration early termination are the most dramatic. Figure 5.6 shows the speed up when using FRaZ over OptZConfig with two different compressors on different slices from the Hurricane CLOUDf48 dataset to a user-defined PSNR tolerance. FRaZ does not support user-defined metrics, only compression ratio. For sake of comparison, we took the FRaZ

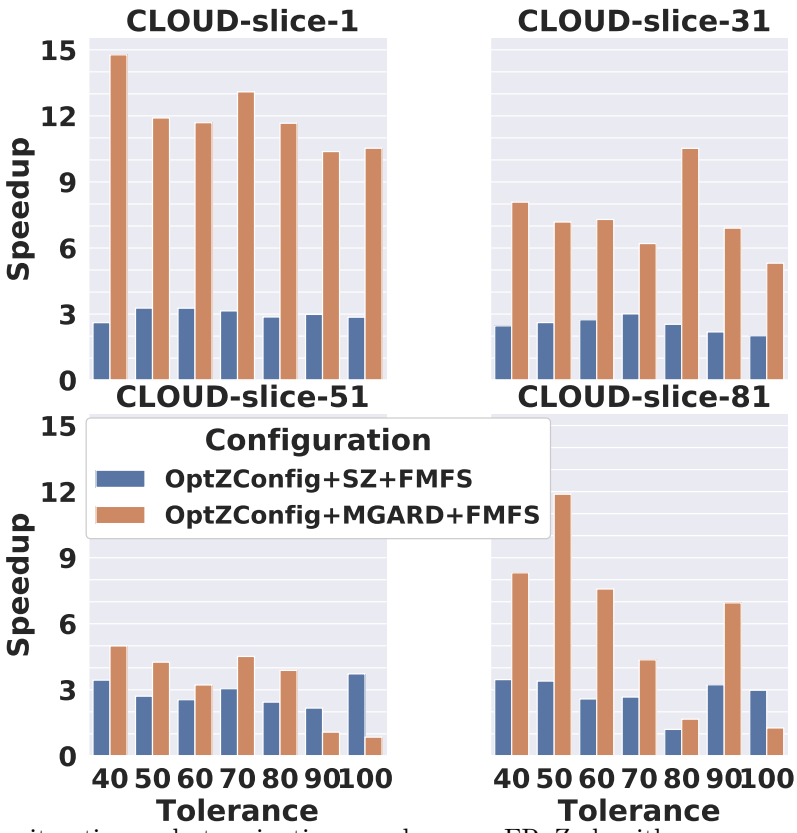


Figure 5.6: Inter-iteration early termination speedup over FRaZ algorithm on a search for a desired PSNR.

algorithm and re-implemented it in OptZConfig to enable us to compare only the improvements from early termination making only the changes required to provide a user-defined metric. In this example, we allow FMFS and FRaZ to terminate early as soon as a feasible solution is found by setting the acceptable PSNR threshold to 0 to represent an upper bound for the effectiveness of this technique when used with PSNR and these compressors. PSNR was chosen because it can be evaluated in linear time, and is easy to implement. We then compute the speedup of the inter-iteration early termination over the intra-iteration early termination from earlier approaches [2]. When used with MGARD, inter-iteration early termination offers improvements as great as 15×. In contrast, when used with SZ, the improvements are closer to 2×. The key explanation of the difference here is the relative fraction of the search time spent invoking the compressor and computing the metrics. In its current implementation, MGARD is nearly 1000× slower than SZ. However, this time would effectively be increased with additional metrics. These metrics from the perspective of the search code are "part of" the compression time, meaning the more metrics or the more complex the metrics being computed, the more SZ would also benefit from inter-iteration early termination.

These three FMFS improvements affect different aspects of the search process, and could be used in conjunction if supported by the compressor and sufficient hardware is available. We ran an experiment that used the ZFP compressor with each of the various speedups enabled in order to find a configuration with a PSNR greater than 105 for the SCALE-LETKF V data buffer on a single node. SCALE-LETKF has the largest buffers of the datasets we consider. We found that a configuration of 3 MPI processes (2 workers and 1 leader), where the workers spawned 5 search threads which in turn spawned 14 compressor threads [6] maximized performance on a single node [7]. In this configuration, we measured a 56.03 × speedup over FRaZ. 83% of the time is spent in tuning the search, the remaining 17% corresponds to a single compress, decompress, compute metrics cycle to compute the final results. A single cycle of compression, decompression, and computing metrics took 780ms (17%), 900ms (19.7%), and 2900ms (63.3%) on average respectively. In total, only 6 invocations were used vs the 101 for FRaZ.

5.7.3 Comparison Versus MGARD-QOI

MGARD's quantity of interest mode (MGARD-QOI) introduces an analytic approach that relies on mathematical properties of supported metrics being computed to solve for an error bound that is mathematically guaranteed to preserve the metrics [8]. It works by computing a scaling factor called the "norm of the quantity of interest" and properties of metrics it supports to bound the error in the metric in terms of the L_s norm chosen by the user.

Specifically, MGARD-QOI requires that the metric be a bounded linear functional computed on a regular grid. The term bounded linear functional implies that the distributive property hold for the metric. An example of a bounded linear functional is the arithmetic mean. If one scales the dataset or add two datasets together, the mean is the same regardless of the order of these operations. The spatial error metric is an example of a metric that is not a bounded linear functional because it allows unbounded error on a point by counting the percentage of points that exceed a threshold. In fact, all of the other metrics mentioned in this paper are not bounded linear functionals. The term regular grid applies to many HPC applications where a domain is discretized into equal-sized

[6] Our implementation requires that the thread count be a multiple of the data dimension in order to reduce copies

[7] Oversubscription in this case overlaps serial and highly parallel portions of ZFP's compression and decompression code from different thread teams maximizing performance

[8] What we call metrics in this paper are referred to as quantities of interest in [26]. The reason is that in the mathematics community the term metrics has a strict mathematical definition. We choose to use it in the broader computer science usage here.

and equally-spaced distinct chunks to compute the effects inside the domain. While MGARD-QOI places restrictions on the metrics and problems supported, these restrictions allow the quantities of interest for many scientific codes. The complete proof of how this works is in their paper [26]. In contrast, the approach offered by OptZConfig supports non-regular grids inputs, and does not require the metric to be a bounded linear functional.

We do not consider the arithmetic mean of the entire dataset because any compressor that bounds the maximum error on a dataset bounds the mean by the same error bound. Consider the mean of the points $x_i \in d_{f,t}$. If the compressor bounds the absolute error, the corresponding decompressed points are $\tilde{x}_i \in \tilde{d_{f,t}}\left(\vec{c};\vec{\theta_c}\right)$ and by definition $\forall x_i - \epsilon_{abs} \leq \tilde{x}_i \leq x_i + \epsilon_{abs}$ The error in the mean is maximized when $\sum_i (x_i + \epsilon_{abs})/\|d_{f,t}\|$. Since we sum $\|d_{f,t}\|$ ϵ_{abs} and divide by $\|d_{f,t}\|$, the maximum error is ϵ_{abs}. The minimum follows similarly, leading to a bound on the mean of $\pm\epsilon_{abs}$

The *weighted* mean is appealing for evaluation because it is a bounded linear functional but less trivial to bound. This is because some entries may have a weighting near zero [9]. Therefore, an implementation that uses the max of absolute error bound times its weight for a cell grossly over-preserves information if that entry never has much error from compression.

We attempted a comparison between MGARD-QOI mode and OptZConfig using the weighted mean metric using a full-sized buffer; however this was performance prohibitive. As written, however, the MGRAD-QOI calculation takes $O(\|d_{f,t}\|)$ evaluations of the metric function. The weighted mean itself takes $O(\|d_{f,t}\|)$ time, making the time to find the norm of the quantity of interest $O(\|d_{f,t}\|^2)$. Even if parallelized (current implementation is serial), this requires $O(\|d_{f,t}\|^2/p)$ time where p is the number of processors. If run on a full-size problem, finding the norm of the quantity of interest is expected to take between 17 and 19 days. We confirm this analysis by timing the first 250,000 basis values (1% of the basis values of the full dataset) on a full sized problem, which takes nearly 4.5 hours.

Instead we focus on the four 3% slices of the input data. we present results from only four $3 \times 500 \times 500$ slices from the 48th time-step of the CLOUDf48 field in the Hurricane dataset from SDRBench[74]. Results from other datasets and their full sized counterparts are similar. This approach is consistent with the slice-by-slice analysis used in the Climate community for CESM [15]. We choose the four slices to represent different problem difficulties: slice-1 is sparse and therefore easy to compress to a specified tolerance; slice-81 is a little bit harder; slice-31 is harder still; and

[9] a weighting near zero is useful when you want to ignore "ghost cells" or the edges of detectors for analysis

slice-51 has the most visual features, making it the hardest to compress. Empirically, the time required to compute the norm of the quantity of interest on a 3% sample MGARD-QOI mode takes on average 23.36 minutes ± 0.04. This is scaled down from the full computation in two ways. First, in this problem, we are computing only the first 3% of the basis values. Second, in this problem, each basis value takes 3% of the time that it takes on the full problem.

For fair comparison, this expensive norm-finding operation does not have be performed with every evaluation of the compressor. The norm can be reused as long as the dimensions of the input data remain constant and the metric does not change. Thus, for some quantities of interest that are not dependent on the original data, one does not have to even recompute the norm of the quantity of interest even if the values of the input data are changed. Additionally, quantities of interest with this property converge to a constant times the number of bases computed as the number of bases increases. Thus, users can often bootstrap their problem with a smaller version of the problem for faster computing of the norm. However, because the weighting changes with the size of the matrix, this kind of bootstrapping is not viable in this case.

There are two possible fair comparisons between MGARD-QOI and OptZConfig for the metrics that MGARD-QOI supports: (1) MGARD-QOI with-tuning vs OptZConfig with-tuning, and (2) MGARD-QOI with-tuning vs OptZConfig without-tuning. Case (1) is most appropriate when there is the potential for substantial variation between the compressor configurations required to preserve a user error metric – such as a turbulent simulations or shock codes. Case (2) is most appropriate when the configuration is likely to be similar between time-steps for fields as prior work has shown is true for many HPC codes [76].

To evaluate the differences between these two cases, we used MGARD-QOI and OptZ-Config with SZ in absolute error bound mode to bound the weighted average for each of the different slices listed above. We attempted to bound the absolute error in the weighted average to $\pm 10^{-3}, 10^{-4}$, and 10^{-6} with both compressors and measured the tuning time for OptZConfig, with-tuning compression time, and compression ratio. We attempted to include OptZConfig+MGARD to have a OptZConfig+MGARD vs MGARD-QOI evaluation as well, but were unable to do so due to implementation flaws in MGARD. OptZConfig+ZFP results were similar to the OptZConfig+SZ results.

Figure 5.7 summarizes the results of this experiment. In the figure, the black line represents the time that MGARD-QOI took post tuning and was identical for all tolerances and files. The

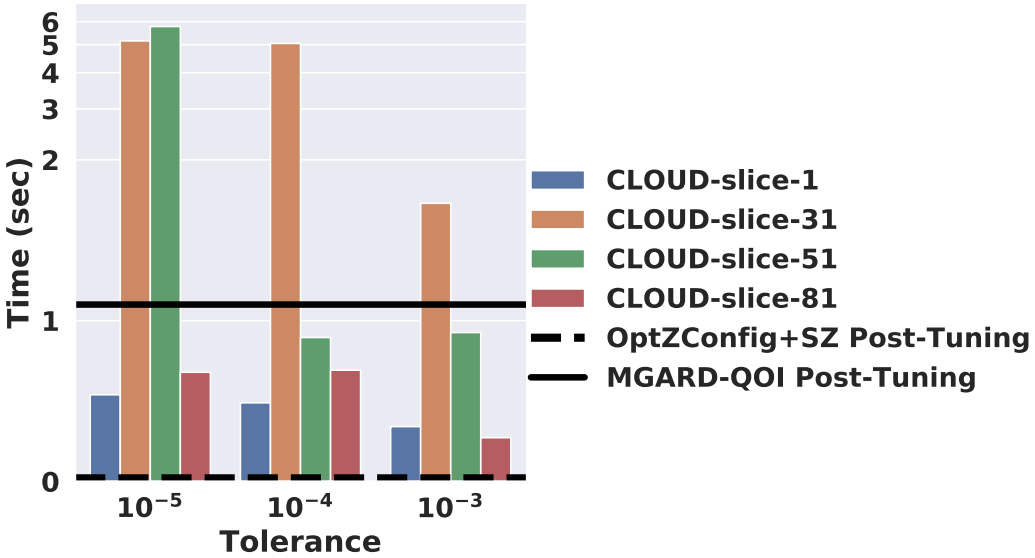


Figure 5.7: OptZConfig vs MGARD-QOI for weighted mean: OptZConfig is 83% faster 75% of the time without tuning OptZConfig; with tuning OptZConfig+SZ is over 1000× faster in all cases.

dashed line represents the post-tuning for OptZConfig and was the same for all tolerances and files. The bars represent the time for compression and tuning using OptZConfig. We are able to tune SZ compression faster than MGARD compresses the same buffer using QOI mode when the QOI norm is already found in 75% of the cases we test. In those cases where OptZConfig is faster, the average time taken by OptZConfig using SZ is 0.6 seconds compared to the 1.1 taken by MGARD-QOI mode. In the other 25% of cases, OptZConfig was $2 - 5\times$ slower than MGARD with its tuning already complete. After OptZConfigs' tuning was preformed, OptZConfig with SZ was over 1,000 times faster than MGARD-QOI with tuning in all cases.

5.7.4 Performance on Non-trivial Metrics

To provide more evidence of OptZConfig's effectiveness on previously un-automated problems, we consider a real-world set of constraints from the climate community that cannot be bounded by any other currently available compressor including MGARD-QOI and FRaZ. Section 5.2 states metrics and bounds for PSNR, the p-value of the KS test, the spatial error percentage, and the R value for the raw data using the thresholds from Table 5.1. We add PSNR to this evaluation as it is discussed in [15]. However, the threshold we use is arbitrarily determined since a bound for PSNR was not identified. Figure 5.8 shows that PSNR never binds over the search space.

For this evaluation, we use a buffer from the CESM application. Table 5.1 shows the

Table 5.6: Runtime for systematic sampling vs OptZConfig

method	**samples (1 node)**	**runtime**	**CR found**
systematic	10	17s	no solution
systematic	100	33s	no solution
systematic	1000	3m45s	no solution
systematic	10000	35m40s	no solution
systematic	100000	6h (4 nodes)	7.264
OptZConfig	n/a	13m	7.3046

thresholds we use. For this evaluation, we use SZ in its absolute error bound mode and consider one parameter, the error bound, for the sake of tuning time. OptZConfig finds a solution that meets all of the constraints with a compression ratio of 7.3046× in 13.1642 minutes using 40 cores of one node. Of this time, 95% is spent evaluating the various metrics on the decompressed data. Figure 5.8 shows a graphical summary of that evaluation.

Figure 5.8 shows that over the search range, the KS-test p-value was the only binding metric. Most of the other values were not even close to their allowed tolerance. The R value is not able to even detect loss in this error bound range. The spatial error percentage is able to detect at most 0.0017% spatial error at its most extreme value. Over the search range, the spatial error percentage, KS Test p-value, and the PSNR are not monotonic. This nonmonotonic behavior defeats more naïve approaches such as binary search [2].

Because binary search-like approaches are not applicable to this problem, users would have to resort to an approach similar to OptZConfig or some kind of systematic evaluation. For a performance comparison, we conducted a systematic evaluation of various numbers of points between the error bounds we provide to OptZConfig 10^{-18} and 10^{-12}. For the systematic evaluation, we divide the state space into evenly sized bins and evaluate the compressor on the midpoint of each bin, and record all points that meet the constraints outlined in this section. The results of this process are in Table 5.6. With even a moderate number of points, the systematic approach was not able to identify a valid solution. This is because a wide range of allowed error bounds result in infeasible requests given the user's constraints. If the user had placed a time limit on the search sorted than the time required, our approach would have failed and returned the result which satisfied the most constraints. The only systematic process which found a solution evaluated 100,000 points and took a little over 6 hours running in parallel on 4 nodes compared to the 13 minutes for OptZConfig on one node. The best result from the systematic search has a compression ratio of 7.264×, which is

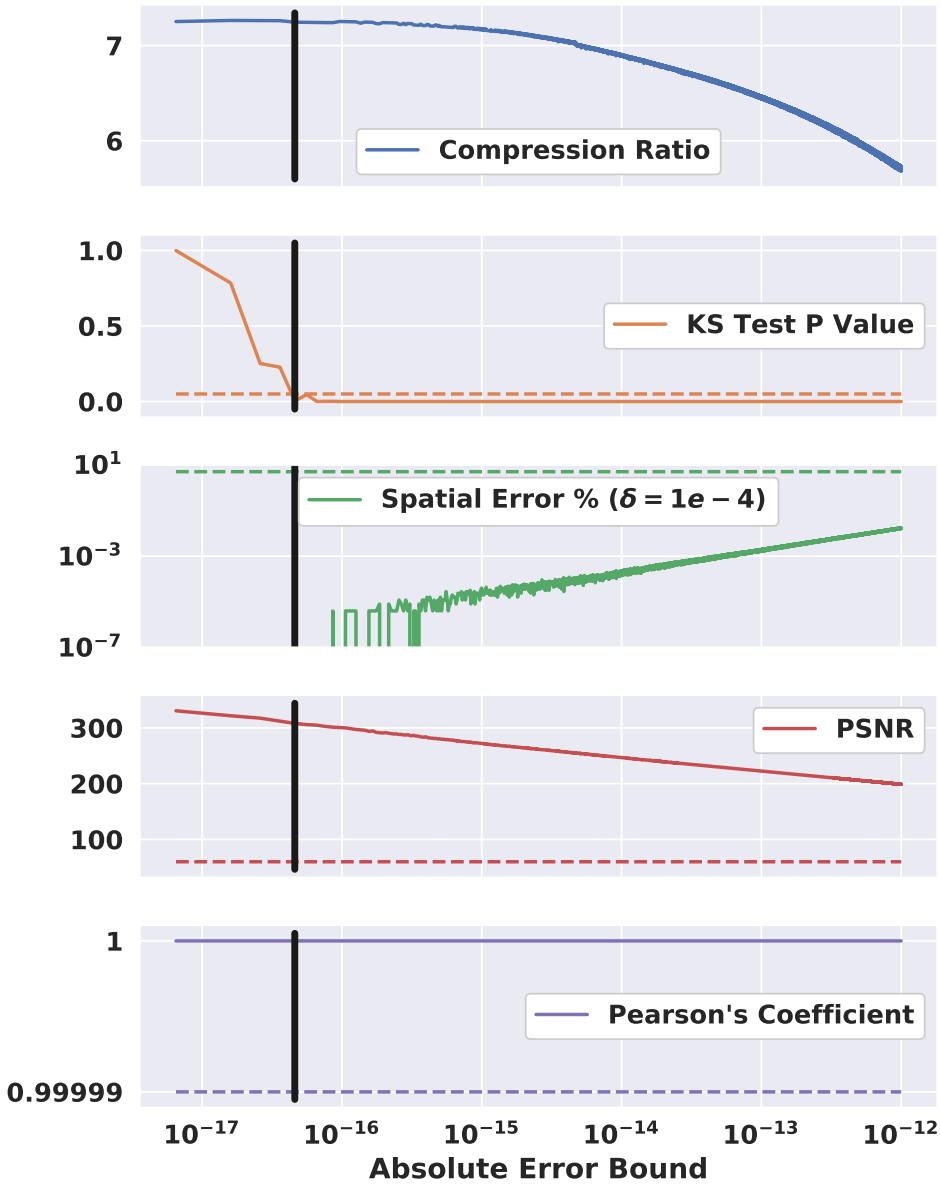


Figure 5.8: Evaluation metrics at 100,000 error bounds between 10^{-18} and 10^{-12}. The PSNR, KS-test p-value, and R value must be above their threshold in order to satisfy the constraints. The spatial error % must be below its threshold. All points to the left of the black line satisfy the constraints.

about .5% worse than using OptZConfig's solution taking 27× more time on 4× as much hardware.

5.8 Conclusions

Techniques such as OptZConfig and FMFS have great promise in helping applications cope with moving and storing ever-increasing volumes of data. It offers a higher-performance method of setting error bounds to preserve user metrics than do prior methods such as FRaZ or MGARD-QOI. OptZConfig also offers new capabilities to bound arbitrary user metrics and provide fine-grained control over the search. We demonstrate that OptZConfig can get up to a 3× improvement in compression ratio for equivalent PSNR requirements as SZ's specialized mode. We further show up to a 56× speed up over FRaZ when using compressors that support being used from a threaded context. OptZConfig further offers a 233× improvement over MGARD-QOI tuning time, and are 1000× faster than MGARD-QOI post-tuning. Lastly, we demonstrate our method on 3 metrics from the climate community achieving a 110× performance improvement over the systematic approach used previously for these metrics.

Chapter 6

Understanding The Effectiveness of Applying Lossy Compression in Machine/Deep Learning Applications

6.1 Introduction

In recent years, there have been increasing focus on the use of machine learning (ML) and deep learning (DL) techniques in high performance computing (HPC) applications [83]–[86]. Many domains including cancer research [87], molecular dynamics [83], and even understanding IO patterns to accelerate super computing [88] take advantage of these techniques. With these techniques, comes a desire for processing an ever increasing volume of data for training and validation to improve the ability to store, transport, and process the data efficiently. The Candle project – which does cancer research – alone anticipates more than *1PB* of data required for training *per experiment*, for just *one class* of problem they consider [87]. Today's advanced instruments such as Linac Coherent Light Source (LCLS)-II [89] and Advanced Photon Source (APS) [90] may produce the X-ray imaging data at a very high acquisition rate (250GB/s [13]).

With the fast growth of training and validation data, it is quite difficult to store or transfer such large volumes of data, especially for the use-cases that require sharing data on wide area network (e.g., using Globus [91]) so other researchers can perform experiments on shared data-sets for comparison or require transferring data from instruments/devices to data centers or supercomputers even within the same site. In addition to the examples (Candle and X-ray imaging data) mentioned above, more typical examples include streaming input data from some instrument or sensor such as structural health monitoring [92] or in traffic safety applications [93]. As these applications are posed, data needs to be transferred over a rural cellular network back to a HPC facility for real-time or near real-time processing. However for these applications, streaming the data becomes challenging due to limitations in the available bandwidth. Moreover, it is also challenging to share data between scientific institutes/supercomputers even on a highly-optimized data transferring platform such as Globus [91]. According to [94], one data transfer node (DTN) on wide area network (WAN) has only about 1GB/s bandwidth and one user generally uses only one DTN, which means transferring petabytes of data between two different sites may take several weeks. Another critical issue is that scientists may experience performance challenges related to the IO involved with training on the input data. IO operations for large volume of data could be very slow – especially when data cannot fit into main memory. The work by [10] has shown that even with advanced IO subsystems, most applications experience *peak* IO performance less than that of a handful of USB mass storage media – less than 0.1% of the peak storage bandwidth capacity. Last but not the least, limitations on storage space is a serious concern for scientists. In many cases, there are huge costs for maintaining a large amount of data in storage: just for the hard drives alone storing the data for one experiment would cost over $20,000 not including the power, cooling, equipment, and staff to maintain the drives over the their lifetime [95].

The challenges of storing/moving massive data for scientific applications have led researchers to consider using efficient lossy compression methods to significantly reduce the data volumes. Compared with lossless compression that achieves only modest compression ratios over numerical datasets for many scientific applications [13], lossy compression techniques [5], [6], [13] yield larger compression ratios and have saw rapid development in recent years.

For lossy compression, the most critical concern for users is what is the impact of the information loss that occurs in lossy compression on the applications. Without a good understanding of such impact, it is very difficult to construct or select the most efficient/qualified lossy compressor

with a specific required fidelity on the quality metrics for applications. Therefore, an efficient methodology to assess the impacts of lossy compression on the quality metrics for ML/DL scientific applications is urgently needed.

In this paper, we present a methodology for assessing the impacts of lossy compression on ML/DL and apply it to several different problems from different domains to identify some best practices for applying training and validation data for ML/DL. We answer the following key research questions:

- Is quality best preserved by compressing the data independently by row, by column, as collectively as a matrix?
- What compression principles are most effective (and why) at preserving quality for each compressed size?
- What indicators can be used to determine whether a problem is "well conditioned" and robust to a particular form of lossy compression?

In order to answer the above critical questions, we need to resolve a series of key challenges, as listed below. (1) This work requires to run several ML/DL-based data-analysis applications designed for different use-cases (image analysis, scientific research, etc.) at different scales on supercomputers. (2) This work requires an in-depth understanding of different types of compressors that could be used on the numerical datasets, and also combining the various compressors with he ML/DL applications to be studied. (3) This work requires an in-depth understanding of the critical quality metrics regarding both compression and ML/DL applications, in order to design a comprehensive, efficient, fair assessment framework for our investigation work.

The remainder of the paper is organized as follows: In Section 6.2, we highlight some key background in data reduction techniques. Following the related work, we discuss in Section 6.3 our overview of our experimental environment. In Section 6.4, we describe how we design the measure system, including how to determine the interesting points for an ML/DL application and how to measure inconsistent effect and threshold effects. In Section 6.5, we present two classes of results that show the breadth and scalability of our approach in Section . In Section 6.6 we discuss related work on how lossy compression is currently used in ML/DL. Finally, in Section 6.7, we present our conclusions and future work.

6.2 Data Reduction Techniques Studied

In order to understand the impact of lossy compression to ML/DL analysis results, it is necessary to have an in-depth understanding of the various data reduction techniques' design principles as well as their pros and cons. To this end, we classify and describe the data reduction techniques carefully in this section. All the techniques mentioned here will be studied and compared in our following comprehensive analysis.

In Machine/Deep Learning Applications (such as the CANDLE project [87]), data is often represented as tables of information where each row represents some observation of the phenomena in question and each column represents quantified proprieties of each observation. This model generalizes to image recognition where each image represents a row and each pixel of information corresponds to a particular column in a 2D representation.

There are three major types of techniques used for data reduction: dimensionality reduction which reduces the number of features, numerosity reduction which reduces the number of observations, and data compression which neither reduces the number of observations or the number of features but instead reduces the space that the existing features and observations take in memory or storage. We discuss numerosity and dimensionality reduction in Section 6.2.1 and data compression in Section 6.2.2.

6.2.1 Dimensionality Reduction and Numerosity Reduction

One class of data reduction methods is dimensionality reduction techniques such as Principle Component Analysis. These methods generally operate by projecting the data into a lower dimensional subspace with fewer features. We view these techniques as orthogonal to the techniques we explore in this paper because they can be used in conjunction with the later techniques we discuss to remove one or more features from the data.

Another class of data reduction methods reduce the number of observations in the data in either the spatial or temporal dimension. For example, in order to control the data volume, many scientific applications or use-cases output the simulation data every K time steps periodically [96]–[98] or just store sampled data and reconstruct the full data by interpolation methods or compressed sensing technology [99]. In our study, we use 3 sampling methods: with replacement, without replacement, and naive sampling. Sampling with or without replacement are very commonly used to

reduce data volumes in ML/DL applications. The user selects a number of records at random either the same record to retrieved multiple times (with replacement) or only once (without replacement) during the sampling. Naive sampling is a degenerate case of sampling that is still commonly used in some domains to accounts for large volumes of streaming data. In the naive method, the user selects records systematically using some arbitrary method such as picking every k^{th} record or the the first k records. Often, systematic sampling is used when the event of some limitation of the system such as having too many readings to fit in memory or too many records to process before the next arrive. Since sampling is probably the most commonly used lossy compression technique used in data mining, it is an important benchmark for comparison.

6.2.2 State-of-the-art Data Compression Techniques

Data compression is generally divided into two categories: lossless compression and lossy compression. Lossy compression methods can now be further divided into traditional and error bounded. The remainder of this section is divided into three subsections: lossless methods, lossy methods, and error bounded methods.

6.2.2.1 Lossless Compression Methods

Lossless compression is the most generic data compression method, because it perfectly preserves the information through the compression and decompression process for all possible inputs. Formally, for a compression function, $C(x)$, and a corresponding decompression function, $D(x)$, a compression scheme is lossless if and only if $\forall x, D(C(x)) = x$. Typical examples of lossless compression include DEFLATE [100] used by the zip utility, LZ77 used by GNU's gzip[69]. Many lossless compressors such as Zlib [69] and Zstd [25] have been integrated different scientific I/O libraries (such as HDF5 [101], Adios [102], and GIO [96]) and well-known scientific application packages (such as HACC [96] and LAMMPS [98]), in case of the data compression need.

However, there are limitations to the effectiveness of lossless compression. A key finding by Shannon in 1948 indicates that the amount of compression possible for a dataset is based on its entropy [103]. This finding leaves researchers looking for other methods which can get higher compression for high entropy datasets like images and floating point numbers.

Generally for these methods, there is a trade-off between high compression ratios and high compression bandwidths. We considered 4 lossless compressors that are commonly used to strike

different trade-offs between bandwidth and compression ratio: zip (also known as DEFLATE) [100], gzip [69], lz4 [104], and zstd [25]. Zip or DEFLATE is known for its higher compression ratios. It applies a LZ77 similar method (described in detail in [105]) with a 32Kb window followed by Huffman Encoding of the resulting data (described in detail [106]). It is one of a few lossless algorithms standardized for use on the Internet [100], with slight modifications [69]. More modern lossless methods have been developed by Yann Collet who developed both lz4 and zstd. lz4 is known for its high decompression rate [104]. It used a byte encoding to replace common subsequences of data with a compact representation similar to [105], but uses a fixed size block which improves compression time. zstd builds on the concepts in lz4 to achieve higher compression ratios at a cost of some speed. zstd uses LZ77 as a first stage and uses two bit reduction phases based on finite state entropy [107] and huffman encoding respectively to improve compression.

6.2.2.2 Traditional Lossy Compression Methods

Lossy compression allows to have data distortion in the reconstructed data compared with the original input and tries to keep a high "fidelty" to the original data, in that many practical use-cases do not need "complete facsimile" [14]. Formally, for a compression function $C(x)$, and a corresponding decompression function, $D(x)$, a compressor is lossy if $\forall x, \exists \epsilon, D(C(x)) = x + \epsilon$. This technique has been widely used in ML/DL applications because the input data are generally stored in the JPEG, PNG, or other image formats.

A JPEG compressed image is highly compressed relative to the image that a camera captures, but still "faithfully" displays the people, nature, or objects that are captured, but without regard to what errors a machine might perceive [14].. Many of these methods, however, are not designed for generic numeric data compression but rather for images. In our study, we still include one method in this category: *truncation*. Truncation computes a lower-order approximation of the model by neglecting the states that have relatively low effect on the overall model. In our experiment we have truncated the size of the data attributes from 64 bits size to 32 and 16 bits.

6.2.2.3 Error Bounded Lossy Methods

Error Bounded Lossy Compression (EBLC) is a recent technique that attempts to strike a balance between traditional lossless and lossy compression. Like lossy compression, EBLC loses some information in the compression process. However, unlike traditional measures, the user is

able to specify *aprori* how much error and where the error can be located in their data set with a mathematical bound on the error. This allows users to adopt the lossy compression on their datasets with high performance and control the data distortion within the error bound. As different programs and analysis need different concepts of error, these compressors often offer multiple kinds of error bound.

There are two state-of-the-art EBLC compressors currently being used widely: SZ [5] and ZFP [6]. ZFP was developed by Peter Lindstrom et al at Lawrence Livermore National Lab (LLNL) in 2013-2014. ZFP fundamentally works by providing a different encoding of floating point numbers that more easily compresses than the standard IEEE floating point notations (c.f. [6], [108]). ZFP provides 3 error bounding modes: *Fixed Precision Error Bounds* (PREC), *Fixed Accuracy Error Bounds* (ACC), and *Fixed Rate Distortion Error Bounds* (RATE). Fixed Precision (PREC) uses a fixed number of bit planes (number of most significant uncompressed bits) to encode each number. Fixed Accuracy (ACC) bounds the absolute difference between a uncompressed and decompressed value. Fixed Rate (RATE) uses a fixed number of bits to encode each value. ZFP splits the each dataset into equal-sized blocks and compresses the data block by block (the block size is set to 4^d, where d is the number of dimensions). In each block, ZFP performs an orthogonal transform on the exponent-aligned data followed by an embedded encoding algorithm. For the transformed data, ZFP truncates their insignificant bits, which is calculated based on the requested precision (i.e., different types of error bounds). Details and implementations can be found in [109]. ZFP has been widely used in many scientific applications [110]–[112], while its performance and impact to the ML/DL applications is still unknown.

In 2016, Di and Cappello developed SZ [5]. SZ fundamentally works by using a set of heuristics to approximate the error, and falling back to lossless compression if that failed to meet a given error tolerance. As time progresses, SZ has evolved and gained additional heuristics and metrics to help it improve its estimates for different kinds of data [20], [21]. For sake of time and clarity, we have chosen a subset error bounding metrics from SZ to include in this paper. We have included: Absolute Error Bound (ABS), Relative Error Bounds (REL), Peak Signal to Noise Ratio (PSNR), and Point-Wise Relative Error Bounds (PW-REL). Absolute Error Bound ensures that each value is with some constant of the original value. Relative Error Bound first computes the range of values input into the compressor and keeps the bound of each below that a specified fraction of that range. Peak Signal to Noise Ratio is a variant of Absolute Error Bounds based on the Peak

Signal to Noise Ratio metric from image compression. Finally, Point-Wise Relative Error Bound bounds the error in a similar manner to relative, but using the value of the point rather than the range of values as the scaling factor. Details and implementations can be found in [38]. SZ has been widely used in many scientific applications [13], [96] across different domains (cosmology, chemistry, mathematics, etc.), while its performance and impact to ML/DL applications is still unknown.

6.3 Experimental Overview

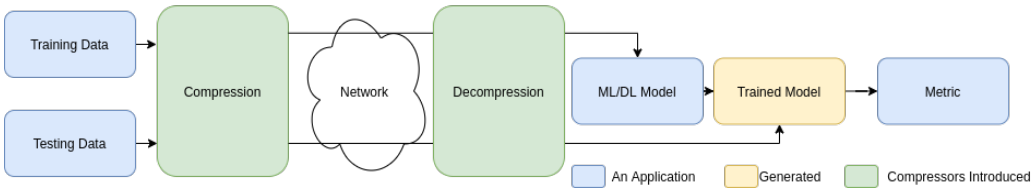


Figure 6.1: Workflow Overview

For each problem that we consider in the following sections, we will need applications to evaluate. For the purpose of this paper, a problem is some specific use case for ML/DL such as predicting if particular drug treats a particular kind of tumor (the P:Drug problem from Candle). An application is a particular solution to the problem (i.e. Candle NT-3). An application consists of a training and a testing dataset, some code that applies ML/DL to the application (a model), and at least one metric that determines how well the model performs on the problem (such as accuracy, gmean, pearson R coefficent, etc...). There may be many applications constructed for a particular problem. To investigate the impacts of applying compression to ML/DL, We treat the applications as a control variables for the purpose of the paper – that is we reused available applications that perform the well without compression according to the metric for their particular problem while just introducing compression. However, to account for differences for working on lossy data, we will re-train the hyper parameters of the models if any.

We consider the application of compression to two possible points of the training and evaluation process. We consider decompressing the training data prior to training, and decompressing the validation data prior to validation. This model is consistent with storing or receiving the data in compressed form prior to training or validating on it. While we are focusing on training and validation data in this paper, we expect that this methodology to extend to other points of applying compression.

Table 6.1: Summary of Data Reduction Methods

Method	Description
GZIP	[69] lossless compressor known for its compression ratios
LZ4	[104] lossless compressor known for its decompresion speed
NONE	no compression (control case)
SAMPLE NAIVE	sample every k^{th} entry
SAMPLE WR	sample randomly with replacement
SAMPLE WOR	sample randomly without replacement
SZ ABS	SZ [5] Absolute Error Mode
SZ PSNR	SZ [31] Peak Signal to Noise Mode
SZ PW_REL	SZ [21] Point-Wise Relative Error Mode
SZ REL	SZ [5] Value-Range Relative Error Mode
TRUNC	64 $\rightarrow$ 32 bit float truncation
ZFP ACC	ZFP [6] Fixed-Accuracy Mode
ZFP PREC	ZFP [6] Fixed-Precision Mode
ZFP RATE	ZFP [6] Fixed-Rate Mode
ZSTD	[107] ZStandard lossless compressor which improves on LZ4 with better compression ratios

Table 6.2: Hardware and Software Details

Component	**Description**	**Component**	**Version**
CPU	Intel Xeon 6148G (40 Cores)	Compiler	GCC 8.3.1
RAM	372 GB	OS	CentOS 8
Interconnect	100 GB/s HDR Infiniband	MPI	OpenMPI 4.0.5
GPU	2 Nvidia v100	Singularity	v3.7.1
MGARD	v0.1.0	SZ	v2.1.11.1
ZFP	v0.5.5	LibPressio	v0.60.0

After we have identified various applications and have performed compression and decompression, we turn our attention to how to analyze the impacts of compression using a variety of different compression techniques. Table 6.1 summarizes the methods we considered in this paper.

For our experiments, we use the following hardware and software in Table 6.2. We selected this hardware based on the availability of GPUs and CPU cores to run the models, and the availability of a high speed network interconnect.

6.4 Problem Formalization and Methodology

Table 6.3 lists all the notations used in our design and analysis, which can be split into two classes: compression-related parameters and analysis-related parameters. Basically, the compressor's parameters can be further split into two categories - nonfixed and fixed. The former refers to the parameters allowed to be tuned by users (such as error bound) and the latter is composed of all other parameters that are needed for running the compressor. Q is a data fidelity metric function

Table 6.3: Key Notations

Notation	Description
$\vec{c}$	Vector of nonfixed compressor parameters
$\vec{\theta_c}$	Vector of fixed compressor parameters
$\tilde{d_{f,t}}(\vec{c};\vec{\theta_c})$	Decompressed buffer for field, f, and time-step, t
$\vec{\theta_m}$	Vector of fixed parameters of the user-specified metrics function
U	Set of feasible nonfixed compressor parameters
Ω	Set of feasible fixed and nonfixed compressor parameters
$\mathcal{Q}\left(d_{f,t}, \tilde{d_{f,t}}\left(\vec{c};\vec{\theta_c}\right);\vec{\theta_m}\right)$	data fidelity metric function
$\phi = \mathcal{Q}\left(d_{f,t}, d_{f,t};\vec{\theta_m}\right)$	data fidelity metric for lossless data
N	The number of desired interesting points
Λ	the list of all "intresting points" ($\vec{c}$) from smallest to largest. (see Algorithm 5)
$\psi_{\vec{c}} = \mathcal{Q}\left(d_{f,t}, \tilde{d_{f,t}}\left(\vec{c};\vec{\theta_c}\right);\vec{\theta_m}\right)$	data fidelity metric evaluated at $\vec{c}$
τ	some user-determined threshold
s_p	parallel speedup on p cores
b_n	the bandwidth of the network/storage
b_c	the decompression bandwidth
$\mathcal{C}$	the compression ratio (uncompressed/compressed)

(or analysis metric) given by users for some application they are interested in (i.e. the validation accuracy for Candle-NT3).

We also adopt b_n and b_c to represent network or storage bandwidth and compression bandwidth respectively. It is also useful to refer to the compression ratio, $\mathcal{C}$, which indicates how much the volume of data has been reduced. We introduce the remaining notation in the following subsections.

6.4.1 Finding Interesting Points

One key aspect of this work is to find "interesting points" for users. Since we do not have an analytical representation of the relationship between the compressor configuration and the application's quality metric, we will have to invoke the application multiple times at different quality levels to determine their relationship. However, within the domain of inputs to the compressor many points are not interesting. Some error bounds are so small that they are indistinguishable from the original or lossless values. They are not interesting because there is no meaningful impact on the quality of the application. Some error bounds are so large that the user would not use them because the quality falls too much. Therefore, we need a way to determine these points.

A set of interesting points (Λ) can be identified using Algorithm 5 when the greater the values of some metric is, the better the analysis result or accuracy. There is a similar definition where one wishes to minimize the quality metric (such as a maximum error). First, we determine the value of the quality metric when evaluated on a lossless configuration – we call this ϕ the lossless quality metric. Next, we need to determine a configuration of the lossy compressor which has the

greatest error bound which minimizes the difference from lossless quality metric – we call this value $\vec{u}$ the upper bound. After that, we determine a configuration of the lossy compressor which has the greatest error bound which is still interesting to the user $\psi_{\vec{c}} < \tau$ – we call this value $\vec{l}$ the lower bound. We identified this points using a binary search like approach. With $\vec{l}$ and $\vec{u}$ determined, we need to sample the space in between to construct a set of interesting points. The number of points N presents a tradeoff: the larger N, the longer the process of measuring the points will take longer, the smaller N, the less likely we are to capture the behavior between $\vec{l}$ and $\vec{u}$ faithfully. Users should the maximum N that they can accept.

Input: $\mathcal{Q}, d_{f,t}, \tilde{d_{f,t}}, U, \vec{\theta_c}, \vec{\theta_m}, \tau, N$
Output: the set of interesting points
begin find_interesting_points
  $\phi \leftarrow \mathcal{Q}\left(d_{f,t}, d_{f,t}; \vec{\theta_m}\right)$
  $\vec{u} \leftarrow \text{argmin}_{\vec{c} \in U} |\phi - \psi_{\vec{c}}|$
  $\vec{l} \leftarrow \text{argmin}_{\vec{c} \in U; \psi_{\vec{c}} > \tau} \psi_{\vec{c}}$;
  $\Lambda \leftarrow \text{logspace}\left(\vec{l}, \vec{u}, N\right)$ `// see numpy.logspace`
  return Λ
end

Algorithm 5: Finding Interesting Points

There are many possible ways to sample this space. We choose the evenly distributed points on a logarithmic scale – what many libraries often call logspace. This is for a few reasons: First, prior work has shown that uniform random sampling does not work well because the relationship between a compressor error bound and even simple metrics tend to have spatial properties [2]. Second, preliminary evaluations showed that there tended to be more interesting behavior at smaller error bounds which the logscale samples.

We will use these interesting points in each of our experiments to examine the trade-offs between the compressors.

6.4.2 Measuring Inconsistent Effects

Sometimes the effect of applying data reduction is not consistent effect as the error bound for the compressor is increased. This represents an indication that types of errors induced by the reduction technique make it such that it no longer represents the original dataset well. However to compare the size of this effect, it is helpful to be able to measure it.

The metric we propose is "sortedness" and intuitively measures the maximum of how much

each observation would have to change in order to put the points into sorted order. Techniques whose interesting points have a small "sortedness" value represent values which have relatively consistent effects. However, techniques whose interesting points have large "sortedness" values would indicate techniques who the user cannot reasonably predict what impact the technique may have of the metric of interest. This metric is preferable over more classical measures for how sorted a dataset is because it treats values adjacent values which are unsorted but have similar value as less significant than those whose values are unsorted and are grossly different. This is useful because lossless methods may have several entries which are unsorted, but are close in value; whereas lossy methods may be more likely to be sorted, but have much larger adjacent differences.

To compute "sortedness", we propose the procedures outlined in Algorithms 6 and 7. Algorithm 7 describes a $O(N)$ merging procedure used to emulate the entries changing value by at most ϵ. It works by sliding a window through the values and expanding the upper and lower threshold for merging points as long as we find values within our existing window. We we encounter a point that is outside of the window, we start a new window at that point.

Algorithm 6 uses this merge procedure to preform a binary search for the smallest threshold that achieves a sorted order. We know to increase the lower bound of our search when we find a sorted ordering, and we know to decrease our upper bound of our search when we no longer find a sorted ordering. The total algorithm is $O(N \log N)$.

6.4.3 Measuring Threshold Effects

Sometimes the effect of applying data reduction techniques results is relatively consistent up until a point and then rapidly deteriorates. We call this behavior "thresholding". Measuring this behavior is useful because it allows us to understand which kinds of techniques could have unexpectedly poor performance within the range of interesting points. To measure this effect we propose using the maximum of the adjacent differences of the interesting points.

Techniques that measure a large maximum adjacent difference of the interesting points represent techniques that have a large drop of in the quality of the results. Techniques that measure a small maximum adjacent difference of the interesting points represent techniques that are relatively consistent over the interesting points. One may note that this does not account for sharp rises in the quality metric, this is because such a change would be a non-monotonic and thus captured by our "sortedness" measure.

```
Input: Λ, ψ_c⃗
Output: the smallest epsilon that "sorts" the input
begin stability
    ψ ← ψ_c⃗, ∀c⃗ ∈ Λ
    ϵ_low ← min_{i,j∈ψ}(i − j)
    ϵ_high ← max_{i,j∈ψ}(i − j)
    ϵ_current ← midpoint(ϵ_low, ϵ_high)
    τ ← .001
    ϵ_best ← ϵ_high
    while ϵ_high > ϵ_low ∧ τ ≤ |ϵ_high − ϵ_low| do
        merged ← merge(ψ, ϵ_current)
        if is_sorted(merged) then
            ϵ_best ← min(ϵ_best, ϵ_current)
            ϵ_low ← ϵ_current
        else
            ϵ_high ← ϵ_current
        end
        ϵ_current ← midpoint(ϵ_low, ϵ_high)
    end
    return ϵ_best
end
```

Algorithm 6: Stability

```
Input: ϵ, ψ_c⃗ ∀c⃗ ∈ Λ
Output: The merged values
begin merge
    merged ← []
    for i ← 0 ; i ¡ length(ψ); i += step do
        step ← 1
        append(merged, ψ_i)
        min_epsilon ← ψ_i − ϵ
        max_epsilon ← ψ_i + ϵ
        for j ← i + 1; j < length(ψ); j ← j + 1 do
            if min_epsilon ≤ ψ ≤ max_epsilon then
                min_epsilon ← min(min_epsilon, ψ_j − ϵ)
                max_epsilon ← max(max_epsilon, ψ_j + ϵ)
                step ← step + 1
            else
                break
            end
        end
    end
    return merged
end
```

Algorithm 7: Merge

Additionally one may argue that the position of the drop matters: A sharp drop in the middle of the set of the interesting points is more important than a drop near $\vec{l}$. To account for this concern we also recommend consulting the pareto front of the compression ratio and the quality metric. Pareto fronts are the set of points for which one property (in this case the compression ratio) cannot be improved without degrading another property (in this case the quality metric for the application).

6.5 Experimental Results

In Section 6.5.1, we first demonstrate broad applicability of lossy compression on a several of different problem classes with different kinds of solutions. This section is primarily concerned with the quality of the compression and the results. Due to data and solution availability constraints, this section will use many smaller problems. For this section we will consider variations on on layout, compression principle, and the data being compressed (training/validation/both) before we scale to larer problems.

Later in Section 6.5.2, we will show how this translates to larger problem sizes which benefit more for compression.

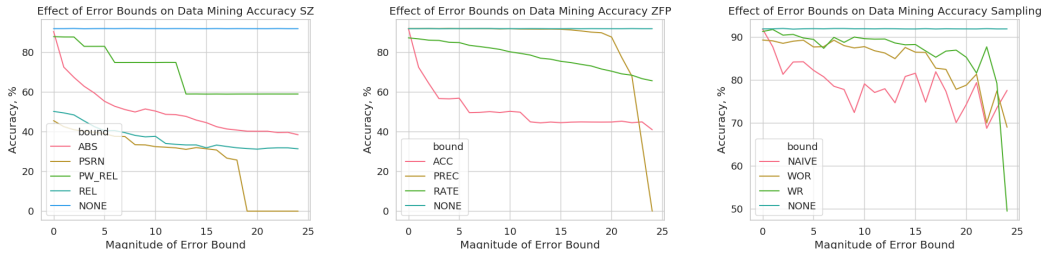
For Section 6.5.1, we use problems, solutions, and datasets from the from the UCI datasets, several datasets MLSVM application [113], as well as the current publicly available data from Candle which had all had publicly available reproducible code and data.

6.5.1 Breadth of Applicability

6.5.1.1 Interesting Points

We begin by creating a pareto front for each of the applications and methods described in this for each of the methods in Table 6.1. Instead of preparing these figures for the lossless methods, we simply re-ran the application several times to measure the natural variation in the quality. Figure 6.2 illustrates the results of finding "interesting points" for the Superconductor data sets.

In this figure, x-axis corresponds to the configuration ID numbers that have increasing error bounds (i.e. more lossy), and y-axis represents the analysis result (i.e., accuracy of the ML/DL model after training: the higher the better). We selected the values that generated these bounds



(a) SZ (b) ZFP (c) Sample Sizes

Figure 6.2: Tuning the Superconductor Dataset

using a binary search-like algorithm to identify the error bound value that resulted in the greatest compression with little to no loss in accuracy. Then we tuned the right error bound to be the value at which the amount of error no longer increased or fell below a threshold of 30%.

We can observe some interesting properties of the compressors using this algorithm. First, as shown in Figure 6.2a), SZ (ABS) or SZ(REL) has the most smooth transitions from lossless to very lossy. This is a nice property when using these algorithms because it implies that users have a fine-grained trade-off between level of error and compression ratio. This is not surprising given that SZ uses a quantization-based method to bound its error such that the data distortion changes gradually with the error bound. This kind of approach allows for fine-grain trade offs in error as you can simply compress an increasing number of points losslessly.

Second, we note that ZFP (Figure 6.2b) has more of a thresholding effect. ZFP experiences very little loss in accuracy for a given increase in an error bound until it reaches a given level and falls rapidly. This too is not particularly surprising given how ZFP operates. Basically, ZFP chooses accuracy levels by selecting a number of bits to preserve. This threshold therefore is likely comparable to the amount of entropy in the uncompressed data. This property is nice in that it allows much higher levels of accuracy to be for higher levels of compression, but it does have a worrying aspect. The value of the threshold is non trivially related to the dataset and properties of ZFP and difficult to predict without running the compressor multiple times. This means that users may have a rapid loss in accuracy just by subtlety increasing the error bound.

Finally, we see that Sampling (Figure 6.2c) is sort of a half way between ZFP and SZ. It has a smoother transition that ZFP does from lossless to lossy, but still features that thresholding effect. It is also interesting to note that the curves are much more “noisy” than either ZFP or SZ.

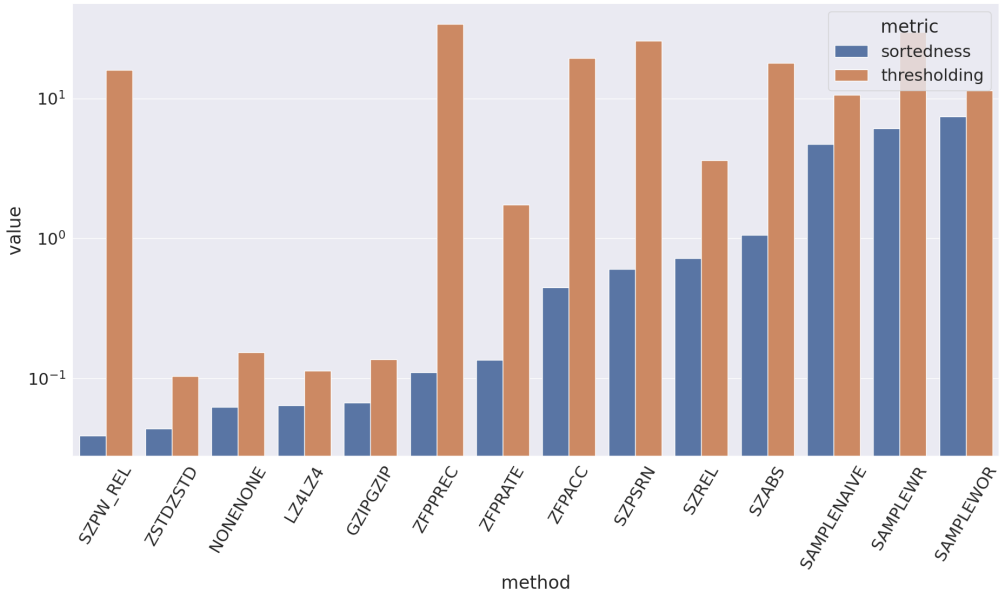


Figure 6.3: Consistency Metrics

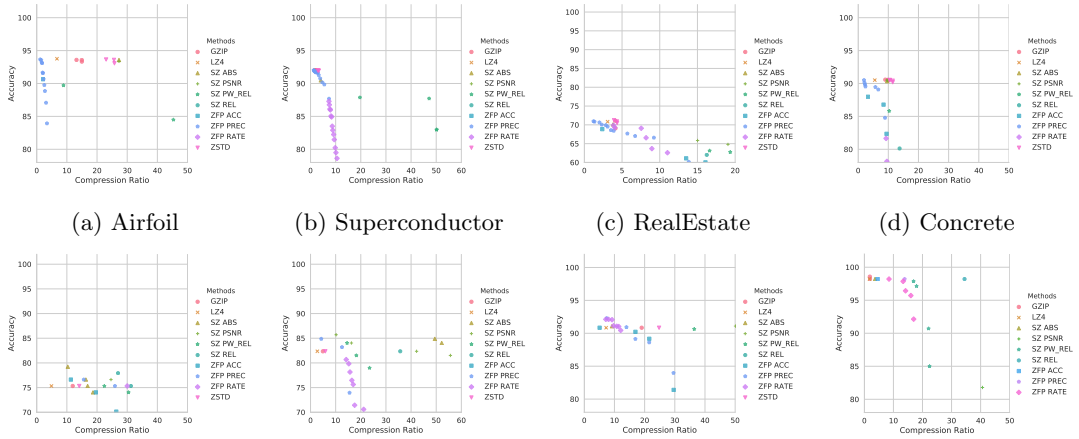
6.5.1.2 Threshold and Sortedness Metrics

Now we attempt to quantify our intuitions from the previous sections with the metrics that presented in Section 6.4.2 and 6.4.3. For this section we again consider the results from Superconductor we considered in the previous section for sake of space. We present our results in Figure 6.3.

We note that consistent with our expectations, the lossless methods perform well on both metrics with small sortedness and thresholding effects. If this were not true, it would indicate a significant flaw in our approach.

Additionally we observe for sortedness, there are 3 distinct clusters: the lossless methods and SZ PW_REL mode, the rest of the lossy compressors, and finally the sampling based methods. There is a large difference between the lossy methods that we consider and the sampling based approaches. This is a strong reason to adopt lossy compression methods over sampling based approaches.

When considering thresholding, there are some similarities for the grouping of methods, but also some important differences. First, SZ PW_REL and ZFP_PREC has are now among the worst performers. This is because they both experience a significant drop in performance: PW_REL towards the middle of its range and ZFP_PREC towards the end of its range. While these values are indicative of inconsistent performance, a different choice about what the user considers acceptable as the upper bound for loss will effect this decision, and the users should also consider the Pareto



(a) Airfoil (b) Superconductor (c) RealEstate (d) Concrete

(e) haberman (f) Musk (g) Nonskin (h) Candle NT-3 (small)

Figure 6.4: Pareto Optimal Points Compression Ratio and Quality for Various Applications

plots.

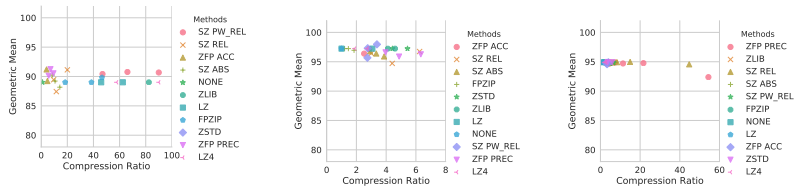
6.5.1.3 Pareto Optimal Points

After preforming the analysis in each of the previous subsections for the remaining applications, We consider pareto optimal points for each of the smaller applications. We present the results of this consideration in Figures 6.4 and 6.5. In these figures, points that are best are to the upper right.

The results in Figure 6.4, applied error bounded lossy compression independently to each column. Where as the results in Figure 6.5 applied error bounded lossy compression to the entire data as a matrix. We did this in part because of differences in the design of MLSVM that made integration challenging, but also to study the impacts of different data layouts.

For some datasets, the ZSTD lossless compressor also performed well, but this was very dataset dependent. Datasets with many identical values performed well with ZSTD, those with more true floating point values did poorly. Additionally only achieved this level of compression ratio but this also required highest compression level setting which has significant run time on most data sets, often 4 times the next closest compressor. GZIP also required substantial time to run, but tended to preform worse than ZSTD.

Of the lossy compressors, SZ's value range relative mode and ZFP's precision modes consistently preformed well on many datasets. At higher allowed drops in quality, the SZ PW_REL compressor also performed well.



(a) MLSVM Advertisement

(b) MLSVM Clean

(c) MLSVM Buzz

Figure 6.5: Pareto Optimal Points for MLSVM

6.5.1.4 What compression approaches were the most effective at preserving quality at a given compression ratio?

We have found that independent column based lossy compression is most effective for ML/DL problems when the inputs are formulated as tables where rows are observations and columns are features. This is because the values in each column are more likely to be spatially correlated with each other where as the contributions of rows are likely to reduce the accuracy of transforms and predictions. This does come at some slight overhead for some compressors such as SZ where redundant huffman trees are stored for each row regardless of input size and runtime performance overheads due to repeated memory allocation latency when columns are small. In the future, a specialized compressor could alleviate these overheads.

We have found that prediction based error bounded lossy compression was generally the most effective. Even though the compressor modes in SZ share substantial amounts of code, the value range relative mode was most effective when compressing by column or column, and all modes were similarly effective when compressing as a matrix. This is because using value range relative mode effectively set distinct error bounds for each column or row. Figure 6.6 shows the distribution of the value ranges by column for Candle NT-3. We expect that ZFP would have similar benefits if an adapter was made for ZFP to preserve a value-range relative bound. Sampling methods were too erratic in terms of both non-monotonities and sharp drops to be recommendable.

6.5.2 Performance Evaluation

To evaluate the performance implications of using lossy compression, there are in principle two possible slow downs. One possibility is that training time would change, we found this was not true for the applications we tested. The other possibility is the impact on IO time, this is the focus of the remainder of this section.

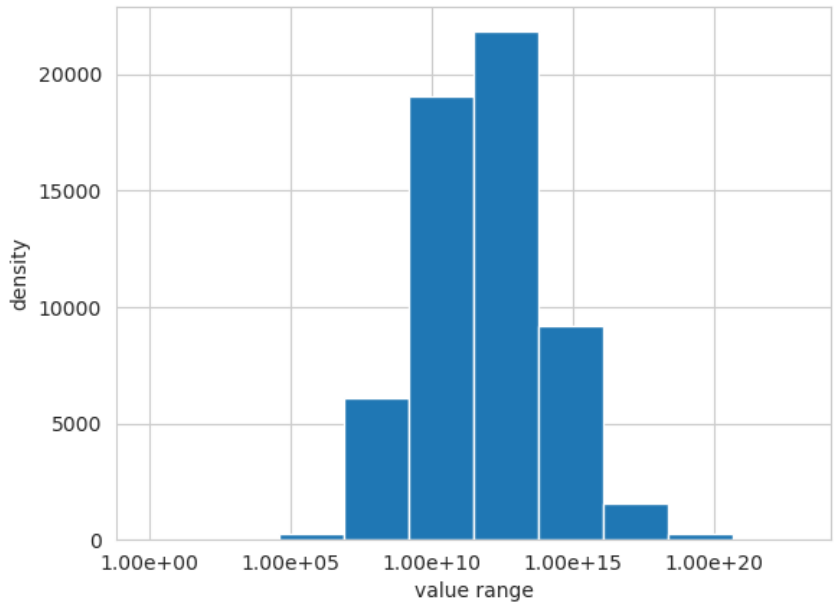


Figure 6.6: Distribution of the value Ranges for Candle NT-3

To evaluate scalability of using error bounded lossy compression, we will use larger scientific data sets. We have two such larger data sets: Candle-NT3 and PtychoDNN. For each application, we generated a 4GB binary input file [1] using the tools provided by the applications. Note that this generalizes to even larger datasets that are processed in chunks; this is to overcome maximum input sizes for some compressors such as LZ4 and ZSTD [25], [104]

The runtime without compression as the time to read the entire data over the network:

$$\frac{||d_{f,t}||}{b_n}$$

One can model the time used to perform read the data with compression can be modeled as the sum of the transfer time in compressed format + the parallel decompression time:

$$\frac{||d_{f,t}||}{s_p b_c} + \frac{||d_{f,t}||}{\mathcal{C} b_n}$$

[1]Candle NT-3's native input format is CSV, however to mitigate the large overheads of parsing the CSV into a binary form for the compressors and processing by Candle. All input sizes reported for Candle assume this binary format instead of the native CSV format; Our results are even stronger when compared against CSV with parsing due to this overhead. PtychoDNN uses numpy format which is a binary format with a small header and thus did not require conversion

Thus the speedup is:

$$\frac{\mathcal{C} s_p b_c}{\mathcal{C} b_n + s_p b_c}$$

We can then compute the number of cores required to achieve a parallel speedup:

$$s_p > \frac{\mathcal{C} b_n}{b_c \left(\mathcal{C} - 1\right)}$$

Network bandwidth can very widely from site to site, and across different times of day. Therefore, we compute the number of parallel compression required to achieve a speedup with varying IO latency. We consider 3 speeds for the literature:

- 3.75 GB/s – this is the speed of the dedicated link between the Advanced Photon Source and Argonne Leadership Computing Facility at Argonne National Laboratory [122].
- 1 GB/s – this is the typical speed that users can transfer data using Globus [94] [2]
- 125 MB/s – this is the typical speed for broadband internet at remote locations [93]

Now we must determine the bandwidth of the compressors. The bandwidth of the compressors depends on the dataset and the configuration of the compressors. In Tables 6.5 and 6.6, we show the bandwidth achieved using the SZ compressor in value-range-relative mode – one of the compressors which performed the best in Section 6.5.1 for the Candle-NT3 and Ptychonn datasets.

6.6 Related Work

In this section, we discuss the related work in two facets: widely prevalent compression techniques and the lossy compression designed for convolutional deep neural network.

6.6.1 Widely Prevalent Techniques

There have been several attempts to apply lossy compression to data mining in the past.

Most of these attempts have been in the case of images. One common method is to convert images to gray scale by normalizing the color channels together. A difference approach is to convert

[2] Greater speeds are possible as described in this paper, but requires cooperation with the Internet Service Providers and other optimizations not available to typical users

the image to a reduced color gamut by truncation. Another technique is to down-sample images by averaging the color of either adjacent or related pixels. A final technique is to use lossy images compression methods such as JPEG [14], JPEG-2000 [123], BPG [124], or WEBP [125] and increase the parameters that control loss[126]. However, there are three major short comings in these classes of work as applied to general numeric data. First, there is no quantification of what information was lost during compression or if it was important to the classification or regression. Second, there is no knowledge *a priori* of how much error will be incurred in the raw data before compression. Third – and perhaps most importantly – general numeric data may not be represented well as an image.

The paper by [99] considers the related domain of compressive sensing. They argue that for some lossy compressors such as JPEG with a given rate distortion, it is possible to approximately lower bound the number of observations required to "successfully" reconstruct the original signal. Thus approximately "bounding", certain classes of traditional lossy compressors commonly used in image storage.

There are some papers that use very early forms [127] of lossy compression in pattern mining [128]. Essentially, the authors construct a decision tree for a subset of columns called predicted columns, if these tree accurately predicts within the error bound and the decision tree is smaller, it is stored instead. However, these techniques were expensive to execute and have a chance of failing to compress at all because of the error bound. They also don't attempt to quantify the effect of the loss on the decisions made on the decompressed data.

Our approach differs from all of these previous approaches in that it is the first to consider error bounded lossy compression. Second, our approach differs in that it the first approach that is reasonable high performing and generally applicable to both balanced and imbalanced numeric features.

6.6.2 Lossy Compression for Convolutional Deep Neural Networks

There have been a few papers which have looked at particular aspects of applying lossy compression to ML/DL applications.

One application area where lossy compression has been used with ML/DL in in pedestrian detection [93]. This work used lossy H.264 with HVEC video compression to compress traffic footage to reduce bandwidth on rural networks. However, the training was performed on lossless data and only validated on lossy data. This work goes beyond the work by Mizanur et al to consider a wider

array of compressors as well as different ways that they can be applied such as different data layouts and the effect of training/validation on lossy data.

Another example, the paper [129] considers the effect of compressing activation data for deep neural networks using JPEG image compression as well as Precision Reduction, Run Length encoding, and Zero Value Compression compression. Like this paper, ours considers the application of lossy compression, but the focus is different: they focused on activation data for deep neural networks whereas we focus on the compression of training and validation data for a greater variety of methods with the goal of providing a method for evaluating the impacts of lossy compression on ML/DL applications. Additionally, the paper by Evans et al considers only two lossy compressors and neither of them are particularly modern of designed for scientific applications.

For that we, we need to consider the paper [130] which studies the impacts of cuSZ's – the gpu implementation of the SZ version 1.4 compressor – on the gradients used in convolutional deep neural networks. Unlike the paper [129] which was primarily empirical, the paper by Jin et al uses an analytical approach which makes a number of simplifying assumptions which are true for the applications they consider and cuSZ compressor specifically. One of these assumptions is that the error distribution imposed by the compressor is uniform; however, this is not true of many of the leading lossy compressors including newer versions of SZ [24]. Further their formulation is also limited in that applies only to networks whose gradient is just sum of the product of the activation data and loss – i.e. not containing any dropout or other more complex operators. Where as this work considers many more classes of models and compressors which do not hold these properties.

6.7 Conclusions and Future Work

Error bounded lossy compression offers a high performing option for storing and transporting training and validation data for a variety of different ML/DL applications – especially value range relative bounds on compressed independently by feature data.

Some areas of future work include considering a greater variety of applications as well as considering a specialized compressor which can apply different error bounds to different columns.

Dataset	Size	Observations	Features	Data Type	Problem Solved	Approach
Haberman [114]	13K	306	3	Integer	Binary Classification	Random Forest
Nonskin [115]	9.4 Mb	245057	3	Integer	Binary Classification	Stochastic Gradient Descent
Musk [116]	635Kb	476	167	Integer	Binary Classification	Random Forest
Real Estate [117]	27Kb	414	6	Real	Regression	Random Forest
Concrete [118]	82Kb	1030	8	Real	Regression	Random Forest
Airfoil [119]	84Kb	1503	5	Real	Regression	Random Forest
Superconductor [120]	14Mb	21263	82	Real	Regression	Random Forest
MLSVM-Clean [113]	13MB	6598	166	Real	Binary Classification	MLSVM (SVM)
MLSVM-Advertisement [113]	59MB	3279	1558	Real	Binary Classification	MLSVM (SVM)
MLSVM-Buzz [113]	125MB	140707	77	Real	Binary Classification	MLSVM (SVM)
Candle-NT3 [87] (Small)	518Mb	1120	60483	Real	Binary Classification	Neural Network
Candle-NT3 [87] (Large)	4Gb	8960	60483	Real	Binary Classification	Neural Network
Ptychonn [121] (Large)	4Gb	8100	131072	Real	Autoencoder	Neural Network

Table 6.4: Overview of the Applications

Table 6.5: Cores for a parallel speedup with SZ-REL: Candle NT-3

psnr	bound ($\vec{c}$)	$\mathcal{C}$	b_c	cores 3.75 GB/s	for a 1GB/s	speedup 125 MB/s
164.7	1e-8	5.4	0.17	27	8	1
144.7	1e-7	7.6	0.28	16	5	1
124.7	1e-6	13.1	0.40	11	3	1
104.1	1e-5	35.2	0.71	6	2	1
93.0	1e-4	201.5	1.28	3	1	1
83.8	1e-3	1476.6	1.44	3	1	1

Table 6.6: Cores for a parallel speedup with SZ-REL: Ptychonn

psnr	bound ($\vec{c}$)	$\mathcal{C}$	b_c	cores 3.75 GB/s	for a 1GB/s	speedup 125 MB/s
164.90	1e-08	2.22	0.06	122	33	5
144.80	1e-07	3.05	0.08	69	19	3
124.87	1e-06	4.61	0.13	38	11	2
105.93	1e-05	10.67	0.21	21	6	1
90.71	1e-04	40.46	0.38	11	3	1

Chapter 7

Conclusions

This dissertation tackles three areas designed to improve the approach-ability of error bounded lossy compression. First, this work provided a generic interface for error bounded lossy compression that enabled productive, high performance, and parallel lossy compression unifying the host of different incompatible interfaces that came before. Second, this work demonstrated how to automatically configure error bounded lossy compression in a variety of circumstances using black box optimization techniques that improved over the performance of prior approaches. Finally, it applied these techniques and developed a methodology for validating the use of EBLC in ML and AI workflows.

In the first work, I demonstrated that using a unified and well-defined interface for loss compression could reduce code volumes for client code by over 50% even when supporting only a single compressor. It reduced the costs of switching compressors unlocking the benefits of improvements from advances in compressor designs without client code changes. I demonstrated that this could consistently be done with statistically insignificant overhead. I also looked at how this could enable productive, high-performance, portable and parallel compression workloads. I demonstrated with less code that I could develop parallel compression techniques that maintained nearly as much of the both run-time performance and scalability as well as compression ratio as two bespoke implementations.

In the second and third works, I demonstrate the effectiveness of using black-box optimization techniques to automatically configure compressors first to bound the compression ratio, and then to bound general metrics that a user might provide. I show how this tuning process for fixed

compression ratio compression with low overhead of ≈ 2 when amortized over a large dataset. I then substantially improve the run-time by $56\times$ while solving the more general problem of bounding deterministic metrics provided by users. I also demonstrate performance advantages of this approach over specialized compressors with tighter control of the metric which yielded higher compression ratios, over the approach used by MGARD-QOI with even more substantial performance improvements, and over systematic approaches which were previously used by complex metrics showing an improvement of $27\times$ while using $4\times$ less hardware.

In the final work, I apply these techniques to ML/AI workflows generalizing my approach again to data-sets where multiple buffers contribute to the analysis. I propose a methodology to investigate the effects of adoption of error bounded lossy compression and apply it to a collection of ML/AI applications that will reasonably need to adopt a strategy such as EBLC to accommodate for their growing dataset sizes. I show that EBLC performs among the best at compressing data while also being less vulnerable to threshold or inconsistent effects than traditionally adopted techniques such as lossless compression, sampling, and truncation.

Together, these techniques will aid researches as they consider the use of techniques of error bounded lossy compression.

Appendices

Appendix A

LibPressio Usage Example

A basic example of using LibPressio with error handling omitted for conciseness. Adapted from the example on [60]. It takes a buffer in memory, and compresses it with the SZ compressor using an absolute error bound of 0.5. To adapt this example for ZFP or another supported compressor, only lines 10, 22, and 23 would need to be changed.

```
1  #include <libpressio.h>
2
3  float* make_input_data();
4
5  int
6  main(int argc, char* argv[])
7  {
8    // get a handle to a compressor
9    struct pressio* library = pressio_instance();
10   struct pressio_compressor* compressor = pressio_get_compressor( library, "sz");
11
12   // configure metrics
13   const char* metrics[] = { "size" };
14   struct pressio_metrics* metrics_plugin = pressio_new_metrics( library, metrics,
     ↪  1);
15   pressio_compressor_set_metrics( compressor, metrics_plugin);
16
17   // configure the compressor
18   struct pressio_options* sz_options = pressio_compressor_get_options(
     ↪  compressor);
19
20   pressio_options_set_string( sz_options, "sz:error_bound_mode_str", "abs");
21   pressio_options_set_double( sz_options, "sz:abs_err_bound", 0.5);
22   pressio_compressor_check_options( compressor, sz_options);
23   pressio_compressor_set_options( compressor, sz_options);
24
```

```
25    // load a 300x300x300 dataset into data created with malloc
26    double* rawinput_data = make_input_data();
27    size_t dims[] = { 300, 300, 300 };
28    struct pressio_data* input_data = pressio_data_new_move( pressio_double_dtype,
      ↪  rawinput_data, 3, dims, pressio_data_libc_free_fn, NULL);
29    // setup compressed and decompressed data buffers
30    struct pressio_data* compressed_data = pressio_data_new_empty(
      ↪  pressio_byte_dtype, 0, NULL);
31    struct pressio_data* decompressed_data = pressio_data_new_empty(
      ↪  pressio_double_dtype, 3, dims);
32
33    // compress and decompress the data
34    pressio_compressor_compress( compressor, input_data, compressed_data);
35    pressio_compressor_decompress( compressor, compressed_data, decompressed_data);
36
37    // get the compression ratio
38    struct pressio_options* metric_results = pressio_compressor_get_metrics_results(
      ↪  compressor);
39    double compression_ratio = 0;
40    pressio_options_get_double( metric_results, "size:compression_ratio",
      ↪  &compression_ratio);
41    printf("compression ratio: %lf\n", compression_ratio);
42
43    // free the input, decompressed, and compressed data
44    pressio_data_free( decompressed_data);
45    pressio_data_free( compressed_data);
46    pressio_data_free( input_data);
47
48    // free options and the library
49    pressio_options_free( sz_options);
50    pressio_options_free( metric_results);
51    pressio_compressor_release( compressor);
52    pressio_release( library);
53    return 0;
54  }
```

[Page 128 is blank]

Appendix B

Glossary

k^{th} order error Metrics module that computes The size of kth largest absolute value of the differences observed between the uncompressed and decompressed data.

ADIOS2 A parallel IO, data movement, and data processing framework.

auto-correlation Metrics module that computes the Pearson's correlation coefficient between the data and itself shifted by one or more "lags". For example for the points $\vec{v} = \{1, 2, 3, 4, 5\}$ with a lag of 2 would compute the correlation between $\vec{v_1} = \{1, 2, 3\}$ and $\vec{v_2} = \{3, 4, 5\}$.

AutoSFX An automated crystallography analysis and processing framework being developed at the Stanford Linear Accelerator Center.

Bit Grooming Compressor that applies various manipulation techniques to increase comparability of IEEE floating point numbers.

BLOSC A family of lossless compressors that have been optimized for performance.

chunking a meta-compressor which divides a dataset into contiguous chunks dispatching each of them to a another meta-compressor. This is useful for automatic parallelization.

CSV IO plugin that consumes Comma Seperated Values and other related formats such as tab seperated values.

delta encoding Meta-compressor that applies a delta encoding a preprocessing step. Delta encoding encodes values encodes the values using adjacent differences. For example $\vec{v} = \{1, 2, 3, 4, 5\}$ would be encoded as $\vec{v} = \{1, 1, 1, 1, 1\}$.

dense tensor a multi-dimensional generalization of an array with a large number of non-zero values often stored contiguously in memory. In C/C++, these are stored in row-major order which has indicies that advance from slowest to fastest. In Fortran, column major order is used where indicies advance from fastest to slowest.

differences-probabilities densities function (pdf) Metrics module that generates an empirical probability density function of the differences between the uncompressed and decompressed values.

Digit Rounding Compressor that applies various rounding techniques to increase comparability of IEEE floating point numbers.

endomorphism A transformation where the set of values allowed as input (i.e. the domain) and the set of values produced as output (i.e. the co-domain) are the same, for example, addition on the integers..

error statistics Metrics module that computes basic descriptive statistics using algorithms that can be computed in a single pass.

Fault Injector Meta-compressor that applies a sequence of single bit errors into the compressed data. Useful for implementing fuzz testing.

fpzip A specialized lossless and lossy compressor for IEEE floating point values.

HDF5 IO plugin that uses the HDF5 parallel IO library and file format.

HDF5 Filter A feature of the HDF5 IO library that allows compression to be preformed inline to dataset access. Supports plugins to support different compressors.

Image Magick A extensive library for image manipulation and compression.

Iota A IO plugin that generates synthetic data using C++'s `std::iota` which fills a buffer with sequentially increasing values.

isomorphism a pair of structure preserving transformations where one reverses the other. Lossless compression is an isomorphism, but lossy compression is not because it is not because not all – in fact most – lossy compression is not reversible.

Kolmogorov-Smirnov (KS) Test for Goodness of Fit Metrics module that compute a non-parametric statistical hypothesis test which test the hypothesis that two distributions are two samples are drawn from the same distribution that operates by determining the largest difference between the empirical cumulative density function.

Kullback-Liebler (KL) Divergence A metrics module that computes A measure of relative entropy from one distribution to another. It is defined as $D(P||Q)_k l = \sum_{x \in X} P(x) \log \left(\frac{P(x)}{Q(x)} \right)$. It is used in information theory and machine learning.

Levenshtein's distance The minimum numbers of characters that need to be inserted, deleted, or changed to make two strings the same. For example, "bad" and "bed" have a distance of 1 while "bed" and "bread" have a distance of 2.

LibPressio-Fuzz A Fuzzer developed for this paper that use LibPressio and Clang/LLVM's libfuzzer.

LibPressio-Opt A meta-compressor that implements an optimizer that can be used to determine an optimal configuration. Previous version of this were named FRaZ and OptZConfig..

LibPressio-Tools A set of tools developed for this paper that use LibPressio to implement a command line interface for LibPressio compressors and meta-compressors.

linear quantization Meta-compressor that preforms linear quantization. Quantization is a transformation that maps a contiguous domain (i.e. floats) to a countable domain (i.e. integers). Linear Quanitization, does so with a mapping like $Q(x) = \lfloor \frac{x-m}{\Delta} \rfloor$ where x is the value, $Q(x)$ is the quantized value, Δ is a scaling factor, and m is some centering term. Quantization is often used in lossy compression because coutable domains often have lower entropy than contiguous domains and are thereby more compress-able.

Many Dependent A Meta-compressor that implements a parallel pipeline that does the following. The first buffer is operated upon and metrics are gathered from it. Metrics from the first buffer

are passed to one or more compression that are done in parallel as configuration options. As each buffer finishes, the value of the latest indexed buffer to complete is stored to be passed to future invocations. This is useful for passing along a guess for an optimal configuration when working on data buffers that are contigious in time..

Many Independent A meta-compressor that implements a embarrassingly parallel compression of multiple data-sets.

masked Metrics module that removes specified data points from a data set prior to computing some other statistic.

Meta-Compressor A concept within LibPressio. Meta-Compressors implement the compressor interface, but are not compressors. Examples may include pre/post processing steps, parallel run-times, optimizer, etc....

MGARD A error bounded lossy compressor based on multi-grid methods.

mmap IO plugin that uses the UNIX system call `mmap` that maps the contents of a file or memory of a device into memory via the virtual memory of a process.

NumPY IO plugin for the custom file format used by the python numeric library NumPY for storing n-dimensional arrays.

Pearson's Correlation Metrics module that computes Pearson's Correlation Coefficient (often denoted r) measures the strength of a linear relationship between two values .

PETSc A IO plugin that reads file created by PETSC, the "Portable, Extensible Toolkit for Scientific Computation".

posix IO plugin that uses the POSIX functions `read` and `write` to read in an array in a native data format.

R Metrics module that uses the scripting language R that is specialized in statistical analysis.

Random Error Injector A meta compressor that applies randomly generated noise to each element of the input dataset according to some specified distribution.

Region of Interest Metrics module that Computes the arithmetic mean of a region of interest within a dataset.

resize A meta-compressor which modifies the dimensions of the data without modifying the values. This is useful for compressors which sometimes benefit from being told the data shape is different than it actually is – i.e. ZFP if you have a 3d dataset that is $A \times B \times 1$ so you can treat it as 2D.

sample A meta-compressor which applies data-sampling techniques such as uniform sampling with and without replacement prior to compression.

select IO plugin that selects a sub-region of an input dataset read in by another IO plugin for compression/analysis.

Spatial Error The percentage of elements of a dataset that exceed some specified threshold.

switch A meta-compressor which allows runtime switching between different compressors based on a configuration setting. This is useful because it allow tools like LibPressio-Opt to select between multiple different compressors types dynamically.

SZ SZ is a prediction based error bounded lossy compressor.

SZ-OMP the parallel CPU version of the SZ prediction based error bounded lossy compressor.

SZ-Threadsafe the threadsafe serial version of the SZ prediction based error bounded lossy compressor.

the Feature Detection Toolkit (FTK) Metrics module that uses the library FTK that tracks features such as maxima, minima, an saddle points in data between time-steps of a simulation.

transpose A meta-compressor which applies a multi-dimensional abstraction of a transpose to the data prior to compression.

tthresh A compressor that uses the principles of singular value decomposition to compress data.

vecSZ A version of SZ optimized to leverage SIMD vector instructions.

ZFP A transform based error bounded lossy compressor.

Bibliography

[1] R. Underwood, V. Malvoso, J. C. Calhoun, S. Di, A. Apon, and F. Cappello, "Productive, performant, and parallel generic lossy data compression with libpressio."

[2] R. Underwood, S. Di, J. C. Calhoun, and F. Cappello, "FRaZ: A Generic High-Fidelity Fixed-Ratio Lossy Compression Framework for Scientific Floating-point Data," presented at the 34th IEEE International Parallel and Distributed Processing Symposium, New Orleans: IEEE, May 18, 2020.

[3] R. Underwood, J. C. Calhoun, S. Di, A. Apon, and F. Cappello, "OptZConfig: Fast parallel optimization of lossy compressor configuration."

[4] ——, "Understanding the effectiveness of applying lossy compression in machine/deep learning applications."

[5] S. Di and F. Cappello, "Fast Error-Bounded Lossy HPC Data Compression with SZ," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2016, pp. 730–739. DOI: 10.1109/IPDPS.2016.11.

[6] P. Lindstrom, "Fixed-Rate Compressed Floating-Point Arrays," *IEEE Transactions on Visualization and Computer Graphics*, vol. 20, no. 12, pp. 2674–2683, Dec. 2014, ISSN: 1077-2626. DOI: 10.1109/TVCG.2014.2346458.

[7] M. Ainsworth, O. Tugluk, B. Whitney, and S. Klasky, "Multilevel techniques for compression and reduction of scientific data—the univariate case," *Computing and Visualization in Science*, vol. 19, no. 5-6, pp. 65–76, Dec. 2018, ISSN: 1432-9360, 1433-0369. (visited on 10/03/2019).

[8] J. Han, J. Pei, and M. Kamber, *Data Mining: Concepts and Techniques*. Elsevier, 2011.

[9] A. H. Baker, H. Xu, D. M. Hammerling, S. Li, and J. P. Clyne, "Toward a Multi-method Approach: Lossy Data Compression for Climate Simulation Data," in *High Performance Computing*, ser. Lecture Notes in Computer Science, J. M. Kunkel, R. Yokota, M. Taufer, and J. Shalf, Eds., vol. 10524, Cham: Springer International Publishing, 2017, pp. 30–42. DOI: 10.1007/978-3-319-67630-2_3. [Online]. Available: http://link.springer.com/10.1007/978-3-319-67630-2_3 (visited on 04/21/2020).

[10] H. Luu, M. Winslett, W. Gropp, R. Ross, P. Carns, K. Harms, M. Prabhat, S. Byna, and Y. Yao, "A Multiplatform Study of I/O Behavior on Petascale Supercomputers," in *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, (Portland, Oregon, USA), ser. HPDC '15, New York, NY, USA: ACM, 2015, pp. 33–44, ISBN: 978-1-4503-3550-8. DOI: 10.1145/2749246.2749269. [Online]. Available: http://doi.acm.org/10.1145/2749246.2749269 (visited on 08/15/2019).

[11] S. Habib, V. Morozov, N. Frontiere, H. Finkel, A. Pope, K. Heitmann, K. Kumaran, V. Vishwanath, T. Peterka, J. Insley, *et al.*, "HACC: Extreme scaling and performance across diverse architectures," *Communications of the ACM*, vol. 60, no. 1, pp. 97–104, 2016.

[12] S. supercomputer at ORNL, https://www.olcf.ornl.gov/summit/, Online.

[13] F. Cappello, S. Di, S. Li, X. Liang, A. M. Gok, D. Tao, C. H. Yoon, X.-C. Wu, Y. Alexeev, and F. T. Chong, "Use cases of lossy compression for floating-point data in scientific data sets," *The International Journal of High Performance Computing Applications*, vol. 33, no. 6, pp. 1201–1220, Nov. 2019, ISSN: 1094-3420, 1741-2846. DOI: 10.1177/1094342019853336. [Online]. Available: http://journals.sagepub.com/doi/10.1177/1094342019853336 (visited on 06/02/2020).

[14] G. K. Wallace, "The JPEG still picture compression standard," *IEEE Transactions on Consumer Electronics*, vol. 38, no. 1, pp. xviii–xxxiv, Feb. 1992, ISSN: 0098-3063. DOI: 10.1109/30.125072.

[15] A. H. Baker, D. M. Hammerling, and T. L. Turton, "Evaluating image quality measures to assess the impact of lossy data compression applied to climate simulation data," *Computer Graphics Forum*, vol. 38, no. 3, pp. 517–528, Jun. 2019. DOI: 10.1111/cgf.13707. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.13707 (visited on 09/03/2019).

[16] X. Liang, S. Di, D. Tao, S. Li, B. Nicolae, Z. Chen, and F. Cappello, "Improving Performance of Data Dumping with Lossy Compression for Scientific Simulation," p. 11, 2019.

[17] J. Calhoun, F. Cappello, L. N. Olson, M. Snir, and W. D. Gropp, "Exploring the feasibility of lossy compression for pde simulations," *The International Journal of High Performance Computing Applications*, vol. 33, no. 2, pp. 397–410, 2019. DOI: 10.1177/1094342018762036.

[18] X.-C. Wu, S. Di, E. M. Dasgupta, F. Cappello, Y. Alexeev, H. Finkel, and F. T. Chong, "Full state quantum circuit simulation by using data compression," in *IEEE/ACM 30th The International Conference for High Performance computing, Networking, Storage and Analysis (IEEE/ACM SC2019)*, 2019, pp. 1–12.

[19] A. M. Gok, S. Di, Y. Alexeev, D. Tao, V. Mironov, X. Liang, and F. Cappello, "PaSTRI: Error-Bounded Lossy Compression for Two-Electron Integrals in Quantum Chemistry," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, Sep. 2018, pp. 1–11. DOI: 10.1109/CLUSTER.2018.00013.

[20] 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 *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2017, pp. 1129–1139. DOI: 10.1109/IPDPS.2017.115.

[21] X. Liang, S. Di, D. Tao, Z. Chen, and F. Cappello, "An Efficient Transformation Scheme for Lossy Data Compression with Point-Wise Relative Error Bound," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, Sep. 2018, pp. 179–189. DOI: 10.1109/CLUSTER.2018.00036.

[22] D. Tao, S. Di, X. Liang, Z. Chen, and F. Cappello, "Fixed-PSNR Lossy Compression for Scientific Data," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, Sep. 2018, pp. 314–318. DOI: 10.1109/CLUSTER.2018.00048.

[23] S. Li, S. Di, X. Liang, Z. Chen, and F. Cappello, "Optimizing Lossy Compression with Adjacent Snapshots for N-body Simulation Data," in *2018 IEEE International Conference on Big Data (Big Data)*, Dec. 2018, pp. 428–437. DOI: 10.1109/BigData.2018.8622101.

[24] P. Lindstrom, "Error Distributions of Lossy Floating-Point Compressors," *Joint Statistical Meetings*, 2017.

[25] I. Facebook. (). "Zstandard - Real-time data compression algorithm," [Online]. Available: https://facebook.github.io/zstd/ (visited on 06/25/2019).

[26] M. Ainsworth, O. Tugluk, B. Whitney, and S. Klasky, "Multilevel techniques for compression and reduction of scientific data—quantitative control of accuracy inderived quantities," *SIAM Journal on Scientific Computing*, vol. 41, no. 4, A2146–A2171, Jan. 2019, ISSN: 1064-8275, 1095-7197. (visited on 10/03/2019).

[27] ——, "Multilevel techniques for compression and reduction of scientific data—the multivariate case," *SIAM Journal on Scientific Computing*, vol. 41, A1278–A1303, Jan. 2019, ISSN: 1064-8275, 1095-7197. DOI: 10.1137/18M1166651. (visited on 10/03/2019).

[28] U. Trottenberg and A. Schuller, *Multigrid*. Orlando, FL: Academic Press, Inc., 2001, ISBN: 0-12-701070-X.

[29] P. Grosset, C. M. Biwer, J. Pulido, A. T. Mohan, A. Biswas, J. Patchett, T. L. Turton, D. H. Rogers, D. Livescu, and J. Ahrens, "Foresight: Analysis That Matters for Data Reduction," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov. 2020, pp. 1–15. DOI: 10.1109/SC41405.2020.00087.

[30] J. Kunkel, A. Novikova, E. Betke, and A. Schaare, "Toward Decoupling the Selection of Compression Algorithms from Quality Constraints," en, in *High Performance Computing*, J. M. Kunkel, R. Yokota, M. Taufer, and J. Shalf, Eds., vol. 10524, Cham: Springer International Publishing, 2017, pp. 3–14. DOI: 10.1007/978-3-319-67630-2_1.

[31] 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 *2018 IEEE International Conference on Big Data (Big Data)*, Seattle, WA, USA: IEEE, Dec. 2018, pp. 438–447, ISBN: 978-1-5386-5035-6. DOI: 10.1109/BigData.2018.8622520. [Online]. Available: https://ieeexplore.ieee.org/document/8622520/ (visited on 06/17/2019).

[32] 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.

[33] M. Schulz, W. Gropp, R. Rabenseifner, R. L. Graham, J. M. Squyres, D. Holmes, G. Bosilca, T. Hoefler, P. Balaji, J. Hammond, D. Solt, Q. Koziol, K. Mohror, and R. Thakur, Eds., *MPI: A Message-Passing Interface Standard*, en, Jun. 2015.

[34] J. Tian, S. Di, K. Zhao, C. Rivera, M. H. Fulp, R. Underwood, S. Jin, X. Liang, J. Calhoun, D. Tao, and F. Cappello, "cuSZ: An Efficient GPU-Based Error-Bounded Lossy Compression Framework for Scientific Data," en, in *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, Virtual Event GA USA: ACM, Sep. 2020, pp. 3–15, ISBN: 978-1-4503-8075-1. DOI: 10.1145/3410463.3414624.

[35] J. Tian, S. Di, C. Zhang, X. Liang, S. Jin, D. Cheng, D. Tao, and F. Cappello, "waveSZ: A hardware-algorithm co-design of efficient lossy compression for scientific data," in *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego California: ACM, Feb. 19, 2020, pp. 74–88, ISBN: 978-1-4503-6818-6. DOI: 10.1145/3332466.3374525. [Online]. Available: https://dl.acm.org/doi/10.1145/3332466.3374525 (visited on 06/02/2020).

[36] W. F. Godoy, N. Podhorszki, R. Wang, C. Atkins, G. Eisenhauer, J. Gu, P. Davis, J. Choi, K. Germaschewski, K. Huck, A. Huebl, M. Kim, J. Kress, T. Kurc, Q. Liu, J. Logan, K. Mehta, G. Ostrouchov, M. Parashar, F. Poeschel, D. Pugmire, E. Suchyta, K. Takahashi, N. Thompson, S. Tsutsumi, L. Wan, M. Wolf, K. Wu, and S. Klasky, "Adios 2: The adaptable input output system. a framework for high-performance data management," *SoftwareX*, vol. 12, p. 100 561, 2020, ISSN: 2352-7110. DOI: https://doi.org/10.1016/j.softx.2020.100561. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2352711019302560.

[37] D. Tao, S. Di, H. Guo, Z. Chen, and F. Cappello, "Z-Checker: A framework for assessing lossy compression of scientific data," *The International Journal of High Performance Computing Applications*, vol. 33, no. 2, pp. 285–303, Mar. 2019, ISSN: 1094-3420, 1741-2846. DOI: 10.1177/1094342017737147. [Online]. Available: http://journals.sagepub.com/doi/10.1177/1094342017737147 (visited on 06/25/2019).

[38] S. Di, *Error-bounded Lossy Data Compressor (for floating-point/integer datasets): Disheng222/SZ*, Jun. 26, 2019. [Online]. Available: https://github.com/disheng222/SZ (visited on 06/27/2019).

[39] A. Sengupta, *Zfp_jll · JuliaHub*, https://juliahub.com/ui/Packages/zfp_jll/DIPUA/0.5.5+0.

[40] N. Kukreja, T. Greaves, G. Gorman, and D. Wade, *Pyzfp*, 2018.

[41] C. Zapart, *Zfp-sys - crates.io: Rust Package Registry*, https://crates.io/crates/zfp-sys.

[42] K. Zhao, S. Di, X. Liang, S. Li, D. Tao, Z. Chen, and F. Cappello, "Significantly Improving Lossy Compression for HPC Datasets with Second-Order Prediction and Parameter Optimization," en, in *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*, Stockholm Sweden: ACM, Jun. 2020, pp. 89–100, ISBN: 978-1-4503-7052-3. DOI: 10.1145/3369583.3392688.

[43] T. McCabe, "A Complexity Measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, Dec. 1976, ISSN: 1939-3520. DOI: 10.1109/TSE.1976.233837.

[44] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," *Soviet physics doklady*, vol. 10, no. 8, pp. 707–710, 1966.

[45] T. Gamblin, M. LeGendre, M. R. Collette, G. L. Lee, A. Moody, B. R. de Supinski, and S. Futral, "The Spack package manager: Bringing order to HPC software chaos," en, in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Austin Texas: ACM, Nov. 2015, pp. 1–12, ISBN: 978-1-4503-3723-6. DOI: 10.1145/2807591.2807623.

[46] Scientific Data Reduction Benchmark, https://sdrbench.github.io/, Online.

[47] NYX simulation, https://amrex-astro.github.io/Nyx, Online.

[48] J. Kay, C. Deser, A. Phillips, A. Mai, C. Hannay, G. Strand, J. Arblaster, S. Bates, G. Danabasoglu, J. Edwards, *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.

[49] I. Foster, M. Ainsworth, B. Allen, J. Bessac, F. Cappello, J. Y. Choi, E. Constantinescu, P. E. Davis, S. Di, W. Di, *et al.*, "Computing just what you need: Online data analysis and reduction at extreme scales," in *European Conference on Parallel Processing*, 2017, pp. 3–19.

[50] Zstd, https://github.com/facebook/zstd/releases, Online.

[51] Gzip, https://www.gzip.org/, Online.

[52] D. King. (Mar. 6, 2018). "Dlib C++ Library - Optimization," [Online]. Available: http://dlib.net/optimization.html#global_function_search (visited on 09/03/2019).

[53] D. Tao, S. Di, X. Liang, Z. Chen, and F. Cappello, "Optimizing Lossy Compression Rate-Distortion from Automatic Online Selection between SZ and ZFP," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 8, pp. 1857–1871, Aug. 2019, ISSN: 1045-9219. DOI: 10.1109/TPDS.2019.2894404.

[54] Bebop, https://www.lcrc.anl.gov/systems/resources/bebop, Online.

[55] N. Sasaki, K. Sato, T. Endo, and S. Matsuoka, "Exploration of lossy compression for application-level checkpoint/restart," in *Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium*, ser. IPDPS '15, Washington, DC, USA: IEEE Computer Society, 2015, pp. 914–922, ISBN: 978-1-4799-8649-1. DOI: 10.1109/IPDPS.2015.67. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2015.67.

[56] P. Triantafyllides, T. Reza, and J. C. Calhoun, "Analyzing the impact of lossy compressor variability on checkpointing scientific simulations," in *In the Proceedings of the 2019 IEEE International Conference on Cluster Computing*, ser. Cluster '19, Washington, DC, USA: IEEE Computer Society, 2019.

[57] Spring8, http://www.spring8.or.jp/en/, Online, 2019.

[58] S. Kim, M. Kim, J. Kim, and H. Lee, "Fixed-Ratio Compression of an RGBW Image and Its Hardware Implementation," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 6, no. 4, pp. 484–496, Dec. 2016.

[59] C. A. Andrews, J. M. Davies, and G. R. Schwarz, "Adaptive data compression," *Proceedings of the IEEE*, vol. 55, no. 3, pp. 267–277, Mar. 1967. DOI: 10.1109/PROC.1967.5481.

[60] *Libpressio*, Codesign Center for Online Data Analysis and Reduction, Sep. 20, 2019. [Online]. Available: https://github.com/CODARcode/libpressio (visited on 09/26/2019).

[61] A. Fischer, "A special Newton-type optimization method," *Optimization*, vol. 24, no. 3-4, pp. 269–284, 1992.

[62] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," Dec. 22, 2014. arXiv: 1412.6980 [cs]. [Online]. Available: http://arxiv.org/abs/1412.6980 (visited on 10/11/2019).

[63] M. J. Powell, "The BOBYQA algorithm for bound constrained optimization without derivatives," *Cambridge NA Report NA2009/06, University of Cambridge, Cambridge*, pp. 26–46, 2009.

[64] M. J. Saltzman, "COIN-OR: An open-source library for optimization," in *Programming languages and systems in computational economics and finance*, Springer, 2002, pp. 3–32.

[65] G. Gabriele and K. Ragsdell, "OPTLIB: An optimization program library," *Mechanical Engineering Modern Design Series*, vol. 4, 1977.

[66] C. Malherbe and N. Vayatis, "Global optimization of Lipschitz functions," Mar. 7, 2017. arXiv: 1703.02628 [stat]. [Online]. Available: http://arxiv.org/abs/1703.02628 (visited on 09/26/2019).

[67] M. J. D. Powell, "The NEWUOA software for unconstrained optimization without derivatives," in *Large-Scale Nonlinear Optimization*, G. Di Pillo and M. Roma, Eds., red. by P. Pardalos, vol. 83, Boston, MA: Springer US, 2006, pp. 255–297. DOI: 10.1007/0-387-30065-1_16. [Online]. Available: http://link.springer.com/10.1007/0-387-30065-1_16 (visited on 09/26/2019).

[68] Zhou Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli, "Image quality assessment: From error visibility to structural similarity," *IEEE Transactions on Image Processing*, vol. 13, no. 4, pp. 600–612, Apr. 2004, ISSN: 1941-0042. DOI: 10.1109/TIP.2003.819861.

[69] L. P. Deutsch, "RFC 1952 GZIP File Format Specification version 4.3," 1996. [Online]. Available: http://www.zlib.org/rfc-gzip.html (visited on 06/25/2019).

[70] M. Burtscher and P. Ratanaworabhan, "FPC: A high-speed compressor for double-precision floating-point data," *IEEE Transactions on Computers*, vol. 58, no. 1, pp. 18–31, Jan. 2009, ISSN: 1557-9956. DOI: 10.1109/TC.2008.131.

[71] S. F. Shandarin and N. S. Ramachandra, "Topology and geometry of the dark matter web: a multistream view," *Monthly Notices of the Royal Astronomical Society*, vol. 467, no. 2, pp. 1748–1762, Jan. 2017.

[72] ——, "Dark matter haloes: a multistream view," *Monthly Notices of the Royal Astronomical Society*, vol. 470, no. 3, pp. 3359–3373, Jun. 2017.

[73] J. L. H. Jr., *The Signifigance Probablity of the Smirnov Two Sample Test*, 43. Arkiv fiur Matematik, 1958, vol. 3, pp. 469–486.

[74] F. Cappello, M. Ainsworth, J. Bessac, M. Burtscher, J. Y. Choi, E. Constantinescu, S. Di, H. Guo, P. Lindstrom, and O. Tugluk. (Jun. 18, 2018). "Scientific Data Reduction Benchmarks," [Online]. Available: `https://sdrbench.github.io/` (visited on 06/02/2020).

[75] A. Fox, J. Diffenderfer, J. Hittinger, G. Sanders, and P. Lindstrom, "Stability Analysis of Inline ZFP Compression for Floating-Point Data in Iterative Methods," *SIAM Journal on Scientific Computing*, vol. 42, no. 5, A2701–A2730, Jan. 2020, ISSN: 1064-8275, 1095-7197. DOI: `10.1137/19M126904X`. [Online]. Available: `https://epubs.siam.org/doi/10.1137/19M126904X` (visited on 01/23/2021).

[76] R. Underwood, *CODARcode/libpressio*, Codesign Center for Online Data Analysis and Reduction, Sep. 20, 2019. [Online]. Available: `https://github.com/CODARcode/libpressio` (visited on 09/26/2019).

[77] A. Zhigljavsky, A. Zilinskas, and J. Birge, *Stochastic Global Optimization*. New York, NY, UNITED STATES: Springer, 2007, ISBN: 978-0-387-74740-8.

[78] "Covering methods," en, in *Global Optimization*, ser. Lecture Notes in Computer Science, A. Törn and A. Žilinskas, Eds., Berlin, Heidelberg: Springer, 1989, pp. 25–52, ISBN: 978-3-540-46103-6. DOI: `10.1007/3-540-50871-6_2`.

[79] A. Malakhov, "Composable multi-threading for Python libraries," presented at the Python in Science Conference, Austin, Texas, 2016, pp. 15–19. (visited on 10/17/2020).

[80] M. P. Matijkiw and M. M. K. Martin, "Exploring coordination of threads in multi-core libraries," p. 8, 2010.

[81] H. Ribic and Y. D. Liu, "AEQUITAS: Coordinated energy management across parallel applications," in *Proceedings of the 2016 International Conference on Supercomputing - ICS '16*, Istanbul, Turkey: ACM Press, 2016, pp. 1–12. (visited on 10/17/2020).

[82] Y. Tanaka, K. Taura, M. Sato, and A. Yonezawa, "Performance evaluation of OpenMP applications with nested parallelism," *5th International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, p. 14, May 1, 2000.

[83] C. lMaxr Jiang, S. Esmaeilzadeh, K. Azizzadenesheli, K. Kashinath, M. Mustafa, H. A. Tchelepi, P. Marcus, M. Prabhat, and A. Anandkumar, "MESHFREEFLOWNET: A Physics-Constrained Deep Continuous Space-Time Super-Resolution Framework," en, in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, Atlanta, GA, USA: IEEE, Nov. 2020, pp. 1–15, ISBN: 978-1-72819-998-6. DOI: `10.1109/SC41405.2020.00013`.

[84] W. Jia, H. Wang, M. Chen, D. Lu, L. Lin, R. Car, E. Weinan, and L. Zhang, "Pushing the Limit of Molecular Dynamics with Ab Initio Accuracy to 100 Million Atoms with Machine Learning," en, in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, Atlanta, GA, USA: IEEE, Nov. 2020, pp. 1–14, ISBN: 978-1-72819-998-6. DOI: `10.1109/SC41405.2020.00009`.

[85] S. Babu, M. Garofalakis, and B. Laboratories, "Model-Based Semantic Compression for Network-Data Tables," p. 7,

[86] A. Srivastava, S. P. Chockalingam, and S. Aluru, "A Parallel Framework for Constraint-Based Bayesian Network Learning via Markov Blanket Discovery," en, in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, Atlanta, GA, USA: IEEE, Nov. 2020, pp. 1–15, ISBN: 978-1-72819-998-6. DOI: 10.1109/SC41405.2020.00011.

[87] J. M. Wozniak, R. Jain, P. Balaprakash, J. Ozik, N. T. Collier, J. Bauer, F. Xia, T. Brettin, R. Stevens, J. Mohd-Yusof, C. G. Cardona, B. V. Essen, and M. Baughman, "CANDLE/Supervisor: A workflow framework for machine learning applied to cancer research," *BMC Bioinformatics*, vol. 19, no. S18, p. 491, Dec. 2018, ISSN: 1471-2105. DOI: 10.1186/s12859-018-2508-4. [Online]. Available: https://bmcbioinformatics.biomedcentral.com/articles/10.1186/s12859-018-2508-4 (visited on 06/25/2019).

[88] M. Isakov, E. del Rosario, S. Madireddy, P. Balaprakash, P. Carns, R. B. Ross, and M. A. Kinsy, "HPC I/O Throughput Bottleneck Analysis with Explainable Local Models," en, in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, Atlanta, GA, USA: IEEE, Nov. 2020, pp. 1–13, ISBN: 978-1-72819-998-6. DOI: 10.1109/SC41405.2020.00037.

[89] G. Marcus, Y. Ding, P. Emma, Z. Huang, J. Qiang, T. Raubenheimer, M. Venturini, and L. Wang, "High Fidelity Start-to-end Numerical Particle Simulations and Performance Studies for LCLS-II," in *Proceedings, 37th International Free Electron Laser Conference (FEL 2015): Daejeon, Korea, August 23-28, 2015*, 2015.

[90] T. E. Fornek, "Advanced photon source upgrade project preliminary design report," Sep. 2017. DOI: 10.2172/1423830.

[91] *Globus*, http://globus.org.

[92] W. Locke, J. Sybrandt, L. Redmond, I. Safro, and S. Atamturktur, "Using drive-by health monitoring to detect bridge damage considering environmental and operational effects," en, *Journal of Sound and Vibration*, vol. 468, p. 115 088, Mar. 2020, ISSN: 0022460X. DOI: 10.1016/j.jsv.2019.115088.

[93] M. Rahman, M. Islam, J. C. Calhoun, and M. Chowdhury, "Dynamic Error-bounded Lossy Compression (EBLC) to Reduce the Bandwidth Requirement for Real-time Vision- based Pedestrian Safety Applications," en, *IEEE TRANSACTIONS ON INTELLIGENT TRANSPORTATION SYSTEMS*, p. 10,

[94] R. Kettimuthu, Z. Liu, D. Wheeler, I. Foster, K. Heitmann, and F. Cappello, "Transferring a petabyte in a day," en, *Future Generation Computer Systems*, vol. 88, pp. 191–198, Nov. 2018, ISSN: 0167-739X. DOI: 10.1016/j.future.2018.05.051.

[95] *The Cost of Hard Drives Over Time*, en-US, Jul. 2017.

[96] S. Habib, A. Pope, H. Finkel, N. Frontiere, K. Heitmann, D. Daniel, P. Fasel, V. Morozov, G. Zagaris, T. Peterka, V. Vishwanath, Z. Lukić, S. Sehrish, and W.-k. Liao, "HACC: Simulating sky surveys on state-of-the-art supercomputing architectures," en, *New Astronomy*, vol. 42, pp. 49–65, Jan. 2016, ISSN: 1384-1076. DOI: 10.1016/j.newast.2015.06.003.

[97] A. S. Almgren, J. B. Bell, M. J. Lijewski, Z. Lukić, and E. Van Andel, "Nyx: A MASSIVELY PARALLEL AMR CODE FOR COMPUTATIONAL COSMOLOGY," en, *The Astrophysical Journal*, vol. 765, no. 1, p. 39, Feb. 2013, ISSN: 0004-637X, 1538-4357. DOI: 10.1088/0004-637X/765/1/39.

[98] S. Plimpton, A. Thompson, S. Moore, A. Kohlmeyer, and R. Berger, *LAMMPS Molecular Dynamics Simulator*, Sandia National Labs and Temple University, 2004.

[99] S. Jalali and A. Maleki, "From compression to compressed sensing," Dec. 17, 2012. arXiv: 1212.4210 [cs, math]. [Online]. Available: http://arxiv.org/abs/1212.4210 (visited on 07/03/2019).

[100] L. P. Deutsch, "RFC 1951 DEFLATE Compressed Data Format Specification ver 1.3," 1996. [Online]. Available: https://www.w3.org/Graphics/PNG/RFC-1951 (visited on 06/25/2019).

[101] M. Folk, G. Heber, Q. Koziol, E. Pourmal, and D. Robinson, "An overview of the HDF5 technology suite and its applications," in *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*, ser. AD '11, New York, NY, USA: Association for Computing Machinery, Mar. 2011, pp. 36–47, ISBN: 978-1-4503-0614-0. DOI: 10.1145/1966895.1966900.

[102] W. F. Godoy, N. Podhorszki, R. Wang, C. Atkins, G. Eisenhauer, J. Gu, P. Davis, J. Choi, K. Germaschewski, K. Huck, A. Huebl, M. Kim, J. Kress, T. Kurc, Q. Liu, J. Logan, K. Mehta, G. Ostrouchov, M. Parashar, F. Poeschel, D. Pugmire, E. Suchyta, K. Takahashi, N. Thompson, S. Tsutsumi, L. Wan, M. Wolf, K. Wu, and S. Klasky, "ADIOS 2: The Adaptable Input Output System. A framework for high-performance data management," en, *SoftwareX*, vol. 12, p. 100 561, Jul. 2020, ISSN: 2352-7110. DOI: 10.1016/j.softx.2020.100561.

[103] C. Shannon and W. Weaver, "The Mathematical Theory of Communication," p. 131, 1948.

[104] Y. Collet. (). "LZ4 - Extremely fast compression," [Online]. Available: https://lz4.github.io/lz4/ (visited on 06/25/2019).

[105] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on Information Theory*, vol. 23, no. 3, pp. 337–343, May 1977, ISSN: 0018-9448. DOI: 10.1109/TIT.1977.1055714.

[106] D. A. Huffman, "A Method for the Construction of Minimum-Redundancy Codes," *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, Sep. 1952, ISSN: 0096-8390. DOI: 10.1109/JRPROC.1952.273898.

[107] Y. Collet, *Finite State Entropy*.

[108] D. Zuras, M. Cowlishaw, A. Aiken, M. Applegate, D. Bailey, S. Bass, D. Bhandarkar, M. Bhat, D. Bindel, S. Boldo, *et al.*, "IEEE standard for floating-point arithmetic," *IEEE Std 754-2008*, pp. 1–70, 2008.

[109] P. Lindstrom, *Library for compressed numerical arrays that support high throughput read and write random access: LLNL/zfp*, Lawrence Livermore National Laboratory, Jun. 27, 2019. [Online]. Available: https://github.com/LLNL/zfp (visited on 06/27/2019).

[110] B. Shan, A. Shamji, J. Tian, G. Li, and D. Tao, "LCFI: A Fault Injection Tool for Studying Lossy Compression Error Propagation in HPC Programs," en, *arXiv:2010.12746 [cs]*, Nov. 2020. arXiv: 2010.12746 [cs].

[111] C. Bradley, N. Emamy, T. Ertl, D. Göddeke, A. Hessenthaler, T. Klotz, A. Krämer, M. Krone, B. Maier, M. Mehl, T. Rau, and O. Röhrle, "Towards realistic HPC models of the neuromuscular system," en, *arXiv:1802.03211 [physics]*, Feb. 2018. arXiv: 1802.03211 [physics].

[112] J. Zhang, X. Zhuo, A. Moon, H. Liu, and S. W. Son, "Efficient Encoding and Reconstruction of HPC Datasets for Checkpoint/Restart," in *2019 35th Symposium on Mass Storage Systems and Technologies (MSST)*, May 2019, pp. 79–91. DOI: 10.1109/MSST.2019.00-14.

[113] E. Sadrfaridpour, T. Razzaghi, and I. Safro, "Engineering fast multilevel support vector machines," *Machine Learning*, vol. 108, no. 11, pp. 1879–1917, Nov. 2019, ISSN: 0885-6125, 1573-0565. DOI: 10.1007/s10994-019-05800-7. [Online]. Available: http://link.springer.com/10.1007/s10994-019-05800-7 (visited on 03/02/2020).

[114] S. Haberman, "Generalized residuals for log-linear models," presented at the Proceedings of the 9th International Biometrics Conference, 1976, pp. 104–122.

[115] R. B. Bhatt, G. Sharma, A. Dhall, and S. Chaudhury, "Efficient Skin Region Segmentation Using Low Complexity Fuzzy Decision Tree Model," in *2009 Annual IEEE India Conference*, Dec. 2009, pp. 1–4. DOI: 10.1109/INDCON.2009.5409447.

[116] T. Dietterich, *Musk (Clean 1) Database*. 1994.

[117] I.-C. Yeh, *Real Estate Valuation Dataset*. 2018.

[118] ——, *Concrete Compressive Dataset*. 2007.

[119] F. Brooks, D. Stuart, and A. Marcolini, "Airfoil Self-Noise and Prediction," p. 146,

[120] K. Hamidieh, "A Data Driven Statistical Model to Predict Critical Temperature of Superconducting Material," in *APS Meeting Abstracts*, 2018, p. X34.013.

[121] M. J. Cherukara, T. Zhou, Y. Nashed, P. Enfedaque, A. Hexemer, R. J. Harder, and M. V. Holt, "AI-enabled high-resolution scanning coherent diffraction imaging," *Applied Physics Letters*, vol. 117, no. 4, p. 044 103, Jul. 2020, ISSN: 0003-6951. DOI: 10.1063/5.0013065.

[122] M. Salim, T. Uram, J. T. Childers, V. Vishwanath, and M. Papka, "Balsam: Near Real-Time Experimental Data Analysis on Supercomputers," in *2019 IEEE/ACM 1st Annual Workshop on Large-Scale Experiment-in-the-Loop Computing (XLOOP)*, Nov. 2019, pp. 26–31. DOI: 10.1109/XLOOP49562.2019.00010.

[123] M. Rabbani and R. Joshi, "An overview of the JPEG 2000 still image compression standard," *Signal Processing: Image Communication*, vol. 17, no. 1, pp. 3–48, Jan. 2002, ISSN: 09235965. DOI: 10.1016/S0923-5965(01)00024-8. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0923596501000248 (visited on 06/17/2019).

[124] (). "BPG Image format," [Online]. Available: https://bellard.org/bpg/ (visited on 07/03/2019).

[125] (). "Compression Techniques — WebP," [Online]. Available: https://developers.google.com/speed/webp/docs/compression (visited on 07/03/2019).

[126] A. Y. Yang, J. Wright, Y. Ma, and S. S. Sastry, "Unsupervised segmentation of natural images via lossy data compression," *Computer Vision and Image Understanding*, vol. 110, no. 2, pp. 212–225, May 2008, ISSN: 10773142. DOI: 10.1016/j.cviu.2007.07.005. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S1077314207001300 (visited on 07/03/2019).

[127] H. Jagadish, J. Madar, and R. T. Ng, "Semantic Compression and Pattern Extraction with Fascicles," in *Proceedings of the 25th VLDB Conference*, 1999. [Online]. Available: https://www.researchgate.net/profile/Raymond_Ng3/publication/221309718_Semantic_Compression_and_Pattern_Extraction_with_Fascicles/links/00b495228937de512e000000.pdf (visited on 07/03/2019).

[128] M. Garofalakis and R. Rastogi, "Data Mining Meets Network Management: The NEMESIS Project," p. 6,

[129] R. D. Evans, L. Liu, and T. M. Aamodt, "JPEG-ACT: Accelerating Deep Learning via Transform-based Lossy Compression," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, May 2020, pp. 860–873. DOI: 10.1109/ISCA45697.2020.00075.

[130] S. Jin, G. Li, S. L. Song, and D. Tao, "A Novel Memory-Efficient Deep Learning Training Framework via Error-Bounded Lossy Compression," en, *arXiv:2011.09017 [cs]*, Nov. 2020. arXiv: 2011.09017 [cs].