TECHNISCHE UNIVERSITÄT
BERGAKADEMIE FREIBERG
Die Ressourcenuniversität. Seit 1765.

Temporal Lossy In-Situ Compression for Computational Fluid Dynamics Simulations

Von der Fakultät für Mathematik und Informatik
der Technischen Universität Bergakademie Freiberg

genehmigte

DISSERTATION

zur Erlangung des akademischen Grades

Doktor-Ingenieur

(Dr.-Ing.)

vorgelegt

von **M.Sc. Henry Lehmann**

geboren am 24. Oktober 1984 in Grimma

Gutachter: Prof. Dr.-Ing. habil. Bernhard Jung, Freiberg, Deutschland
Prof. Dr.-Ing. Gerik Scheuermann, Leipzig, Deutschland

Tag der Verleihung: Freitag, 13. Juli 2018

Versicherung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Das aus fremden Quellen direkt oder indirekt übernommene Gedankengut ist als solches kenntlich gemacht. Bei der Auswahl und Auswertung des Materials sowie bei der Herstellung des Manuskripts habe ich keine Unterstützungsleistungen erhalten. Die Hilfe eines Promotionsberaters habe ich nicht in Anspruch genommen. Weitere Personen haben von mir keine geldwerten Leistungen für Arbeiten erhalten, die nicht als solche kenntlich gemacht worden sind. Die Arbeit wurde bisher weder im Inland noch im Ausland in gleicher oder ähnlicher Form einer anderen Prüfungsbehörde vorgelegt.

Freiberg, den 05. Dezember 2017

Henry Lehmann

Nothing determines who we will become so much
as those things we choose to ignore.
— SANDOR MCNAB

Acknowledgments

It was a great honor for me to work on my dissertation while participating in the Collaborative Research Center 920 at the Freiberg University of Mining and Technology. With great respect I look back to the last years, in which Prof. Dr. Bernhard Jung gave me his trust and attention during the supervision of the research project. Furthermore, my appreciation goes to the entire working group for Virtual Reality and Multimedia, as well as the entire Institute of Computer Science, for the friendly and helpful environment with colleagues always interested and motivated. Furthermore, I thank Prof. Dr. Gerik Scheuermann for making himself available as second reviewer of my dissertation, and Eric Werzner for the interesting discussions and the close cooperation during the project. I wish to thank my girlfriend and my whole family, who have always supported me with love, patience and understanding beyond compare in all life situations.

Danksagung

Mit großer Ehre habe ich die vorliegende Arbeit während meiner Tätigkeit als Doktorand im Sonderforschungsbereich 920 an der Technischen Universität Bergakademie Freiberg angefertigt. Mit großer Achtung schaue ich auf die letzten Jahre zurück, in denen Prof. Dr. Bernhard Jung mir sein Vertrauen und seine Aufmerksamkeit während der Betreuung meines Promotionsvorhabens geschenkt hat. Weiter gilt meine Erkenntlichkeit der gesamten Arbeitsgruppe für Virtuelle Realität und Multimedia, sowie dem ganzen Institut für Informatik, für das freundliche und hilfsbereite Umfeld mit stets interessierten und motivierten Kollegen. Des Weiteren danke ich Herrn Prof. Dr. Gerik Scheuermann dafür, dass er sich als Zweitgutachter meiner Arbeit zur Verfügung gestellt hat, und Eric Werzner für die interessanten Diskussionen und die gewissenhafte Zusammenarbeit während des Forschungsvorhabens. Ganz herzlich danke ich meiner Freundin und meiner Familie, die mich zu jeder Zeit mit sehr viel Liebe, Geduld und Verständnis ohnegleichen in allen Lebenslagen unterstützt haben.

Table of Contents

Table of Contents **i**

List of Publications **vi**

List of Acronyms **ix**

List of Figures **ix**

List of Tables **xiii**

List of Algorithms **xvi**

1 Introduction **1**

1.1 Motivation 1

1.1.1 High-Performance Computing 1

1.1.2 Computational Fluid Dynamics 2

1.1.3 In-Situ Processing Techniques 3

1.2 Research Objectives 5

1.3 Outline of Thesis 7

2 Lossless Compression of Scientific Floating Point Data **9**

2.1 Floating Point Data in Simulation Grids 9

2.2 Linearization of Simulation Grids 13

2.3 General Purpose Lossless Compression 16

2.4 Compression of Scientific Data Sets 21

2.5 Summary 28

3 Lossy In-Situ Compression for Scientific Floating Point Data **29**

3.1 Motivation for Lossy Compression 29

3.2 Error Correction Policy 30
3.3 Lossy Float Compression using Scalar Quantization 33
3.3.1 In-situ Sort-And-B-spline Error-bounded Lossy Abatement 33
3.3.2 Set Based Decomposition with Sequential Encoding 40
3.3.3 Grid Linearization And Truncation Encoding 45
3.4 Evaluation of Lossy Float Compression on Scientific Data Sets 55
3.4.1 ISABELA Compression 55
3.4.2 SBD Compression 61
3.4.3 GLATE Compression 66
3.5 Summary 73

4 Temporal In-Situ Compression for Scientific Floating Point Data 79
4.1 Motivation for Temporal Compression 79
4.2 Temporal Compression Procedure 82
4.3 Lossy Temporal Compression Schemes 83
4.3.1 Differential ISABELA Compression 84
4.3.2 Temporal ISABELA Compression 90
4.3.3 Temporal SBD Compression 96
4.3.4 Temporal GLATE Compression 102
4.3.5 Optimization of t-GLATE Keyframe Compression 110
4.4 Temporal Compression of Scientific Data Sets 113
4.4.1 d-ISABELA Compression 114
4.4.2 t-ISABELA Compression 115
4.4.3 t-SBD Compression 117
4.4.4 t-GLATE Compression 121
4.5 Comparison to ZFP Lossy Floating Point Compressor 123
4.6 Summary 131

5 Application of In-Situ Compression in a Computational Fluid Dynamics Simulation 137
5.1 Motivation for Lossy In-Situ Compression 137
5.2 Simulation of Metal Melt Flow in a Complex Filter Structure 139
5.3 Storage of In-Situ Compressed Data 142
5.3.1 Collective I/O in parallel HPC Environments 143

5.3.2 Column-Major Storage Layout for Uncompressed Grid Data . . . 144
5.3.3 Sequential Storage Layout for Compressed Grid Data 147
5.3.4 Storage Layout for Compressed Low-Resolution Data 151
5.4 In-Situ Tests in the Taurus HPC Cluster 154
5.4.1 In-Situ Compression using GLATE 158
5.4.2 Temporal In-Situ Compression using t-GLATE 164
5.5 Summary 167

6 Conclusion and Future Work 177
6.1 Research Summary 177
6.2 Research Contributions 180
6.3 Future Work 184
6.3.1 Tool Integration for Temporal Compression 184
6.3.2 Selective Decompression using In-situ generated Index 184

List of Publications

- Henry Lehmann and Bernard Jung. “In-situ data compression of Flow in pourous media”. In: *PDPTA’12 - 18th The International Conference on Parallel and Distributed Processing Techniques and Applications.* 2012

- Henry Lehmann and Bernard Jung. “Applying In-Situ Compression to Hierarchical Scientific Voxel Data”. In: vol. 1. EDIS - Publishing Institution of the University of Zilina, 2012, pp. 1953–1958. ISBN: 978-80-554-0606-0

- Henry Lehmann, Katja Fiedler, and Bernhard Jung. “Processing In-Situ Compressed Large Data Sets in VR-based Flow Analysis”. In: *Virtuelle und Erweiterte Realität - 9. Workshop der GI-Fachgruppe VR/AR 2012, Düsseldorf, Deutschland, September 19-20, 2012. Proceedings.* Ed. by Christian Geiger, Jens Herder, and Tom Vierjahn. Shaker, 2012, pp. 61–70. ISBN: 978-3-8440-1309-2

- Henry Lehmann et al. “In Situ Data Compression Algorithm for Detailed Numerical Simulation of Liquid Metal Filtration through Regularly Structured Porous Media”. In: *Advanced Engineering Materials* 15.12 (2013), pp. 1260–1269. ISSN: 1527-2648

- Henry Lehmann and Bernhard Jung. “In-situ multi-resolution and temporal data compression for visual exploration of large-scale scientific simulations”. In: *4th IEEE Symposium on Large Data Analysis and Visualization, LDAV 2014, Paris, France, November 9-10, 2014.* 2014, pp. 51–58

- Henry Lehmann, Matthias Lenk, and Bernhard Jung. “Adding Interlacing to In-Situ Data Compression for Multi-Resolution Visualization of Large-Scale Scientific Simulations”. In: *ICAT-EGVE 2014 - Posters and Demos.* Ed. by Yuki Hashimoto et al. The Eurographics Association, 2014. ISBN: 978-3-905674-77-4

- Henry Lehmann, Eric Werzner, and Christian Degenkolb. “Optimizing In-situ Data Compression for Large-scale Scientific Simulations”. In: *Proceedings of the 24th High Performance Computing Symposium*. HPC ’16. Pasadena, California: Society for Computer Simulation International, 2016, 5:1–5:8. ISBN: 978-1-5108-2318-1

List of Acronyms

$\%_{\texttt{32B}}$	*Percent of* 32 bit `FLOAT`
$\%_{\texttt{64B}}$	*Percent of* 64 bit `DOUBLE`
`ALUR`	*Aluminum Melt Data Set, Turbulent Region*
`BZIP`	*General Purpose Lossless Compressor*
CFD	*Computational Fluid Dynamics*
CPU	*Central Processing Unit*
CRC920	*Collaborative Research Center* 920
`CUBC`	*Aluminum Melt in Cubic Cell, Cropped Center Region*
DF	*Difference Frame*
d-ISABELA	*Differential ISABELA*
`DOUBLE`	*Double Precision* 64 bit *Floating Point*
e_1	*Smallest Number ensured to conform Maximum Relative Error*
e_{MAX}	*Maximum Relative Error*
e_0	*Smallest Number not blanked to Zero*
`FLOAT`	*Single Precision* 32 bit *Floating Point*
FLOPS	*Floating Point Operations Per Second*
GLATE	*Grid Linearization And Truncation Encoding*
GPLC	*General Purpose Lossless Compressor*
GPU	*Graphic Processing Unit*
HPC	*High Performance Computing*
`INTEGER`	32 bit *Signed Integer*
I/O	*Input/Output*
IOPS	*Input/Output operations Per Second*
ISABELA	*In-situ Sort-And-B-spline Error-bounded Lossy Abatement*
`ISOR`	*Isotropic Turbulence Data Set, Turbulent Region*
k_{D}	*Temporal Compression Write-Out Step Width*

k_{MOD}	*Temporal Compression* KF *Insertion Rate*
KF	*Key Frame*
LBM	*lattice-Boltzmann method*
LUT	*Look-Up Table*
LZ4	*General Purpose Lossless Compressor*
LZMA	*General Purpose Lossless Compressor*
MDS	*Metadata Server*
MDT	*Metadata Target*
mLUT	*Mantissa Look-Up Table*
MPEG	*Motion Picture Experts Group*
MPI	*Message Passing Interface*
MPI-I/O	*MPI Input/Ouput*
N	*Length of Linearized Input Stream for Compression*
n_{B}	*Number of B-Spline Control Points in* ISABELA
n_{M}	*Number of Quantized Mantissas in* GLATE
n_{T}	*Number of Truncated Mantissa Bits*
OSS	*Object Storage Server*
OST	*Object Storage Target*
PNG	*Portable Network Graphics*
s	*Sign Bit of* FLOAT *value*
SBD	*Set Based Decomposition*
SFPD	*Scientific Floating Point Data*
SNAPPY	*General Purpose Lossless Compressor*
SPDP	*Lossless Floating Point Compressor*
JETR	*Jet Combustion Data Set, Turbulent Region*
t-GLATE	*Temporal GLATE*
t-ISABELA	*Temporal ISABELA*
TPFOR	*Lossless Integer Bit Packing using PFor Codec*
t-SBD	*Temporal SBD*
UNSIGNED	32 bit *Unsigned Integer*
VTK	*Open-Source Visualization Toolkit*
ZFP	*Lossy Floating Point Compressor*
ZLIB	*General Purpose Lossless Compressor*
ZSTD	*General Purpose Lossless Compressor*

List of Figures

1.1 The evolution of compute hardware and storage devices. 2

1.2 The scientific method cycle in HPC systems. 4

2.1 The IEEE754 `FLOAT` format. 10

2.2 Visualization of exponent E and mantissa M for 100 consecutive `FLOAT` numbers. 11

2.3 Typical simulation grids. 12

2.4 Spatial coherence in simulation grids. 13

2.5 Recursive construction of space filling curves in two dimensions. 14

2.6 Linearization schemes on a three dimensional grid. 14

2.7 Clustering and locality properties of space filling curves. 15

2.8 Compression rate of lossless compressors for random signed and unsigned integer numbers. 19

2.9 Compression speed of lossless compressors for compression and decompression of random unsigned integer numbers. 20

2.10 CFD data sets used for lossless compression tests. 22

2.11 Complex porous filter structure used in LBM based CFD simulations in the CRC920. 27

3.1 Policy for maximum decompression error. 32

3.2 Quantization grid resulting from error mapping defined by e_{MAX} and e_1. . 32

3.3 ISABELA compression procedure. 34

3.4 ISABELA B-spline fit of sorted data. 34

3.5 ISABELA B-spline accuracy and distribution of decompression error. . . 37

3.6 SBD compression procedure. 41

3.7 Calculation of LUT values inside SBD compression procedure. 42

3.8 Quantization of LUT indices inside SBD compression procedure. 43

3.9 Maximum relative error of GLATE quantized float values. 46
3.10 GLATE compression procedure. 47
3.11 GLATE adjustment of error bound for exponent-aligned quantization. . 48
3.12 GLATE mantissa look-up table. 50
3.13 GLATE global step function. 50
3.14 GLATE exponent-aligned quantization of mantissa values. 51
3.15 ISABELA startup costs. 57
3.16 ISABELA compression rate for data sets `ALUR`, `ISOR`, `JETR`. 58
3.17 Growth of LUT size depending on maximum error. 62
3.18 SBD compression rate for data sets `ALUR`, `ISOR`, `JETR`. 64
3.19 GLATE internal data. 67
3.20 GLATE adjusted error bound. 68
3.21 Comparison of GLATE quantization and SBD look-up table. 69
3.22 GLATE compression rate for data sets `ALUR`, `ISOR`, `JETR`. 71
3.23 Overview of compression rate for data sets `ALUR`, `ISOR` and `JETR`. 75
3.24 Overview of compression speed for of data sets `ALUR`, `ISOR` and `JETR`. . . 76

4.1 Temporal coding in MPEG video compression methods. 80
4.2 Temporal coherence between two consecutive time steps of a CFD simulation. 81
4.3 Illustration of temporal compression procedure. 82
4.4 Placement of KFs and DFs in temporal compression procedure. 83
4.5 Selection of time steps for temporal compression procedure during CFD simulation. 84
4.6 *d*-ISABELA temporal compression procedure in DFs. 85
4.7 *d*-ISABELA compression procedure. 86
4.8 *t*-ISABELA procedure in KFs. 90
4.9 *t*-ISABELA procedure in DFs. 91
4.10 *t*-ISABELA compression procedure. 93
4.11 *t*-SBD procedure in KFs. 97
4.12 *t*-SBD procedure in DFs. 97
4.13 *t*-SBD compression procedure. 99
4.14 *t*-GLATE procedure in KFs. 103
4.15 *t*-GLATE procedure in DFs. 103

4.16 *t*-GLATE compression procedure. 106
4.17 Optimized *t*-GLATE procedure in KFs and DF. 111
4.18 Flow of fluid aluminum inside a cubic geometry cell with periodic boundary conditions. 113
4.19 *d*-ISABELA internal data. 114
4.20 *t*-ISABELA internal data. 116
4.21 *t*-SBD internal data. 118
4.22 *t*-SBD fluctuating LUT values. 120
4.23 *t*-SBD compression artifacts. 120
4.24 *t*-GLATE internal data. 121
4.25 Compression rate improvement through optimized *t*-GLATE compression procedure in KFs. 123
4.26 Overview of compression performance for *t*-GLATE and `ZFP` on uniform grid data set. 125
4.27 Scatter plot of relative error for *t*-GLATE and `ZFP` decompressed data. . 127
4.28 CFD simulation of moving vocal cords using unstructured grid with dynamic mesh points. 128
4.29 Overview of compression performance for *t*-GLATE and `ZFP` on unstructured grid data set. 130
4.30 Overview of compression rate for lossy temporal compression of data set `CUBC`. 133
4.31 Overview of compression speed for lossy temporal compression of data set `CUBC`. 135

5.1 Time-to-Analysis of data-intensive HPC applications. 139
5.2 Streaming directions in discrete lattice in the LBM. 140
5.3 Discretization of complex filter structure with 216 pores in a uniform grid. 141
5.4 Global column-major data layout. 145
5.5 File views for global column-major data layout. 146
5.6 Partition of global simulation grid used for the LBM. 148
5.7 Global sequential layout for compressed data on disk. 150
5.8 Visualization of melt data set in two different levels of resolution. 152
5.9 Octree averaging operation for computation of low-resolution data. 153
5.10 Architecture of a Lustre$^{(R)}$ distributed file system. 155

5.11 Distribution of stripe-aligned data fragments across four OSTs. 157
5.12 Distribution of size-varying data fragments across four OSTs. 158
5.13 Compression rate of `ZFP` and GLATE on full-resolution and low-resolution data for simulation variables u, v, w, M, Q. 160
5.14 Simulation run-time when writing uncompressed and compressed data into the Lustre(R) file system using different data layouts. 161
5.15 I/O times for writing uncompressed and compressed data into the Lustre(R) file system using different data layouts. 163
5.16 Varying run-time time of CFD simulation during temporal compression procedure. 166
5.17 I/O time for writing temporal compressed data into the Lustre(R) file system during the CFD simulation. 167
5.18 Overview of compression rate of `ZFP` and t-GLATE in CFD simulation. 169
5.19 Transfer time savings and speed-up of the scientific workflow by using t-GLATE in-situ compression. 173

6.1 Improvement of *Time-to-Analysis* in the scientific workflow. 179
6.2 Selective decompression by using an in-situ generated index. 185

List of Tables

2.1 Compression rate for CFD data sets using different GPLCs. 24

2.2 Run-time for compression of CFD data sets using different GPLCs. . . . 25

2.3 Compression rate and compression run-time for CFD data sets using lossless floating point compressor `SPDP`. 26

2.4 Entropy and unique values of CFD data sets. 26

3.1 Relative error resulting from throwing away lower mantissa bits. 45

3.2 GLATE quantization table for exponent-aligned mantissa values. 52

3.3 ISABELA index start-up costs. 56

3.4 ISABELA compression rate for data sets `ALUR`, `ISOR`, `JETR` 60

3.5 ISABELA run-time on data sets `ALUR`, `ISOR`, `JETR` 60

3.6 LUT size in SBD for data sets `ALUR`, `ISOR` and `JETR`. 61

3.7 SBD compression rate for data sets `ALUR`, `ISOR`, `JETR` 63

3.8 SBD run-time on data sets `ALUR`, `ISOR`, `JETR`. 65

3.9 GLATE compression rate for data sets `ALUR`, `ISOR`, `JETR`. 70

3.10 GLATE compression run-time for data sets `ALUR`, `ISOR`, `JETR`. 72

4.1 *d*-ISABELA compression rate for data set `CUBC`. 115

4.2 *t*-ISABELA compression rate for data set `CUBC`. 117

4.3 *t*-SBD compression rate for data set `CUBC`. 119

4.4 *t*-GLATE compression rate for data set `CUBC` 122

5.1 *t*-GLATE compression rate in parallel CFD simulation. 165

List of Algorithms

3.1 Pseudo code of ISABELA compression procedure ISA_compress(N, n_{B}, `in_data`, e_{MAX}, `out_stream`) 39

3.2 Pseudo code of SBD compression procedure SBD_compress(N, `in_data`, e_{MAX}, `out_stream`) 44

3.3 Pseudo code of GLATE compression procedure GLATE_compress(N, X, e_{MAX}, `out_stream`) 54

4.1 Pseudo code of d-ISABELA compression procedure dISA_compress(`time_step`, N, n_{B}, e_{MAX}, `in_stream`, `out_data`) 88

4.2 Pseudo code of d-ISABELA decompression procedure dISA_decompress(`time_step`, N, n_{B}, `in_stream`, e_{MAX}, `out_data`) 89

4.3 Pseudo code of t-ISABELA compression procedure tISA_compress(`time_step`, N, n_{B}, e_{MAX}, `in_stream`, `out_data`) 95

4.4 Pseudo code of t-ISABELA decompression procedure tISA_decompress(`time_step`, N, n_{B}, `in_stream`, e_{MAX}, `out_data`) 96

4.5 Pseudo code of t-SBD compression procedure tSBD_compress(`time_step`, N, e_{MAX}, `in_stream`, `out_data`) 101

4.6 Pseudo code of t-SBD decompression procedure tSBD_decompress(`time_step`, N, `in_stream`, e_{MAX}, `out_data`) 102

4.7 Pseudo code of t-GLATE compression procedure tGLATE_compress(`time_step`, N, X, e_{MAX}, `out_stream`). 108

4.8 Pseudo code of t-GLATE decompression procedure tGLATE_decompress(`time_step`, N, `in_stream`, e_{MAX}, `out_data`) 109

4.9 Pseudo code of optimized t-GLATE compression procedure tGLATE_compress_opt(N, X, e_{MAX}, `out_stream`) 112

5.1 Opening a shared file using MPI-I/O. 144

5.2 Collective write of uniform grid with global column-major data layout using MPI data types and MPI file views. 147
5.3 Collective write of length-varying compressed data using sequential data layout. 151

1 Introduction

1.1 Motivation

The motivation of this dissertation stems from an important observation in the *High Performance Computing* (HPC) domain: The limited bandwidth of storage systems and the large data sizes are strong hindrances for efficient storage and post-processing of scientific data sets. Storing full-size data sets during large-scale HPC applications leads to the accumulation of massive amounts of data, as it is the case for e.g. *Computational Fluid Dynamics* (CFD) simulations with high spatial and temporal resolution. Here, *In-Situ Compression* helps to reduce the *Input/Output* (I/O) botleneck by employing fast data reduction during running simulations. Typically, CFD simulations describe continuous physical processes in time and generate sequences of data sets with temporal coherence between consecutive time steps. Therefore, *Temporal Compression* is considered a promising approach for further decrease of the storage footprint for large CFD data. Hence, the research in this dissertation focuses on the development of methods for data reduction in HPC applications, so-called *In-Situ Compression* and *Temporal In-Situ Compression*.

1.1.1 High-Performance Computing

HPC has become an essential tool for research and knowledge discovery through large-scale computations. Unfortunately, the imbalance in the advancements in *Floating Point Operations Per Second* (FLOPS) and in *Input/Output operations Per Second* (IOPS) limits the efficient use of large systems [43]. Fig. 1.1 (a)–(b) shows, that FLOPS have increased by two orders of magnitude faster as compared to IOPS in the past 25 years [43]. Particularly, FLOPS increased by a factor of ~$1,000,000\times$, whereas the speed of storage devices has increased only by a factor of ~$1,200\times$. As the trend stays unchanged, data

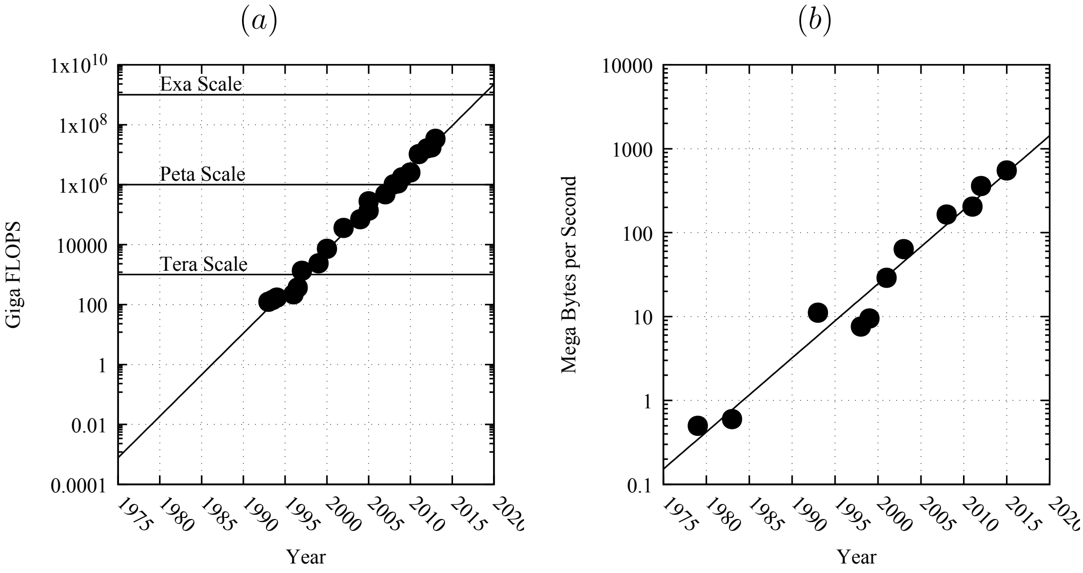


Figure 1.1: The evolution of compute hardware and storage devices. The speed of storage devices evolves much slower than the speed of CPUs.
Image is adapted from [43].

movement between HPC storage and main memory slows down data-intensive HPC for present and, even more, for future applications [43].

As a workaround, scientists often use sub-sampling and store grids in low-resolution in order to decrease the memory footprint. However, this approach leads to a lossy data capture and impedes the data interpretation. In most of the cases, storing results only in low-resolution defeats the main purpose of high-resolution simulations, i.e. the analysis of high-resolution three dimensional data. Therefore, in-situ data reduction is increasingly considered for storage of full-resolution data produced by HPC applications like CFD simulations [45, 55].

1.1.2 Computational Fluid Dynamics

CFD is an important branch of scientific computing and allows for the modeling and simulation of complex fluid flow scenarios. In the *Collaborative Research Center* 920 (CRC920), multi-functional ceramic filters for metal melt filtration are researched aiming towards high-purity metal melt filtration for the production of "zero-defect materials". There, CFD simulations have become an increasingly important tool for the analysis of the metal melt filtration processes [21, 85]. Because of the generally harsh process conditions and high temperatures, the analysis of the on-going filtration processes using real experiments is expensive and difficult. Detailed numerical simulations at the pore-scale level provide insights into the relationship between the filter structure and the flow of the liquid metal. Such simulations require high-resolution grids in order to resolve the flow inside filters with realistic porous structures. Especially when storing high-resolution uniform grid data, data reduction offers a big potential for decreasing the storage requirements for CFD simulations [36, 45]. For scientific applications, it is desirable to control the data quality by providing a point-wise maximum error for the decompressed data.

Specifically motivated by the research and development in the CRC920, this dissertation investigates new temporal compression schemes, which are applied in-situ in a CFD simulation, i.e. directly integrated in the main-loop of the *lattice-Boltzmann method* (LBM) solver used for metal melt flow simulation. The high data reduction allows for the storage of full-resolution data in large-scale HPC systems while circumventing the I/O bottleneck during data transfer for *Visualization & Analysis* [45].

1.1.3 In-Situ Processing Techniques

Although many problems can be solved directly inside the CFD solver without storing any data, the storage of full-resolution data allows for a flexible creation of analyses and visualizations in a post-hoc fashion. However, handling large amounts of data fragmented across many parallel processes is inefficient with traditional workflows, e.g. storing one file per process.

On the one hand, the movement of full-size data puts high load on the storage system, especially if data movement is tightly integrated into tiered storage systems [43], e.g. data is collected into temporary scratch storage or node-local/rack-local storage as burst-buffer before it is moved to workstations. On the other hand, storing full-size data impedes the loading into visualization workstations, as they in general exhibit less powerful hardware than the HPC clusters.

Regarding the I/O bottleneck, the HPC community initiates a shift in paradigm for how to deal with the increasing data size. So-called *In-Situ Processing* employs early data preparation and uses "cheap" CPU time for carrying out data transformations directly after the data has been created [42, 65]. Usually, such *In-Situ Techniques* are run as a part of HPC applications and aim at the reduction of storage space and I/O

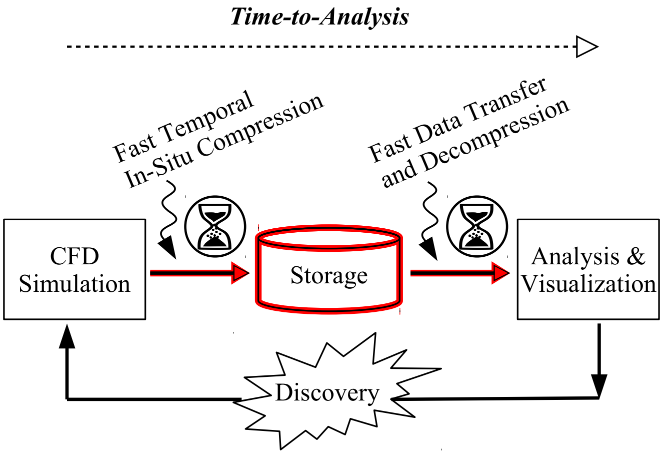


Figure 1.2: The scientific method cycle in HPC systems. I/O bottleneck impedes fast transfer of large-scale scientific data sets between HPC storage and main memory. Transfer of compressed data reduces the I/O bottleneck and decreases the *Time-to-Analysis*.
Image is adapted from [44].

times [46].

Many different types of in-situ algorithms have been developed for various applications. The following list shows successful applications of in-situ techniques inside data-intensive HPC applications:

In-Situ Visualization	Carrying out visualization tasks during the simulation. [3, 7, 8, 23, 73, 87]
In-Situ Feature Extraction	Recognition and tracking of patterns in the data. [20, 49, 83, 92]
In-Situ Indexing	Supporting query-driven analytics of scientific data sets by generating a (compressed) index. [14, 40, 44, 48]
In-Situ Compression	Reduction of the memory footprint of full-resolution data sets. [25, 45, 47, 54, 55, 58, 80]

For the CFD simulations carried out in the CRC920, in-situ processing shall provide the feasibility of data capture and data movement w.r.t. decreasing the *Time-to-Analysis* and accelerating the scientific workflow, as shown in Fig. 1.2. In our case, the *Time-to-Analysis* comprises the time between data production and data analysis, i.e. the time required for:

(a) Computation, storage and compression during the *Simulation* phase.

(b) Data transfer and decompression during the *Analysis & Visualization* phase.

This dissertation aims at the development of new *Temporal In-Situ Compression* methods for further reduction of the memory footprint, i.e. exploiting the temporal coherence inherently present when writing out time series in CFD simulations. In order to improve the *Time-to-Analysis*, the algorithms must be fast and avoid message passing in parallel processing to not infer with the simulation.

1.2 Research Objectives

This work provides contributions in the aspects of *Data Management* for HPC systems. Specifically, *Lossy In-Situ Data Compression* algorithms for *Scientific Floating Point Data* (SFPD) inside numerical CFD simulations are developed and evaluated. New schemes for *Temporal Compression* are developed and investigated in detail.

The starting point for the research are two existing methods for lossy compression of SFPD, namely *In-situ Sort-And-B-spline Error-bounded Lossy Abatement* (ISABELA) [45] and *Set Based Decomposition* (SBD) [36], which achieve a high data reduction by allowing insignificant numerical errors. ISABELA was originally developed for the in-situ context and has an extension for temporal compression, namely *Differential ISABELA* (d-ISABELA). Interestingly, both algorithms rely on using *General Purpose Lossless Compressor*s (GPLCs) as a back end for the compression of internal data. The availability of real-time GPLCs, like `snappy`, `lz4`, `zstd` and `tpfor` renders this a powerful idea, as the in-situ compressors directly benefit from the compression performance and future advancements of those algorithms. Especially the newly developed in-situ compression algorithm *Grid Linearization And Truncation Encoding* (GLATE) and the temporal compression scheme *Temporal GLATE* (t-GLATE) provide an improved compression speed and compression rate.

The GLATE and t-GLATE algorithms are evaluated inside a highly parallel CFD simulation, which is run on the *Taurus* HPC *Cluster* in Dresden, Germany – one of the top 500 supercomputers by June 2017[1]. Based on the evaluation results, the improvement of the *Time-to-Analysis* in the scientific workflow is discussed.

In particular this dissertation addresses the following research objectives:

1. *Development and Implementation* of a new lossy in-situ compression algorithm named GLATE.
2. *Evaluation* of lossy in-situ compression on three different CFD data sets, including melt flow simulations from the CRC920, and two other publicly available CFD data sets.
3. *Evaluation* of lossy in-situ compression back end using different state-of-the-art lossless compressors for the compression of internal data of the in-situ compressors.
4. *Development and Implementation* of several new lossy temporal in-situ compression algorithms namely *Temporal ISABELA* (*t*-ISABELA), *Temporal SBD* (*t*-SBD) and *t*-GLATE.
5. *Evaluation* of lossy temporal in-situ compression w.r.t. the compression performance depending on the temporal resolution of decompressed data, i.e. keyframe insertion and write-out step width.
6. *Comparison* of compression performance between the newly developed algorithms GLATE, *t*-ISABELA, *t*-SBD and *t*-GLATE, and the existing algorithms ISABELA [45], SBD [36] and *d*-ISABELA [45], as well as, the existing lossless float compressor `SPDP` [9] and the novel lossy float compressor `ZFP` [63] w.r.t. compression rate, compression run-time, and error of decompressed data.
7. *Evaluation* of in-situ applicability of temporal lossy in-situ compressor *t*-GLATE in a large-scale CFD simulation [2, 85] by empircal measurement of I/O times for merging compressed data from parallel processes into large files.
8. *Discussion* of acceleration of the scientific workflow by means of improving the *Time-to-Analysis* through the application of *t*-GLATE temporal lossy in-situ compression, and quantification of the improvement for typical data transfer of CFD simulation results within the CRC920.

---

[1] `https://www.top500.org/list/2017/06/`

1.3 Outline of Thesis

This section gives a short overview about the structure of this dissertation.

Chapter 2: *Lossless Compression of Scientific Floating Point Data* – This chapter summarizes the performance of lossless compression on SFPD. First, properties of SFPD stored in simulation grids are introduced and different schemes for grid linearization are presented. Second, an overview about fast standard lossless compressors is given and their performance is tested on three different CFD data sets, i.e. melt casting simulation, isotropic turbulence simulation, and turbulent combustion simulation.

Chapter 3: *Lossy In-Situ Compression for Scientific Floating Point Data* – This chapter describes the reimplementation of existing and the development of new lossy in-situ compressors for CFD simulations. First, the lossy in-situ compressors ISABELA and SBD are introduced as a starting point for in-situ compression. Second, the new in-situ compressor GLATE is developed. Third, the in-situ compressors are evaluated using three CFD data sets, and different ĺossless compressors are evaluated as back end for the compression of internal data of ISABELA, SBD and GLATE.

Chapter 4: *Temporal In-Situ Compression for Scientific Floating Point Data* – This chapter describes the development of new temporal compression algorithms for CFD simulations. First, the internal mechanisms of the temporal in-situ compressor *d*-ISABELA are presented. Second, new ideas for the in-situ compression algorithms *t*-ISABELA, *t*-SBD and *t*-GLATE are developed and implemented. Third, the temporal in-situ compressors are evaluated using high-resolution temporal data. Fourth, the performance of *t*-GLATE is compared to the novel lossy floating point compressor `ZFP`.

Chapter 5: *Application of In-Situ Compression in a Computational Fluid Dynamics Simulation* – In this chapter *t*-GLATE is evaluated in a parallel CFD simulation on the *Taurus* HPC *Cluster*. First, the HPC system and the CFD simulation used for the evaluation are described briefly, and different I/O procedures for writing and merging data into large files on the Lustre$^{(R)}$ parallel file system are presented. Second, simulation run-time and I/O times for storing compressed and uncompressed data are evaluated. For comparison purposes, all tests are repeated using the novel lossy floating point compressor `ZFP` as well. Fourth, the results of the evaluation are discussed w.r.t. the improvement of the *Time-to-Analysis* in the scientific workflow.

Chapter 6: *Conclusion and Future Work* – In the final chapter the contributions of this dissertation are summarized and future research directions are proposed.

2 Lossless Compression of Scientific Floating Point Data

This chapter provides background information on SFPD encountered in the CFD domain, and it investigates the performance of lossless compression on scientific data sets. First, general properties of floating point data are summarized and data structures for the representation of three dimensional simulation grids are explained briefly. Second, different linearization schemes for uniform simulation grids are introduced and used to create data streams of SFPD, which are subject to compression. Third, background information on state-of-the-art lossless compressors are presented, and fourth, the lossless compression is performed on sample *Single Precision* 32 bit *Floating Point* (`FLOAT`) data coming from CFD simulations.

2.1 Floating Point Data in Simulation Grids

The data type of prime importance to be looked at is the `FLOAT` format, which is written out in many CFD simulations. In general, computations inside simulations are carried out using *Double Precision* 64 bit *Floating Point* (`DOUBLE`) precision numbers. For data storage aiming towards subsequent analysis and visualization, `FLOAT` data supplies sufficient precision in most cases, as the data can be converted into `DOUBLE` when higher precision is needed for temporary computations. While requiring only half of the disk space at the cost of reduced precision, the usage of `FLOAT` data is a viable trade-off between data size and data precision for practical storage, transmission and processing of large-scale SFPD.

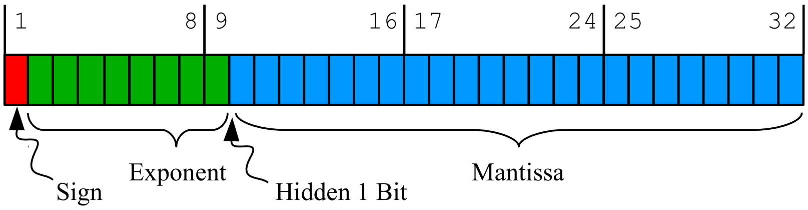


Figure 2.1: The IEEE754 `FLOAT` format. One `FLOAT` value consists of 32 bit. 1 bit for the sign, 8 bit for the exponent and 23 bit for the mantissa. Normalization of `FLOAT` values allows for omitting the *Hidden Bit* in front of the comma.

The IEEE754 Float Standard

The `FLOAT` format is specified in the IEEE754 standard and conforms to the bit pattern shown in Fig. 2.1. All 32 bits are subdivided into three different parts, namely, one bit $s \in \{0 = +, 1 = -\}$ for the sign, eight bits for the exponent $E \in [0, 255]$, and 23 bits $\mathcal{M} = \{m_1 \ldots m_{23}\}$ for the mantissa. The mantissa represents a sum of negative powers of two. The decimal point is shifted using the exponent, and the result sign is provided using the sign bit. For $E = 1, 2, 3, \ldots, 254$, the resulting decimal value x of a *Normalized Floating Point Number* is given by s, E and $\mathcal{M}$ according to the following equation:

$$x = (-1)^s \cdot 2^{E-127} \cdot \left([1] + \sum_{i=1}^{23} m_i \cdot \frac{1}{2^i} \right) \tag{2.1}$$

The IEEE754 format stores binary numbers in normalized form, which means the bit m_0 in front of the decimal point is choosen to be `1` by adjusting the exponent E, i.e. $m_0 \texttt{.} m_1\ m_2\ m_3\ \ldots\ m_{23}$. Thus, $m_0 = \texttt{1}$ is always omitted and called the *Hidden Bit*, denoted as [1] in Eq. (2.1).

The exponent $E = 0$ indicates the *Denormalized Floating Point Numbers*, i.e. $|x| \leq$ 1.1754942E–38 = `0x007fffff`. $E = 255$ indicates *Infinity* ($\mathcal{M} =$ `0x00000000`) or *Not a Number* ($\mathcal{M} \neq$ `0x00000000`). Because of this general structure of normalized floating point numbers, SFPD is commonly known to be high-entropy data with randomized bit and byte patterns [45, 84]. Fig. 2.2 shows a monotonic sequence of 100 floating point numbers, 0.001, 0.002, 0.003, ..., 0.1, and illustrates the behavior of the exponent and mantissa.

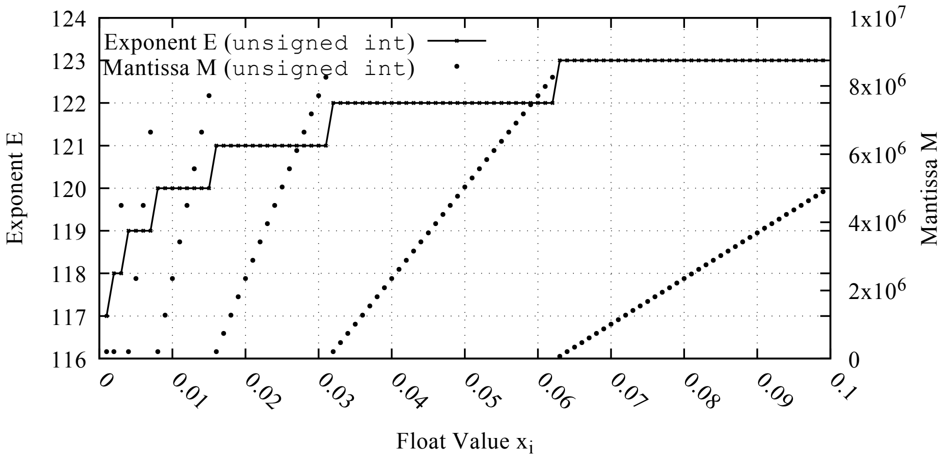


Figure 2.2: Visualization of exponent E and mantissa M for 100 consecutive `FLOAT` numbers. The exponents and mantissa of the numbers $x_i = 0.001 \cdot i$ for $i = 1 \ldots 100$ are plotted as `UNSIGNED`. As can be seen, if the exponent is counted up by one in the sequence, the mantissa jumps.

Floating Point Data in CFD Simulation Grids

Although `FLOAT` numbers span a very large value range between the values −3.403E+38 and +3.403E+38, large amounts of values in CFD data usually form clusters, where values concentrate within a smaller bounded value range [46]. Furthermore, simulations are carried out in grids, which supply a spatial context for coherence of floating point values. As shown in Fig. 2.3 (a)–(b), the grids consist of so-called grid cells containing values for different physical quantities and index numbers, which are computed during the simulation.

The uniform voxel-based grid, as shown in Fig. 2.3 (a), comes from a simulation of metal melt flow on pore-scale using a high-resolution parallel simulation for metal melt in the scope of the CRC920, as explained in Section 1.1.2, p. 2. So-called *Voxels* are small cubic three dimensional grid cells, which compose the whole spatial domain of the simulation. A subset of the voxels describes the filter structure as a staircase approximation allowing for the representation of complex shapes. In this example, each voxel contains one velocity vector (u, v, w). The color corresponds to the velocity magnitude $M = \sqrt{u^2 + v^2 + w^2}$.

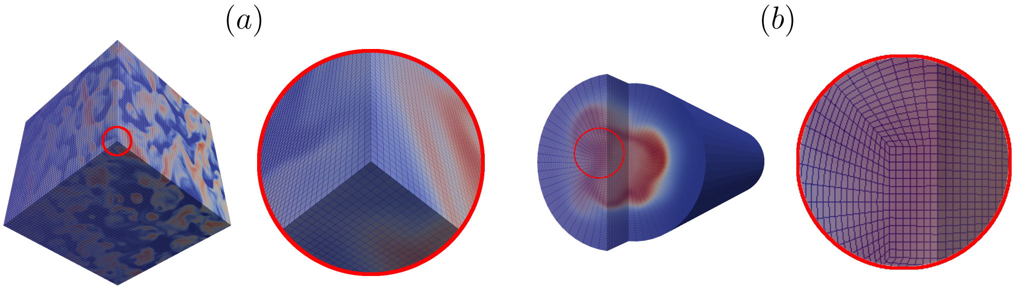


Figure 2.3: Typical simulation grids. (*a*) Uniform simulation grid from metal melt simulation in the CRC920 using the LBM. (*b*) Unstructured simulation grid with varying cell size used in combustion simulation.

Spatial Coherence in CFD Simulation Grids

The smooth color transition on the surface of the voxel grid indicates that the discretization of the underlying flow problem creates a mathematical relationship for adjacent grid cells. Values computed in neighboring grid cells are subject to spatial coherence, i.e. values are usually close to each other, and do not differ by orders of magnitude. Fig. 2.4 shows the distribution of the velocity component u, as well as, the differences between velocity components of adjacent grid cells Δu in x-direction, as extracted from the grid shown in Fig. 2.3 (*a*). As can be seen, the value range of u (−0.01, …, +0.07) is much wider, than the value range of Δu (−0.01, …, +0.01), i.e. neighboring grid cells exhibit spatial coherence. In general, this assumption counts for grids containing data describing continuous physical processes, e.g. computed using CFD codes.

In order to compress `FLOAT` data contained in three dimensional CFD simulation grids, the grid cells are linearized into a one dimensional sequence before they are fed into a compressor.

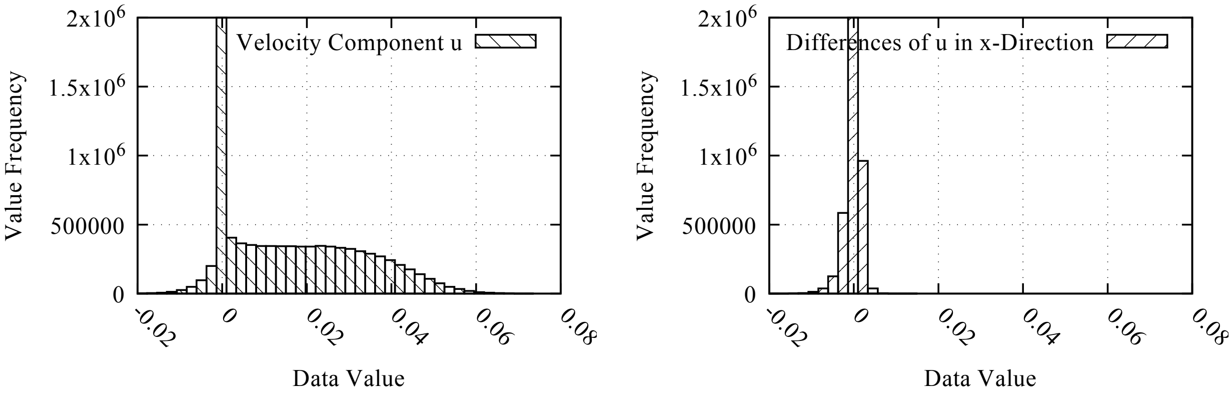


Figure 2.4: Spatial coherence in simulation grids. (a) Distribution of velocity component u. (b) Distribution of differences Δu between velocity component u of adjacent grid cells along the x-direction. Distribution of u is much wider than distribution of differences Δu.

2.2 Linearization of Simulation Grids

In uniform grids, the spatial location of each grid cell is implicitely given by three indices (i, j, k), as well as, the grid origin and the size of one voxel. Typically, uniform grids in main memory constitute three dimensional arrays and are linearized according to the *Column-Major Order*. In general, when dealing with uniform three dimensional data structures, so-called *Space Filling Curves* allow for a mapping between three dimensional grid locations (i, j, k) and a linear sequence of indices $0, 1, 2, \ldots$.

Space Filling Curves

Two prominent space filling curves, the Z-curve and the Hilbert-curve, traverse the grid and build spatial clusters instead of stacked planes. Fig. 2.5 (a)–(b) show the recursive construction of the Z-curve and the Hilbert-curve on a two dimensional grid for 4 iterations, starting with 2×2 cells. Fig. 2.6 (a)–(c) shows the Z-curve and Hilbert-curve in comparison to the column-major linearization on a three dimensional grid consisting of $4 \times 4 \times 4$ cells. As can be seen, the Z-curve and the Hilbert-curve traverse the grid in clusters of nearby grid cells, which form three dimensional blocks. The column-major ordering traverses the grid in columns and slices. All linearization schemes connect grid cells within one single polygonal chain, but only the Hilbert-curve constructs a line passing through direct neighbors of each grid cell. Such a linearization without jumps is beneficial for the partial compression of simulation grids, as similar data is close to each other in the resulting sequence.

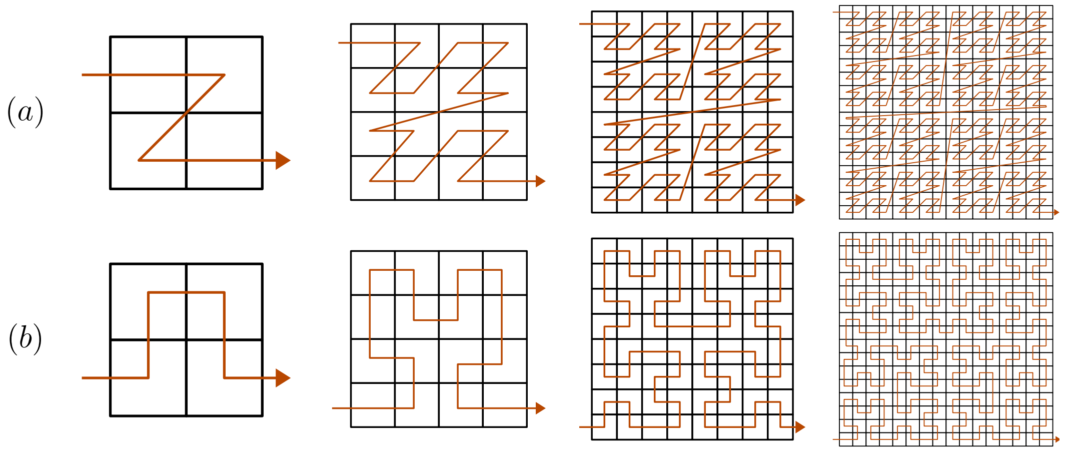


Figure 2.5: Recursive construction of space filling curves in two dimensions. Four iterations of the recursion for (a) the Z-curve, and (b) the Hilbert-curve are shown starting on a 2 × 2 grid and ending on a 16 × 16 grid.

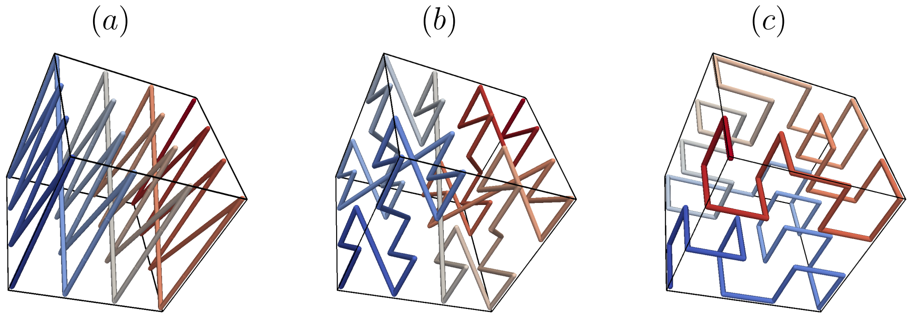


Figure 2.6: Linearization schemes on a three dimensional grid. The grid has 4 × 4 × 4 cells and is linearized using (a) the column-major order, (b) the Z-curve, and (c) the Hilbert-curve.

Partial Compression of Simulation Grids

Especially when sequences of grids are compressed partially, the Hilbert-curve improves the performance of the compression, as well as, the performance of data access [26], e.g. retrieval and decompression of planes and subgrids. In order to increase the compression performance the data should be compressed in large contiguous blocks, which support the cache and block-fetching mechanisms in the memory hierarchies and the CPU. Assuming many data values have to be decompressed at the same time, the way how the sequence groups data into blocks determines the amount of data required to be loaded during the decompression of subregions [89]. Since I/O dominates, a properly chosen data layout benefits the application.

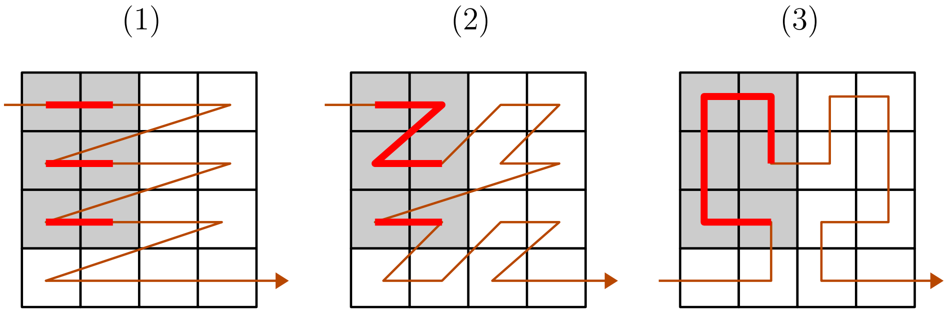


Figure 2.7: Clustering and locality properties of space filling curves. A 3×2 block is intersected with (1) the column-major linearization, (2) the Z-curve and (3) the Hilbert-curve. In general, the intersection with the column-major linearization and the Z-curve yield more jumps than the intersection with the Hilbert-curve.

Depending on the kind of query to be answered during data access, e.g. retrieving planes or connected regions of grid cells, different kinds of linearization schemes may be used. Whereas column-major layout yields the best performance for plane retrieval along the i, j-directions, the Z-curve and the Hilbert-curve are better choices for retrieving subregions of grid cells [41]. Generally, it is desired that bounding boxes intersecting the curve split the sequences in a small amount of distinct clusters, which yield indices close to each other. Fig. 2.7 illustrates the effect of locality and clustering for the column-major order, the Z-curve and the Hilbert-curve on a 4×4 grid, where a 3×2 bounding box is intersected with the curve. The column-major order and the Z-curve, result in 3 and 2 clusters, whereas the Hilbert-curve yields only one cluster. Generally, it has been shown, that the Hilbert-curve achieves better locality and clustering than the Z-curve for retrieving regions of grid cells [69].

As the research for the compression of SFPD is motivated by metal melt casting simulations using the LBM in the CRC920, the compression techniques are primarily applied to CFD data contained in uniform simulation grids. However, other linearization techniques for other grid types, e.g. unstructured grids, can be employed in the same manner. Consequently, instead of extending the compressors for taking into account the particular topology of different grid formats, methods for grid linearization are applied for rearranging the grid cells into a one dimensional sequence in order to feed them into a compressor.

2.3 General Purpose Lossless Compression

Data compression is an important topic in information and computer science, as it allows for minimizing the bits required to encode the information in a block of data while being able to reconstruct the original [31]. Historically, compression was employed, because disk space was limited, and data compression virtually increased the disk capacity [77]. In the context of this dissertation, compression is employed for decreasing the load of the storage system and increasing the data throughput between e.g. numerical simulation and the visualization system, as explained in Section 1.1.3, p. 3. Using lossless compression, a speed-up is achieved when compression, transfer of compressed data, and decompression is faster than the transmission of uncompressed data [50].

History of Lossless Compression

Lossless compression has its roots in the study of the Morse code method, which led to the concept of *Information Theory*. Data compression was carried out by assigning shorter codes to certain letters of the English alphabet in order to save space and time [17]. Today, many popular compression techniques exist, e.g. Huffman coding, arithmetic coding, run-length encoding, etc. [34]. However, some of the most popular GPLCs are based on ideas published in 1977 and 1978 by Abraham Lempel and Jacob Ziv, who developed the LZ77/78 algorithm [93]. Both algorithms are dictionary coders, which replace repeating occurences of data by references to previous occurences in the data stream. LZ77/78 was superseded by the LZW algorithm [84], which runs faster and introduced variable-width codes.

State-of-the-Art Lossless Compression

Some of the most important compression libraries are based on the the ideas of the LZW algorithm: `zlib` [18], `bzip` [76], `lzma` [1], `bsc` [30], `snappy` [28], `lz4` [15] and `zstd` [22] –

more implementations of lossless compression algorithms exist, e.g. *Zopfli* [29], *Brotli* [27], *ZPAQ* [66], *LZO* [70], but those are not suited for the present compression task and disproportionately trade compression speed for compression rate.

The `zlib` library uses the DEFLATE algorithm, which is based on LZW and *Huffman Coding* [35, 72]. It is used in *Portable Network Graphics* (PNG) files and the ZIP file format. `bzip` combines LZW algorithms with the *Burrows-Wheeler transform* [68], which is a reversible transformation for text-like data, acting like a pre-conditioner, and creating optimized input streams for the `zlib` DEFLATE. `lzma` extends the LZW algorithm to use *Range Encoding* [72], which generalizes the Huffman coding and employs a complex model for the prediction of the next data in the stream using *Markov Chains* [13]. `bsc` aims at being a high-performance block-sorting compressor, which uses Burrows-Wheeler transform while performing radix sort on the *Graphic Processing Unit* (GPU). Compared to `bzip`, `bsc` yields a better compression rate, and `lzma` yields faster decompression speed.

Real-Time Lossless Compression

Especially interesting for in-situ compression are real-time GPLCs, namely `snappy`, `lz4` and `zstd`, which have been developed recently. `lz4` and `zstd` constitute efficient implementations of the DEFLATE algorithms, mainly optimized for high speed and high throughput. While `lz4` focuses on real-time compression for data transfer over 1 Gbit/s ethernet, `zstd` allows for a fine-grained tweaking of the compression performance by trading run-time for compression rate. Both algorithms are developed for real-time compression scenarios and yield compression rates comparable to `zlib`. In contrast, the `snappy` implementation aims for high speeds using a conservative compression policy and may result in nearly no compression if the data cannot be encoded fast enough.

Contrary to LZW based approaches for lossless compression, algorithms like `tpfor` [10] use the PFor bit packing [59, 60] which encodes streams of unsigned integer numbers. The PFor bit packing codec works on small blocks of `UNSIGNED` values, where it determines the minimum value. The minimum value is stored uncompressed, and the differences between the minimum value and the other values are encoded using the minimal number of bits. If the values are in a narrow range, PFor yields efficient encoding at very good compression rates. Further, sorted lists can be compressed using *Differential Coding* [60]. Using differential coding, the compression rate can be improved by processing the differences between consecutive elements of a sorted list using the PFor bit packing codec.

Compression Performance Test

In order to evaluate the compression performance of `zlib`, `bzip`, `lzma`, `lz4`, `zstd`, `bsc`, `snappy` and `tpfor` for the in-situ compression context, range-bounded random 32 bit *Signed Integer* (`INTEGER`) and range-bounded random `UNSIGNED` numbers are compressed. Specifically, $128^3 = 2{,}097{,}152$ random values in two intervals $\mathcal{X}_S = [-X_{\mathrm{MAX}}, +X_{\mathrm{MAX}}]$ and $\mathcal{X}_U = [0, +X_{\mathrm{MAX}}]$, for $X_{\mathrm{MAX}} = 2^2, 2^3, 2^4, \ldots, 2^{30}$, are generated and compressed. During each run, the compression rate, and the run-time for compression and decompression are recorded. All algorithms are applied using their default compression level, which is 6 for `zlib`, `bzip`, `lz4` and `zstd`, and 5 for `lzma`. Higher compression levels are not considered for the in-situ compression task, as they increase run-time disproportionally compared to the gain in compression rate. For comparison, `zlib`, `bzip`, `lz4`, `zstd` and `lzma` are also operated with lower compression levels between 1–4. The `bsc`, `snappy` and `tpfor` compression algorithms, do not provide advanced adjustment of the compression performance. Therefore, for `bsc` and `snappy` the default parameters are used. For `tpfor`, the default block size of 128 values is used.

Compression Rate

The term compression rate refers to the percentage of data which is left after the compression has been executed, as shown in the following equation:

$$\text{Compression Rate} = \frac{\text{Size of Compressed Data}}{\text{Size of Raw Uncompressed Data}}$$

For the compression of `FLOAT` data inside CFD simulation grids, the term compression rate is explained in more detail in the next section.

Fig. 2.8 (*a*)–(*b*) show the compression rate for random `INTEGER` and `UNSIGNED` numbers in the intervalls $\mathcal{X}_S$ and $\mathcal{X}_U$. As explained before, `tpfor` is designed for the compression of `UNSIGNED` data only. As negative `INTEGER` numbers are represented using the *two's complement*, `tpfor` handles them as high `UNSIGNED` numbers. Therefore, the compression performance of `tpfor` on the interval $\mathcal{X}_S$ is poor, as can be seen in Fig. 2.8 (*a*). However, as shown in Fig. 2.8 (*b*), for the compression of positive values, `tpfor` yields a strong compression similar to the best LZW based GPLCs, namely `bzip`, `lzma` and `bsc`.

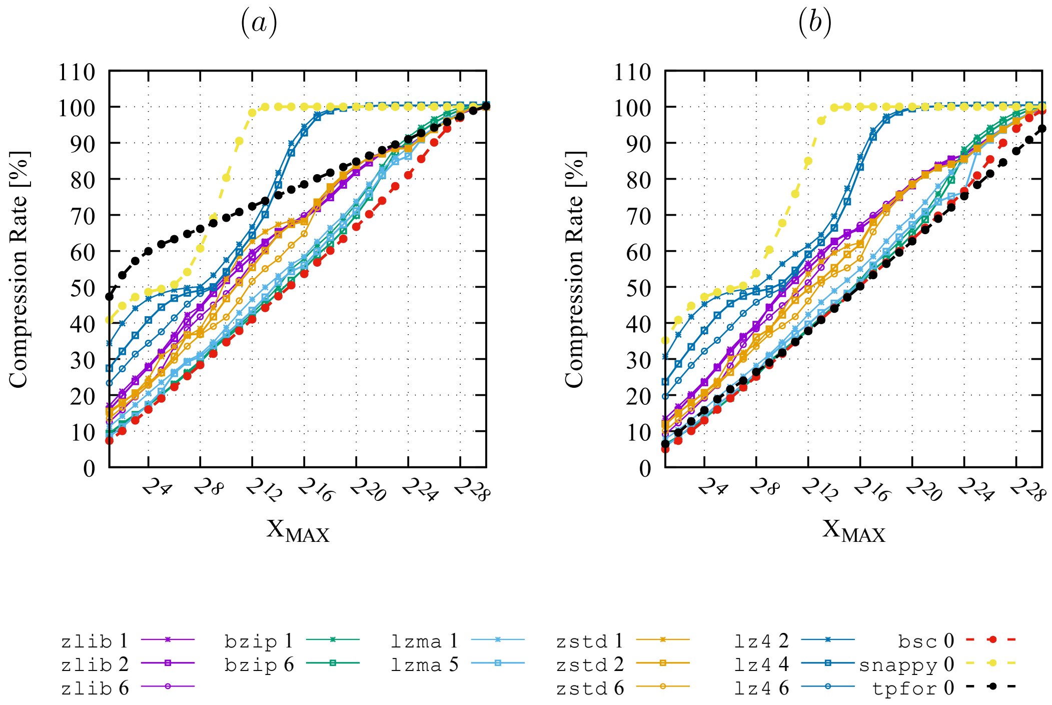


Figure 2.8: Compression rate of lossless compressors for random signed and unsigned integer numbers. The compressors `zlib`, `bzip`, `lzma`, `lz4`, `zstd`, `bsc`, `snappy`, `tpfor` are used for compression of (a) random `INTEGER` numbers in $\mathcal{X}_S = [-X_{\mathrm{MAX}}, +X_{\mathrm{MAX}}]$, and (b) random `UNSIGNED` numbers in $\mathcal{X}_U = [0, +X_{\mathrm{MAX}}]$ for $X_{\mathrm{MAX}} = 2^2, 2^3, 2^4, \ldots, 2^{30}$.

Compression Run-Time

Fig. 2.9 (a)–(b) show the run-time for compression and decompression of `UNSIGNED` data plotted against the compression rate as shown in Fig. 2.8 (b). As the compressors yield similar speed on `INTEGER` data, only the statistics for `UNSIGNED` data is shown. As can be seen, `tpfor` yields the highest speed for compression and decompression of `UNSIGNED` data.

The fastest LZW based algorithms are `snappy`, `zstd` and `lz4` which yield high-speed compression and decompression for the expense of a slightly lower compression rate as compared to `bzip`, `lzma` and `bsc`. It has to be noted the time scale is plotted logarithmic,

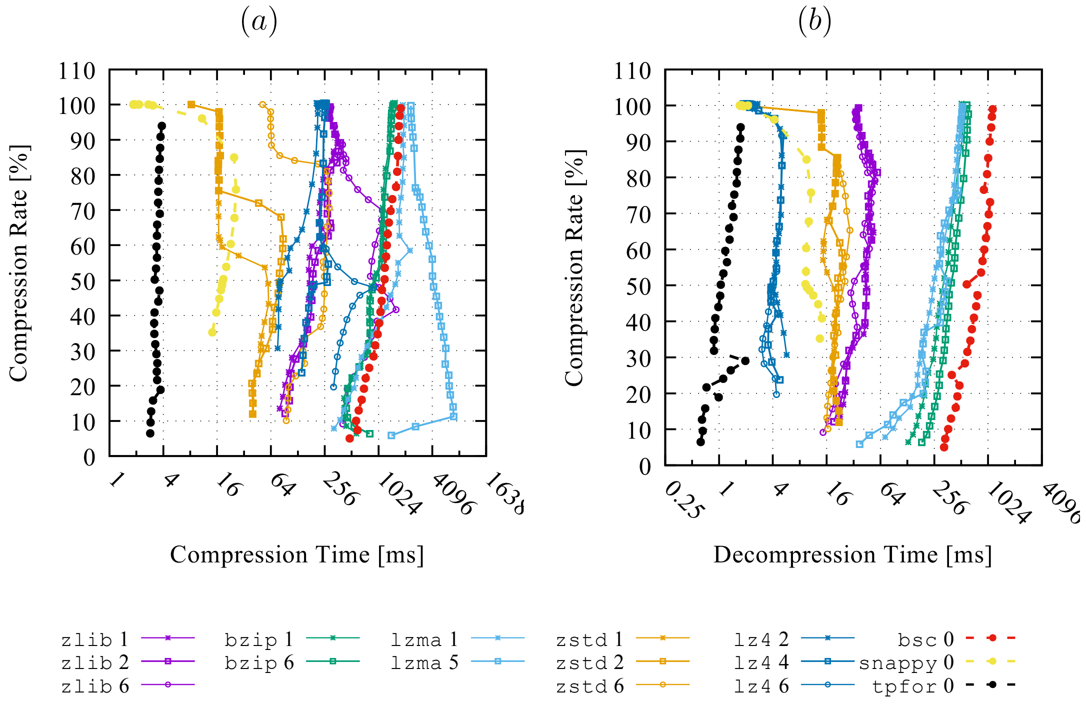


Figure 2.9: Compression speed of lossless compressors for compression and decompression of random unsigned integer numbers. The compressors `zlib`, `bzip`, `lzma`, `lz4`, `zstd`, `bsc`, `snappy`, `tpfor` are used to compress and decompress random `UNSIGNED` numbers in the interval $[0, +X_{\text{MAX}}]$ for $X_{\text{MAX}} = 2^2, 2^3, 2^4, \ldots, 2^{30}$. The graph shows (a) compression run-time, and (b) decompression run-time plotted against the compression rate shown in Fig. 2.8 (*b*).

and `bzip`, `lzma` and `bsc` take more than 16× longer for compression than the other algorithms. While `lz4` yields the fastest decompression speed, `zstd` yields very good compression rates at high decompression speeds and offers the best trade-off between level of compression and compression run-time.

Selection of Fast Lossless Compressors

Because of the simplicity of bit packing, `tpfor` is always the fastest algorithm and yields a better trade-off between compression rate and compression speed as compared to the LZW algorithms. However, as can be seen in Fig. 2.8 (a), `tpfor` can be applied to `UNSIGNED` values only, otherwise the compression rate declines. Concluding, `zlib`, `zstd`, `lz4`, `snappy` and `tpfor` are well-suited for compression of integer streams and are used as back end for fast lossy in-situ compression and temporal compression of SFPD.

2.4 Compression of Scientific Data Sets

Before lossy in-situ compression is elaborated in the next chapter, different lossless compressors, namely `zlib`, `bzip`, `lzma` and `zstd`, are tested on SFPD coming from three different CFD simulations. All data sets are composed of voxels in uniform simulation grids, which conform to the setup of the simulations for metal melt casting used in the CRC920. At first, voxels in each simulation grid are linearized before they are fed into different GPLCs. Additionally to feeding the data into GPLCs, the data is also compressed with a lossless compressor `SPDP` [9] specially developed for `FLOAT` data. For linearization, the row-major and the column-major ordering, as well as, the Z-curve and the Hilbert-curve are used.

CFD Data Sets

The data sets denoted as `ALU`, `ISO` and `JET`, correspond to an *Aluminum Metal Melt Simulation in Porous Media* [67], an *Isotropic Turbulence Simulation* [61], and a *Turbulent Combustion Simulation* [88]. Different indices `A`, `C` and `R` of the data sets denote the following different versions.

`A` The entire simulation grid including boundary regions.

`C` A cropped data set without boundary regions.

R A unsteady center region capturing simulation specific phenomena.

The cropped version does not contain boundary regions, which typically do not contain important phenomena and are needed for the flow to evolve during the simulation. The turbulent center regions of size $64 \times 64 \times 64 = 262,144$ voxels contain 60 time steps in the temporal end stage of the simulation, where the flow has fully developed. Compressing different regions of the data sets illustrates the performance of the compressors during different states of the simulation.

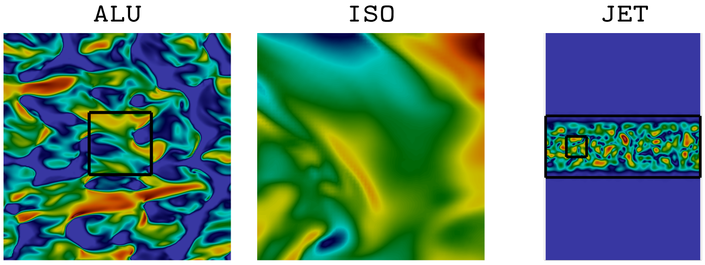


Figure 2.10: CFD data sets used for lossless compression tests. ALUC, ALUR Aluminum Metal Flow in Porous Media, showing the whole data set and the center region. ISOC, ISOR Isotropic Turbulence, showing the cropped data set and the center region, which have the same size but different amount of time steps. JETA, JETC, JETR Turbulent Combustion Simulation, showing whole data set, cropped version and the center region.

The following list describes the data sets, and Fig. 2.10 shows a color plot of the data sets and their different versions.

ALU Aluminum Metal Flow in Porous Media [67] from in-house CFD code [85].

- ALU*C* Cropped version. Consists of 164 time steps with $256 \times 256 \times 256$ voxels containing velocity vector (u, v, w) – 30.75 GB.
- ALU*R* Center region. Consists of time steps $100 \dots 160$ with $64 \times 64 \times 64$ voxels containing velocity vector (u, v, w) – 180 MB.

ISO Isotropic Turbulence [61] from the *Johns Hopkins Turbulence Databases*.

- ISO*C* Cropped Version. Consists of 1024 time steps with $64 \times 64 \times 64$ voxels containing velocity vector (u, v, w) – 3 GB.
- ISO*R* Center region. Consists of time steps $960 \dots 1020$ with $64 \times 64 \times 64$ voxels containing velocity vector (u, v, w) – 180 MB.

`JET` Turbulent Combustion Simulation [88] from the *S3D Direct Numerical Solver*.

`JETA` Whole data set. Consists of 122 time steps with $448 \times 704 \times 64$ voxels containing the length of the vorticity vector V – 9.2 GB.

`JETC` Cropped version. Consists of 122 time steps with $448 \times 192 \times 64$ voxels containing the length of the vorticity vector V – 2.5 GB.

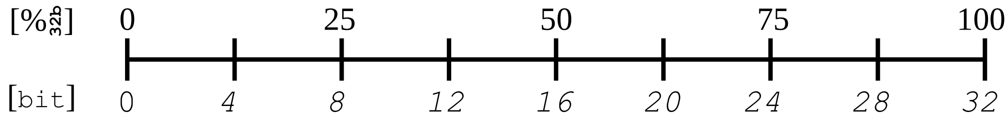
`JETR` Center region. Consists of time steps $60 \ldots 120$ with $64 \times 64 \times 64$ voxels containing the length of the vorticity vector V – 60 MB.

The data sets `ALU`, `ISO` and `JET` are compressed using the GPLCs `zlib`, `bzip`, `lzma` and `zstd` on their default levels, i.e. level 6 for `zlib`, `bzip` and `zstd`, and level 5 for `lzma`. As the compression tests mimic the compression of scientific data inside numerical simulations, which typically are run in parallel *Message Passing Interface* (MPI) environments, the data sets are divided into subgrids of size $64 \times 64 \times 64$ voxels before compression. The subgrids are linearized using the schemes described in Section 2.2, p. 13, and are compressed independently.

Definition of Compression Rate

The term *Compression Rate* refers to the amount of data left after compression, relative to the original data size. The compression rate relative to `FLOAT` data is referred in the unit $[\,\%_{\mathrm{32b}}]$. In the original publication of ISABELA and SBD, the compression rate is referred relative to `DOUBLE` data. However, since a simple conversion into the `FLOAT` data type already yields a data reduction by factor two, the compression rates are specified relative to `FLOAT` data $[\,\%_{\mathrm{32b}}]$. The compression rate for `FLOAT` data is given by the following equation:

$$\text{Compression Rate} = \frac{\text{Size of Compressed Data}}{\text{Size of Raw Binary FLOAT Data}} \times 100\,\%_{\mathrm{32b}} \tag{2.2}$$



Compression Performance of GPLC

The compression rates resulting on the data sets `ALU`, `ISO` and `JET` are shown in Table 2.1. As can be seen, GPLCs do not reduce SFPD by more than ~10–25 % while consuming a reasonable amount of run-time for compression of the noisy `FLOAT` data. Increasing the level of compression would improve the compression rate slightly. However, it also would increase the run-time and memory requirements disproportionately. The `lzma` algorithm always yields the best compression rates of ~74–80 % on data sets linearized using the Hilbert-curve. Using different linearization schemes makes a difference of ~1–3 %.

Table 2.2 shows the compression run-time in seconds for the compression of the data sets `ALU`, `ISO` and `JET` using the same compressors. Although, `lzma` yields the best compression rate, it also takes the longest time for compression. `zlib` and `zstd` are at least ~2× faster than `lzma` and `bzip`.

$[\%_{32b}]$	`zlib`	`bzip`	`lzma`	`zstd`		`zlib`	`bzip`	`lzma`	`zstd`	
`ALUC`	83.43	85.03	77.74	83.16	`ALUR`	80.77	82.17	75.26	80.44	i–j–k
	83.27	85.01	76.75	83.00		80.59	82.17	74.34	80.31	k–j–i
	82.43	85.13	76.71	82.90		79.69	82.20	74.24	80.11	Z–curve
	82.17	84.98	**76.49**	82.74		79.43	82.11	**74.02**	79.90	Hilbert
`ISOC`	90.17	91.57	82.09	89.85	`ISOR`	89.48	90.72	81.47	89.16	i–j–k
	89.98	91.45	81.30	89.69		89.56	90.85	81.13	89.25	k–j–i
	88.63	91.42	79.88	89.13		88.58	90.86	79.56	89.02	Z–curve
	88.50	91.49	**79.47**	89.13		88.39	90.82	**79.10**	89.02	Hilbert
`JETC`	89.36	91.17	82.07	89.13	`JETR`	89.21	91.10	82.12	88.98	i–j–k
	89.30	91.17	81.09	89.09		89.18	91.11	81.26	88.97	k–j–i
	88.54	91.18	80.12	88.85		88.50	91.13	79.97	88.79	Z–curve
	88.47	91.02	**79.72**	88.85		88.43	91.05	**79.50**	88.79	Hilbert

Table 2.1: Compression rate for CFD data sets using different GPLCs. The compressors `zlib`, `bzip`, `lzma` and `zstd` are used for the compression of the data sets `ALU`, `ISO` and `JET`. `zlib`, `bzip` and `zstd` operate at default compression level 6, and `lzma` operates at default compression level 5.

Compression Performance of Lossless Float Compressor

As shown, GPLCs do not reduce the size of SFPD efficiently and also require reasonable amount of run-time. In order to achieve a better compression rate while requiring

[s]	`zlib`	`bzip`	`lzma`	`zstd`		`zlib`	`bzip`	`lzma`	`zstd`	
`ALUC`	1550.4	4588.2	9038.7	1626.7	`ALUR`	9.7	25.7	48.4	7.5	i–j–k
	1445.3	4489.8	8870.1	1552.2		10.1	26.5	47.0	7.3	k–j–i
	1643.7	4489.8	8753.9	**1290.5**		10.6	26.1	47.8	**5.2**	Z–curve
	1600.1	4517.5	8868.1	1361.7		10.6	25.7	46.7	6.5	Hilbert
`ISOC`	182.8	491.7	878.4	60.8	`ISOR`	9.7	28.7	52.0	**3.5**	i–j–k
	180.3	480.9	851.6	**54.9**		9.5	28.3	50.3	3.8	k–j–i
	180.9	478.8	882.4	60.0		9.7	28.1	52.1	4.0	Z–curve
	186.4	486.3	875.7	64.0		9.7	28.2	51.4	4.6	Hilbert
`JETC`	136.0	410.8	762.8	41.4	`JETR`	3.3	9.8	17.0	1.1	i–j–k
	135.8	401.2	778.7	40.0		3.3	9.6	16.9	0.9	k–j–i
	136.5	404.2	774.1	39.9		3.3	9.7	17.6	**0.8**	Z–curve
	139.6	400.6	765.2	**36.7**		3.3	9.8	17.2	1.0	Hilbert

Table 2.2: Run-time for compression of CFD data sets using different GPLCs. The compressors `zlib`, `bzip`, `lzma` and `zstd` are used to compress the data sets `ALU`, `ISO` and `JET`. `zlib`, `bzip` and `zstd` operate at default compression level 6, and `lzma` operates at default compression level 5.

less run-time, special lossless compressors, namely `FPC` and `SPDP` [9], for `DOUBLE` and `FLOAT` have been designed. As compared to GPLCs, `FPC` and `SPDP` are designed for the compression of `FLOAT` and `DOUBLE` data. However, the data reduction is primarily implemented for high speed and high throughput, and does not contain complicated features for improving its compression rate [9]. `FPC` and `SPDP` use a predictor to estimate the next floating point value in the sequence and perform an `XOR`-operation on the next original and predicted value. Since `XOR` creates `0` bits in the case the operands match, bit-wise `XOR` of two `FLOAT` values create a lot of leading `0` bits if original and prediction are close to each other. By employing an efficient encoding of the residuals and leading `0` bits, `FPC` and `SPDP` achieve data compression. Table 2.3 shows the compression rate and run-time of `SPDP` on the data sets `ALU`, `ISO` and `JET`. As can be seen, the compression rate is best at ~85–92 %, and the run-time is comparable to `zstd`, which is the fastest GPLC.

	CR [$\%_{32b}$]	Time [s]		CR [$\%_{32b}$]	Time [s]	
`ALUC`	93.47	2116.2	`ALUR`	93.00	5.1	i-j-k
	92.43	**1947.9**		91.43	3.7	k-j-i
	88.74	2112.3		**86.76**	3.8	Z-curve
	91.92	3014.2		89.72	**3.6**	Hilbert
`ISOC`	92.28	60.9	`ISOR`	91.66	3.5	i-j-k
	91.98	**55.9**		**91.46**	3.0	k-j-i
	91.96	58.8		91.83	3.4	Z-curve
	95.88	59.0		95.69	**2.9**	Hilbert
`JETC`	87.17	31.5	`JETR`	85.78	1.0	i-j-k
	86.92	**30.4**		**85.43**	**0.8**	k-j-i
	88.15	32.2		87.64	1.0	Z-curve
	93.50	31.1		92.96	1.0	Hilbert

Table 2.3: Compression rate and compression run-time for CFD data sets using lossless floating point compressor `SPDP`. The data sets `ALU`, `ISO` and `JET` are compressed using `SPDP` operating at compression level 9.

Complexity of CFD Data Sets

The bad performance of GPLCs on SFPD indicates the inherent complexity and randomness of SFPD, which is high-entropy data without many repeating bit and byte patterns [84]. Thus, SFPD in common is hardly compressible using GPLCs based on the LZ77/78 algorithms [44], e.g. implemented in the well known `zlib`, `bzip`, `lzma` and `zstd` libraries which have been tested. Table 2.4 shows the entropy and the amount of unique values for the data sets `ALU`, `ISO` and `JET`.

Variable	Data Set	Entropy [bit]	Unique Values [%]
(u, v, w)	`ALUC`	17.9	79.3
	`ALUR`	17.9	85.9
(u, v, w)	`ISOC`	21.5	99.4
	`ISOR`	15.8	99.4
V	`JETA`	24.6	86.5
	`JETC`	22.8	86.9
	`JETR`	17.9	99.5

Table 2.4: Entropy and unique values of CFD data sets. Amount of unique values in the data sets `ALU`, `ISO`, `JET` and their different versions `A`, `C`, `R`.

The uniqueness of the data is given by the fraction of the number of unique values in the data set divided by the overall number of values. The entropy is a measure for the amount of self-information carried by messages sent out by an information source X, i.e. the number of bits required to label all possible messages x that are sent out by X [12]. Mathematically, entropy is measured in bit and defined by the following equation:

$$H(X) = -\sum_{x \in X} P(x) \cdot \log_2 \left(P(x)\right) \tag{2.3}$$

All data sets exhibit a high entropy of at least ~15.8 bit. However, due to memory and run-time limitations, as well as, the fragmentation of the grid data in subgrids before compression, GPLCs only operate on a smaller local context of the data leading to less efficient coding. As can be seen, the data sets also exhibit a high amount of unique values corresponding to individual bit patterns. The uniqueness reflects the noise introduced by the normalized mantissa of `FLOAT` values as explained in Section 2.1, p. 9 and increases in the turbulent center regions. As shown in Fig. 2.11, the `ALU` data set describes a porous filter structure containing regions blanked to zero, i.e. the melt flow data set `ALU` has slightly lower entropy and less unique values.

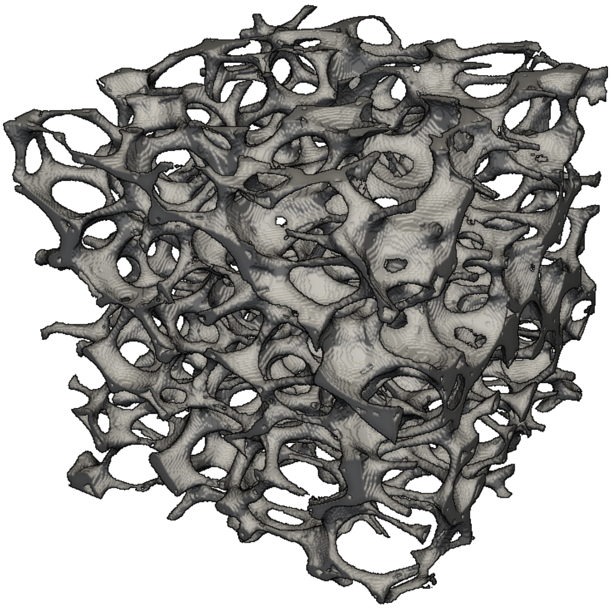


Figure 2.11: Complex porous filter structure used in LBM based CFD simulations in the CRC920. The filter is obtained from a CT scan and used for the production of the `ALU` data set describing a aluminum metal melt filtration scenario.

2.5 Summary

As elaborated in this chapter, GPLC and lossless floating point compressors hardly yield data reduction on SFPD. GPLCs are not suited for the compression of SFPD, as floating point data exhibits a lot of randomness in its bit and byte patterns. Special-purpose floating point compressors solve the speed problem of GPLC on SFPD. However, they are only slightly better than GPLCs w.r.t. the resulting compression rate.

If data compression is fast enough and reduces space requirements, it allows for the acceleration of data transfer [82]. In HPC systems, computational power is available on a larger scale as compared to network bandwidth and storage space. In this regard, early data compression is a powerful approach for scaling up e.g. CFD simulations and visualization applications while keeping the I/O bottleneck in mind.

For this reason it is argued, that *Lossy In-Situ Compression* of SFPD constitutes a promising approach for reducing space requirements, as well as, the transfer time of data by offering better data reduction as compared to lossless compression. *Lossy In-Situ Compression* for SFPD is described in the next chapter.

3 Lossy In-Situ Compression for Scientific Floating Point Data

In this chapter, existing algorithms for in-situ compression are reviewed and new algorithms are developed. First, lossy compression for SFPD is motivated, and the requirements on the quality of the decompressed data are defined through an error correction policy for a point-wise maximum decompression error. Second, two algorithms, namely ISABELA [45] and SBD [36], are presented that serve as an entry point for practical in-situ compression of SFPD. Furthermore, based on the ideas of ISABELA and SBD, the GLATE algorithm is newly developed in this chapter. Third, the compression rate and the compression speed of ISABELA, SBD and GLATE are evaluated on three CFD data sets. In this context, different real-time GPLCs are evaluated as back end for the compression of internal data of ISABELA, SBD and GLATE.

3.1 Motivation for Lossy Compression

The availability of powerful HPC systems allows scientists to run CFD simulations with increased spatial and temporal resolution. However, such high-resolution simulations can generate virtually any amount of data, and scientists are forced to manage large amounts of data for single simulation runs. As a result, many data-intensive applications of the future are expected to be limited by available memory and bandwidth instead of processor performance [50], and the storage of uncompressed full-resolution data will be practically impossible in the future [4]. Here, data compression constitutes a way to trade off required bandwidth and computation time. However, as shown in Section 2.4 lossless compression yields poor performance on SFPD, reducing data by ~10–20 % only. The lossless float compressor `SPDP` achieves only slightly better compression but improved speed.

Although many problems can be addressed by analysis and visualization directly in the HPC simulation [23], which would decrease the amount of stored data to a higher extent than data compression, the data analysis in a post-hoc fashion allows for a more flexible generation of visualizations from simulation data stored in the file system and is inevitable for many visualization and analysis tasks [46]. Such workflows may greatly benefit from lossy in-situ data compression, which limits the amount of simulation data written out to the file system while keeping the decompression error low. With regard to these circumstances, error-bounded lossy in-situ compression for SFPD is argued to constitute a practical solution for storing large amounts of data at low error-rates.

3.2 Error Correction Policy

Lossy compression is argued to improve the data capture in large scale CFD simulations because it allows for storage of full-resolution data at low error rates, which is sufficient for e.g. visual analysis and exploration of scientific data sets. For scientific CFD simulations the quality of decompressed data is controlled by a user-defined maximum error e_{MAX} in percent.

Point-Wise Maximum Relative Error

Lossy compression is realized by a scalar quantization method which restricts the maximum relative error $E_{\mathrm{REL}}(x, \hat{x}) = {}^{|x-\hat{x}|}/_{|x|}$ between original x and decompressed data value $\hat{x}$. The decompressed value is allowed to differ from its original value based on a user-defined maximum relative error e_{MAX} [%]. The maximum difference between decompressed and original equals to $\pm|x| \cdot e_{\mathrm{MAX}}$ and is given by the following relation:

$$
\begin{aligned}
E_{\mathrm{REL}}(x, \hat{x}) \quad = \quad \frac{x - \hat{x}}{x} \quad &< \quad e_{\mathrm{MAX}} \\
|x - \hat{x}| \quad &< \quad |x| \cdot e_{\mathrm{MAX}}
\end{aligned}
\tag{3.1}
$$

In order to prevent overhead for error correction of near-to-zero numbers, an epsilon value e_1 for the restriction of the denominator $|x|$ has to be supplied according to the following equation:

$$E_{\mathrm{REL}}(x,\hat{x}) = \begin{cases} \dfrac{|x-\hat{x}|}{|x|} & \text{if } e_1 \leq |x| \qquad (i) \\[2ex] \dfrac{|x-\hat{x}|}{e_1} & \text{if } e_0 \leq |x| < e_1 \qquad (ii) \\[2ex] 0 & \text{if } |x| < e_0 \qquad (iii) \end{cases} \tag{3.2}$$

The parameters e_{MAX} and e_1 describe the mapping, as shown in Fig. 3.1. The five different sections (1)–(5), which are shown in the figure, correspond to the three different cases of Eq. (3.2), i.e. section (1) and (5) correspond to case (i), section (2) and (4) correspond to case (ii), and section (3) corresponds to case (iii). As can be seen, the parameter e_1 describes the value for which the error correction mechanism switches over to the relaxed definition, i.e. using an absolute error bound. Consequently, all values in $\hat{x} \in [-e_0, +e_0]$ with $e_0 = e_{\mathrm{MAX}} \cdot e_1$ are mapped to zero, as shown by the following relation:

$$\begin{aligned} E_{\mathrm{REL}}(0,\hat{x}) &< e_{\mathrm{MAX}} \\ \frac{|0-\hat{x}|}{e_1} &< e_{\mathrm{MAX}} \\ |\hat{x}| &< e_{\mathrm{MAX}} \cdot e_1 \\ \hat{x} &\in [-e_0, +e_0] \end{aligned} \tag{3.3}$$

Error Quantization

Fig. 3.2 shows three different possible sets of quantized values in the interval [0, 1E–3] conforming to Eq. (3.2) *(i)*–*(iii)*. The three following different configurations are shown:

(a) $e_{\mathrm{MAX}} = 1.5\,\%$, $e_1 = 1\mathrm{E}{-6}$.

(b) $e_{\mathrm{MAX}} = 3.0\,\%$, $e_1 = 1\mathrm{E}{-5}$.

(c) $e_{\mathrm{MAX}} = 4.5\,\%$, $e_1 = 1\mathrm{E}{-4}$.

As can be seen, for configuration (a), the quantization grid is finer and $e_0 = e_{\mathrm{MAX}} \cdot e_1$ is smallest, which results in a better reconstruction of small data values, as compared to configuration (b) and (c).

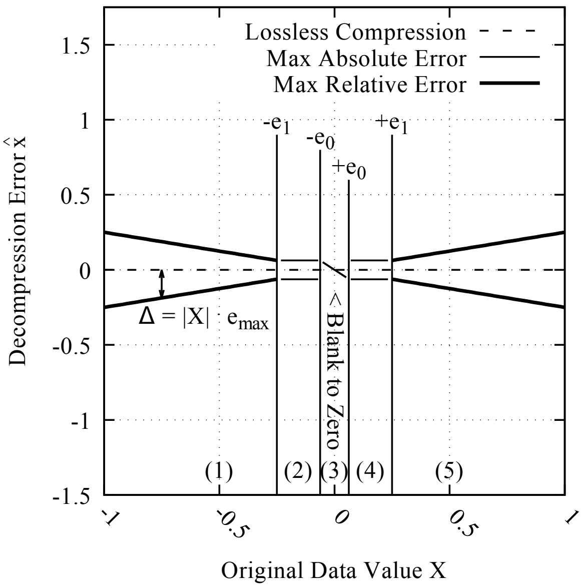


Figure 3.1: Policy for maximum decompression error. Depending on e_{MAX} and e_1, the x-axis is partitioned into five sections: (1),(5) $e_1 \leq |x|$, values are handled with a relative error definition according to Eq. (3.2) (*i*). (2),(4) $e_0 \leq |x| < e_1$, values are handled using a relaxed error definition according to Eq. (3.2) (*ii*). (3) $|x| < e_0 = e_{\mathrm{MAX}} \cdot e_1$, values are blanked to zero according to Eq. (3.2) (*iii*).

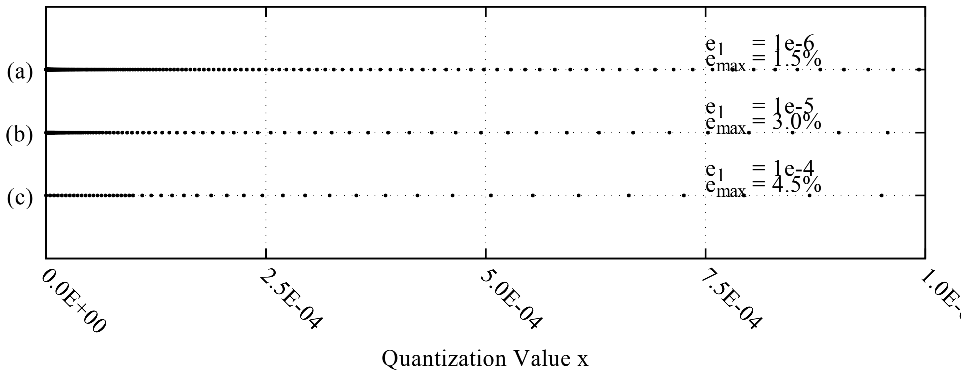


Figure 3.2: Quantization grid resulting from error mapping defined by e_{MAX} and e_1. The quantization grid is constructed using Eq. (3.2) for (a) $e_{\mathrm{MAX}} = 1.5\,\%$, $e_1 = 1\mathrm{E}{-6}$, (b) $e_{\mathrm{MAX}} = 3.0\,\%$, $e_1 = 1\mathrm{E}{-5}$, and (c) $e_{\mathrm{MAX}} = 4.5\,\%$, $e_1 = 1\mathrm{E}{-4}$.

3.3 Lossy Float Compression using Scalar Quantization

In the next sections, two different approaches for *In-Situ Compression* are presented, namely *In-situ Sort-And-B-spline Error-bounded Lossy Abatement* (ISABELA) [45] and *Set Based Decomposition* (SBD) [36]. The algorithms transform `FLOAT` data into an internal representation, which is compressed using fast GPLCs. Based on the ideas of ISABELA and SBD, the *Grid Linearization And Truncation Encoding* (GLATE) algorithm is proposed for the first time. GLATE constitutes a new method for scalar quantization and compression of `FLOAT` data.

3.3.1 In-situ Sort-And-B-spline Error-bounded Lossy Abatement

In-situ Sort-And-B-spline Error-bounded Lossy Abatement (ISABELA) [45] is a lossy in-situ compression algorithm for SFPD which supports a user-defined bound on the maximum error. ISABELA is fast and does not require communication in parallel applications. As ISABELA uses sorting as a preconditioner, it achieves high data reduction for turbulent CFD simulations at excellent error rates [45].

ISABELA Compression Procedure

The ISABELA algorithm is based on four steps illustrated in Fig. 3.3 (1)–(4). First, a linear stream of floating point data is produced, and second, the data is sorted. Third, a B-spline fit is obtained for the sorted data, and forth, an error correction mechanism is applied in order to ensure the bound on the maximum error. ISABELA replaces the noisy `FLOAT` values by a sort order, B-spline control points and quantized errors. In step (2), the sort order has to be stored, because the sorting needs to be reversed during decompression.

The internal mechanisms of ISABELA are visualized using a stream of $N = 512$ normal distributed random values $\mathcal{N}(0, 1)$. The parameter N specifies the length of the input stream of floating point values to be processed with ISABELA. After sorting the data sample, the sorted sequence has a smooth monotonic shape, as illustrated in

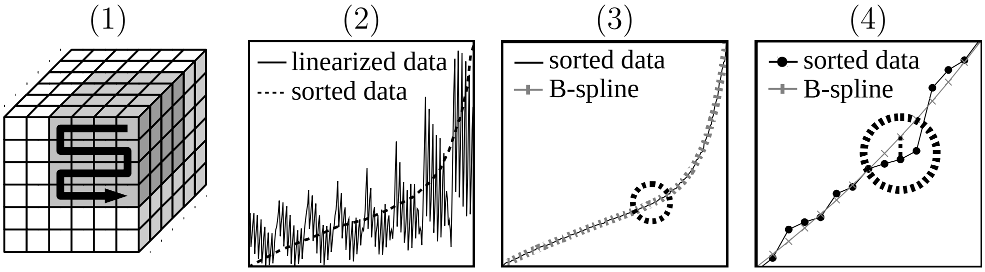


Figure 3.3: ISABELA compression procedure. (1) Linearize data, (2) sort data, (3) B-spline regression, and (4) error quantization.

Fig. 3.4 (*a*).

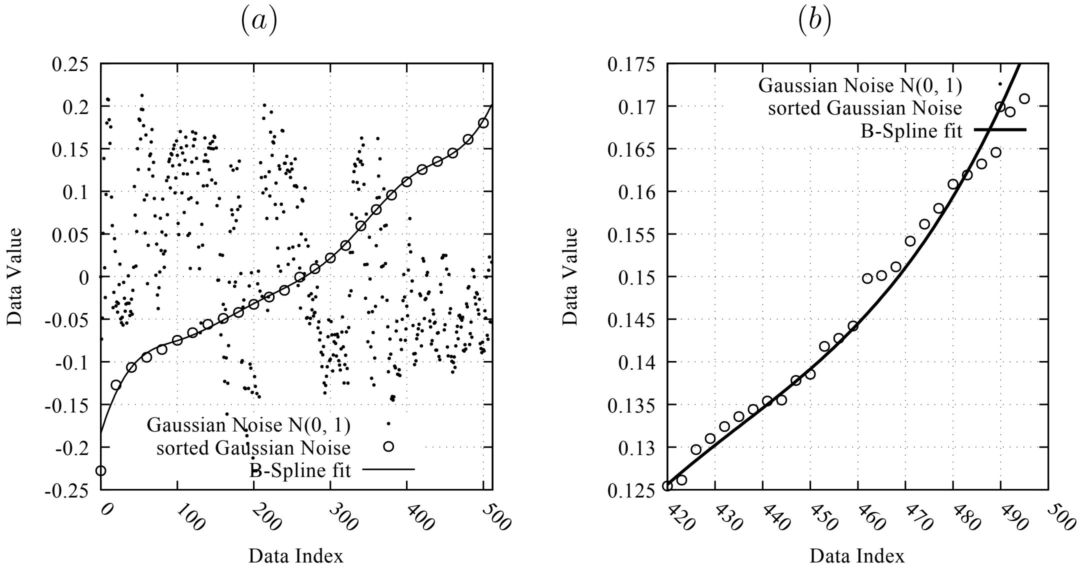


Figure 3.4: ISABELA B-spline fit of sorted data. (a) B-spline approximation $\hat{X}$ of $N = 512$ sorted data values, and (b) magnified differences between B-spline approximation $\hat{X}$ and original data X.

The B-spline fit is computed using $n_\mathrm{B} = 16$ B-spline control points, corresponding to only ~1.5 % (16/512 × 100 %) of the input stream size. Fig. 3.4 (*b*) shows the magnified differences between the B-spline and the sorted data. As the sorted data has a smooth shape, the B-spline fit can be carried out using a low number of B-spline control points $n_\mathrm{B} \ll N$ [45].

ISABELA Error Correction Mechanism

For each approximate value in the B-spline $\hat{X}[i]$ $(i = 0 \dots N-1)$, the error residual $X[i] - \hat{X}[i]$ between original data and approximation is represented by a integer value q_i. The value quantizes the error residual using the step width $\Delta > 0$ and is calculated according to the following equation:

$$q_i = \left\lfloor \frac{(X[i] - \hat{X}[i])}{\Delta} + 0.5 \right\rfloor \tag{3.4}$$

In order to ensure a point-wise maximum decompression error of e_{MAX} percent, the decompressed value $\tilde{X}[i]$ is allowed to fluctuate around the original value $\tilde{X}[i] = X[i] \pm e_{\text{MAX}} \cdot |X[i]|$. Given the quantization step width Δ, the decompressed value is reconstructed using the B-spline approximation $\hat{X}[i]$ and the quantized error q_i according to the following equation:

$$\tilde{X}[i] = \hat{X}[i] + q_i \cdot \Delta \tag{3.5}$$

The smallest valid quantization step width depends on the original data value, i.e. $\Delta = 2 \cdot e_{\text{MAX}} \cdot |X[i]|$. Since the original data value is not known during decompression, the step width has to be predicted. In order to implement the policy described in Section 3.2, four indices $0 \leq i^-_{e1} \leq i^-_{e0} \leq i^+_{e0} \leq i^+_{e1} \leq N-1$ are stored together with the sort order. The indices are used to partition the sorted input stream X into five sets $X_{(1)-(5)}$, which correspond to the five sections (1)–(5) labeled on the x-axis shown in Fig. 3.1.

$$\begin{array}{rclcrcccc}
i \in X_{(1)} & = & [0 \dots i^-_{e1} - 1] & \rightarrow & & & X[i] & < & \text{-}e_1 \\
i \in X_{(2)} & = & [i^-_{e1} \dots i^-_{e0} - 1] & \rightarrow & \text{-}e_1 & \leq & X[i] & < & \text{-}e_0 \\
i \in X_{(3)} & = & [i^-_{e0} \dots i^+_{e0} - 1] & \rightarrow & \text{-}e_0 & \leq & X[i] & < & +e_0 \\
i \in X_{(4)} & = & [i^+_{e0} \dots i^+_{e1} - 1] & \rightarrow & +e_0 & \leq & X[i] & < & +e_1 \\
i \in X_{(5)} & = & [i^+_{e1} \dots N - 1] & \rightarrow & +e_1 & \leq & X[i] & &
\end{array} \tag{3.6}$$

The indices $\{i^-_{e1}, i^-_{e0}, i^+_{e0}, i^+_{e1}\}$ are determined by scanning the sorted input stream X while checking the relations given above. Depending on the indices referring to positions in the sorted data, the corresponding case (i)–(ii) in Eq. (3.2) is used for error compensation. All values at positions $i \in X_{(3)}$ are blanked to zero. For values at positions $i \in X_{(2)} \cup X_{(4)}$ the constant step width $\Delta = e_0 = e_{\text{MAX}} \cdot e_1$ is used for the relaxed error definition for small numbers. The step width for values at positions $i \in X_{(1)} \cup X_{(5)}$ is predicted to be close to the optimal step width $\Delta = 2 \cdot e_{\text{MAX}} \cdot |X[i]|$.

For brevity, the prediction is illustrated for positive numbers at positions $i \in X_{(5)} = [i^{+}_{e1} \ldots N-1]$ only. For negative numbers at positions $i \in X_{(1)}$ the same procedure is carried out. The choice of a valid step width $\Delta \leq 2 \cdot e_{\text{MAX}} \cdot |X[i]|$ is accomplished through a value $\delta[i] \leq X[i]$, which is smaller or equal to the i-th sorted original value and used to predict the step width $\Delta = 2 \cdot e_{\text{MAX}} \cdot \delta[i]$.

The first value $\delta[i = i^{+}_{e1}] = \bar{\delta}$ is initialized as the smallest positive value larger or equal to e_1, i.e. $\bar{\delta} = X[i^{+}_{e1}]$. $\bar{\delta}$ is stored together with the B-spline control points and used to start the estimation of the quantization step width during the error correction procedure. The next value $\delta[i+1]$ is predicted using the last decompressed value $\tilde{X}[i]$ according to $\delta[i+1] = \tilde{X}[i] \cdot p_{\Delta}$. The correction factor p_{Δ} is chosen, to that $\tilde{X}[i] \cdot p_{\Delta} \leq X[i+1]$ holds for all $i \in X_{(5)}$ and $\Delta = 2 \cdot e_{\text{MAX}} \cdot \delta[i+1]$ is a valid step width for the quantization of the error of the next value.

In the worst case, two subsequent values in the sorted stream $X[i] = X[i+1]$ are equal, and the factor p_{Δ} is used to scale the decompressed value so that $\tilde{X}[i] \cdot p_{\Delta} \leq X[i+1]$. The scaling factor is chosen as $p_{\Delta} = {}^{1}/_{(1+e_{\text{MAX}})}$, which is calculated according to the following relation:

$$
\begin{aligned}
p_{\Delta} \cdot \cancelto{=(1+e_{\text{MAX}})\cdot X[i] \text{ in worst case}}{\tilde{X}[i]} \quad &\leq \quad X[i+1] \\
p_{\Delta} \cdot (1 + e_{\text{MAX}}) \cdot X[i] \quad &\leq \quad X[i+1] \\
p_{\Delta} \quad &\leq \quad \frac{1}{(1+e_{\text{MAX}})} \cdot \cancelto{=1 \text{ in worst case}}{\frac{X[i+1]}{X[i]}} \\
p_{\Delta} \quad &\leq \quad \frac{1}{(1+e_{\text{MAX}})}
\end{aligned}
\tag{3.7}
$$

The accuracy of the B-spline without error correction is given in Fig. 3.5 (a). There, the distribution of the relative error between sorted data X and B-spline approximation $\hat{X}$ is shown. The error is given by $E_{\text{REL}}(x, \hat{x})$ as described in Section 3.2, p. 30. Although only $n_{\text{B}} = 16$ B-spline control points have been used, the error of more than half of all data values is ~5 % or better. The largest error occurs at the intersection with zero, where the error correction demands small quantization steps in order to achieve the desired precision. Fig. 3.5 (b) shows the distribution of the relative error for decompressed data using $e_{\text{MAX}} = 1\,\%$ and $e_1 = 1\text{E}{-6}$.

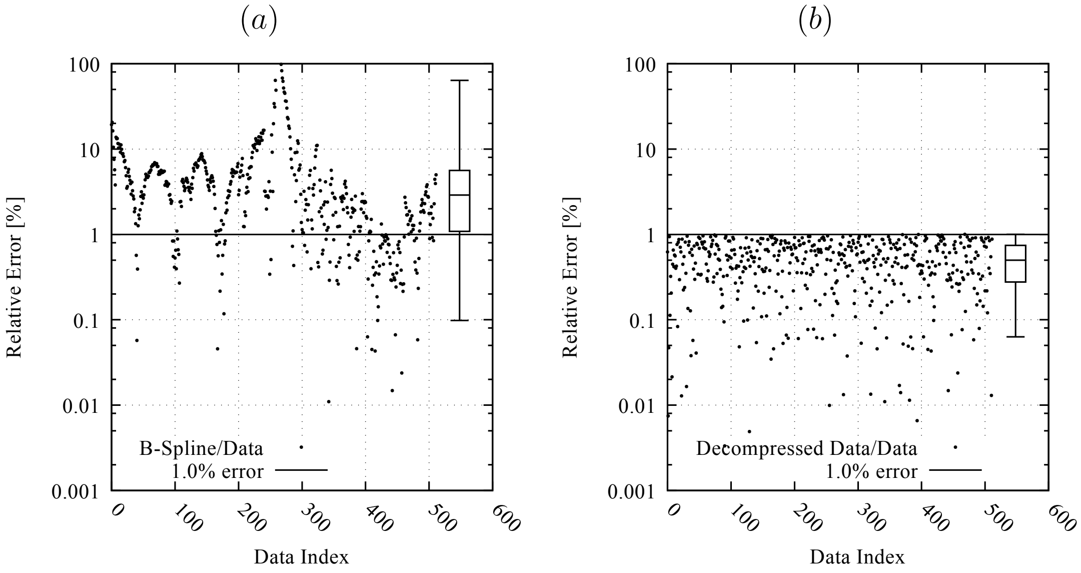


Figure 3.5: ISABELA B-spline accuracy and distribution of decompression error. (a) Error between sorted data values X and B-spline approximation $\hat{X}$, and (b) error between sorted data values and decompressed values.

ISABELA Compression Rate

Sorting the data values creates smooth sequences for the B-spline fit. However, the sort order $\mathcal{S}$ has to be stored in order to rearrange the values during decompression. Compared to the B-spline control points, the sort order takes up more space, e.g. for $N = 512$, the sort order takes up ~28.1 $\%_{32\text{b}}$ corresponding to 9 bit per value, and the B-spline control points take up ~3.1 $\%_{32\text{b}}$ for $n_\text{B} = 16$ control points. As the size for the sort order and the B-spline control points are independent of the data values to be compressed, ISABELA has fixed start-up costs defined by N and n_B. The overall ISABELA compression rate for `FLOAT` data including start-up costs is given by the following equation:

$$\text{CR}^{\text{ISA}}(N, n_\text{B}) = \left(\frac{\lceil \text{ld}(N) \rceil}{32} + \frac{n_\text{B}}{N} + \frac{\text{ERR}^{\text{ISA}}}{4 \cdot N} \right) \times 100\,\%_{32\text{b}} \tag{3.8}$$

ERR^{ISA} denotes the number of bytes required to compress the quantized errors using a GPLC. This size is unknown and determined during run-time of the algorithm. For `FLOAT` data, the B-spline control points take up $(n_\text{B} \cdot 32\,\text{bit})/(N \cdot 32\,\text{bit}) = n_\text{B}/N \times 100\,\%_{32\text{b}}$. The integer values of the sort order occupies $(N \cdot \lceil \text{ld}(N) \rceil)/(N \cdot 32\,\text{bit}) =$

$\lceil \mathrm{ld}(N) \rceil / (32\,\mathrm{bit}) \times 100\,\%_{32b}$.

ISABELA Implementation

The ISABELA compression procedure ISA_ compress (...) is given in Alg. 3.1 and corresponds to the algorithmic steps after data linearization, as depicted in Fig. 3.3 (2)–(4). The stream of sorted data X is obtained using a sorting algorithm called SpreadSort, which is a distribution-based sorter like bucket-sort using quicksort for sorting bins [74]. SpreadSort turned out to outperform Std: : Sort and other famous sort implementations on scientific data [57], and is contained in the `Boost C++` library. The sorting step yields the sorted data X, as well as, the sort order $\mathcal{S}$.

The indices $\{i^-_{e1},\, i^-_{e0},\, i^+_{e0},\, i^+_{e1}\}$ are required for error quantization and are found using the binary search implementation Std: : UpperBound(`lo`, `hi`, `ptr`, `val`) available in the `C++` standard library. After the indices are obtained, the minimum value $\bar{\delta}$ in the input stream greater than e_1 is determined. $\bar{\delta}$ is required to start the estimation of the quantization step width during error quantization.

Using the sorted data, a B-spline of degree three is fitted and reconstructed using the procedure calls BSplineFit3(`size`, `data`, `num_coeff`) and BSplineSample3(`size`, `coeff`, `num_coeff`). The proedure call ISA_ quant_ error (...) performs the quantzation of errors between sorted data X and the reconstructed B-spline approximation $\hat{X}$ using the indices $\{i^-_{e1},\, i^-_{e0},\, i^+_{e0},\, i^+_{e1}\}$ for determination of the corresponding case in Eq. (3.2) (i)–(iii), p. 31.

At the end, of the ISABELA algorithm, the sort order $\mathcal{S}$ is encoded into $\lceil \mathrm{ld}\, N \rceil$ bit numbers. The indices $\{i^-_{e1},\, i^-_{e0},\, i^+_{e0},\, i^+_{e1}\}$, as well as, the n_B B-spline control points are copied in a stream. The quantized errors $\mathcal{Q} = (q_0 \ldots q_{N-1})$ are stored compressed using a GPLC.

```
def ISA_compress($N$, $n_B$, in_data, $e_{MAX}$, out_stream):
  // $N$ – size of input data
  // $n_B$ – number of B-spline control points
  [$X$, $\mathcal{S}$] = Boost::SpreadSort(in_data)
  // $X$ – sorted data
  // $\mathcal{S}$ – sort order
  $i^-_{e1}$ = Std::UpperBound( 0, $N$ – 1, $X$, $-e_1$)
  $i^-_{e0}$ = Std::UpperBound($i^-_{e1}$, $N$ – 1, $X$, $-e_0$)
  $i^+_{e0}$ = Std::UpperBound($i^-_{e0}$, $N$ – 1, $X$, $+e_0$)
  $i^+_{e1}$ = Std::UpperBound($i^+_{e0}$, $N$ – 1, $X$, $+e_1$)
  $\bar{\delta}$ = min(abs($X$[$i^-_{e1}$]), $X$[$i^+_{e1}$])
  coeff = BSplineFit3($N$, $X$, $n_B$) // calc B-spline control points
  $\hat{X}$ = BSplineSample3($N$, coeff, $n_B$) // reconstruct B-spline
  $\mathcal{Q}$ = ISA_quant_error($N$, $X$, $\hat{X}$, [$i^-_{e1}$, $i^-_{e0}$, $i^+_{e0}$, $i^+_{e1}$, $\bar{\delta}$], $e_{MAX}$)
  out_stream.nbit_encode($\lceil \mathrm{ld}\, N \rceil$, [$\mathcal{S}$[0], ..., $\mathcal{S}$[$N$ – 1]])
  out_stream.write([$i^-_{e1}$, $i^-_{e0}$, $i^+_{e0}$, $i^+_{e1}$, $\bar{\delta}$])
  out_stream.write([coeff[0], ..., coeff[$n_B$ – 1]])
  out_stream.gplc_compress([$\mathcal{Q}$[0], ..., $\mathcal{Q}$[$N$ – 1]])
```

Algorithm 3.1: Pseudo code of ISABELA compression procedure ISA_compress(N, n_B, `in_data`, e_{MAX}, `out_stream`). `out_stream.nbit_encode`(n_{BIT}, `data`) copies a vector of n_{BIT}-bit numbers into the stream, `out_stream.write`(`data`) copies uncompressed data into the stream, and `out_stream.gplc_compress`(`data`) copies compressed data into the stream using a GPLC.

3.3.2 Set Based Decomposition with Sequential Encoding

Set Based Decomposition (SBD) with *Sequential Encoding* [36] is a lossy compression algorithm for SFPD based on scalar quantization. SBD was designed for the compression of floating point data associated with unstructured grids.

SBD Compression Procedure for Unstructured Grids

The underlying topology of the simulation grid is taken into account by employing a mathematical graph $G = (V, E)$, which contains vertices $v_i \in V$ and edges in E. The vertices v_i represent the grid cells of the numerical simulation grid and are associated with data values x_i, e.g. flow vectors for CFD simulations. The edges describe the neighboring relation between grid cells.

The SBD algorithm decomposes the vertex set V into so called ϵ-sets W_j, containing indices i of the vertices v_i. Each index i is contained in only one set. The union of all ϵ-sets contains the indices of all vertices. The index i of vertex v_i is contained in the ϵ-set W_j, if and only if x_i is contained in an interval around the center point $\hat{w}_j$ associated with the ϵ-set W_j.

$$\hat{w}_j - \epsilon_W < x_i \leq \hat{w}_j + \epsilon_W \quad \rightarrow \quad i \in W_j \tag{3.9}$$

The value ϵ_W describes the maximum allowed error around the center point. At the end of the decomposition process, a total amount of n_L different center points $\hat{w}_j$ together with their corresponding ϵ-sets W_j are present for $j = 1, 2, 3, \ldots, n_L$. The compression effect is produced by replacing the data value x_i with the index of their corresponding ϵ-set W_j according to the following relation:

$$i \in W_j \quad \rightarrow \quad x_i := j \tag{3.10}$$

Therefore, the set of center points $\hat{w}_j$ constitutes a so-called *Look-Up Table* (LUT) $L[i] = \hat{w}_j$, which is used for replacement of the data by the so-called LUT index j. At the end of the procedure, each grid cell i contains an integer index j. Because SFPD is clustered and typically covers a relatively small value range of the floating point domain, the procedure results in a total amount n_L of different center points $\hat{w}_j$ smaller than the input stream size N. As the stream of LUT indices contains many repetitions, it can be compressed efficiently using GPLCs.

SBD Compression Procedure for Uniform Grids

In order to compress CFD data contained in uniform grids, the SBD compression procedure is simplified. For the present compression task, the error of each decompressed data value is restricted by e_{MAX}, which conforms to the error policy described in Section 3.2, p. 30.

The SBD algorithm on uniform grids is based on four steps, as illustrated in Fig. 3.6 (1)–(4). First, a linear stream of floating point data is produced. Second, the data is sorted. Third, a step function is determined on the sorted data using SBD version 1 [36]. The step function ensures the error bound e_{MAX} according to Eq. (3.2). Fourth, all distinct values of the step function are stored in a so-called LUT $L[j]$, and data values x_i are replaced by their so-called LUT index j. The resulting stream of LUT indices can be compressed efficiently using a GPLC.

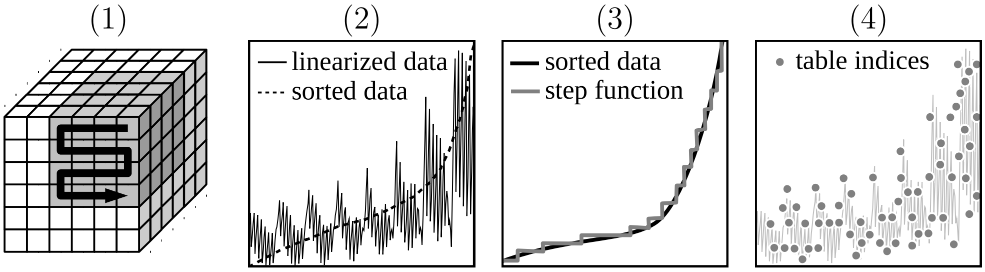


Figure 3.6: SBD compression procedure. (1) Linearize data, (2) sort data, (3) determine LUT values, and (4) map data values to LUT indices.

Fig. 3.7 (a) shows the same input stream consisting of $N = 512$ normal distributed data values as used for visualizing the internal mechanisms of ISABELA. The figure also shows the sorted data and the step function containing the LUT values $L[j]$. The step function is determined using $e_{\mathrm{MAX}} = 1.00\,\%$ and $e_1 = 1\mathrm{E}{-6}$. Fig. 3.7 (b) illustrates the magnified differences between the step function and the sorted data. As can be seen, several values are mapped to one and the same LUT value $L[j]$.

Fig. 3.8 (a) shows the LUT for the example data. The number of LUT values $n_L = 196$ is much smaller than the number of values in the input stream $N = 512$. Fig. 3.8 (b)

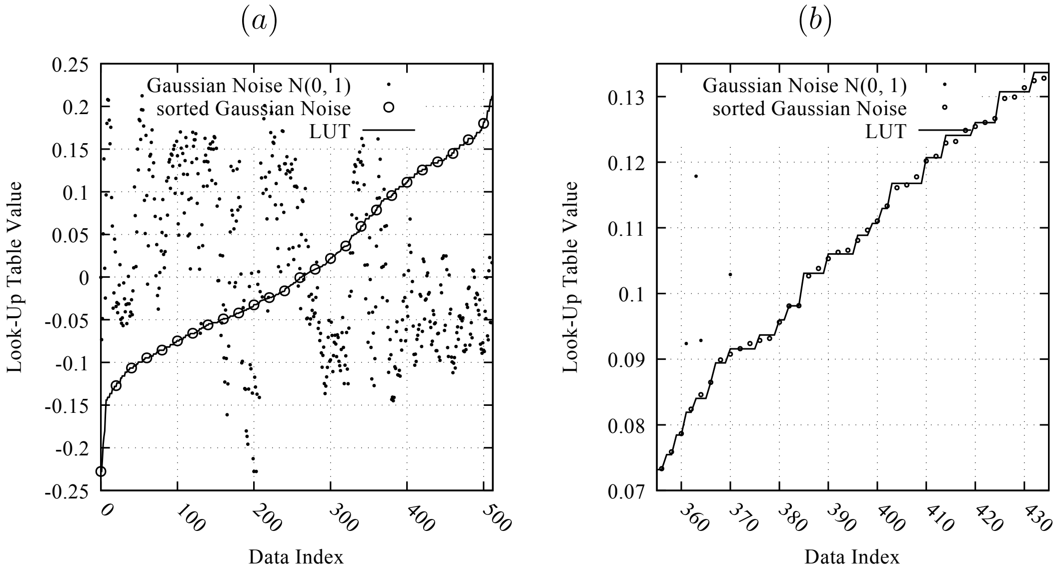


Figure 3.7: Calculation of LUT values inside SBD compression procedure. (a) Each sorted value is mapped to its LUT value. (b) Magnified differences between LUT values and original data.

shows the resulting stream of LUT indices, which reflects the trend of the original data values plotted in Fig. 3.7 (*a*).

SBD Compression Rate

The resulting compression rate of SBD is given by the ratio between the size of the LUT relative to the size of the input stream, as stated by the following equation:

$$\mathrm{CR}^{\mathrm{SBD}}(N, n_L) = \frac{n_L}{N} + \frac{\mathrm{IDX}^{\mathrm{SBD}}}{4 \cdot N} \times 100\,\%_{\text{32b}} \tag{3.11}$$

$\mathrm{IDX}^{\mathrm{SBD}}$ denotes the number of bytes required to store the compressed LUT indices. The compression effect in SBD is created as the number n_L of data values in the LUT is much lower compared to the input stream size N. Further, due to repeating LUT indices j, the stream of LUT indices exhibits increased redundancy allowing for very good compression rates using GPLCs.

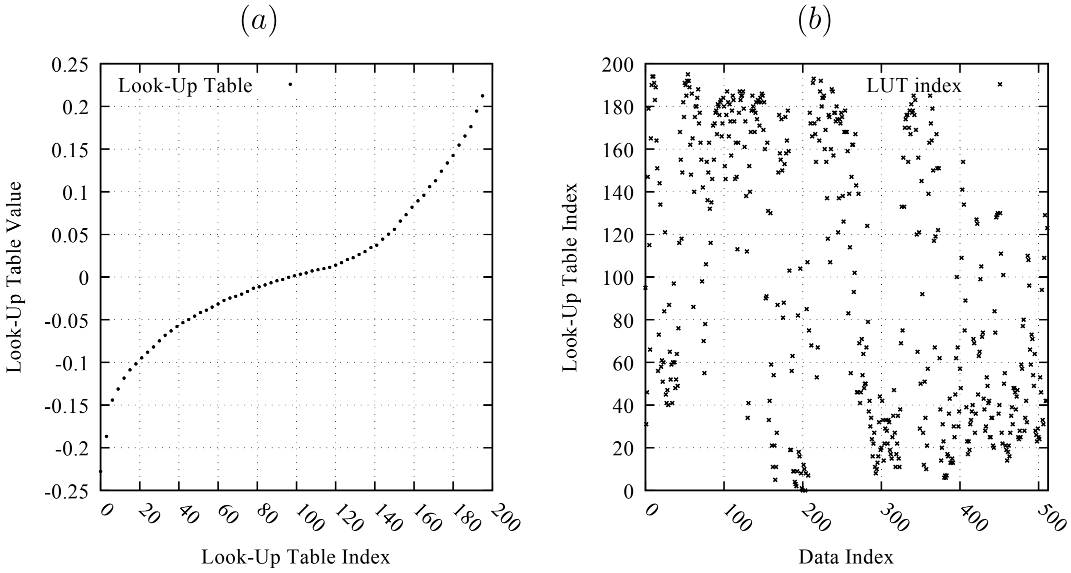


Figure 3.8: Quantization of LUT indices inside SBD compression procedure. (a) LUT containing $n_L = 196$ LUT values. The original data sample contained $N = 512$ normal distributed random values. (b) LUT indices in the order of original data values. LUT indices reflect the trend of the data.

SBD Implementation

The SBD compression procedure SBD_compress(...), given in Alg. 3.2, corresponds to the algorithmic steps after data linearization depicted in Fig. 3.6 (2)-(4). As for ISABELA, the stream of sorted data X and the sort order `perm` is obtained using the SpreadSort algorithm. The LUT is constructed by scanning the sorted sequence X and collecting indices of data values until the bounds of the bin exceed the limits of the maximum error e_{MAX}. The function SBD_bin_error(`binmin`, `binmax`) indicates if the next value in the input stream exceeds the error bound, and a new bin has to be created. The function SBD_bin_value(`binmin`, `binmax`) returns the center point w_j for the current bin and reflects the decompressed value. The LUT size n_L and the LUT itself are stored uncompressed. The LUT indices are compressed using a GPLC.

```
def SBD_compress($N$, in_data, $e_{\mathrm{MAX}}$, out_stream):
  // $N$ - size of input data
  $n_L$ = 0 // LUT size
  lutv = [] // LUT values
  luti = [] // LUT indices
  [$X$, perm] = Boost::SpreadSort(in_data)
  // $X$ - sorted data
  // perm - sort order
  binmin = $X$[0] // minimum bin value
  for i in range(1, $N$):
    if SBD_bin_error(binmin, $X$[i]) > $e_{\mathrm{MAX}}$:
      // new value in old bin exceeds maximum error
      lutv.append(SBD_bin_value(binmin, $X$[i - 1]))
      binmin = $X$[i]
      $n_L$ += 1
    luti[perm[i]] = $n_L$ // set LUT index
  if luti[perm[$N$ - 1]] = $n_L$:
    // if last value has a separate bin
    lutv.append($X$[$N$ - 1])
  out_stream.write([$n_L$, lutv[0], ..., lutv[$n_L$ - 1]])
  out_stream.gplc_compress([luti[0], ..., luti[$N$ - 1]])
```

Algorithm 3.2: Pseudo code of SBD compression procedure SBD_ compress (N, `in_data`, e_{MAX}, `out_stream`). `out_stream.write`(`data`) copies uncompressed data into the stream, and `out_stream.gplc_compress`(`data`) copies compressed data into the stream using a GPLC.

3.3.3 Grid Linearization And Truncation Encoding

Grid Linearization And Truncation Encoding (GLATE) is a lossy compression algorithm for 32 bit `FLOAT` data. GLATE uses scalar quantization and uses GPLC for the compression of streams of quantized floating point data. Such streams are created using linearization schemes on simulation grids as explained in Section 2.2, p. 13. GLATE uses an optimized quantization scheme, which differentiates between the exponents and the mantissa of `FLOAT` values. Whereas the streams for exponents are smooth, the streams containing mantissas are subject to the noise in the `FLOAT` data.

Unlike ISABELA and SBD, GLATE operates on the IEEE754 bit-representation for `FLOAT` values. Instead of using the full set of 23 bit for `FLOAT` mantissas, GLATE performs quantization and maps the noisy `FLOAT` mantissa to a set of predetermined values, much smaller than 2^{23}. Effectively, this corresponds to the truncation of mantissa bits to a number lower than 23 bit.

Truncation of Lower Mantissa Bits

As can be seen in Eq. (2.1), p. 10, the mantissa in the IEEE754 format describes a sum of negative powers of two multiplied by a power of two. Therefore, the maximum error e_{MAX} for the decompressed data can be controlled by the number n_{T} of highest mantissa bits restored. This corresponds to truncation of the 23 bit mantissa to its n_{T} highest bits, i.e. $n_{\text{T}} < 23$. Table 3.1 shows the relationship between the number of bits and the resulting relative error.

n_{T}	[bit]	7	8	9	10	...	14	15	16
$1/2^m \times 100$	[%]	0.781	0.390	0.195	0.097	...	0.006	0.003	0.001

Table 3.1: Relative error resulting from throwing away lower mantissa bits. Number of reconstructed highest mantissa bits n_{T} is directly related to the relative error of the `FLOAT` value.

As can be seen, the reconstruction of the highest 7 bit during decompression ensures an error bound of less than 1.00 %. The reconstruction of the highest 8 bit ensures an error bound of less than 0.50 %. However, truncating the mantissa by throwing away the lowest bits results in a non-uniform maximum relative error, as shown in Fig. 3.9 (*a*).

The error varies, because the resulting mantissas for quantization are distributed in a uniform manner. Therefore, the relative error decreases as values increase.

However, an uniform error maximum for decompressed data is desired. Fig. 3.9 (*b*) shows the desired error distribution with a uniform maximum for the relative error. In the following, the GLATE compression procedure is extended to produce the error distribution as shown in Fig. 3.9 (*b*).

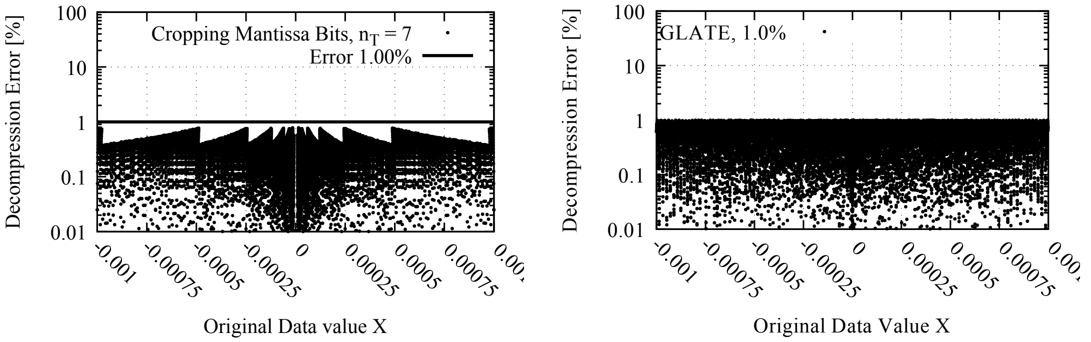

(*a*) (*b*)

Figure 3.9: Maximum relative error of GLATE quantized float values. (a) Non-uniform maximum error resulting from throwing away lower bits of the mantissa. (b) Uniform maximum relative error of decompressed data to be implemented in GLATE.

GLATE Compression Procedure

As stated above, truncation of the lower mantissa bits does not lead to the desired result. Instead, the GLATE algorithm is based on three steps illustrated in Fig. 3.10 (1)–(3). First, a locality-preserving linearization scheme is applied. Second, each value is decomposed into 1 bit sign, 8 bit exponent and 23 bit mantissa. The mantissa values are encoded using indices referring to a LUT containing n_{M} quantized mantissa values according to the precision demanded by the maximum allowed error e_{MAX}. The sign bit is `XOR`-ed with the sign of the previous value and merged with the mantissa as higher bits. Third, the exponents and the mantissas are separated into two streams of integer values, which are compressed independently using a GPLC and the `tpfor` bit packing codec.

As the data differences between adjacent grid cells are unlikely to differ by orders of magnitude unless simulation-specific physical phenomena are present [37], the exponents

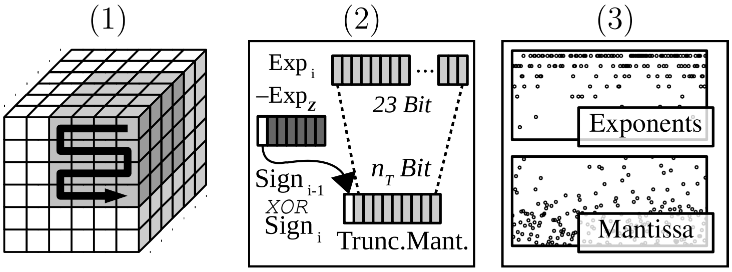


Figure 3.10: GLATE compression procedure. (1) Linearize data, (2) decompose `FLOAT` data into exponents and signs & mantissas, (3) compress exponents and signs & mantissas.

are extracted from `FLOAT` values and compressed separately. Similarities between the signs of consecutive values are encoded using the `XOR`-operation. Whereas the exponents are compressed using a GPLC, the noise of the truncated mantissas is bounded by a maximum `UNSIGNED` value and compressed using `tpfor` bit packing, as motivated in Section 2.3, p. 16.

GLATE Mantissa Look-Up Table

As can be seen in Fig. 3.9 (a), throwing away lower mantissa bits results in a non-uniform error distribution. In order to construct a set of mantissa values which results in a uniform maximum relative error throughout the quantization of all values $|x| \geq e_0$, the set of mantissas used for quantization is determined according to the error policy proposed in Section 3.2, p. 30. Therefore, all numbers $|x| < e_0$ are blanked to zero during decompression. Further, for $|x| \geq e_0$, a step function S_i beginning at e_0 is constructed using the following equation:

$$\begin{aligned} T_i &= e_0 \cdot \left(\frac{1 + e_{\mathrm{MAX}}}{1 - e_{\mathrm{MAX}}} \right)^i \\ S_i &= T_{\lfloor i+0.5 \rfloor} \end{aligned} \tag{3.12}$$

Fig. 3.11 shows S_i with $e_0 = 1.0$ and $e_{\mathrm{MAX}} = 10.00\,\%$ for $i = 0, 1, 2, 3, 4$. The high error bound of $e_{\mathrm{MAX}} = 10.00\,\%$ is only chosen for illustrative purposes in this explanation and is not to be used for actual compression of `FLOAT` data. Since the step function increases the step width, an error distribution similar to Fig. 3.9 (b) is created. The quantization of value x requires the determination of the index i for $x = T_i$, which involves the calculation of the inverse function, e.g. $T_i = 2.0 \to i = 3.45$. The equation for the step function index i is derived from Eq. (3.12) and shown in the following equation:

$$i = \frac{\log \dfrac{x}{e_0} - \log \dfrac{1 + e_{\text{MAX}}}{1 - e_{\text{MAX}}}}{\log \dfrac{1 + e_{\text{MAX}}}{1 - e_{\text{MAX}}}} \tag{3.13}$$

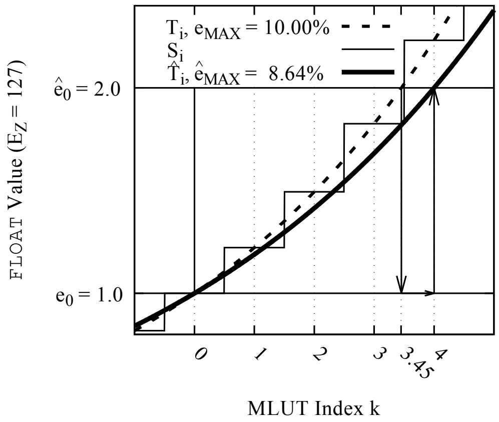


Figure 3.11: GLATE adjustment of error bound for exponent-aligned quantization. The adjustment of the error bound $e_{\text{MAX}} = 10.00\,\%$ for alignment of step function with floating point exponents of e_0, $\hat{e}_0$. A slightly lower error bound $\hat{e}_{\text{MAX}} = 8.64\,\%$ is computed by rounding up the intersection point $j = \lceil i = 3.45 \rceil = 4$ with $\hat{e}_0 = 2.0$, as illustrated using arrows.

GLATE Step Function Adjustment

In order to be able to decompose exponents and mantissas during quantization, a new step function $\hat{T}_j$ is introduced and adjusted in a manner to pass $\hat{T}_0 = e_0$ at $j = 0$ and $\hat{T}_j = \hat{e}_0$ at one later step $j > 0$. Interestingly, by choosing $e_0 = 1.0$ with exponent $E_Z = 127$ and mantissa $0 =$ `0x000000`, and choosing $\hat{e}_0 = 2 \cdot e_0$ with exponent $E_Z + 1 = 128$ and mantissa $0 =$ `0x000000`, the resulting step function $\hat{S}_i$ is aligned with the floating point exponents $E_Z = 127$ and $E_Z + 1 = 128$, as also shown in Fig. 3.11.

In particular, the alignment of the step function T_i is accomplished by slightly decreasing the maximum allowed error e_{MAX}, as depicted in Fig. 3.11 using arrows. First, the non-integer position i where T_i intersects $\hat{e}_0 = 2.0$ is determined using Eq. (3.13), and rounded up to the next integer value, i.e. for $e_{\text{MAX}} = 10.00\,\%$ and $e_0 = 1.0$, the adjustment of $T_i = \hat{e}_0 = 2.0$ yields $i = 3.45 \rightarrow j = \lceil i \rceil = 4$. Second, the decreased error bound $\hat{e}_{\mathrm{MAX}}$ is calculated by solving T_j for e_{MAX}, as shown in the following equation derived from Eq. (3.12):

$$\hat{e}_{\mathrm{MAX}} = \frac{\left(\dfrac{\hat{e}_0}{e_0}\right)^{1/j} - 1}{\left(\dfrac{\hat{e}_0}{e_0}\right)^{1/j} + 1} = 8.64\,\% \tag{3.14}$$

Third, the decreased error bound $\hat{e}_{\mathrm{MAX}}$ is used for the construction of the adjusted step function $\hat{S}_i$, as shown in the following equation:

$$\begin{aligned} \hat{T}_i &= e_0 \cdot \left(\frac{1 + \hat{e}_{\mathrm{MAX}}}{1 - \hat{e}_{\mathrm{MAX}}}\right)^i \\ \hat{S}_i &= \hat{T}_{\lfloor i+0.5 \rfloor} \end{aligned} \tag{3.15}$$

As can be seen in Fig. 3.12, $\hat{S}_i$ is adjusted to the minimum and maximum floating point values $1.0 = e_0 \leq x < \hat{e}_0 = 2.0$ within the scope of the floating point exponent $E_Z = 127$. The values of $\hat{S}_i$ for $i = 0, 1, 2, 3$ constitute the so-called *Mantissa Look-Up Table* (MLUT), which contains 23 bit `UNSIGNED` numbers, i.e. for $\hat{e}_{\mathrm{MAX}} = 8.64\,\%$ the MLUT consists of $n_{\mathrm{M}} = 4$ values $\hat{S}_0$ =`0x000000`, $\hat{S}_1$ =`0x1837f0`, $\hat{S}_2$ =`0x3504f3` and $\hat{S}_3$ =`0x5744fd`. The MLUT is used for the quantization of the floating point mantissa for values $|x| > e_0$ with any exponent $E \geq E_Z$. Denormalized `FLOAT` values in the range $0 < |x| \leq 1.1754942\mathrm{E}{-38}$ are not supported by the current implementation of GLATE, as they are typically much smaller than e_0.

GLATE Global Step Function

By repeating the values of the MLUT for all exponents E_Z, E_Z+1, E_Z+2, ..., as shown in Fig. 3.13 for $e_{\mathrm{MAX}} = 10.00\,\%$, a global step function is constructed. Furthermore, for any given `FLOAT` value x with any exponent $E = 1, \ldots, 254$, the MLUT index only depends on the 23 bit mantissa of x. As a consequence, the exponents and mantissas can be encoded separately using GPLC and `tpfor` bit packing.

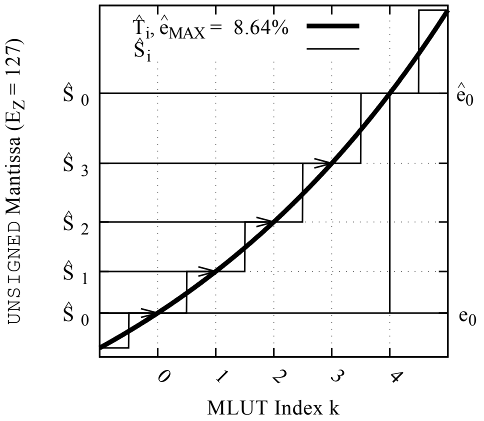


Figure 3.12: GLATE mantissa look-up table. Mantissa values resulting from adjusted error bound $\hat{e}_{\mathrm{MAX}} = 8.64\,\%$ consisting of $n_{\mathrm{M}} = 4$ mantissas for quantization: $\hat{S}_0$ =`0x000000`, $\hat{S}_1$ =`0x1837f0`, $\hat{S}_2$ =`0x3504f3`, $\hat{S}_3$ =`0x5744fd`. The value $\hat{S}_0$ =`0x000000` is repeated, because it corresponds to the first mantissa for the next exponent $E_Z + 1 = 128$.

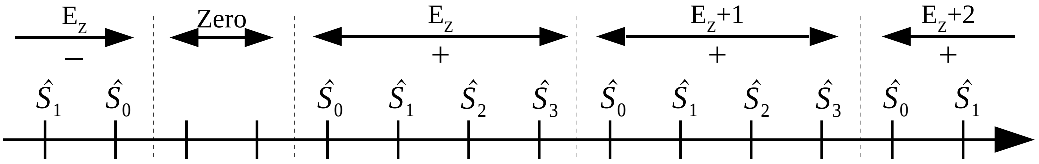


Figure 3.13: GLATE global step function. The global step function used in GLATE is constructed by repeating the quantized mantissas obtained from the MLUT for each exponent E_Z, $E_Z + 1$, $E_Z + 2$, …, e.g. $\hat{S}_0$ =`0x000000`, $\hat{S}_1$ =`0x1837f0`, $\hat{S}_2$ =`0x3504f3`, $\hat{S}_3$ =`0x5744fd` for $e_{\mathrm{MAX}} = 10.00\,\%$. By using the sign bit, positive and negative values on the number line are addressed.

GLATE Mantissa Quantization

As shown in Fig. 3.13 for $e_{\mathrm{MAX}} = 10.00\,\%$, all values enclosed in $e_0 \le x < \hat{e}_0$ exhibit the same exponent $E_Z = 127$. Therefore, each value $e_0 \le x < \hat{e}_0$ is quantized into an exponent E and an mLUT index $k \in \{0, 1, 2, 3\}$ for the corresponding mantissa $\hat{S}_k$.

In order to find the mLUT index k for the quantization of one floating point value x, the sequence $\bar{M} = (\hat{M}_{0.0}, \hat{M}_{0.5}, \hat{M}_{1.5}, \hat{M}_{2.5}, \hat{M}_{3.5}, \hat{M}_{4.0})$ is introduced. Using $\bar{M}$, the mLUT index k can be determined for any float value $|x| \ge e_0$ using binary search. Specifically, two consecutive values in $\bar{M}$ describe the range of 23 bit `UNSIGNED` mantissas, which are mapped to one mLUT value $\hat{S}_k$, e.g. all mantissas M in $\hat{M}_{0.5} \le M < \hat{M}_{1.5}$ are mapped to $\hat{S}_1$ for mLUT index $k = 1$. Therefore, the mantissa values contained in $\bar{M}$ correspond to the transitions of one step of $\hat{S}_k$ to the next step $\hat{S}_{k+1}$, as shown in Fig. 3.14 using arrows.

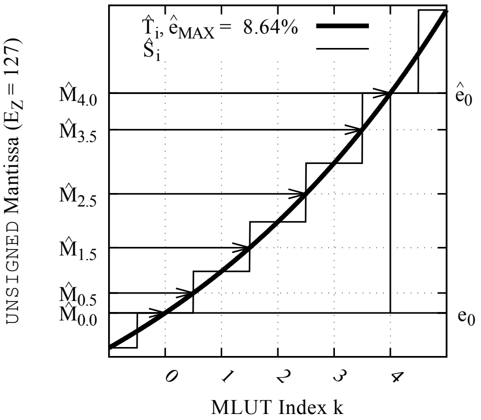


Figure 3.14: GLATE exponent-aligned quantization of mantissa values. mLUT indices are determined by binary search in $\bar{M}$. For adjusted error bound $\hat{e}_{\mathrm{MAX}} = 8.64\,\%$ the sequence $\bar{M}$ consists of $n_{\mathrm{M}} + 1 = 4$ mantissas: $\hat{M}_{0.0}$ =`0x000000`, $\hat{M}_{0.5}$ =`0x0b95c2`, $\hat{M}_{1.5}$ =`0x25fed7`, $\hat{M}_{2.5}$ =`0x45672a`, $\hat{M}_{3.5}$ =`0x6ac0c7`. The last value $\hat{M}_{4.0}$ =`0x800000` is fictitious and corresponds to the next exponent $E_Z + 1 = 128$.

The last interval $\hat{M}_{3.5} \le M < \hat{M}_{4.0}$ (= `0x800000` = 2^{23}) is fictitious and used to map values to the next exponent $E_Z + 1$ and mantissa $\hat{S}_0$ =`0x000000` during quantization. The fictitious interval is introduced in order to be able to rely on binary search in $\bar{M}$ during quantization. Table 3.2 summarizes the GLATE quantization mapping for $e_{\mathrm{MAX}} = 10.00\,\%$.

k	Minimum Mantissa			$\leq$	m	$<$	Maximum Mantissa			$\rightarrow$	Quantized Mantissa		
0	$\hat{M}_{0.0}$	=	`0x000000`	$\leq$	m	$<$	`0x0b95c2`	=	$\hat{M}_{0.5}$	$\rightarrow$	$\hat{S}_0$	=	`0x000000`
1	$\hat{M}_{0.5}$	=	`0x0b95c2`	$\leq$	m	$<$	`0x25fed7`	=	$\hat{M}_{1.5}$	$\rightarrow$	$\hat{S}_1$	=	`0x1837f0`
2	$\hat{M}_{1.5}$	=	`0x25fed7`	$\leq$	m	$<$	`0x45672a`	=	$\hat{M}_{2.5}$	$\rightarrow$	$\hat{S}_2$	=	`0x3504f3`
3	$\hat{M}_{2.5}$	=	`0x45672a`	$\leq$	m	$<$	`0x6ac0c7`	=	$\hat{M}_{3.5}$	$\rightarrow$	$\hat{S}_3$	=	`0x5744fd`
4	$\hat{M}_{3.5}$	=	`0x6ac0c7`	$\leq$	m	$<$	`0x800000`	=	$\hat{M}_{4.0}$	$\rightarrow$	$\hat{S}_0$	=	`0x000000`

Table 3.2: GLATE quantization table for exponent-aligned mantissa values. A truncated set of mantissa values is obtained from an adjusted step function $\hat{S}_i$ for $e_{\mathrm{MAX}} = 10.00\,\%$. The quantization process operates on the mantissa M only. Each mantissa receives a mLUT index $k \in \{0, 1, 2, 3\}$ determined using binary search. The index $k = 4$ is mapped to $k = 0$ for the next exponent.

GLATE Sign Encoding

As stated, the similarities between the signs in the linearized input stream are encoded using the `XOR`-operation on the current sign s and the previous sign $\hat{s}$ in the stream, i.e. the operation $s\,\texttt{XOR}\,\hat{s}$ identifies a flip of the sign bit in the stream. If a sign flip is indicated, the value of the corresponding mLUT index is increased. The decision for encoding the sign bits in the mLUT indices was made due to spatial coherence in linearized streams of `FLOAT` values in CFD grids, where long sequences of values share the same sign. By marking only the positions in the stream where the sign flips, the sequence of sign bits can be transformed to contain many zeros. In detail, if a sign flip is detected using $s\,\texttt{XOR}\,\hat{s}$, the mLUT index k is increased by the mLUT size n_{M}, as shown in the following example.

$$
\begin{array}{c|ccccccccccccccccccc}
\hat{s} & \boldsymbol{\mathit{0}} & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\
s & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & \times \\
\hline
s\,\texttt{XOR}\,\hat{s} & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & \mathbf{1} & \mathbf{1} & \times \\
\hline
k & 0 & 3 & 3 & 2 & 1 & 1 & 0 & 2 & 3 & 2 & 1 & 1 & 0 & 0 & 2 & 3 & 3 & 2 & \times \\
{\scriptstyle (n_{\mathrm{M}}=4)} & & & & & & {\scriptstyle +n_{\mathrm{M}}} & & & {\scriptstyle +n_{\mathrm{M}}} & & & & & & {\scriptstyle +n_{\mathrm{M}}} & & {\scriptstyle +n_{\mathrm{M}}} & {\scriptstyle +n_{\mathrm{M}}} & \\
k' & 0 & 3 & 3 & 2 & 1 & \mathbf{5} & 0 & 2 & \mathbf{7} & 2 & 1 & 1 & 0 & 0 & \mathbf{6} & 3 & \mathbf{7} & \mathbf{6} & \times
\end{array}
\tag{3.16}
$$

Since the first value in the stream has no predecessor, the stream of previous signs $\hat{s}$ is right-shifted by one and the the gap in the beginning is filled with a `0` bit. Concretely, the encoding of the sign transforms the mLUT indices k obtained form quantization into so-called *signed* mLUT indices k'. The signed mLUT indices are equal the mLUT indices k, unless a sign flip was detected through $s \,\texttt{XOR}\, \hat{s} = 1$, then the signed mLUT index is given by $k' = k + n_\mathrm{M} < 2 \cdot n_\mathrm{M}$, as highlighted in the previous example using bold numbers. Therefore, signed mLUT indices k' are in the interval $[0,\, 2 \cdot n_\mathrm{M})$.

GLATE Compression Rate

After the exponents and the mLUT indices are determined for each value in the input stream, the compression effect in GLATE is produced by applying a GPLC to the exponents, and applying `tpfor` bit packing to the signed mLUT indices k'. Since e_0 is the smallest value not blanked to zero, the exponent E_Z of e_0 is subtracted from each exponent E before the compression of the exponents according to $E' = 1 + E - E_Z$. Therefore, the stream of signed mLUT indices reflects the noise, whereas the exponents constitute a smooth stream describing the order of magnitude for the `FLOAT` values. The exponents are compressed using a GPLC. The signed mLUT indices are bounded by the maximum value $2 \cdot n_\mathrm{M}$ corresponding to twice the size of the mLUT and are compressed using `tpfor` bit packing.

The compression rate of GLATE is given by the storage required for the compressed exponents, plus the compressed signed mLUT indices. The overall compession rate of GLATE is given by the following equation:

$$\mathrm{CR}^{\mathrm{GLA}}(N) = \frac{\mathrm{MNT}^{\mathrm{GLA}} + \mathrm{EXP}^{\mathrm{GLA}}}{4 \cdot N} \times 100\,\%_{\text{32b}} \tag{3.17}$$

$\mathrm{EXP}^{\mathrm{GLA}}$ denotes the bytes required for exponents, which have been compressed using a GPLC, and $\mathrm{MNT}^{\mathrm{GLA}}$ denotes the bytes required for signed mLUT indices k' compressed using `tpfor`. Both sizes are determined during run-time of GLATE.

GLATE Implementation

The procedure GLATE_compress(...), given in Alg. 3.3, corresponds to the algorithmic steps after data linearization depicted in Fig. 3.10 (2)–(3). First, the exponent of e_0 is determined using the function GLATE_float_expo(`val`). Second, the size of the mLUT n_M is determined, and the sequence $\bar{M}$ required for mantissa quantization is computed using the function GLATE_float_quant(`errmax`, `erreps`). Third, the streams for exponents and LUT indices are constructed in a `for`-loop. If the exponent is smaller

```
def GLATE_compress($N$, $X$, $e_{\mathrm{MAX}}$, out_stream):
  expo_zero = GLATE_float_expo($e_0$)
  [$n_{\mathrm{M}}$, $\bar{M}$] = GLATE_float_quant($e_{\mathrm{MAX}}$, $e_1$)
  luti = [] // MLUT indices
  expo = [] // exponents
  sign = 0
  for i in range(0, $N$):
    expo[i] = 1 + GLATE_float_expo($X$[i]) − expo_zero
    if expo[i] $\leq$ 0:
      // absolute zero
      expo[i] = 0
      luti[i] = 0
    else:
      // absolute value
      luti[i] = Std::UpperBound(0, $n_{\mathrm{M}}$ + 1, $\bar{M}$, GLATE_float_mant($X$[i]))
      if luti[i] = $n_{\mathrm{M}}$:
        // if first mantissa of next exponent
        expo[i] += 1
        luti[i] = 0
      if GLATE_float_sign($X$[i]) XOR sign = 1:
        // if sign flip detected
        luti[i] += $n_{\mathrm{M}}$ // signed MLUT index
        sign = flip(sign)
  out_stream.gplc_compress([expo[0], ..., expo[$N$ − 1]])
  out_stream.tpfor_pack([luti[0], ..., luti[$N$ − 1]])
```

Algorithm 3.3: Pseudo code of GLATE compression procedure GLATE_compress(N, X, e_{MAX}, `out_stream`). `out_stream.gplc_compress(data)` copies compressed data into the stream using a GPLC, and `out_stream.tpfor_pack(data)` copies compressed data into the stream using `tpfor`.

than the exponent of e_0, a zero is encoded, otherwise the binary search implementation Std::UpperBound(`lo`, `hi`, `ptr`, `val`) from the `C++` standard library is used in order to determine the MLUT index. If the sign changes w.r.t. the previous sign, the MLUT index is increased by n_{M}. The sign and the mantiassa are extracted from float values using GLATE_float_sign(`val`) and GLATE_float_mant(`val`). Finally, the exponents are compressed using a GPLC, and the signed MLUT indices are compressed using `tpfor`.

3.4 Evaluation of Lossy Float Compression on Scientific Data Sets

The compression performance of ISABELA, SBD and GLATE is evaluated on the same data sets `ALU`, `ISO` and `JET` as used for compression tests in Section 2.4, p. 21. Before compression, the grids are subdivided into three dimensional blocks of size $64 \times 64 \times 64$, which are linearized using the Hilbert-curve in order to produce a linear input stream for the in-situ compressors. The compression tests are carried out using $e_{\mathrm{MAX}} = 1.00$, 0.50, 0.25, 0.12 %, and $e_1 = 10^{-6}$.

The subdivision into blocks is considered in order to evaluate the compression rate in simulations, where grids are typically fragmented across parallel processes. The next three sections cover the in-detail compression tests on ISABELA, SBD and GLATE. As explained in Section 2.5, grid linearization based on the Hilbert-curve provides better locality properties than the Z-curve, row-major or column-major ordering.

3.4.1 ISABELA Compression

As mentioned in Section 3.3.1, p. 33, the start-up costs for ISABELA compression depend on the size of the input stream N and on the number of B-spline control points n_{B}. On the one hand, ISABELA requires to store the sort order of the data, and on the other hand, the B-spline fit needs enough parameters to model the sorted sequence accurately. The fixed ISABELA start-up costs bound the minimal compression rate that can be achieved.

ISABELA Fixed Start-Up Costs

ISABELA start-up costs depend on the input stream size N, which implies the number of bits required for the encoding of one index value, i.e. $\lceil \mathrm{ld}(N) \rceil$ bit each. As can be seen in Table 3.3, for ISABELA start-up costs to be less than 50 $\%_{\mathrm{32b}}$ of the original data size, one index value can have at most 16 bit. The sort order is stored uncompressed using ${}^{N}/_{8} \cdot \lceil \mathrm{ld}(N) \rceil$ byte, e.g. ${}^{512}/_{8} \cdot \lceil \mathrm{ld}\, 512 \rceil = 576$ byte corresponding to ~28.1 $\%_{\mathrm{32b}}$ of the input stream size.

The n_{B} B-spline control points required to model the sorted sequence take up ${}^{n_{\mathrm{B}}}/_{N} \times 100$ $\%_{\mathrm{32b}}$, e.g. ~3.1 $\%_{\mathrm{32b}}$ for $n_{\mathrm{B}} = 16$ and $N = 512$. Since SFPD is clustered and usually has

N		256	512	1024	2048	4096	8192	16384	32768
$\lceil \mathrm{ld}(N) \rceil$	[bit]	8	9	10	11	12	13	14	15
$\lceil \mathrm{ld}(N) \rceil / 32$	[$\%_{32b}$]	25.0	28.1	31.2	34.3	37.5	40.6	43.7	46.8

Table 3.3: ISABELA index start-up costs. The storage of the sort order containing N `UNSIGNED` numbers with $\lceil \mathrm{ld}(N) \rceil$ bit each.

a bounded value range, the sorted sequence is smooth, and a low number of B-spline control points is sufficient for modeling the sorted data accurately [45].

Fig. 3.15 (*a*) shows the start-up costs depending on the input stream size N including the storage required for $n_{\mathrm{B}} = 16$ B-spline control points. For $n_{\mathrm{B}} = 16$, the start-up cost is minimal for 256 and 512 `FLOAT` values accounting to ~31.2 $\%_{32b}$ of the input stream size. For comparison, the figure also shows the start-up costs relative to the size of `DOUBLE` data, as stated in the original ISABELA publication [45]. Since `DOUBLE` uses 64 bit, the sort order occupies two times less space relative to `DOUBLE` data, as compared to `FLOAT`. Fig. 3.15 (*b*) shows the start-up cost on `FLOAT` data excluding the space required for compressed errors $\mathrm{ERR}^{\mathrm{ISA}}$ for $n_{\mathrm{B}} = 16$, 32, 64, 128, 256, according to Eq. (3.8), p. 37. The start-up costs for sort order and B-spline are between ~30–40 $\%_{32b}$.

ISABELA Number of B-Spline Control Points

The compression tests are carried out using input stream size $N = 8^3$ and 16^3. The performance of GPLC inside ISABELA is evaluated in terms of compression rate and compression speed. Input streams larger than $N = 16^3$ are not considered for ISABELA, because the storage of the sort order for $N = 16^3$ values demands for ~37.5 $\%_{32b}$ already and would increase even more for larger input streams.

Fig. 3.16 shows the compression rate of ISABELA on the data sets `ALUR`, `ISOR` and `JETR`. `zstd` is used for the compression of quantized errors at different error bounds $e_{\mathrm{MAX}} = 1.00$, 0.50, 0.25, 0.12 %. The plots show the trade-off between additional B-spline control points and the resulting noise in the error quantization. Hence, the plots show the optimal choice of n_{B} for the compression of the data sets `ALUR`, `ISOR` and `JETR` for input streams with size $N = 8^3$ and 16^3.

For $e_{\mathrm{MAX}} = 1.00$ %, the compression rate of ISABELA ranges from ~37.0–41.5 $\%_{32b}$ for $N = 8^3$, and ~39.0–41.5 $\%_{32b}$ for $N = 16^3$. Compared to the theoretical compression rate

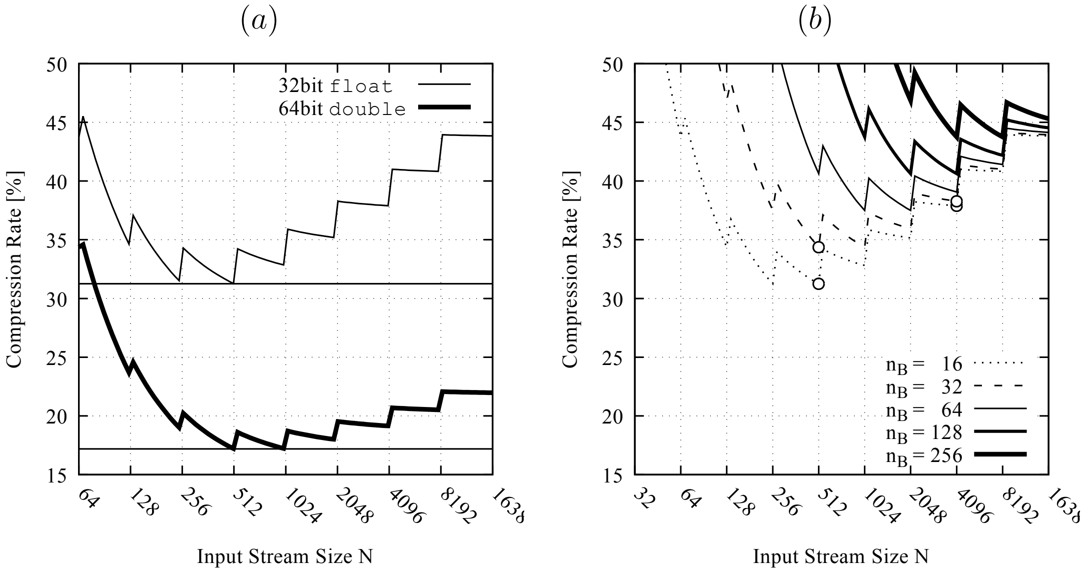


Figure 3.15: ISABELA startup costs. (a) Start-up cost for sort order relative to the compression of `FLOAT` and `DOUBLE` data using $n_{\mathrm{B}} = 16$ B-spline control points. (b) Start-up cost for `FLOAT` data using $n_{\mathrm{B}} = 16, 32, 64, 128, 256$. The space required for compressed errors $\mathrm{ERR}^{\mathrm{ISA}}$ is not included. The white circles in the plot show the start-up costs for ISABELA using input stream size $N = 8^3 = 512$ and $N = 16^3 = 4096$.

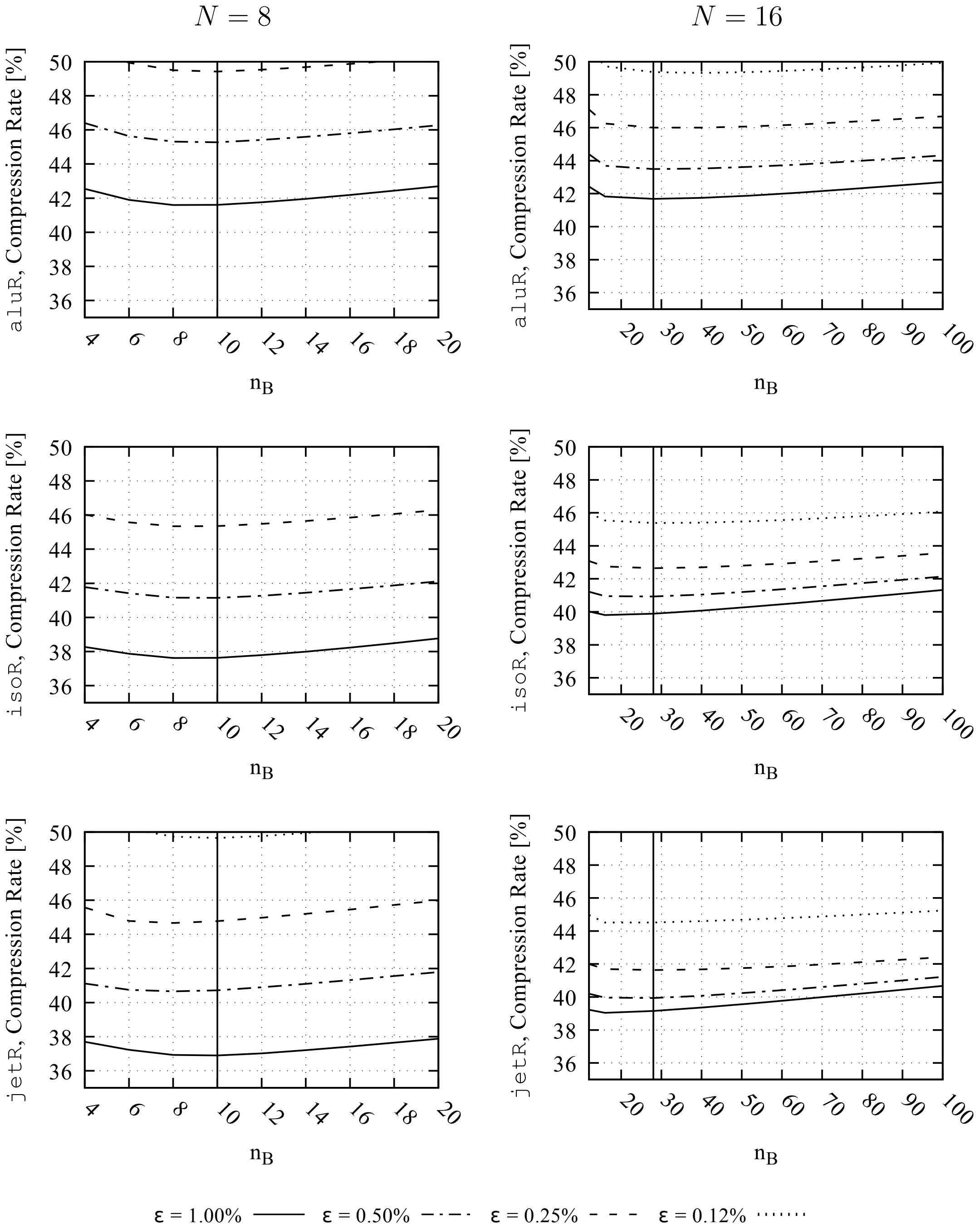


Figure 3.16: ISABELA compression rate for data sets **ALUR**, **ISOR**, **JETR**. ISABELA uses GPLC `zstd` for input stream size (LEFT) $N = 8^3$, and (RIGHT) $N = 16^3$, and varying error bounds $e_{\mathrm{MAX}} = 1.00$, 0.50, 0.25, 0.12 %. `zstd` operates at default compression level 6.

without any error correction, ∼28.1 $\%_{32\text{b}}$ for $N = 8^3$, and ∼37.5 $\%_{32\text{b}}$ for $N = 16^3$, the B-spline control points and quantized error introduce an overhead of ∼9.0–13.5 $\%_{32\text{b}}$ for $N = 8^3$, and ∼1.5–4.0 $\%_{32\text{b}}$ for $N = 16^3$. The application of ISABELA using $e_{\text{MAX}} = 0.12\,\%_{32\text{b}}$ results in a bad compression performance of ∼45–50 $\%_{32\text{b}}$ and worse. Therefore, the application of ISABELA with lower error bounds is not useful. For $N = 8^3$ and 16^3, a number of $n_{\text{B}} = 10$ and 28 B-spline control points is used for further testing of the compression performance of ISABELA.

ISABELA Compression Performance

Tables 3.4 and 3.5 show the compression rate and run-time achieved using ISABELA for the compression of the data sets `ALUR`, `ISOR`, and `JETR` using maximum error $e_{\text{MAX}} = 1.00\,\%$ and $n_{\text{B}} = 10$ and 28 B-spline control points for $N = 8^3$ and 16^3. The quantized errors of ISABELA are compressed using `zstd`, `lz4`, `snappy` and `tpfor`. `zstd` and `lz4` are operated on default level 6, as well as, compression level 2 for `zstd`, and 4 for `lz4`. Those levels provide a trade-off between compression rate and speed, as elaborated in Section 2.3, p. 16.

As can be seen, the difference in compression rate between `zstd` and `lz4` on level 6, and 2 and 4, is less than 1 $\%_{32\text{b}}$. Therefore, a small penalty in compression performance allows the application of faster compression of ISABELA. `zstd` on level 6 yields the best compression rate of ∼37.6–41.6 $\%_{32\text{b}}$. `snappy` and `tpfor` yield the worst compression rates of ∼41–55 $\%_{32\text{b}}$ but are the fastest algorithms. Although `lz4` yields the faster decompression, `zstd` yields a faster compression and an improvement of the compression rate by ∼1–3 $\%_{32\text{b}}$.

[$\%_{32\text{b}}$]	`zstd 2`	`zstd 6`	`lz4 4`	`lz4 6`	`snappy`	`tpfor`	N
`ALUR`	42.8	**41.6**	45.6	45.2	50.0	55.0	8^3
	42.2	**41.6**	43.7	43.4	48.0	50.3	16^3
`ISOR`	38.5	**37.6**	39.4	39.2	43.7	46.4	8^3
	40.1	**39.8**	40.8	40.7	45.1	44.0	16^3
`JETR`	37.7	**36.9**	37.7	37.5	41.6	44.4	8^3
	39.2	**39.1**	39.6	39.5	43.8	41.0	16^3

Table 3.4: ISABELA compression rate for data sets `ALUR`, `ISOR`, `JETR` using GPLCs `zstd`, `lz4`, `snappy` and `tpfor` with input stream size $N = 8^3$ and 16^3, B-spline control points $n_\mathrm{B} = 10$ and 28, and error bound $e_\mathrm{MAX} = 1.00\,\%$. `zstd` operates on level 2 and 6, `lz4` operates on level 4 and 6. `tpfor` operates on block size 128.

(*a*)

[s]	`zstd 2`	`zstd 6`	`lz4 4`	`lz4 6`	`snappy`	`tpfor`	N
`ALUR`	5.16	6.72	7.75	8.84	4.12	**4.07**	8^3
	6.13	6.88	6.97	8.06	7.40	**5.77**	16^3
`ISOR`	4.81	5.96	7.61	8.15	**4.26**	4.33	8^3
	6.03	6.49	6.72	7.17	5.88	**5.85**	16^3
`JETR`	1.52	1.87	2.50	2.85	**1.39**	**1.39**	8^3
	1.68	1.84	1.88	2.06	**1.68**	1.69	16^3

(*b*)

[s]	`zstd 2`	`zstd 6`	`lz4 4`	`lz4 6`	`snappy`	`tpfor`	N
`ALUR`	1.89	1.84	**1.34**	1.35	1.41	**1.34**	8^3
	2.57	2.56	2.41	2.40	2.45	**2.37**	16^3
`ISOR`	1.78	1.71	1.37	1.35	1.42	**1.32**	8^3
	2.49	2.50	2.43	2.41	2.48	**2.37**	16^3
`JETR`	0.52	0.54	**0.44**	0.46	0.46	0.45	8^3
	1.11	0.81	0.79	0.85	0.82	**0.78**	16^3

Table 3.5: ISABELA run-time for (a) compression and (b) decompression of data sets `ALUR`, `ISOR`, `JETR` using GPLCs `zstd`, `lz4`, `snappy` and `tpfor` with input stream size $N = 8^3$ and 16^3, B-spline control points $n_\mathrm{B} = 10$ and 28, and error bound $e_\mathrm{MAX} = 1.00\,\%$. `zstd` operates on level 2 and 6, `lz4` operates on level 4 and 6. `tpfor` operates on block size 128.

3.4.2 SBD Compression

The compression rate of SBD only depends on the error bound e_{MAX}, which directly determines the amount of different unique values in the LUT and in the decompressed data. Unlike ISABELA, which uses a B-spline and point-wise error quantization, SBD replaces the original values with values determined according to the user-defined maximum error e_{MAX}.

SBD Look-Up Table Size

Table 3.6 shows the size of the LUT L resulting from the data sets `ALUR`, `ISOR` and `JETR` using SBD version 1 with the error bounds $e_{\mathrm{MAX}} = 2.00, 1.00, 0.50, 0.25, 0.12\,\%$, and $e_1 = 10^{-6}$. The column $\min|L|$ shows the smallest LUT, and $\max|L|$ shows the largest LUT encountered when compressing all time steps of the data sets. The table illustrates the clustering of SFPD, i.e. all values in the `ALUR` data set are mapped to ~1000–1100 values for $e_{\mathrm{MAX}} = 1.00\,\%$.

e_{MAX} [%]	`ALUR` $\min\|L\|$	`ALUR` $\max\|L\|$	`ISOR` $\min\|L\|$	`ISOR` $\max\|L\|$	`JETR` $\min\|L\|$	`JETR` $\max\|L\|$
0.12	6025	7047	3208	5410	1784	2163
0.25	3324	3833	1838	2930	914	1112
0.50	1842	2085	1060	1616	483	587
1.00	997	1095	602	882	255	311
2.00	525	566	331	478	133	162

Table 3.6: LUT size in SBD for data sets `ALUR`, `ISOR` and `JETR`. Since SFPD is clustered, all data set values are mapped to ~1000–1100 LUT values using error bound $e_{\mathrm{MAX}} = 1.00\,\%$.

Fig. 3.17 shows the relation between error bound e_{MAX} and LUT size $|L|$ for all three data sets. Decreasing the error bound results in a strong growth of the size of the LUT. The figure shows the absolute size of the LUT $\max|L|$, as well as, the fraction of the LUT size relative to the size N of the input stream. As can be seen, using an error bound of $e_{\mathrm{MAX}} = 1.00, 0.50, 0.25, 0.12\,\%$, results in a LUT which requires at most ~2.5 %$_{32b}$ of storage relative to the full data set size.

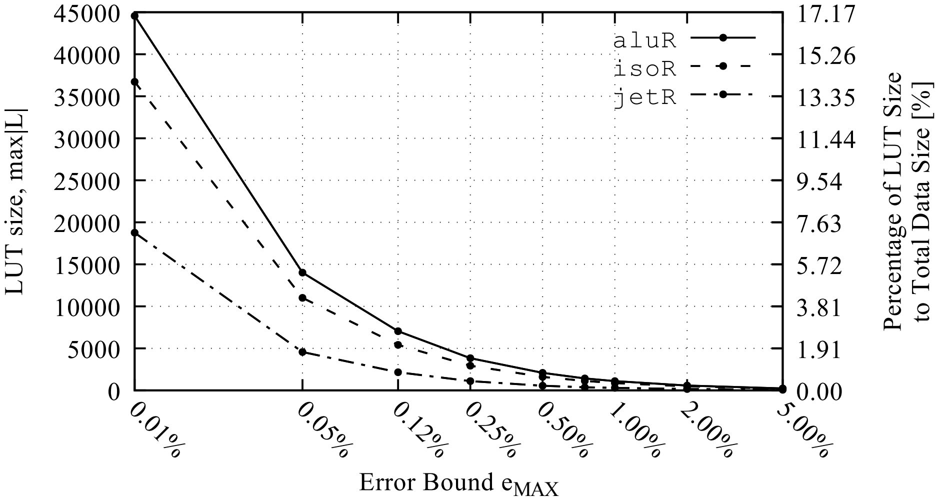


Figure 3.17: Growth of LUT size depending on maximum error. LUT size increases strongly for decreasing error bound e_{MAX}. For error bounds $e_{\mathrm{MAX}} \geq 0.12\,\%$ the LUT is smaller than 2.5 % of the data set.

SBD Compression Rate

In contrast to ISABELA, SBD has no start-up costs which are independent of the data to be compressed. Since SBD uses scalar quantization in order to map each value to a smaller set of LUT values, the internal integer data in SBD differs from that in ISABELA. For LUT indices, the smallest index is zero, and the largest index corresponds to the size of the LUT. Therefore, the internal data of SBD are `UNSIGNED` values with a bounded value range, i.e. LUT indices are amenable to bit packing compression like `tpfor`. For ISABELA in contrast, the quantized errors consist of mostly small signed integers, which can grow large in order to compensate the regression error of the B-spline.

Table 3.7 and Fig. 3.18 (a)–(c) show the compression rate of SBD on the data sets `ALUR`, `ISOR` and `JETR` for $e_{\mathrm{MAX}} = 1.00,\ 0.50,\ 0.25,\ 0.12\,\%$ using `zstd`, `lz4`, `snappy` and `tpfor` for lossless compression of LUT indices. The compression is carried out on data blocks of size $N = 8^3$, 16^3, 32^3, 64^3 in order to observe the effect of data fragmentation inside parallel CFD simulations. `zstd` and `lz4` operate on compression level 6, as well as, on 2 and 4 respectively. `tpfor` operates on the default block size of 128 values.

As can be seen, the performance of the GPLC improves for larger input streams $N = 64^3$, because the lossless compressors can perform more efficient coding on the data. `lz4` and `snappy` yield poor performance on LUT indices in SBD. For smaller block sizes $N = 8^3$ and 16^3, `tpfor` yields better compression rates than `zstd` on level 6. For `zstd` on level 2, the difference in compression rate between `zstd` and `tpfor` declines, and `tpfor` outperforms `zstd`. However, for large block sizes, `zstd` is ~4 $\%_{32b}$ better than `tpfor` on the `ALUR` data set.

[$\%_{32b}$]	`zstd 2`	`zstd 6`	`lz4 4`	`lz4 6`	`snappy`	`tpfor`	N
`ALUR`	35.3	32.9	49.8	49.5	53.9	**31.1**	8^3
	32.3	**28.4**	41.0	39.7	48.5	30.8	16^3
	30.1	**27.3**	37.7	34.7	47.4	30.7	32^3
	29.5	**26.7**	37.1	33.6	47.4	30.7	64^3
`ISOR`	33.9	30.6	47.6	47.1	53.6	**26.6**	8^3
	31.2	26.5	40.4	37.8	49.3	**26.2**	16^3
	28.7	**26.0**	38.3	34.1	48.6	26.2	32^3
	28.0	**25.4**	37.9	33.4	48.6	26.2	64^3
`JETR`	34.3	30.8	48.0	47.7	53.2	**25.9**	8^3
	31.2	26.3	40.6	37.9	48.8	**25.5**	16^3
	28.5	25.7	38.8	34.2	48.3	**25.5**	32^3
	27.3	**25.2**	38.6	33.6	48.3	25.5	64^3

Table 3.7: SBD compression rate for data sets `ALUR`, `ISOR`, `JETR` using GPLCs `zstd`, `lz4`, `snappy` and `tpfor` with input stream size $N = 8^3$, 16^3, 32^3, 64^3, and error bound $e_{\mathrm{MAX}} = 1.00\,\%$. `zstd` operates on level 2 and 6, `lz4` operates on level 4 and 6. `tpfor` operates on block size 128.

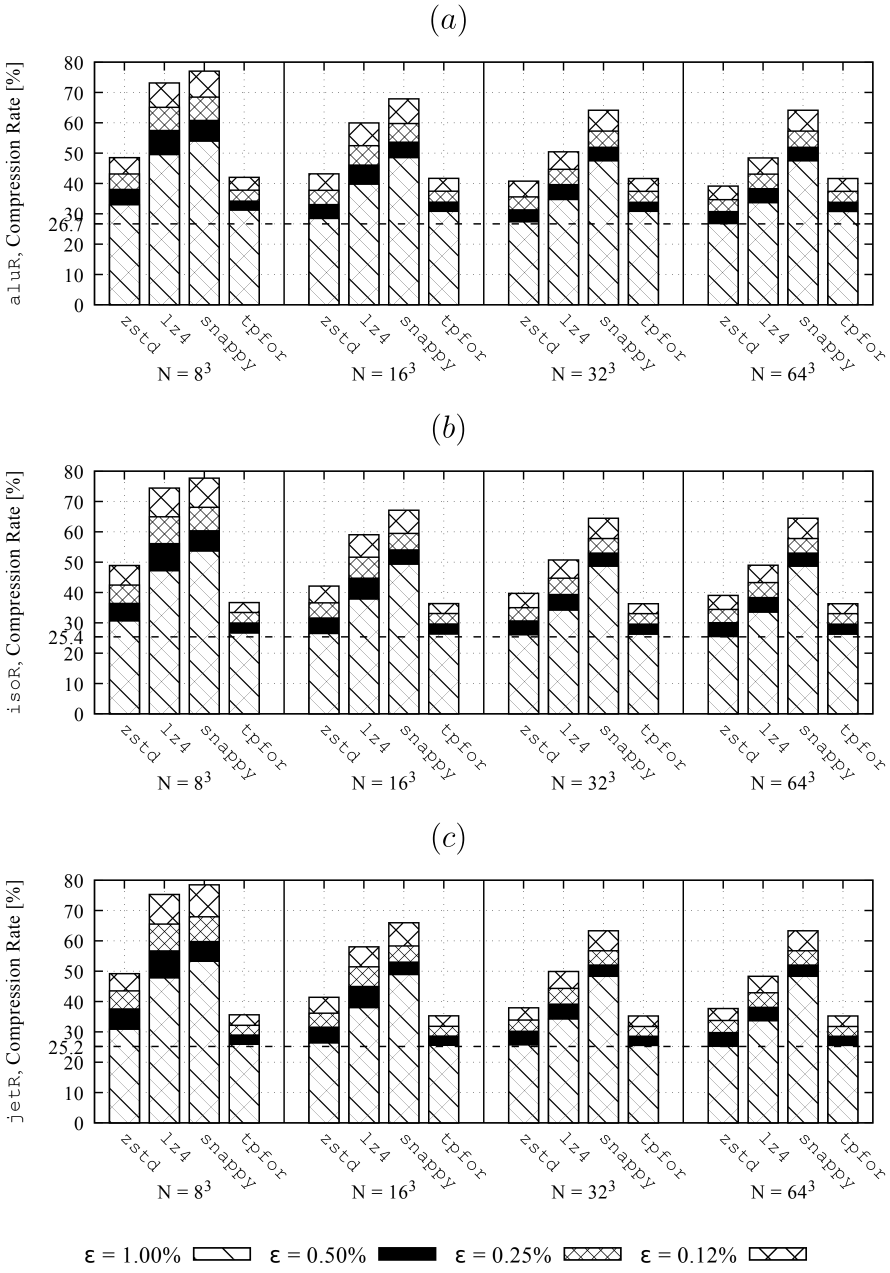


Figure 3.18: SBD compression rate for data sets **`ALUR`**, **`ISOR`**, **`JETR`**. SBD uses GPLCs `zstd`, `lz4`, `snappy` and `tpfor` on input stream size $N = 8^3,\ 16^3,\ 32^3,\ 64^3$ with error bound $e_{\mathrm{MAX}} = 1.00,\ 0.50,\ 0.25,\ 0.12\,\%$. `zstd` and `lz4` operate on level 6. `tpfor` operates on block size 128.

SBD Compression Run-Time

Table 3.8 (a)–(b) shows the SBD run-time during compression and decompression using input stream size of $N = 8^3$, 64^3, and error bound $e_{\mathrm{MAX}} = 1.00\,\%$. As can be seen, during compression and decompression, `tpfor` is constantly faster than `zstd`, `lz4` and `snappy`. For compression, `lz4` on level 4 yields a run-time comparable to `zstd` on level 6. However, during decompression `lz4` is consistently faster than `zstd` on level 2. `snappy` is fast, but has a bad compression rate, as LUT indices have a complicated structure. The bounded value range of LUT values allows for the application of `tpfor`, whereas the maximum number of bits required for encoding the LUT indices is determined by the `tpfor` compressor. Since `tpfor` operates on small blocks of 128 values, it can perform fast coding on short input streams.

(a)

[s]	`zstd` 2	`zstd` 6	`lz4` 4	`lz4` 6	`snappy`	`tpfor`	N
`ALUR`	5.86	7.94	7.92	8.06	4.04	**3.82**	8^3
	4.63	6.83	7.17	11.23	4.13	**3.69**	64^3
`ISOR`	5.71	8.22	8.27	9.07	4.14	**3.83**	8^3
	4.67	6.66	7.22	11.84	4.07	**3.81**	64^3
`JETR`	1.78	2.63	2.65	2.87	1.24	**1.17**	8^3
	1.42	2.08	2.27	3.92	1.21	**1.11**	64^3

(b)

[s]	`zstd` 2	`zstd` 6	`lz4` 4	`lz4` 6	`snappy`	`tpfor`	N
`ALUR`	1.09	1.24	0.33	0.33	0.43	**0.23**	8^3
	0.61	0.63	0.29	0.28	0.43	**0.22**	64^3
`ISOR`	1.02	1.27	0.32	0.33	0.48	**0.23**	8^3
	0.63	0.65	0.30	0.29	0.46	**0.22**	64^3
`JETR`	0.35	0.43	0.10	0.10	0.16	**0.06**	8^3
	0.20	0.22	0.09	0.09	0.14	**0.06**	64^3

Table 3.8: SBD run-time for (a) compression and (b) decompression of data sets `ALUR`, `ISOR`, `JETR` using GPLCs `zstd`, `lz4`, `snappy` and `tpfor` with input stream size $N = 8^3$ and 64^3, and error bound $e_{\mathrm{MAX}} = 1.00\,\%$. `zstd` operates on level 2 and 6, `lz4` operates on level 4 and 6. `tpfor` operates on block size 128.

3.4.3 GLATE Compression

Similarly to SBD, the compression rate of GLATE depends on the error bound e_{MAX} only. The error bound directly determines the size of the resulting mLUT required for the encoding of `FLOAT` values using a truncated set of mantissas for quantization. The exponents are compressed using a GPLC, and the range bounded signed mLUT indices are compressed using `tpfor`, as explained in Section 3.3.3.

GLATE Float Decomposition

The GLATE compression procedure decomposes each `FLOAT` value in the input stream into two streams for exponents and signed mLUT indices. As can be seen in Fig. 3.19 (a)-(b) for $e_{\mathrm{MAX}} = 1.00\,\%$, the signed mLUT indices k' contain the sign, as well as, the noise of the floating point mantissas, whereas the exponents E' constitute a narrow band of mostly smoothly changing values. As e_0 is the smallest value not blanked to zero, the actual floating point exponents E are subtracted by E_Z before compression, according to $E' = 1 + E - E_Z$, whereas $E' = 0$ encodes a zero. The exponent E_Z of e_0 is stored in the compressed data block.

The signed mLUT indices k' shown in Fig. 3.19 (b) are bounded in the range $[0, 2 \cdot n_{\mathrm{M}})$, i.e. if the sign in the input stream stays the same, then the signed mLUT index equals the mLUT index $k' = k$, and if the sign in the input stream changes, then the mLUT index is incremented by n_{M} according to $k' = k + n_{\mathrm{M}}$.

Fig. 3.20 (a)-(b) shows the adjustment of the error bound e_{MAX} for alignment with floating point exponents during the GLATE compression procedure, as well as, the number of quantized mantissas n_{M} resulting from the adjusted error bound $\hat{e}_{\mathrm{MAX}}$. As can be seen, the adjusted error bound $\hat{e}_{\mathrm{MAX}}$ is only slightly lower than the requested error bound. Using the error bounds $e_{\mathrm{MAX}} = 1.00, 0.10, 0.01\,\%$ encodes `FLOAT` mantissas with ~5.13, 8.45, 11.77 bit, e.g. the error bound $e_{\mathrm{MAX}} = 1.00\,\%$ implies $n_{\mathrm{M}} = 35$ mLUT indices per exponent.

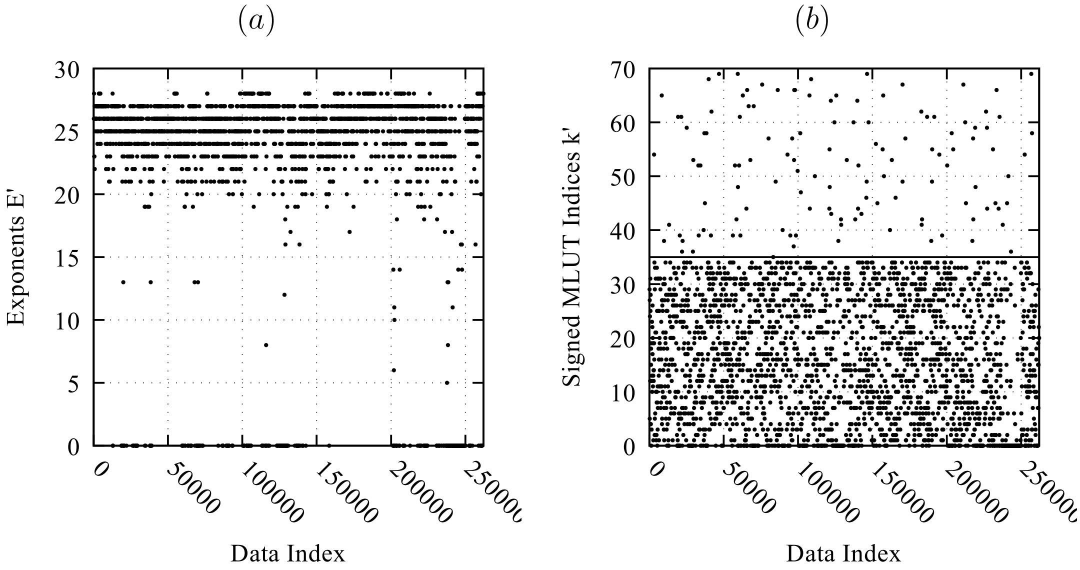


Figure 3.19: GLATE internal data. The internal data has been computed using $e_{\mathrm{MAX}} = 1.00\,\%$ and consists of (a) exponents $E' = 1 + E - E_Z$ and (b) signed MLUT indices k'. The exponent E_Z of e_0 is subtracted from every `FLOAT` exponent, and $E' = 0$ encodes a zero. The signed MLUT indices k' contain signs and truncated mantissas. In the case the sign does not change $k' = k$, and in the case the sign flips $k' = k + n_{\mathrm{M}}$. Both streams contain `UNSIGNED` data.

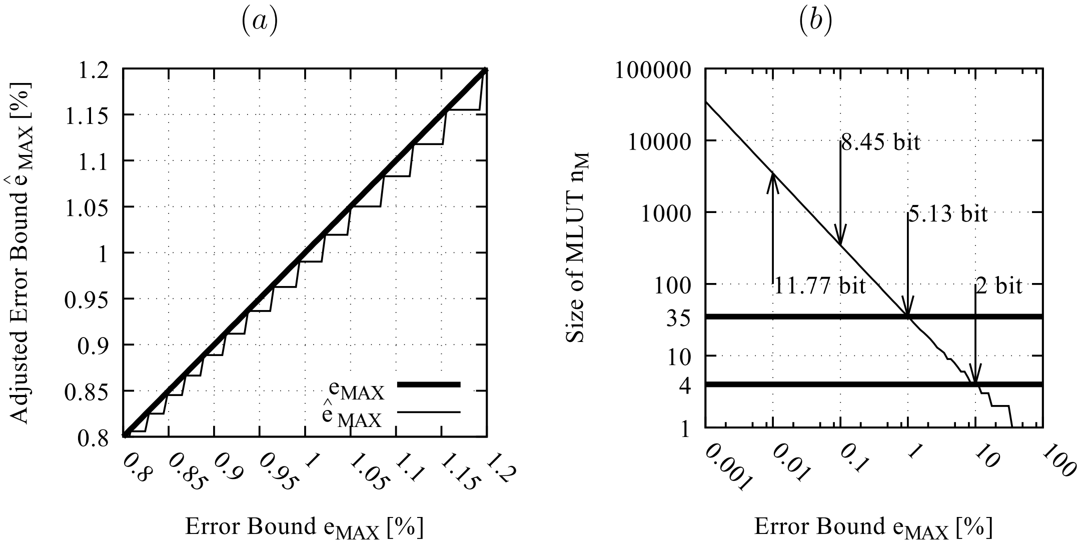


Figure 3.20: GLATE adjusted error bound. (a) Adjusted error bound $\hat{e}_{\text{MAX}}$ depending on the requested error bound e_{MAX}. For alignment with floating point exponents, the error bound is only lowered slightly. (b) The size of the mLUT using logarithmic x- and y-scale. The GLATE quantization scheme directly relates e_{MAX} and the size of the mLUT n_{M}.

GLATE Quantization and SBD Look-Up Table

The signs s, the exponents E', and mLUT indices k, which are determined in GLATE are similar to the encoding of LUT indices encountered in SBD. Given a sign s, an exponent E' and a mLUT index k, a so-called *global* mLUT index i, which is a representation similar to LUT indices in SBD, can be computed using the following equation:

$$i = (\text{-}1)^s \cdot (E' \cdot n_{\text{M}} + k) \tag{3.18}$$

As shown in Fig. 3.21 (*a*)-(*b*), the LUT indices i from SBD and the global mLUT indices i exhibit a similar shape and reflect trends of the data. SBD uses a GPLC in order to compress all LUT indices in one block. In comparison, GLATE compresses exponents E' and signed mLUT indices k' separately using different lossless compression techniques. This way GLATE carries the trend of the data in the exponents, and the sign and the noise are contained in the signed mLUT indices k'. By using a GPLC for the exponents and `tpfor` for signed mLUT indices, GLATE achieves an efficient encoding of `FLOAT` values.

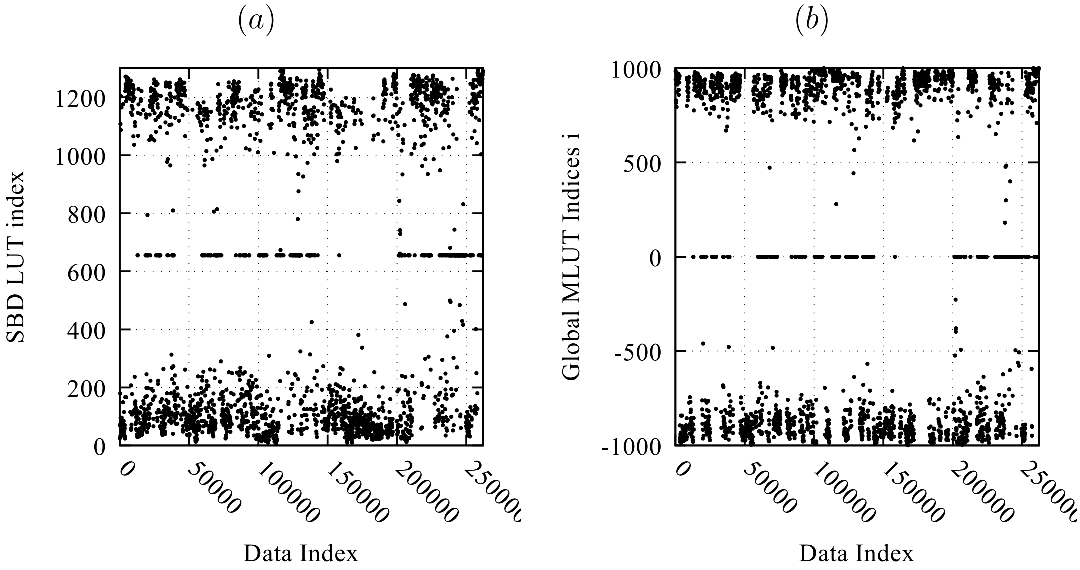


Figure 3.21: Comparison of GLATE quantization and SBD look-up table. (a) LUT indices encountered in SBD compression. (b) Global MLUT indices i computed from internal data of GLATE, i.e. sign s, exponent E' and MLUT index k, according to $i = (\text{-}1)^s \cdot (E' \cdot n_\mathrm{M} + k)$. In contrast to SBD, GLATE allows for separate encoding of smooth exponents and noisy signed MLUT indices.

GLATE Compression Rate

Table 3.9 and Fig. 3.22 (a)-(c) show the compression rate of GLATE on the data sets `ALUR`, `ISOR` and `JETR` for $e_{\mathrm{MAX}} = 1.00$, 0.50, 0.25, 0.12 % using `zstd`, `lz4`, `snappy` and `tpfor` for lossless compression of the exponents stream. The signed mLUT indices constitute a noisy stream of range-bounded `UNSIGNED` numbers $k' \in [0, 2 \cdot n_{\mathrm{M}})$ and are always compressed using `tpfor`. `zstd` and `lz4` operate on compression level 6, as well as, on 2 and 4 respectively. `tpfor` operates on the default block size of 128 values.

$[\%_{32b}]$	`zstd 2`	`zstd 6`	`lz4 4`	`lz4 6`	`snappy`	`tpfor`	N
`ALUR`	25.5	**24.9**	27.6	27.5	28.1	38.1	8^3
	23.2	**22.6**	25.6	25.4	26.2	37.6	16^3
	22.7	**22.3**	24.7	24.2	25.7	37.6	32^3
	22.6	**22.2**	24.3	23.7	25.6	37.5	64^3
`ISOR`	25.2	**24.6**	25.9	25.8	26.4	41.8	8^3
	22.7	**22.1**	24.2	23.9	24.8	41.2	16^3
	22.3	**21.9**	23.7	23.2	24.5	41.1	32^3
	22.1	**21.8**	23.5	22.9	24.4	41.1	64^3
`JETR`	25.6	**24.9**	25.6	25.5	26.1	42.6	8^3
	22.7	**22.0**	24.0	23.7	24.6	42.0	16^3
	22.3	**21.7**	23.6	23.0	24.4	42.0	32^3
	22.1	**21.6**	23.5	22.9	24.3	42.0	64^3

Table 3.9: GLATE compression rate for data sets `ALUR`, `ISOR`, `JETR` using GPLCs `zstd`, `lz4`, `snappy` and `tpfor` with input stream size $N = 8^3$, 16^3, 32^3, 64^3, and error bound $e_{\mathrm{MAX}} = 1.00\,\%$. `zstd` operates on level 2 and 6, `lz4` operates on level 4 and 6. `tpfor` operates on block size 128. Signed mLUT indices are always compressed using `tpfor`.

Unlike in SBD, `lz4` and `snappy` yield a reasonable performance on compressing the exponents, but `zstd` clearly yields a better compression rate. GLATE shows the same behavior as SBD on small block sizes $N = 8^3$ and 16^3, where the GPLC performs coding less efficient on short streams of exponents. Furthermore, the difference in compression rate between `zstd` on level 2 and 6 is minor, which allows for a faster execution of `zstd` on level 2 with nearly the same compression performance, as compared to SBD, where `zstd` should by applied on level 6. Overall, GLATE consistently outperforms SBD by ~3.6–5.6 $\%_{32b}$.

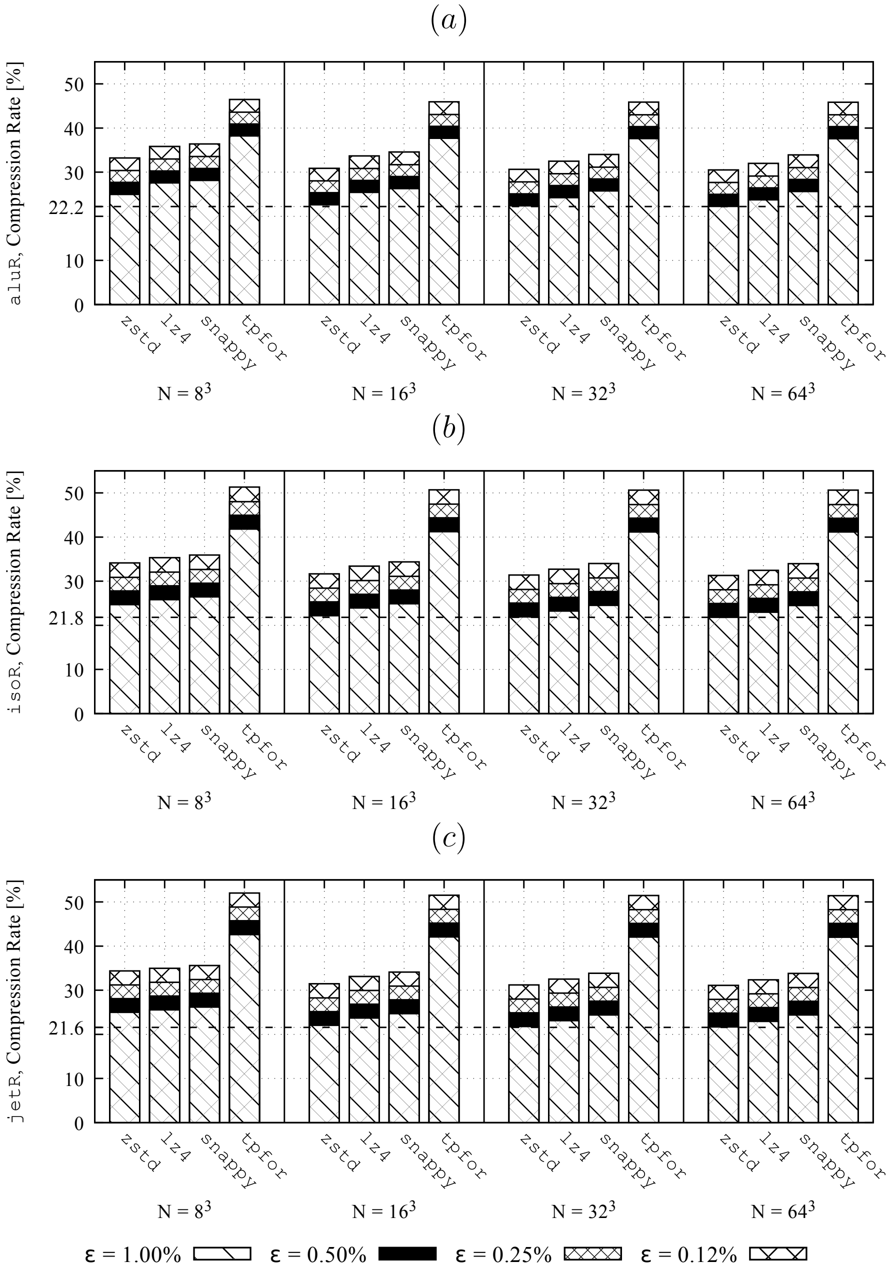


Figure 3.22: GLATE compression rate for data sets **ALUR**, **ISOR**, **JETR**. GLATE uses GPLCs `zstd`, `lz4`, `snappy` and `tpfor` with input stream size $N = 8^3$, 16^3, 32^3, 64^3, and error bound $e_{\mathrm{MAX}} = 1.00, 0.50, 0.25, 0.12\,\%$. `zstd` and `lz4` operate on level 6. `tpfor` operates on block size 128. Signed mLUT indices are always compressed using `tpfor`.

GLATE Compression Run-Time

The run-time for compression and decompression is shown in Table 3.10 (*a*)–(*b*). As can be seen, `snappy` is faster during compression of exponents than `lz4` and `zstd` on level 6. `snappy` yields a compression rate only ~2 $\%_{32b}$ worse than `lz4` on level 6 and ~3.5 $\%_{32b}$ worse than `zstd`. `tpfor` is the fastest algorithm, but is not amenable to the compression of the exponents. `snappy` constitutes an alternative for the compression of exponents with the speed of `tpfor` and a compression rate comparable to other GPLCs. Although `zstd` yields the best compression rate, `zstd` on compression level 2 and level 6 is slower than `snappy` and `lz4` during decompression. However, as the compression rate is to be minimized, `zstd` is preferred.

(*a*)

[s]	`zstd 2`	`zstd 6`	`lz4 4`	`lz4 6`	`snappy`	`tpfor`	N
`ALUR`	2.25	3.05	4.48	4.62	1.90	**1.77**	8^3
	1.99	2.60	2.78	3.78	1.80	**1.77**	64^3
`ISOR`	2.16	2.81	4.55	4.58	1.81	**1.76**	8^3
	1.92	2.45	2.58	3.63	1.80	**1.76**	64^3
`JETR`	0.75	1.00	1.43	1.66	**0.63**	0.67	8^3
	0.66	0.85	0.90	1.32	0.62	**0.61**	64^3

(*b*)

[s]	`zstd 2`	`zstd 6`	`lz4 4`	`lz4 6`	`snappy`	`tpfor`	N
`ALUR`	0.53	0.57	0.30	0.30	**0.28**	**0.28**	8^3
	0.35	0.37	**0.28**	0.29	0.31	0.29	64^3
`ISOR`	0.48	0.49	0.30	0.30	**0.27**	0.32	8^3
	0.33	0.35	**0.28**	0.29	0.30	**0.28**	64^3
`JETR`	0.13	0.15	0.09	0.10	0.09	**0.08**	8^3
	0.10	0.11	**0.08**	0.10	0.10	**0.08**	64^3

Table 3.10: GLATE compression run-time for data sets `ALUR`, `ISOR`, `JETR` using GPLCs `zstd`, `lz4`, `snappy` and `tpfor` with input stream size $N = 8^3$ and 64^3, and error bound $e_{\text{MAX}} = 1.00\,\%$. Time in seconds for (a) compression and (b) decompression. `zstd` operates on level 2 and 6, `lz4` operates on level 4 and 6. `tpfor` operates on block size 128. Signed mLUT indices are always compressed using `tpfor`.

3.5 Summary

In this chapter, three schemes for lossy in-situ compression of SFPD, namely ISABELA, SBD and GLATE were investigated. The algorithms ISABELA, SBD and GLATE require less run-time and achieve a higher data reduction on noisy `FLOAT` data, as compared to the direct application of GPLCs. During compression, all algorithms ensure a global point-wise error bound e_{MAX} according to the error correction policy described in Section 3.2, p. 30.

ISABELA Algorithm

The ISABELA algorithm was specifically developed for the in-situ compression context [45]. It achieves ~20 $\%_{64b}$ on `DOUBLE` data, and ~40 $\%_{32b}$ on `FLOAT` data for $e_{\mathrm{MAX}} = 1.00\,\%$. The compression rate depends on the number of B-spline control points n_{B} and on the size of the input stream N, which also determines the storage required for the sort order. Nevertheless, the B-spline control points and the sort order constitute fixed start-up costs, which are independent from the data to be compressed. As the B-spline fit yields low errors for a large amount of the data values, quantized errors in ISABELA mainly consist of zeros and small signed numbers, which are compressed effectively using a GPLC. By introducing only a small penalty of ~1 $\%_{32b}$ in compression rate, `zstd` level 2 can be used for increased compression speed over level 6.

SBD Algorithm

The SBD algorithm is a scalar quantization algorithm originally proposed as an offline compression algorithm for unstructured simulation grids [36]. In order to emulate the data fragmentation in parallel CFD simulations, SBD has been applied to smaller subgrids containing `FLOAT` data. The application of SBD to smaller grids decreases the compression performance compared to the original implementation of SBD, where compression is applied to global data sets as a post-processing step. In contrast to compression of subgrids, GPLCs perform much more efficient coding on LUT indices on the whole data set. A parallel version of SBD, which works on a global data set inside a numerical simulation is not considered here, as communication during the simulation is avoided for the in-situ compressors developed in this thesis. Depending on the input stream size $N = 8^3$, 16^3, 32^3, 64^3, the compression rate of SBD for `FLOAT` data is

~25–31 $\%_{32b}$ for $e_{\mathrm{MAX}} = 1.00\,\%$. Compared to ISABELA quantized errors, LUT indices have a more complex structure, and using `zstd` level 2 results in a penalty of up to ~3 $\%_{32b}$ in compression rate. Since LUT indices consist of bounded `UNSIGNED` values, the internal data of SBD is amenable to `tpfor` bit packing. However, `zstd` on level 6 results in a improvement of compression rate by at most ~4 $\%_{32b}$ for $e_{\mathrm{MAX}} = 1.00\,\%$, as compared to the faster `tpfor`.

GLATE Algorithm

The GLATE algorithm is a lossy compression algorithm for `FLOAT` numbers operating on the IEEE754 `FLOAT` bit representation and benefits from spatial coherence in linearized sequences of data values from CFD simulation grids. GLATE compresses exponents and signed mLUT indices separately using a GPLC and `tpfor` bit packing. Although the algorithm is only applied to `FLOAT` data in this chapter, GLATE can be applied to `DOUBLE` values with minor modifications as well. When applied to `DOUBLE`, GLATE is expected to result in higher data reduction if the same error bound is used. Because of the spatial coherence in the linearized data streams, the exponents of nearby grid cells in CFD simulation grids fluctuate only slightly. Therefore, the compression speed can be increased by applying `zstd` on level 2 resulting in a negligible penalty of ~0.5 $\%_{32b}$ in compression rate only. For different sizes of the input stream $N = 8^3$, 16^3, 32^3, 64^3, GLATE achieves a compression rate of ~21.6–25.0 $\%_{32b}$ on `FLOAT` data. As compared to SBD, GLATE decomposes `FLOAT` values into smooth exponents and noisy mLUT indices and achieves a better compression also on small blocks.

Compression Rate Overview

Fig. 3.23 shows the compression rate of ISABELA, SBD and GLATE depending on the maximum error of decompressed data. The error bounds are chosen to be $e_{\mathrm{MAX}} = 1.00$, 0.50, 0.25, 0.12 %. Lowering the error bound from 1.00 % to 0.12 % results in an increase of ~9.5 $\%_{32b}$ for ISABELA and GLATE. For SBD, the increase was ~12 $\%_{32b}$. Concluding, ISABELA, SBD and GLATE establish a trade-off between error bound e_{MAX} and the quality of decompressed data. However, GLATE constantly yields the best compression performance between ~20–30 $\%_{32b}$ on all data sets tested.

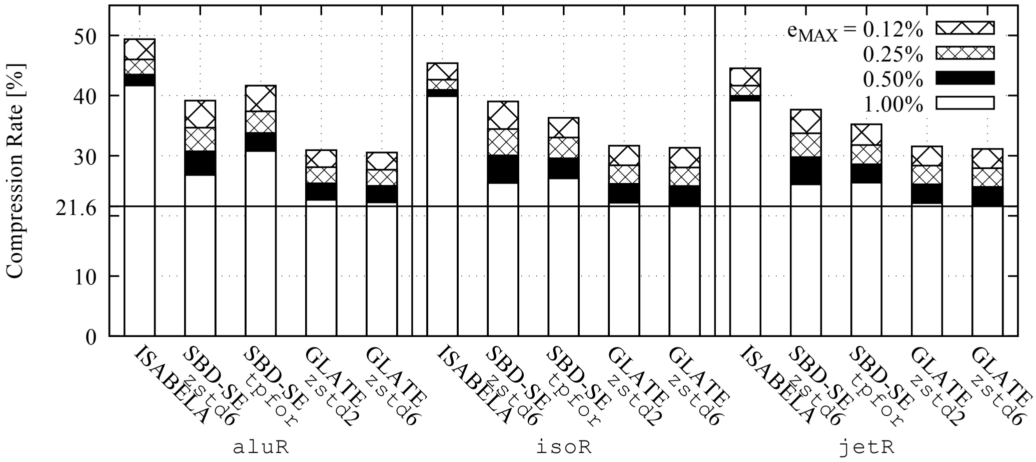


Figure 3.23: Overview of compression rate for data sets `ALUR`, `ISOR` and `JETR`. Compression is applied using error bound $e_{\mathrm{MAX}} = 1.00, 0.50, 0.25, 0.12\,\%$. The data sets `ALUR`, `ISOR` and `JETR` are compressed using ISABELA with `zstd` level 2, SBD with `zstd` level 6, SBD with `tpfor` block size 128, GLATE with `zstd` level 2, and GLATE with `zstd` level 6. For GLATE, signed MLUT indices are always compressed with `tpfor` block size 128.

Compression Run-Time Overview

Fig. 3.24 (*a*)-(*b*) shows the run-time for compression and decompression using ISABELA, SBD and GLATE depending on the maximum error of decompressed data. As can be seen, for ISABELA the run-time increases slightly with decreasing error bound e_{MAX}. During compression GLATE outperforms ISABELA and SBD when being used with `zstd` on level 2. During decompression GLATE is less than two times slower than SBD with `tpfor`. However, GLATE is still faster than SBD with `zstd` in level 6.

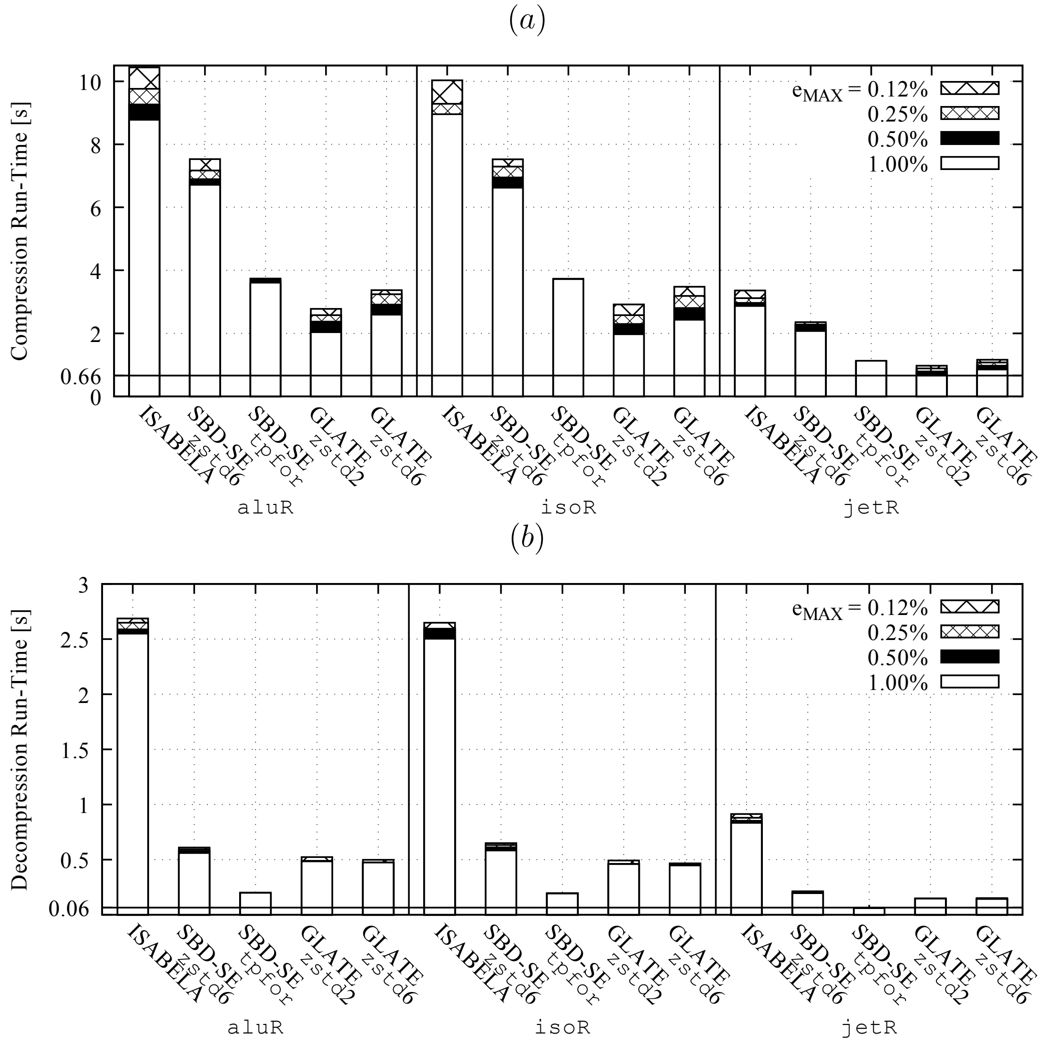


Figure 3.24: Overview of compression speed for of data sets `ALUR`, `ISOR` and `JETR`. Compression is applied using error bound $e_{\mathrm{MAX}} = 1.00, 0.50, 0.25, 0.12\,\%$. Time in seconds for (a) compression and (b) decompression. The data sets `ALUR`, `ISOR` and `JETR` are compressed using ISABELA with `zstd` level 2, SBD with `zstd` level 6, SBD with `tpfor` block size 128, GLATE with `zstd` level 2, and GLATE with `zstd` level 6. For GLATE, signed MLUT indices are compressed with `tpfor` block size 128 always.

Closing Statement

Summarizing, lossy in-situ compression for SFPD overcomes the limitations of lossless compression on SFPD. In particular, ISABELA, SBD and GLATE deliver improved compression speed, improved compression rate and ensure decompression within a user-defined error bound e_{MAX}. Therefore, the algorithms also establish a trade-off between compression rate and quality of decompressed data. In the tests conducted, the newly developed GLATE compression algorithm achieved the best compression rate and fastest compression speed using `zstd` and `tpfor` as lossless compression back end. ISABELA and SBD using `zstd` are slower, and SBD with `tpfor` is faster but yields a worse compression rate. However, by trading compression rate against compression speed, the run-time of GLATE can be further improved by using `snappy` instead of `zstd` on level 2. For storage of large-scale SFPD, error-bounded lossy compression using GLATE offers a compression rate of ~21.6–31.5 %$_{32b}$ within a maximum error of $e_{\text{MAX}} = 1.00, 0.50, 0.25, 0.12$ %, e.g. for $e_{\text{MAX}} = 1.00$ %, GLATE encodes `FLOAT` values with an average of ~6.9 bit per value.

Concluding, the lossy in-situ compression algorithms ISABELA, SBD and GLATE constitute a practical approach for data reduction of CFD simulation girds and help to reduce the I/O bottleneck when moving large-scale data. In the next chapter, ISABELA, SBD and GLATE are extended for new temporal compression procedures, which improve the compression rate even further.

4 Temporal In-Situ Compression for Scientific Floating Point Data

In this chapter, new algorithms for temporal in-situ compression are developed. The algorithms focus on the improvement of the compression rate by exploiting the temporal coherence, which is inherently present in time series of CFD data sets. First, temporal compression is motivated for CFD simulations, and the temporal compression procedure is explained briefly. Second, the existing algorithm *d*-ISABELA [45] is presented as an entry point for practical temporal in-situ compression. Third, based on the ideas of *d*-ISABELA, new temporal compression algorithms *t*-ISABELA, *t*-SBD and *t*-GLATE are developed and implemented. Fourth, the performance of the temporal compression schemes *d*-ISABELA, *t*-ISABELA, *t*-SBD and *t*-GLATE is evaluated on a temporal CFD data set w.r.t. the compression speed and the improvement of the compression rate.

4.1 Motivation for Temporal Compression

As shown in Chapter 3, lossy in-situ compression achieves a fourfold to fivefold data reduction in CFD simulations for `FLOAT` data. As CFD simulations typically describe continuous processes in time, the CFD data exhibits strong temporal coherence. Therefore, it is argued, that *Temporal Compression* is a promising approach for further reduction of the data size of CFD data sets. Such compression algorithms which use temporal schemes are already used in video compression.

Temporal Schemes in Video Compression

Video compression dramatically reduces the storage space of sequences of images through exploiting temporal redundancy between consecutive images [32], e.g. the background in videos usually is static and objects move through the scene. Image elements found in previous frames might be found in future frames and do not need to be reencoded entirely new. Instead, similar parts of the image are matched and only differences are encoded. Such procedures are called *Motion Compensation*, and are used for *Temporal Coding* in typical *Motion Picture Experts Group* (MPEG) video compression methods. From temporal coding, the following different frame types arise:

I-frames	So-called *Intra Frames* are encoded independently of any other frames and constitute entry points into the compressed video stream.
P-frames	So-called *Predictive Frames* are encoded using information from the previous frame.

For the compression of I-frames, typically spatial blocks of 8 × 8 pixels are encoded separately. In P-frames, the image data also is encoded in blocks. However, for each block, a motion vector is estimated pointing to the location of the best matching block in the previous frame, as shown in Fig. 4.1. The resulting error between the current frame and the prediction is quantized and encoded.

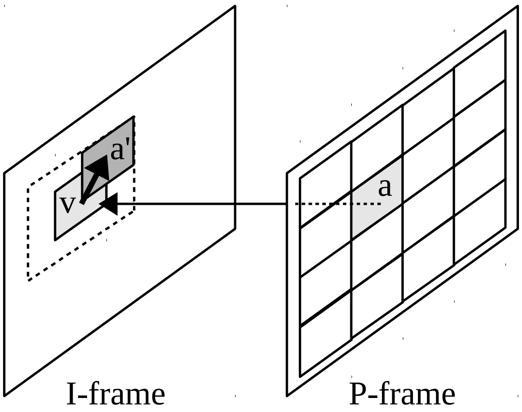


Figure 4.1: Temporal coding in MPEG video compression methods. Motion compensation is used for exploiting temporal redundancy in image sequences. I-frames are encoded independently. P-frames are encoded using information from the previous frame, e.g. block a in a P-frame is matched with block a' in the previous frame using motion vector v.

Image is adapted from [32].

As video compression methods for exploiting temporal redundancy typically consume a reasonable amount of run-time [6], such methods are not suited for the application in the in-situ context. Further, as data in CFD simulations typically is fragmented across many parallel processes, approaches like motion compensation would require message passing and would introduce more overhead. Instead, a difference encoding on the algorithm-specific internal data of ISABELA, SBD and GLATE is employed for the development of new temporal compression schemes for CFD simulations.

Temporal Coherence in CFD Simulations

Typically CFD simulations are iterative procedures. Therefore, the data of the current time step depend on the data of previous time steps. Based on a current solution, typically a numerical solver propagates the flow through the simulation domain. Depending on the physical process and the temporal resolution of the CFD simulation, the strength of the underlying forces change the current solution. The process puts the data of adjacent grid cells in a relationship on the temporal dimension. Fig. 4.2 (a)–(c) illustrates the temporal coherence of the velocity magnitude $M_t = \sqrt{u_t^2 + v_t^2 + w_t^2}$ in an aluminum melt casting simulation for two consecutive time steps t and $t+1$ [67]. As can be seen, the differences between same grid cells $M_{t+1} - M_t$ are much smaller than the original velocities and can be encoded more efficiently than the new data on its own. Other effects also support temporal coherence, e.g. regions with stagnating or constant flow are amenable to temporal compression.

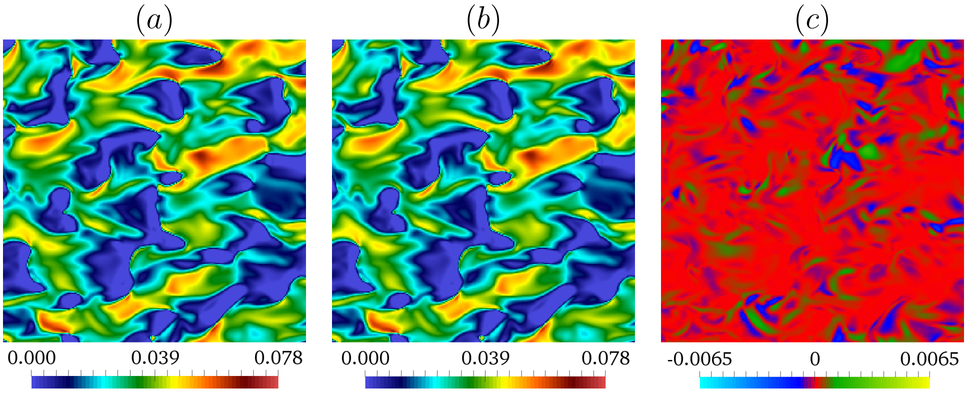


Figure 4.2: Temporal coherence between two consecutive time steps of a CFD simulation. The color plot on the slice shows (a),(b) velocity magnitude M_t and M_{t+1} at time points t and $t+1$, and (b) difference of velocity magnitudes $M_{t+1} - M_t$ of two consecutive time steps $t+1$ and t in a metal melt casting simulation. The differences are determined between same grid cells.

4.2 Temporal Compression Procedure

Similar to *I-frames* and *P-frames* in video compression, the temporal compression procedure proposed in this section differentiates between so-called *Key Frame*s (KFs), and *Difference Frame*s (DFs). KFs are compressed and decompressed independently, whereas DFs require data from the previous frame. Fig. 4.3 shows a sequence of KFs and DFs, in which three DFs follow one KF.

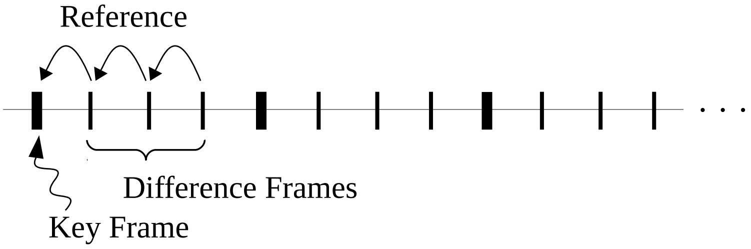


Figure 4.3: Illustration of temporal compression procedure. Temporal compression differentiates between so-called KFs and DFs. In KFs, lossy in-situ compression is applied independently. In DFs, the in-situ compression is extended for a temporal compression scheme which improves the compression rate.

Similarly to I-frames and P-frames, KF are decompressed independently and constitute entry points into the compressed stream. In contrast, DFs refer to information of the previous frame and allow for higher data reduction through encoding of small differences between frames. The temporal compression procedure employed for CFD simulations has two parameters, which control the insertion of KFs and DFs, and the temporal resolution of decompressed data.

Insertion of Key Frames and Difference Frames

The parameter $k_{\text{MOD}} \geq 1$ controls the insertion rate for KFs. Setting $k_{\text{MOD}} = 1$ corresponds to non-temporal compression. Using the parameter k_{MOD}, KFs are inserted at every time step $t = 0, 1, 2, \ldots$ of the CFD simulation for which $t \,\texttt{MOD}\, k_{\text{MOD}} = 0$ holds. Fig. 4.4 shows different distributions of KFs on the temporal dimension for $k_{\text{MOD}} = 1, 4, 16$. The remaining frames between two KFs are filled up with DFs. The DFs reference information from previous frames, which usually necessitates algorithm-specific data caching for the realization of difference encoding in ISABELA, SBD and GLATE. An increasing amount of DFs, e.g. 15 DFs per KF for $k_{\text{MOD}} = 16$, improves the compression rate but only allows limited direct access to compressed contents.

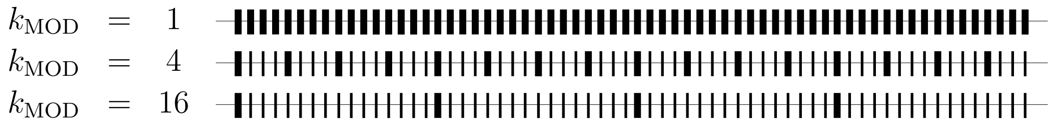


Figure 4.4: Placement of KFs and DFs in temporal compression procedure. A KF is inserted in every k_{MOD}-th time step to be compressed. The remaining frames in the sequence are encoded as difference frames for improved compression performance. $k_{\text{MOD}} = 1$ corresponds to no temporal compression, since all frames are encoded as KFs.

Temporal Resolution of Decompressed Data

The parameter $k_{\text{D}} \geq 1$ is used for control of the temporal resolution of decompressed data. By setting $k_{\text{D}} = 1$ the decompressed data has the highest resolution, i.e. the same temporal resolution as the CFD simulation. For values $k_{\text{D}} > 1$, the time steps $t = 0, 1, 2, \ldots$ are subsampled, i.e. only every k_{D}-th time step is included into the temporal compression procedure. The KFs and DFs are distributed among the resulting set of time steps. Fig. 4.5 shows the resulting temporal resolution for different choices of $k_{\text{D}} = 1, 4, 16$ and the distribution of KFs and DFs within the selected time steps using $k_{\text{MOD}} = 2, 4$. The parameter k_{D} has a direct influence on the distance between difference encoded frames. Therefore, k_{D} determines the order of magnitude of the data differences between time steps, which directly influences the compression rate of the temporal compression procedure.

4.3 Lossy Temporal Compression Schemes

In this section, d-ISABELA [45] is presented as a starting point for practical temporal in-situ compression, and three schemes t-ISABELA, t-SBD and t-GLATE for temporal compression are newly developed and implemented. The algorithms transform data differences into an internal representation, which is compressed using fast GPLCs.

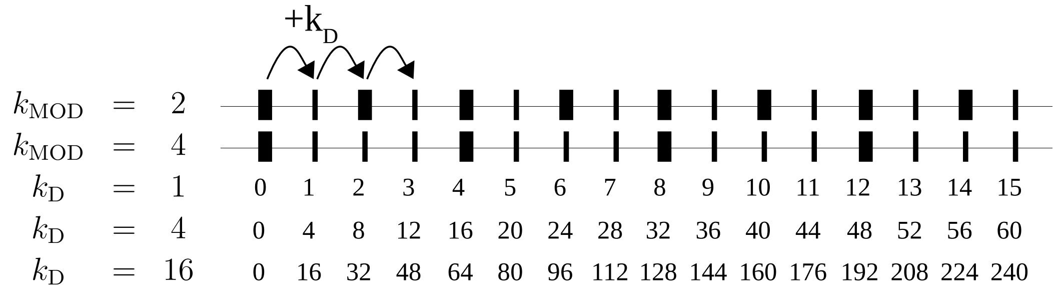


Figure 4.5: Selection of time steps for temporal compression procedure during CFD simulation. The time steps to be compressed are determined by skipping $k_{\mathrm{D}} - 1$ time steps. The temporal compression procedure is applied to the resulting subset of time steps, which are mapped to KFs and DFs according to e.g. $k_{\mathrm{MOD}} = 2, 4$. k_{D} determines the effective temoral resolution of decompressed data. $k_{\mathrm{D}} = 1$ corresponds to the temporal resolution of the CFD simulation.

4.3.1 Differential ISABELA Compression

The original publication of ISABELA [45] proposes the d-ISABELA extension for temporal compression, which improves the compression rate on sequences of SFPD. As explained in Section 3.3.1, p. 33, the sort order constitues fixed start-up costs and takes up a reasonable amount of storage in the ISABELA compressed format, e.g. ~28.1 $\%_{32\mathrm{b}}$ for input stream size $N = 512$, and ~37.5 $\%_{32\mathrm{b}}$ for $N = 4096$. By improving the storage scheme of the sort order, d-ISABELA improves the overall compression rate of ISABELA.

d-ISABELA Temporal Compression Procedure

In KFs, the d-ISABELA algorithm applies the original ISABELA procedure, except the sort order is cached at the end of each KF. As the indices of the sort order contain N different numbers in arbitrary order, the sort order by itself is not amenable to compression. Therefore, in DFs the sort order is encoded using differences relative to the cached sort order of the previous frame. Fig. 4.6 shows the d-ISABELA compression procedure in DFs.

Step (1), (3) and (4) correspond to the original ISABELA procedure. In step (2), the linearized data is sorted, and differences between the previously cached and the current sort order are computed. Given the fact that the data changes smoothly, it is assumed that the sort order also changes smoothly over time, and a difference encoding improves the ISABELA compression rate [45]. The difference encoding, as illustrated

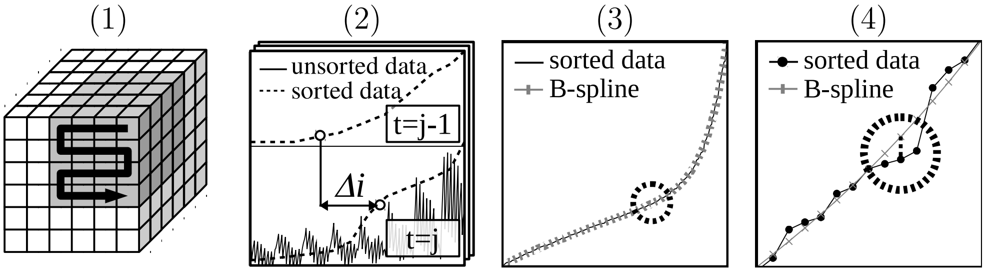


Figure 4.6: *d*-ISABELA temporal compression procedure in DFs. (1) Linearize data, (2) sort data, compute sort order differences and update cache, (3) B-spline regression on sorted data, and (4) error quantization.

in Fig. 4.6 (2), produces many small repetitive integers, which can be compressed using a GPLC efficiently. Similar to KFs, at the end of step (2) in the DF, the sort order cache is updated for continuation of the temporal compression procedure.

d-ISABELA Illustration

Fig. 4.7 (1)–(9) illustrate the *d*-ISABELA compression procedure for one KF at $t = 0$, and four DFs at $t = 1, 2, 3, 4$. First, after carrying out the ISABELA procedure in the KF, as shown in Fig. 4.7 (1), the indices of the sort order are obtained and cached. During the caching, the sort order $\mathcal{S}$ is inverted into a list $\mathcal{S}^{-1}$ of positions in the order of the data in the input stream. As the sort order constitutes a permutation $\mathcal{S}$: (NEW POS)→(OLD POS), it always can be inverted, i.e. $\mathcal{S}^{-1}$: (OLD POS)→(NEW POS). Second, in the subsequent DF as shown in Fig. 4.7 (2), the ISABELA procedure is carried out on new input data yielding a new sort order $\bar{\mathcal{S}}$ for the DF. Instead of storing the new sort order in the DF directly, the difference is computed between the new position i in the new sort order and the old position $\mathcal{S}^{-1}[\bar{\mathcal{S}}[i]]$ of the same value in the cached list of the KF, i.e. DIFF $= \mathcal{S}^{-1}[\bar{\mathcal{S}}[i]] - i$. As can be seen in in Fig. 4.7 (3), the resulting differences are small. For each value in the input stream, the differences describe where the value has moved relative to the position stored in the list cached in the KF. Before *d*-ISABELA moves on to the next DF at $t = 2$, the new sort order $\bar{\mathcal{S}}$ is inverted and used to update the list cached in the KF. Using this procedure, the differences in the sort order between consecutive time steps stay small and can be compressed using a GPLC efficiently.

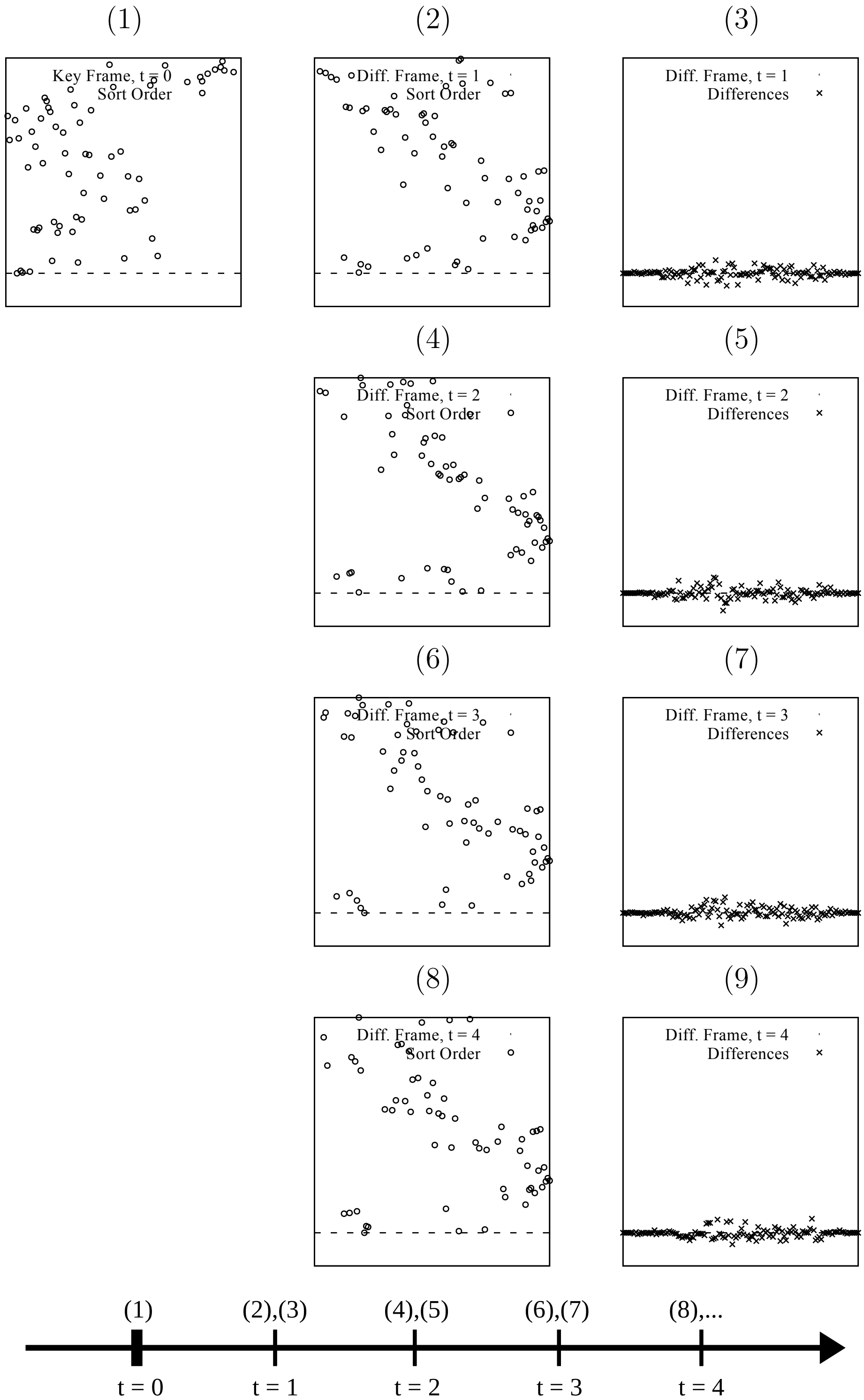


Figure 4.7: *d*-ISABELA compression procedure. (1) In a KF, and (2)–(9) in DFs: (1) shows the indices of the sort order obtained from sorting data in the KF, (2),(4),(6),(8) show the indices of the sort order obtained from sorting the new data in the DF, and (3),(5),(7),(9) show differences between sort orders, relative to the sort order from the previous frame.

d-ISABELA Implementation

The procedure dISA_compress(...), given in Alg. 4.1, corresponds to the algorithmic steps depicted in Fig. 4.6 (1)–(4). In KFs, the original ISABELA compression procedure ISA_compress(...), given in Alg. 3.1, is executed. At the end of the execution of the KF, the inverted sort order `invsort` $= \mathcal{S}^{-1}$ is cached. In DFs, first, the inverted sort order is loaded, which describes how unsorted values are rearranged in order to obtain the sorted sequence. Second, the new sort order for the new input data is determined, inverted and cached for the next DF. Using the inverted sort order and the new sort order, the difference encoding is computed by subtracting old position from the new position, i.e. `diff[i]` $=$ `invsort[`$\mathcal{S}$`[i]]` $-$ `i`. Third, the original ISABELA procedure is continued, except differences `diff[i]` are stored instead of the sort order itself. The procedure dISA_decompress(...), given in Alg. 4.2, illustrates the caching mechanism and the reconstruction of the sort order from differences during decompression. In KFs, the sort order is inverted and cached according to the compression procedure. In DFs, the cached inverted sort order is loaded, and the original sort order is reconstructed using the differences.

```
def dISA_compress(time_step, $N$, $n_\mathrm{B}$, in_data, $e_\mathrm{MAX}$, out_stream):
  if time_step mod $k_\mathrm{MOD}$ = 0:
    ISA_compress($N$, $n_\mathrm{B}$, in_data, $e_\mathrm{MAX}$, out_stream)
    invsort = [] // invert sort order
    for i in range(0, len($\mathcal{S}$)):
      invsort[$\mathcal{S}$[i]] = i
    dISA_cache_set(invsort) // init cache
  else:
    [$X$, $\mathcal{S}$] = Boost::SpreadSort(in_data)
    invsort = dISA_cache_get() // load cache
    diff = []
    for i in range(0, len($\mathcal{S}$)):
      // diff = (old sort pos) - (new sort pos)
      diff[i] = invsort[$\mathcal{S}$[i]] - i
    invsort = [] // invert sort order
    for i in range(0, len($\mathcal{S}$)):
      invsort[$\mathcal{S}$[i]] = i
    dISA_cache_set(invsort) // update cache
    // determine $i^-_{e1}$, $i^-_{e0}$, $i^+_{e0}$, $i^+_{e1}$, ...
    // ... $\bar{\delta}$, coeff, $\hat{X}$, quant - see ISA_compress(...)
    out_stream.write([$i^-_{e1}$, $i^-_{e0}$, $i^+_{e0}$, $i^+_{e1}$, $\bar{\delta}$])
    out_stream.write([coeff[0], ..., coeff[$n_\mathrm{B}$ - 1]])
    out_stream.gplc_compress([diff[0], ..., diff[$N$ - 1]])
    out_stream.gplc_compress([quant[0], ..., quant[$N$ - 1]])
```

Algorithm 4.1: Pseudo code of d-ISABELA compression procedure dISA_compress (`time_step`, N, n_B, e_MAX, `in_stream`, `out_data`). The procedure illustrates the caching mechanism during compression, as well as, the calculation of the difference encoding. `out_stream.write(data)` copies data into the stream, and `out_stream.gplc_compress(data)` copies compressed data into the stream using a GPLC.

```
def dISA_decompress(time_step, $N$, $n_{\mathrm{B}}$, in_stream, $e_{\mathrm{MAX}}$, out_data):
  if time_step mod $k_{\mathrm{MOD}}$ = 0:
    // decompress keyframe ...
    // reconstruct decompressed data ...
    out_data = ...
    invsort = [] // invert sort order
    for i in range(0, len($\mathcal{S}$)):
      invsort[$\mathcal{S}$[i]] = i
    dISA_cache_set(invsort) // init cache
  else:
    // load data $i^{-}_{e1}$, $i^{-}_{e0}$, $i^{+}_{e0}$, $i^{+}_{e1}$, $\bar{\delta}$, coeff, ...
    // ... diff, quant from in_stream ...
    // reconstruct B-spline $\hat{X}$ ...
    invsort = dISA_cache_get() // load cache
    $\mathcal{S}$ = []
    for i in range(0, len(diff)):
      // (new sort pos) = (old sort pos) + diff
      $\mathcal{S}$[i] = invsort[i + diff[i]]
    // reconstruct decompressed data ...
    out_data = ...
    invsort = [] // invert sort order
    for i in range(0, len($\mathcal{S}$)):
      invsort[$\mathcal{S}$[i]] = i
    dISA_cache_set(invsort) // update cache
```

Algorithm 4.2: Pseudo code of *d*-ISABELA decompression procedure dISA_decompress(`time_step`, N, n_{B}, `in_stream`, e_{MAX}, `out_data`). The procedure illustrates the caching mechanism during decompression, as well as, the reconstruction of the sort order from the difference encoding.

4.3.2 Temporal ISABELA Compression

The ISABELA algorithm relies on sorting for the creation of smooth sequences and a B-spline fit for the approximation of sorted data values. Compared to SBD and GLATE, ISABELA has a lower compression performance due to start-up costs for storing the sort order and the B-spline control points, as explained in Section 3.3.1, p. 33. The original implementation of ISABELA [45] proposed a temporal compression scheme d-ISABELA based on a difference encoding of the sort order, as explained in the previous section. d-ISABELA exploits the fact that the indices of the sorted data change smoothly between consecutive time steps in data obtained from CFD simulations. However, by reusing the B-spline and the sort order of the previous time steps for modeling the new data, the storage for the sort order and the B-spline can be omitted entirely.

t-ISABELA Procedure in KFs

Fig. 4.8 illustrates the procedure of t-ISABELA in KFs. There, the original ISABELA algorithm is applied, except that the sign bit is extracted from the `FLOAT` values, and only positive values are processed in the steps (2)–(4). The original sign bits are stored in a bit stream and take up $^N/_8$ byte. At the end of step (2) and (3), the sort order and the B-spline is cached.

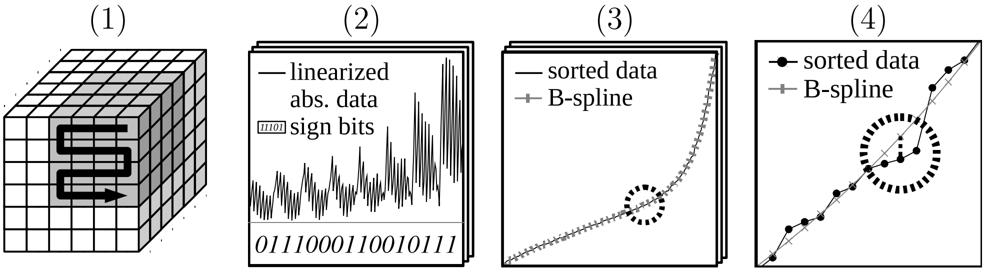


Figure 4.8: t-ISABELA procedure in KFs. (1) Linearize data, (2) determine absolute values by extracting signs, sort data and cache sort order, (3) B-spline regression and cache B-spline, and (4) error quantization.

t-ISABELA Procedure in DFs

Fig. 4.9 shows the new procedure for temporal compression in DFs, which reuses the sort order, as well as, the B-spline from the previous frames. Step (1) and (2) in DFs are conducted similarly to the procedure in KFs. However, in step (3), instead of applying sorting and B-spline regression, the sort order from the previous frame is used for reordering the new data values around the B-spline from the previous frame. Since high-resolution temporal data exhibits strong temporal coherence, the differences between data values in the same grid cells of consecutive time steps are typically small and the reordering produces a point cloud close to the B-spline. In step (4), the noise is compensated through quantization of the differences between B-spline and reordered data values, similar to the error correction mechanism employed in ISABELA [55, 57]. During the error quantization in step (4), the decompressed data is reconstructed, sorted and used to update the B-spline. The new sort order and the new B-spline are used to update the cache for continuation of the temporal compression procedure.

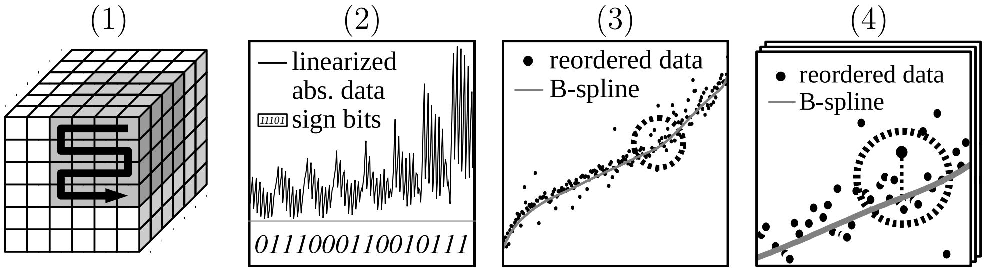


Figure 4.9: t-ISABELA procedure in DFs. (1) Linearize data, (2) determine absolute values by extracting signs, and (3) reorder data using cached sort order and restore B-spline from cache, and (4) error quantization, restore & sort decompressed data and update cache with new sort order and new B-spline.

t-ISABELA Illustration

Fig. 4.10 (1)–(8) illustrates the t-ISABELA temporal compression procedure for one KF at $t = 0$, and four DFs at $t = 1, 2, 3, 4$. First, in the KF at $t = 0$, shown in Fig. 4.10 (1)–(2), the ISABELA compression is applied, and a B-spline fit and a sort order are obtained. The B-spline and the sort order are cached, since they are needed in the DF. Second, in the DF at $t = 1$, shown in Fig. 4.10 (3), the new input data is reordered using the sort order from the previous frame. Because of the smooth data changes in consecutive time steps, the reordering creates a point cloud close to the B-spline from the previous frame. At this point, compared to d-ISABELA, which stores index differences and quantized errors separately, t-ISABELA applies error quantization to the reordered new data directly. During the error quantization in the DF, the decompressed data values of the DF are reconstructed at the same time. Before transitioning into the next time step at $t = 2$, the decompressed data values are used to update the sort order and the B-spline fit, as shown in Fig. 4.10 (4). Using this procedure, the sort order and the B-spline are updated and yield a better prediction for the input data of the next frame, shown in Fig. 4.10 (4)–(5). At the end of every frame, the decompressed data is reconstructed and used to update the B-spline cache and the sort order cache. Hence, a sequence of DFs can be compressed with improved performance.

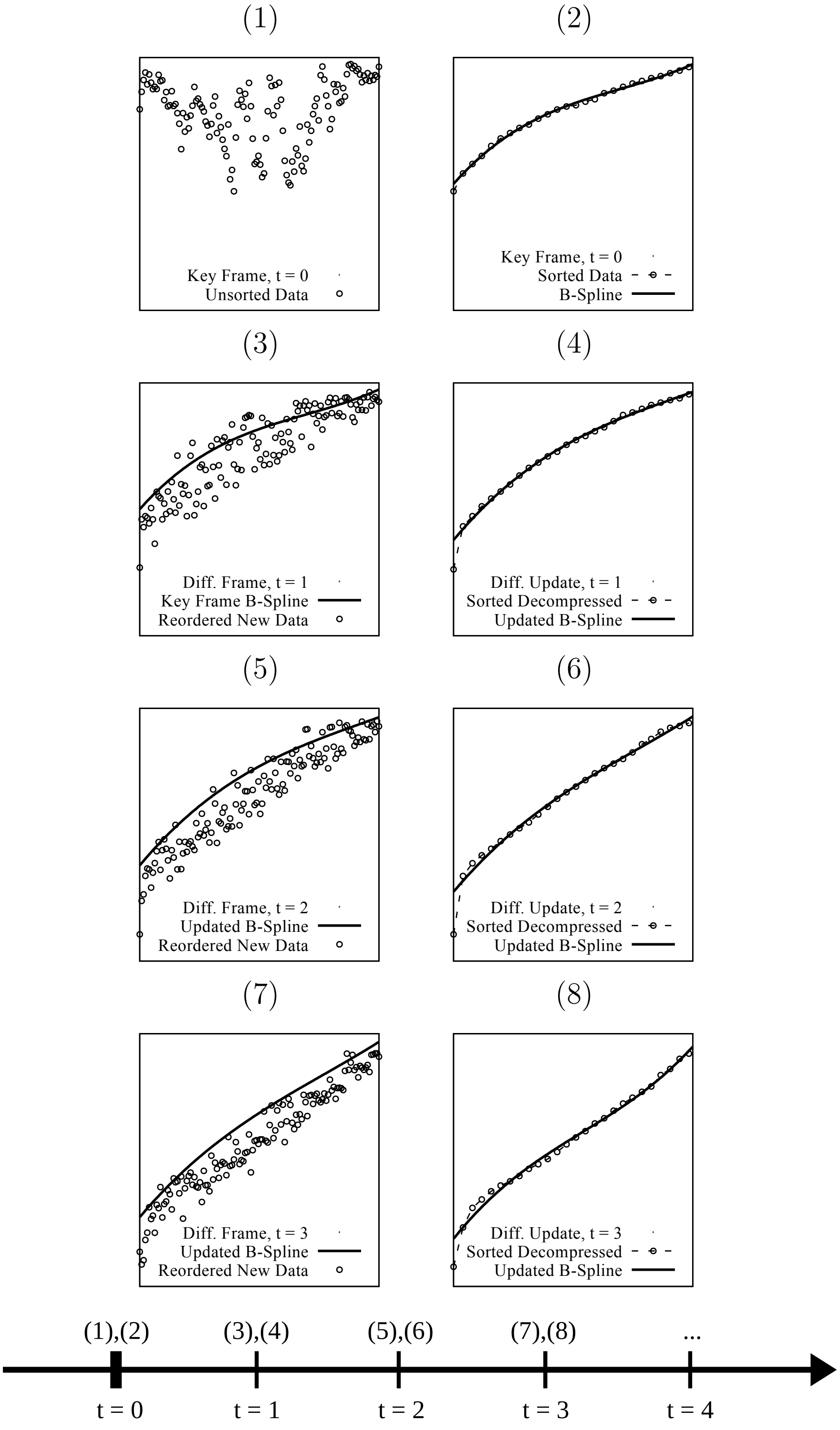


Figure 4.10: t-ISABELA compression procedure. (1)–(2) in a KF, and (3)–(8) in DFs: (1) shows the unsorted data to be compressed in the KF. (2) shows the sorted data and the B-spline fit obtained in the KF. (3),(5),(7) show the new input data of the DF reordered around the B-spline from the previous frame, and (4),(6),(8) show the new B-spline, which has been updated in the DF using the sorted decompressed values.

t-ISABELA Implementation

The procedure tISA_ compress(...), given in Alg. 4.3, corresponds to the algorithmic steps depicted in Fig. 4.8 (1)–(4) and Fig. 4.9 (1)–(4). In KFs, the original ISABELA compression procedure ISA_ compress(...), as shown in Alg. 3.1, is executed on the positive data after the signs have been extracted. At the end of the execution of the KF, the sort order $\mathcal{S}$ and the B-spline are cached. In DFs, first, the signs are extracted, the cached sort order and the B-spline are loaded. Second, the new input data is reordered around the cached B-spline using the cached sort order, and errors are quantized. Third, decompressed data is sorted and fitted in order to update the cache.

The procedure tISA_ decompress(...), given in Alg. 4.4, illustrates the caching mechanism and cache update during decompression. In KFs, the sort order is cached according to the compression procedure. In DFs, the reordered data is reconstructed using the cached sort order, the cached B-spline and quantized errors. After the data has been reconstructed, the data is sorted and fitted in order to update the cache. The signs are used to reconstruct the decompressed data.

```
def tISA_compress(time_step, $N$, $n_{\mathrm{B}}$, in_data, $e_{\mathrm{MAX}}$, out_stream):
  if time_step mod $k_{\mathrm{MOD}}$ = 0:
    [signs, abs_data] = tISA_get_signs(in_data)
    ISA_compress($N$, $n_{\mathrm{B}}$, abs_data, $e_{\mathrm{MAX}}$, out_stream)
    // $\mathcal{S}$ - sort order
    // $\hat{X}$ - B-spline
    // ... obtained from ISA_compress(...)
    out_stream.nbit_encode(1, [signs[0], ..., signs[$N$ - 1]])
    tISA_cache_set($\mathcal{S}$, $\hat{X}$) // init cache
  else:
    [$\mathcal{S}$, $\hat{X}$] = tISA_cache_get()
    [signs, abs_data] = tISA_get_signs(in_data)
    [$X$, $\bar{\delta}$] = tISA_reorder(abs_data, $\mathcal{S}$)
    // $\bar{\delta}$ for estimation of quantization step width
    [quant, out_data] = tISA_quant_error($N$, $X$, $\hat{X}$, $\bar{\delta}$, $e_{\mathrm{MAX}}$)
    [$X$, $\mathcal{S}$] = Boost::SpreadSort(out_data)
    coeff = BSplineFit3($N$, $X$, $n_{\mathrm{B}}$)
    $\hat{X}$ = BSplineSample3($N$, coeff, $n_{\mathrm{B}}$)
    tISA_cache_set($\mathcal{S}$, $\hat{X}$) // update cache
    out_stream.write($\bar{\delta}$)
    out_stream.nbit_encode(1, [signs[0], ..., signs[$N$ - 1]])
    out_stream.gplc_compress([quant[0], ..., quant[$N$ - 1]])
```

Algorithm 4.3: Pseudo code of t-ISABELA compression procedure tISA_compress (`time_step`, N, n_{B}, e_{MAX}, `in_stream`, `out_data`). The procedure illustrates the caching mechanism during compression, as well as, the reordering of new data around the old B-spline. `out_stream.nbit_encode`(n_{BIT}, `data`) copies a vector of n_{BIT}-bit numbers into the stream, `out_stream.write`(`data`) copies data into the stream, and `out_stream.gplc_compress`(`data`) copies compressed data into the stream using a GPLC.

```
def tISA_decompress(time_step, $N$, $n_B$, in_stream, $e_{MAX}$, out_data):
  if time_step mod $k_{MOD}$ = 0:
    // decompress keyframe ...
    // reconstruct positive decompressed data ...
    abs_data = ...
    // load signs from in_stream ...
    tISA_cache_set($\mathcal{S}$, $\hat{X}$) // init cache
    out_data = tISA_set_signs(signs, abs_data)
  else:
    // load $\bar{\delta}$, signs, quant from in_stream ...
    [$\mathcal{S}$, $\hat{X}$] = tISA_cache_get()
    // reconstruct positive decompressed data ...
    abs_data = ...
    [$\tilde{X}$, $\mathcal{S}$] = Boost::SpreadSort(abs_data)
    // $\tilde{X}$ - sorted decompressed data
    coeff = BSplineFit3($N$, $\tilde{X}$, $n_B$)
    $\hat{X}$ = BSplineSample3($N$, coeff, $n_B$)
    tISA_cache_set($\mathcal{S}$, $\hat{X}$) // update cache
    out_data = tISA_set_signs(signs, abs_data)
```

Algorithm 4.4: Pseudo code of t-ISABELA decompression procedure tISA_ decompress (`time_step`, N, n_B, `in_stream`, e_{MAX}, `out_data`). The procedure illustrates the caching mechanism during decompression and the cache update mechanism using sorted decompressed data.

4.3.3 Temporal SBD Compression

The SBD algorithm replaces original data values by indices referring to discrete values inside a LUT, as explained in Section 3.3.2, p. 40. Unlike ISABELA, which requires the storage of the sort order, SBD yields a better compression rate and has no start-up costs for compression. The t-SBD algorithm is a new temporal compression extension for the SBD algorithm and is built on the observation that many LUT indices are similar between subsequent time steps, or only change slightly. Therefore, a differential encoding of LUT indices in the temporal dimension is applied in order to improve the compression rate.

t-SBD Temporal Compression Procedure

Fig. 4.11 shows the original compression procedure of SBD, which is applied in KFs without modification. However, at the end of step (4) in KFs, the LUT indices are cached for the next iteration, as they are needed for the difference encoding in the next DF.

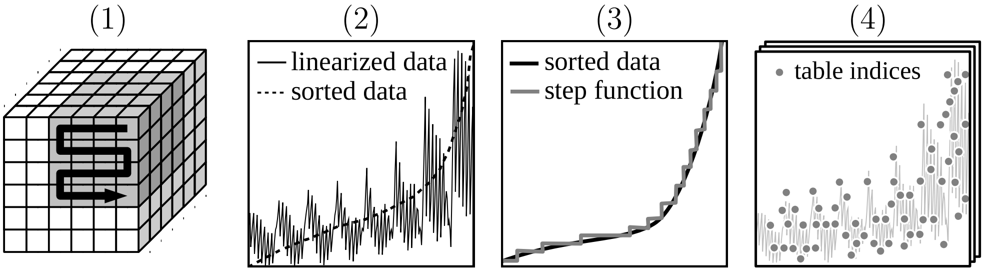


Figure 4.11: t-SBD procedure in KFs. (1) Linearize data, (2) sort data, (3) determine LUT values, and (4) map LUT values to LUT indices and update cache.

The procedure for DFs is shown in Fig. 4.12. The steps (1)–(3) remain unchanged compared to the original SBD procedure in KFs. In step (4), new LUT indices for the data values in the DF are determined in order to calculate the differences between the new indices in the current DF and the old indices in the previous frame. Unlike the LUT indices themselves, the resulting stream of differences contains many small repetitive numbers allowing for more efficient compression using a GPLC. At the end of step (4) in DFs, the new LUT indices are used to update the cache for continuation of the temporal compression procedure.

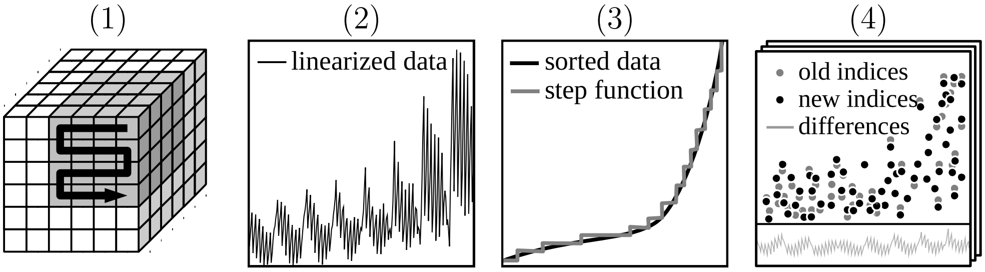


Figure 4.12: t-SBD procedure in DFs. (1) Linearize data, (2) sort data, (3) determine LUT values, and (4) map LUT values to LUT indices, compute differences between old and new LUT indices, and update cache.

t-SBD Illustration

Fig. 4.13 (1)–(9) illustrates the t-SBD temporal compression procedure for one KF at $t = 0$, and four DFs at $t = 1$, 2, 3, 4. First, in the KF at $t = 0$, shown in Fig. 4.13 (1), the SBD compression procedure is applied, and LUT indices for the input data are determined and cached. Second, in the DF at $t = 1$, shown in Fig. 4.13 (2), the LUT indices for the new input data are determined. Using the cached LUT indices from the last KF, and the new LUT indices of the current DF, the differences between LUT indices are computed, as shown in Fig. 4.13 (3). Similarly to the procedure in t-ISABELA, before transitioning in the next DF, the indices cached in the KF are replaced with the new indices determined in the current DF. Using this procedure, small changes in a sequence of consecutive DFs can be encoded with improved compression performance.

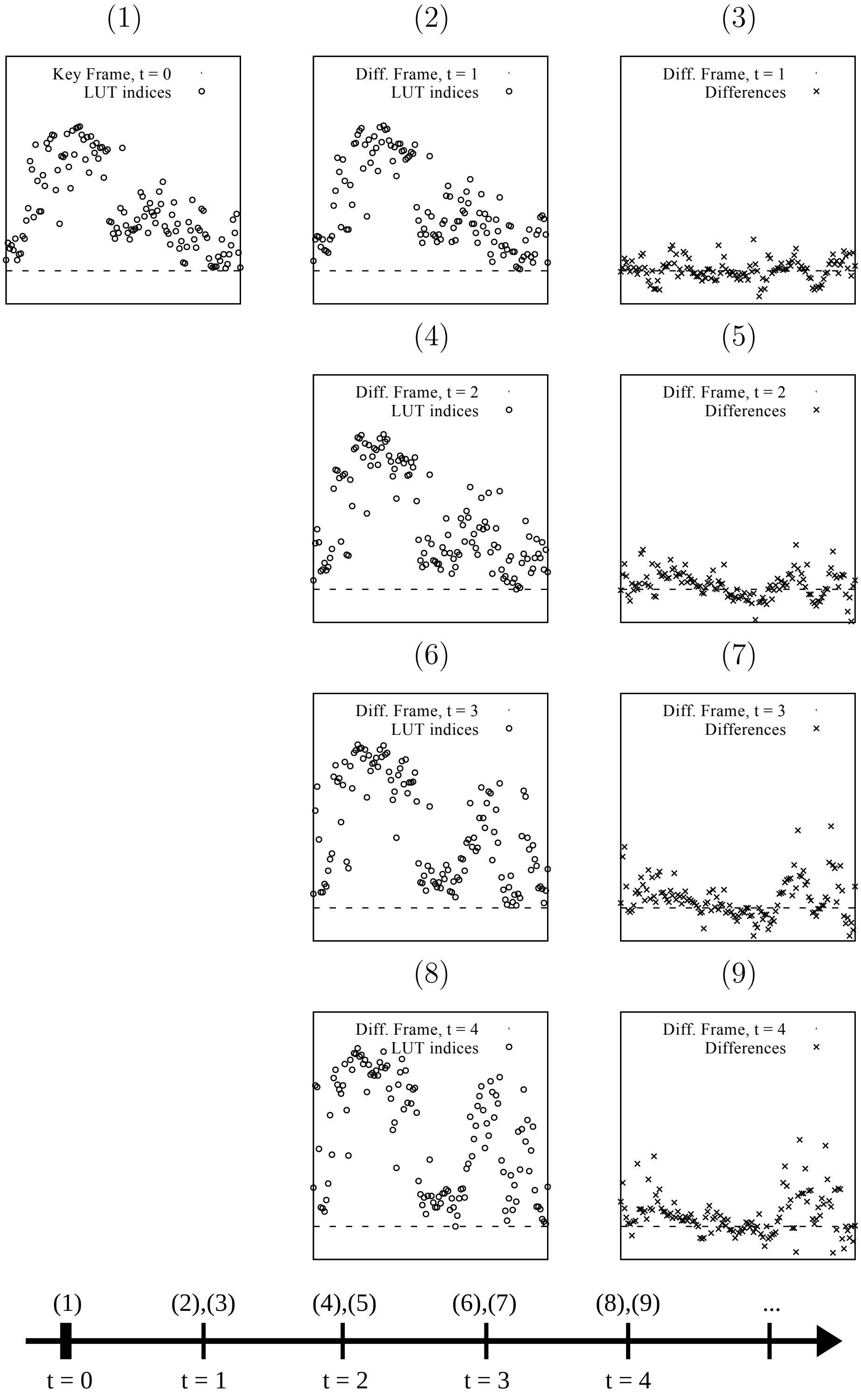


Figure 4.13: *t*-SBD compression procedure. (1) in a KF, and (2)–(9) in DFs: (1) shows the LUT indices computed in the KF, (2),(4),(6),(8) show the LUT indices computed in DFs, and (3),(5),(7),(9) show the differences between LUT indices of the current and the previous frame.

t-SBD Implementation

The procedure tSBD_ compress(...), given in Alg. 4.5, corresponds to the algorithmic steps depicted in Fig. 4.11 (1)–(4) and Fig. 4.12 (1)–(4). In KFs and in DFs, the LUT values and LUT indices are computed for the requested error bound using the input data. The algorithm follows the original SBD compression procedure SBD_ compress (...), given in Alg. 3.2. However, at the end of the execution, the caching mechanism for KFs and DFs is applied. In KFs, LUT indices are determined and cached. In DFs, cached LUT indices are loaded and used for the calculation of differences between new and old LUT indices. The procedure tSBD_ decompress(...), given in Alg. 4.6, illustrates the caching mechanism and cache update during decompression.

```
def tSBD_compress(time_step, $N$, in_data, $e_{\mathrm{MAX}}$, out_stream):
  $n_L$ = 0 // LUT size
  lutv = [] // LUT values
  luti = [] // LUT indices
  [$X$, perm] = Boost::SpreadSort(in_data)
  binmin = $X$[0]
  for i in range(1, $N$):
    if SBD_bin_error(binmin, $X$[i]) > $e_{\mathrm{MAX}}$:
      // if new bin value exceeds maximum error
      lutv.append(SBD_bin_value(binmin, $X$[i - 1]))
      binmin = $X$[i]
      $n_L$ += 1
    luti[perm[i]] = $n_L$
  if luti[perm[$N$ - 1]] = $n_L$:
    // if last value has a separate bin
    lutv.append($X$[$N$ - 1])
  out_stream.write([$n_L$, lutv[0], ..., lutv[$n_L$ - 1]])
  if time_step mod $k_{\mathrm{MOD}}$ = 0:
    // compress keyframe ...
    out_stream.gplc_compress([luti[0], ..., luti[$N$ - 1]])
    tSBD_cache_set(luti) // init cache
  else:
    luticache = tSBD_cache_get() // load cache
    diff = [] // LUT index differences
    for i in range(0, len(luti)):
      diff[i] = luti[i] - luticache[i]
    out_stream.gplc_compress([diff[0], ..., diff[$N$ - 1]])
    tSBD_cache_set(luti) // update cache
```

Algorithm 4.5: Pseudo code of t-SBD compression procedure tSBD_compress (`time_step`, N, e_{MAX}, `in_stream`, `out_data`). The procedure illustrates the caching mechanism during compression, as well as, the calculation of the differences between LUT indices. `out_stream.write`(`data`) copies data into the stream, and `out_stream.gplc_compress`(`data`) copies compressed data into the stream using a GPLC.

```
def tSBD_decompress(time_step, $N$, in_stream, $e_{MAX}$, out_data):
  $n_L$ = 0 // LUT size
  lutv = [] // LUT values
  luti = [] // LUT indices
  if time_step mod $k_{MOD}$ = 0:
    // load lutv, luti from in_stream ...
    tSBD_cache_set(luti) // init cache
  else:
    diff = [] // LUT index differences
    // load lutv, diff from in_stream ...
    luti = tSBD_cache_get()
    for i in range(0, len(luti)):
      luti[i] += diff[i]
    tSBD_cache_set(luti) // update cache
  // reconstruct decompressed data ...
  for i in range(0, $N$):
    out_data[i] = lutv[luti[i]]
```

Algorithm 4.6: Pseudo code of t-SBD decompression procedure tSBD_decompress (`time_step`, N, `in_stream`, e_{MAX}, `out_data`). The procedure illustrates the caching mechanism during decompression and the cache update mechanism.

4.3.4 Temporal GLATE Compression

The GLATE algorithm decomposes `FLOAT` values into exponents and MLUT indices referring to a truncated set of mantissas used for quantization, as explained in Section 3.3.3, p. 45. Because of the clustering properties of SFPD [37], GLATE yields a very good compression rate by applying GPLC to exponents and `tpfor` bit packing to signed MLUT indices. The approach to t-GLATE compression employs a novel difference encoding between consecutive time steps of CFD data. Since data differences between time steps are usually smaller than the absolute values stored in the grid cells, a differential encoding on the GLATE global stepfunction, depicted in Fig. 3.13, p. 50, is employed for improving the compression performance.

t-GLATE Temporal Compression Procedure

Fig. 4.14 shows the original compression procedure of GLATE, which is applied in KFs without modification. However, after completing compression of a KF, the exponents and signed MLUT indices are cached for the next DF, as they are needed for computing the difference encoding between consecutive time steps. Instead of caching the exponents and signed MLUT indices separately, the values are merged into global MLUT indices referring to the global step function. The global MLUT indices are computed using Eq. (3.18), p. 68, as explained in Section 3.4.3. On the global step function, each decompressed value is represented by a unique `INTEGER` value.

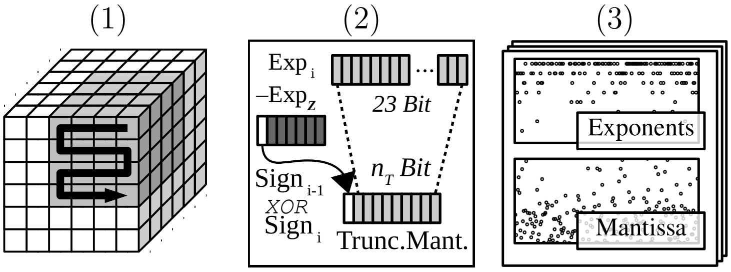


Figure 4.14: *t*-GLATE procedure in KFs. (1) Linearize, (2) decompose `FLOAT` into exponents and signed MLUT indices, (3) compress exponents and signed MLUT indices, and update cache.

The procedure for DFs is shown in Fig. 4.15. The steps (1) and (3) remain unchanged. In step (2), after the exponents and signed MLUT indices are merged into new global MLUT indices representing the new input data, the old global MLUT indices are loaded from the cache. Following, the difference encoding computes *differential* MLUT indices using old and new indices according to (NEW INDEX − OLD INDEX).

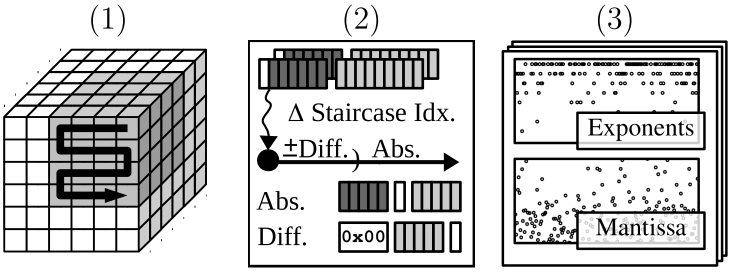


Figure 4.15: *t*-GLATE procedure in DFs. (1) Linearize, (2) compute differential MLUT indices on global step function. If differential MLUT index is too large, the signed MLUT index is used instead, (3) compress exponents and signed/differential MLUT indices, and update cache.

As explained in Section 3.4.3, the MLUT size n_{M} bounds the value range for MLUT indices. Therefore, differential MLUT indices are only encoded if they are contained in the interval $(-{}^{n_{\mathrm{M}}}/_{2}, +{}^{n_{\mathrm{M}}}/_{2})$, otherwise the absolute value is encoded according to the GLATE compression procedure using the corresponding signed MLUT index. Using this procedure, the difference encoding automatically regards differential MLUT indices in the case they would require more bits than a signed MLUT index, and t-GLATE falls back to the absolute encoding of the signed MLUT index instead. Since differential MLUT indices are usually smaller than signed MLUT indices, `tpfor` bit packing can encode the new stream with a lower number of bits improving the compression performance.

As shown in Fig. 3.19 (b), p. 67, and as implemented in GLATE_ compress(...) shown in Alg. 3.3, all values $|x| \geq e_0$ receive an exponent $E' \geq 1$. Further, GLATE uses the zero exponent $E' = 0$ for indication of a absolute zero, which leaves room for encoding of a zero difference and a non-zero difference. Therefore, t-GLATE uses the following encoding.

$$\begin{array}{lclcrl} \textsc{absolute value} & \rightarrow & E' > 0 & \text{and} & \text{signed} & \textsc{mLUT index} \geq 0 \\ \textsc{difference value} & \rightarrow & E' = 0 & \text{and} & \textit{differential} & \textsc{mLUT index} > 1 \\ \textsc{zero difference} & \rightarrow & E' = 0 & \text{and} & \textit{differential} & \textsc{mLUT index} = 1 \\ \textsc{absolute zero} & \rightarrow & E' = 0 & \text{and} & \text{signed} & \textsc{mLUT index} = 0 \end{array} \tag{4.1}$$

$$\begin{array}{rlcccl} & \text{Exponent} & \rightarrow & E' & = & E - E_Z \\ & \textsc{mLUT index} & \rightarrow & k & \in & [0, n_\mathrm{M}) \\ \text{signed} & \textsc{mLUT index} & \rightarrow & k' & \in & [0, 2 \cdot n_\mathrm{M}) \\ \text{global} & \textsc{mLUT index} & \rightarrow & i & = & (\text{-}1)^s \cdot (E' \cdot n_\mathrm{M} + k) \\ \text{differential} & \textsc{mLUT index} & \rightarrow & i_2 - i_1 & \in & (\text{-}{}^{n_\mathrm{M}}/_2, +{}^{n_\mathrm{M}}/_2) \end{array} \tag{4.2}$$

t-GLATE Illustration

Fig. 4.16 (1)–(4) illustrates the t-GLATE temporal compression procedure for one KF at $t = 0$, and five DFs at $t = 1, 2, 3, 4, 5$. First, in the KF at $t = 0$, shown in Fig. 4.16 (1), the exponents E' and the signed MLUT indices k' are determined according to GLATE compression procedure and used to compute the global MLUT indices. At the end of the KF, the global MLUT indices are cached, since they are needed for the temporal compression procedure. Second, in the DF at $t = 1$, the exponents E' and the signed MLUT indices k', as well as, the global MLUT indices are computed for the new input data. After loading the old global MLUT indices from the previous frame, each individual value can be encoded using signed/differential MLUT indices according to Eqs. (4.1) and (4.2), as shown in Fig. 4.16 (2). Before the temporal compression moves on to the next DF, the new global MLUT indices are used to update the cache. Using this procedure, the changes between consecutive time steps remain small and can be compressed with improved performance. The same procedure is repeated for all DFs, as shown in Fig. 4.16 (3)–(4).

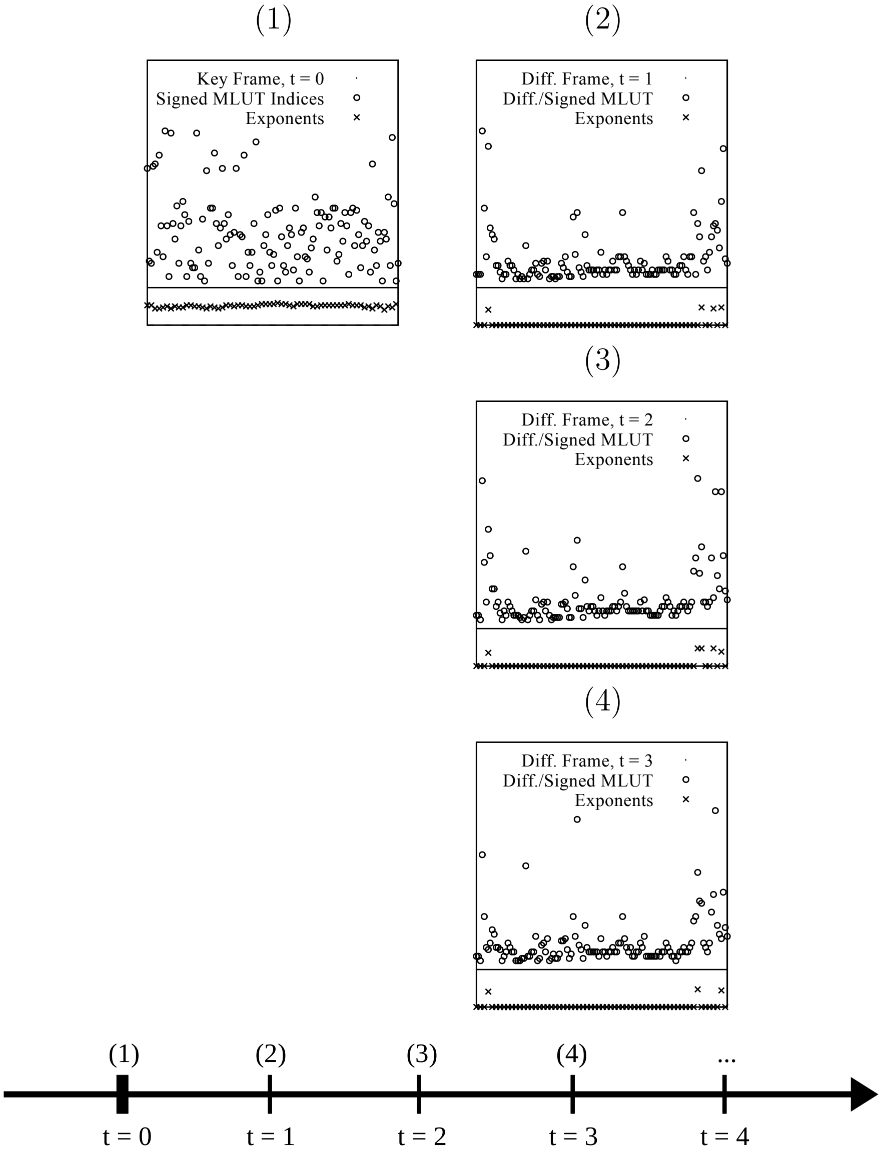


Figure 4.16: t-GLATE compression procedure. (1) in a KF, and (2)–(4) in DFs: (1) shows the exponents E' and signed MLUT indices obtained from encoding the input data in the KF, (2)–(4) show the t-GLATE difference encoding according to Eqs. (4.1) and (4.2). As compared to the KF, the differential MLUT indices reduce the largest `UNSIGNED` number for large parts of the stream in DFs, which allows for improved compression performance using `tpfor`. Further, as differential MLUT indices are indicated by a zero exponent, the difference encoding produces a lot of repetitive zeros in the exponent stream.

t-GLATE Implementation

The procedure tGLATE_compress(...), given in Alg. 4.7, corresponds to the algorithmic steps after data linearization, as depicted in Fig. 4.14 (2)–(3) and Fig. 4.15 (2)–(3). In KFs and in DFs, the GLATE compression procedure GLATE_compress(...), given in Alg. 3.3, is executed in order to determine the exponents and signed MLUT indices. In KFs, after the original GLATE compression procedure has been executed, the global MLUT indices are computed for all input values and cached, as described in Eq. (3.18). In DFs, the cached global MLUT indices are loaded from the cache, and the global MLUT indices for the new input data are computed. Thereby, the `dummy_stream` is passed to GLATE_compress(...) in order to postpone writing of the data until the difference encoding has been carried out. As shown in Eqs. (4.1) and (4.2), *t*-GLATE differentiates between an absolute encoded value (`expo[i]` > 0 and `luti[i]` ≥ 0), the absolute zero (`expo[i]` $= 0$ and `luti[i]` $= 0$), a difference value (`expo[i]` $= 0$ and `luti[i]` > 1), and a zero difference (`expo[i]` $= 0$ and `luti[i]` $= 1$). As signed MLUT indices are bounded by the MLUT size in the interval $[0, n_{\mathrm{M}})$, the differential MLUT values are allowed to fluctuate in the interval $(-{}^{n_{\mathrm{M}}}/_2, +{}^{n_{\mathrm{M}}}/_2)$ before they are discarded and encoded as signed MLUT indices. For transforming differential MLUT indices into `UNSIGNED`, the value is left-shifted by one bit and the sign is encoded as the lowest bit, as illustrated in Fig. 4.15 (3). After the difference encoding has been carried out, the new global MLUT indices are used to update the cache. The procedure tGLATE_decompress (...), given in Alg. 4.8, illustrates the caching mechanism and the reconstruction of the data during decompression. The procedure call tGLATE_float_compose(`expo`, `luti`) composes a `FLOAT` value using an exponent E' and an MLUT index k, and the call tGLATE_float_reconstruct(...) reconstructs a `FLOAT` value using a global MLUT index i.

```
def tGLATE_compress(time_step, $N$, $X$, $e_{\mathrm{MAX}}$, out_stream):
  luti = [] // signed/differential MLUT indices
  expo = [] // exponents - see GLATE_compress(...)
  if time_step mod $k_{\mathrm{MOD}}$ = 0:
    GLATE_compress(time_step, $N$, $X$, $e_{\mathrm{MAX}}$, out_stream)
     // luti, expo obtained from GLATE_compress(...)
    valcache = [] // global MLUT indices
    for i in range(0, len(expo))
      if luti[i] < $n_{\mathrm{M}}$:
        valcache[i] = +( $n_{\mathrm{M}} \cdot$ expo[i] + luti[i] )
      else:
        // signed MLUT index
        valcache[i] = -( $n_{\mathrm{M}} \cdot$ expo[i] + (luti[i] - $n_{\mathrm{M}}$) )
    tGLATE_cache_set(valcache) // init cache
  else:
    GLATE_compress(time_step, $N$, $X$, dummy_stream, $e_{\mathrm{MAX}}$) // compute luti, expo
    valcache = tGLATE_cache_get()
    valquant = 0 // global MLUT index, temporary variable
    for i in range(0, len(expo)):
      if luti[i] < $n_{\mathrm{M}}$:
        valquant = +( $n_{\mathrm{M}} \cdot$ expo[i] + luti[i] )
      else:
        // signed MLUT index
        valquant = -( $n_{\mathrm{M}} \cdot$ expo[i] + (luti[i] - $n_{\mathrm{M}}$) )
      diff = valquant - valcache[i] // differential MLUT index
      valcache[i] = valquant
      if $-^{n_{\mathrm{M}}}/_2$ < diff < 0:
        // differential MLUT index, negative
        expo[i] = 0
        luti[i] = ((-diff) << 1) | 0
      elseif 0 $\leq$ diff < $+^{n_{\mathrm{M}}}/_2$:
        // differential MLUT index, positive or zero
        expo[i] = 0
        luti[i] = ((+diff) << 1) | 1
      else:
        // differetial MLUT index is too large, ...
        // ... use signed MLUT index instead
        pass
    tGLATE_cache_set(valcache) // update cache
    out_stream.gplc_compress([expo[0], ..., expo[$N$ - 1]])
    out_stream.tpfor_pack([luti[0], ..., luti[$N$ - 1]])
```

Algorithm 4.7: Pseudo code of t-GLATE compression procedure tGLATE_compress (`time_step`, N, X, e_{MAX}, `out_stream`). `out_stream`.gplc_compress(`data`) copies compressed data into the stream using a GPLC, and `out_stream` .tpfor_pack(`data`) copies compressed data into the stream using `tpfor`. `dummy_stream` does not perform writes and is used to postpone writing in GLATE_compress(...) for the determination of exponents and signed MLUT indices.

```
def tGLATE_decompress(time_step, $N$, in_stream, $e_{\text{MAX}}$, out_data):
  if time_step mod $k_{\text{MOD}}$ = 0:
    // decompress keyframe ...
    // load expo, luti from in_stream ...
    // reconstruct decompressed data ...
    out_data = ...
    valcache = [] // global MLUT indices
    // build up valcache ... - see tGLATE_compress(...)
    tGLATE_cache_set(valcache) // init cache
  else:
    // load expo, luti from in_stream ...
    valcache = tGLATE_cache_get() // global MLUT indices
    // reconstruct decompressed data ...
    for i in range(0, len(expo)):
      // update valcache ...
      if expo[i] = 0:
        if luti[i] = 0:
          // absolute zero ...
          out_data[i] = 0
        elseif luti[i] & 1 = 0:
          // positive difference ...
          valcache[i] += (luti[i] >> 1)
          out_data[i] = tGLATE_float_reconstruct(valcache[i])
        else:
          // zero or negative difference ...
          valcache[i] -= (luti[i] >> 1)
          out_data[i] = tGLATE_float_reconstruct(valcache[i])
      else:
        // absolute value ...
        out_data[i] = tGLATE_float_compose(expo[i], luti[i])
    // build up new valcache ... - see tGLATE_compress(...)
    tGLATE_cache_set(valcache) // update cache
```

Algorithm 4.8: Pseudo code of *t*-GLATE decompression procedure tGLATE_decompress(`time_step`, N, `in_stream`, e_{MAX}, `out_data`). The procedure illustrates the caching mechanism during decompression, as well as, the data reconstruction from differences.

4.3.5 Optimization of t-GLATE Keyframe Compression

t-GLATE employs a difference encoding on a global step function in order to realize temporal compression. As explained in Section 4.1, the temporal compression exploits the temporal coherence inherently present in high-resolution CFD data. Concretely in DFs, *t*-GLATE encodes data values in *signed* and *differential* mLUT indices, as shown in Fig. 4.15 (2). The differential mLUT indices describe differences on the GLATE global step function, for two values in the same grid cell at different points in time. In principle, any difference between two data values can be encoded this way, as long as the values are expected to be close to each other. As the Hilbert-curve is used for grid linearization, the resulting sequence conserves the spatial coherence, and the same difference encoding can be applied in the spatial dimension for compression of KFs.

t-GLATE Spatial Difference Encoding in KFs

As explained in Section 2.2, the linearization methods used for the creation of the linear input stream for *t*-GLATE conserve the spatial coherence of data values from the three dimensional simulation grid. Concretely, grid cells which are located close to each other in the three dimensional simulation grid are also close to each other in the linearized sequence, shown in Fig. 4.15 (1). Consequently, the spatial coherence inherently present in the linearized sequences from high-resolution CFD simulation grids can be exploited for the improvement of the compression rate in KFs in the same manner as in DFs.

t-GLATE Optimized Implementation in KFs

For the realization of the spatial difference encoding, the *t*-GLATE step (2) in KFs, as shown in Fig. 4.14 (2), is extended for the difference encoding according to Eqs. (4.1) and (4.2) applied in DFs, shown in Fig. 4.15 (2). Concretely, in KFs, the global mLUT indices, as well as, differential mLUT indices are computed directly after the exponents and signed mLUT indices have been determined using GLATE_ compress(...). Thereby, a `dummy_stream` is passed to GLATE_ compress(...) in order to postpone writing of the data until the difference encoding is carried out. According to the *t*-GLATE compression procedure in DFs, the differences of consecutive values in the spatial dimension are encoded in the same manner. The optimized *t*-GLATE compression procedure for KFs and DFs is illustrated in Fig. 4.17 (1)–(3). The pseudo code for steps (2)–(3) of the optimized t-GLATE compression procedure tGLATE_ compress_ opt (. . .) is given in Alg. 4.9.

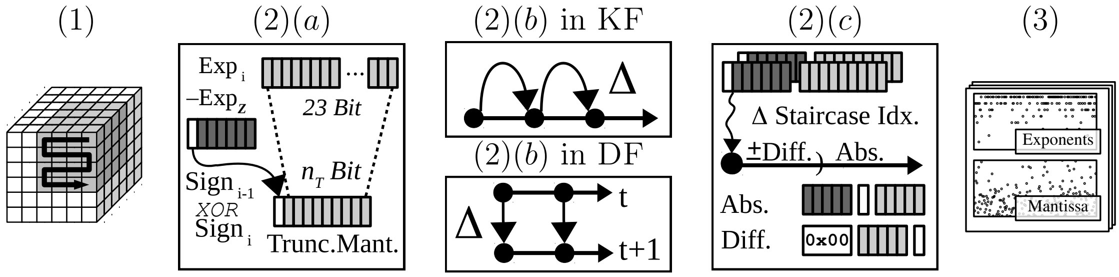

(1) (2)(a) (2)(b) in KF (2)(b) in DF (2)(c) (3)

Figure 4.17: Optimized t-GLATE procedure in KFs and DF. t-GLATE uses difference encoding in the spatial and temporal dimension. (1) Linearize, (2)(a) decompose `FLOAT` into exponents and signed mLUT indices, and compute and cache global mLUT indices, (2)(b) in KF compute differential mLUT indices between values in the linearized sequence, (2)(b) in DF compute differential mLUT indices between values of consecutive time steps, (3) compress exponents and signed/differential mLUT indices, and update cache.

```
def tGLATE_compress_opt(time_step, $N$, $X$, $e_{\text{MAX}}$, out_stream):
  luti = [] // signed/differential MLUT indices
  expo = [] // exponents − see GLATE_compress(...)
  if time_step mod $k_{\text{MOD}}$ = 0:
    GLATE_compress(time_step, $N$, $X$, dummy_stream, $e_{\text{MAX}}$)
     // luti, expo obtained from GLATE_compress(...)
    valquant = 0 // global MLUT index
    valcache = [] // global MLUT indices
    for i in range(0, len(expo))
      if luti[i] < $n_{\text{M}}$:
        valcache[i] = +( $n_{\text{M}} \cdot$ expo[i] + luti[i] )
      else:
        // signed MLUT index
        valcache[i] = −( $n_{\text{M}} \cdot$ expo[i] + (luti[i] − $n_{\text{M}}$) )
      diff = valcache[i] − valquant // differential MLUT index
      valquant = valcache[i]
      if $-^{n_{\text{M}}}/_2$ < diff < 0:
        // differential MLUT index, negative
        expo[i] = 0
        luti[i] = ((−diff) << 1) | 0
      elseif 0 $\leq$ diff < $+^{n_{\text{M}}}/_2$:
        // differential MLUT index, positive or zero
        expo[i] = 0
        luti[i] = ((+diff) << 1) | 1
      else:
        // differetial MLUT index is too large, ...
        // ... use signed MLUT index instead
        pass
    tGLATE_cache_set(valcache) // init cache
    out_stream.gplc_compress([expo[0], ..., expo[$N$ − 1]])
    out_stream.tpfor_pack([luti[0], ..., luti[$N$ − 1]])
  else:
    // compress difference frame ...
    // ... see tGLATE_compress(...)
```

Algorithm 4.9: Pseudo code of the optimized t-GLATE compression procedure tGLATE_compress_opt(N, X, e_{MAX}, `out_stream`). The statement `out_stream.gplc_compress(data)` copies compressed data into the stream using a GPLC, and `out_stream.tpfor_pack(data)` copies compressed data into the stream using `tpfor`. `dummy_stream` does not perform writes and is used to postpone writing in GLATE_compress(...).

4.4 Temporal Compression of Scientific Data Sets

The different temporal compression procedures *d*-ISABELA, *t*-ISABELA, *t*-SBD and *t*-GLATE presented in the previous section are evaluated through a CFD simulation in a cubic cell used to study particle dispersion [11]. The numerical model of the simulation is based on the LBM [85]. The model is used to simulate the periodic flow of fluid aluminum with inlet velocity 3.85 cm/s and temperature of 730 °C, as shown in Fig. 4.18. The simulation has a Reynolds number of $Re = 270$ w.r.t. to the strut width and a temporal step width of $\Delta t = 5.13\,\mu\text{s}$.

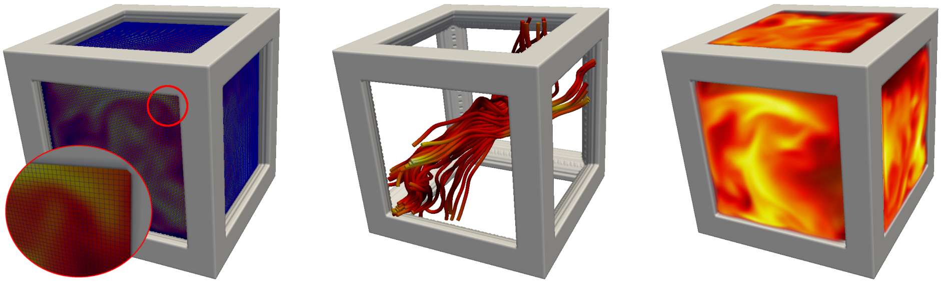


Figure 4.18: Flow of fluid aluminum inside a cubic geometry cell with periodic boundary conditions. The grid consits of 128 × 128 × 128 voxel cells. Colors show the magnitude of the velocity vector (u, v, w) (*Black* corresponds to slow, *Yellow* corresponds to fast).

The LBM is used to produce a sequence comprising 1024 time steps of the aluminum flow flow field (u, v, w) denoted as `CUBC`. Each time step contains $128 \times 128 \times 128$ voxels corresponding to a cell size of $3 \times 3 \times 3$ mm (voxel size 23.4 µm). Depending on the configuration of the temporal compression procedure through k_{MOD} and k_{D}, a subset of all 1024 time steps is compressed according to the placement of KFs and DFs, described in Section 4.2.

The parameter k_{D} controls how many time steps are skipped and not included in the temporal compression procedure, thus k_{D} determines the temporal resolution of the decompressed data. k_{MOD} decides how many DFs follow one KF. For $k_{\text{MOD}} = 1$ no DFs are present and full-random access is possible. For $k_{\text{MOD}} > 1$ the direct access is only possible to KFs, and DFs are stored with improved compression rate. In the following, *d*-ISABELA, *t*-ISABELA, *t*-SBD and *t*-GLATE are applied for the compression of the data set `CUBC`.

4.4.1 d-ISABELA Compression

As explained in section Section 4.3.1, *d*-ISABELA is part of the original ISABELA algorithm and improves the compression rate in a temporal compression procedure [45]. *d*-ISABELA exploits the fact, that the ordering of data values in input streams is similar between subsequent time steps.

d-ISABELA Difference Encoding

Fig. 4.19 (*a*) shows a sort order obtained inside step (2) in the KF procedure of *d*-ISABELA. Since the sort order itself does not contain redundant values, it can be considered incompressible. In contrast, the differences between positions in the sort order of the previous and the current frame yield repetitive small numbers, which are amenable to lossless compression.

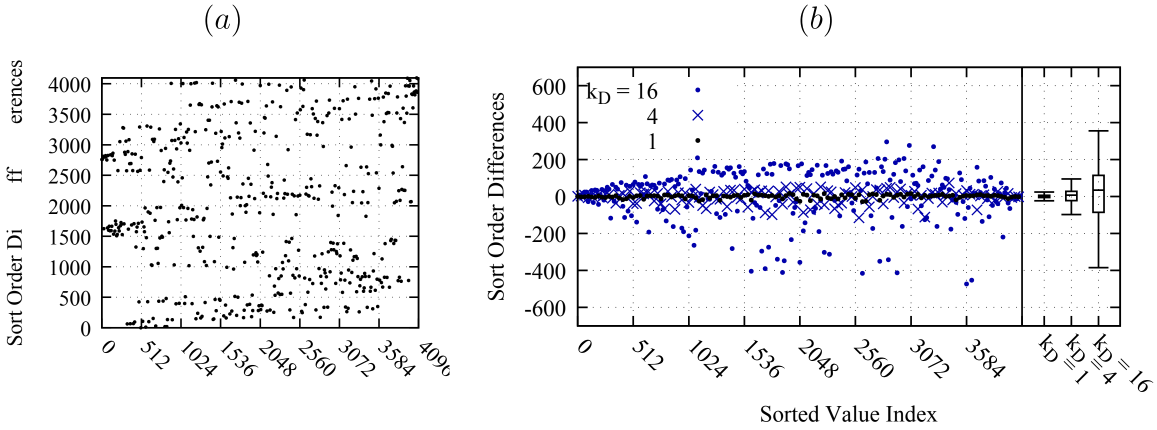
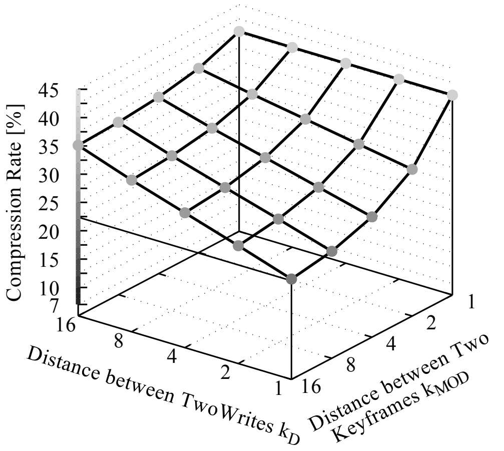


Figure 4.19: *d*-ISABELA internal data. (a) Sort order in a KF. (b) Differences between positions of the sort order from current DF and previous frame. The boxplots show the distribution of differences for $k_\mathrm{D} = 1,\ 4,\ 16$ varying in $[-50, 50]$ up to $[-400, 400]$.

Fig. 4.19 (*b*) shows the differences between positions for a DF in *d*-ISABELA using $k_\mathrm{D} = 1,\ 4,\ 16$. As can be seen, with increasing temporal resolution, the fluctuation of differences increases from close to zero up to ~100 and more.

d-ISABELA Compression Rate

Table 4.1 shows the compression rate of *d*-ISABELA resulting for k_{MOD}, $k_{\mathrm{D}} = 1$, 2, 4, 8, 16. As can be seen, for $k_{\mathrm{MOD}} = 16$ and $k_{\mathrm{D}} = 1$, the compression rate is best at ~22.4 %$_{32b}$. Increasing the temporal resolution $k_{\mathrm{D}} > 1$, or decreasing the number of DFs $k_{\mathrm{MOD}} < 16$, decreases the compression performance. For $e_{\mathrm{MAX}} = 1.00\,\%$, *d*-ISABELA achieves a compression rate between ~22.4–40.0 %$_{32b}$ depending on the amount of DFs and on the temporal resolution of decompressed data.



[%$_{32b}$] k_{D}	1	2	4	8	16
$k_{\mathrm{MOD}} = 1$	40.0	40.0	40.0	40.0	40.0
$k_{\mathrm{MOD}} = 2$	30.7	32.3	33.9	35.5	37.3
$k_{\mathrm{MOD}} = 4$	26.0	28.4	30.8	33.2	35.9
$k_{\mathrm{MOD}} = 8$	23.6	26.5	29.3	32.1	35.2
$k_{\mathrm{MOD}} = 16$	22.4	25.5	28.5	31.5	34.8

Table 4.1: *d*-ISABELA compression rate for data set `CUBC` with k_{MOD}, $k_{\mathrm{D}} = 1$, 2, 4, 8, 16 and input stream size $N = 16^3$. The compression is carried out using `zstd` on level 2 and error bound $e_{\mathrm{MAX}} = 1.00\,\%$.

4.4.2 t-ISABELA Compression

Unlike *d*-ISABELA, *t*-ISABELA aims at compressing differences between new input data and a B-spline from a previous frame. The new input values are reordered using the sort order obtained from the same previous frame. Once the values have been reordered using the cached sort order, the remaining error between B-spline and the data is quantized yielding low `INTEGER` numbers amenable to lossless compression. At the end of the DF, the decompressed data is used to update the sort order and the B-spline, which are used as prediction in the next frame.

t-ISABELA Difference Encoding

t-ISABELA completely avoids the storage of the sort order and the B-spline inside DFs. Therefore, t-ISABELA overcomes the fixed start-up costs induced by ISABELA. As shown in Fig. 4.19 (a), the sort order is very noisy. However, once the data is sorted, the B-spline yields a accurate approximation and the remaining error is quantized into small `INTEGER` numbers, as shown in Fig. 4.20 (a).

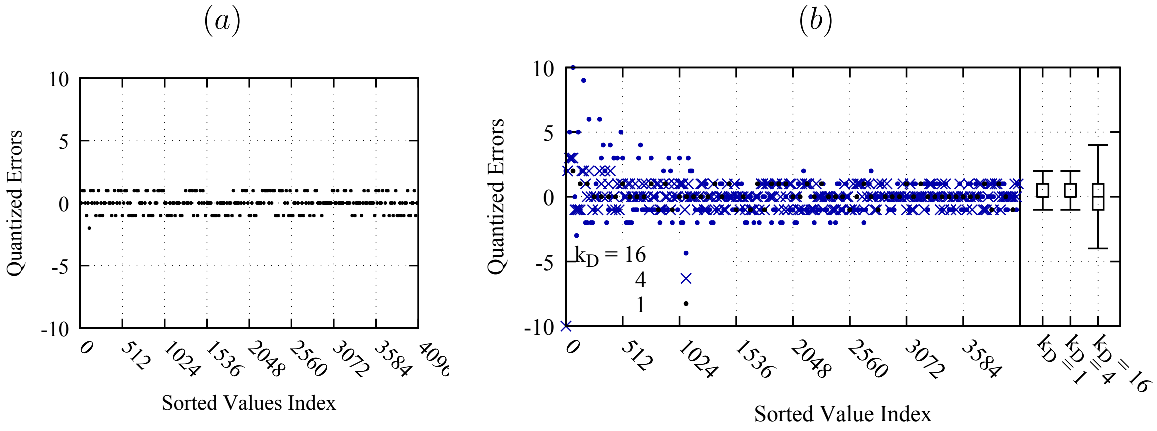
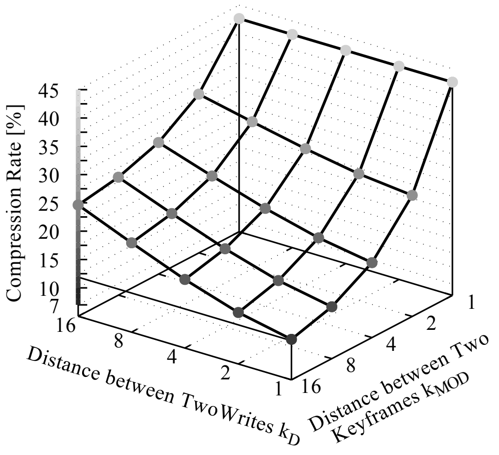


Figure 4.20: t-ISABELA internal data. (a) Quantized errors in a KF. (b) Quantized errors in a DF. The boxplots show the distribution of quantized errors for $k_{\mathrm{D}} = 1,\ 4,\ 16$ varying in $[-2, 2]$ up to $[-10, 10]$.

Fig. 4.20 (a) shows the quantized errors in a KF, which are mostly zero because of the accurate B-spline fit. The same observation holds when observing the error between the old B-spline and the reordered data in a DF. Fig. 4.20 (b) shows the quantized errors resulting for $k_{\mathrm{D}} = 1,\ 4,\ 16$. As can be seen, quantized errors are low and the reordered data follows the trend of the old B-spline. As quantized errors are less noisy than differences between sort order positions shown in Fig. 4.19 (b), t-ISABELA yields better compression rates than d-ISABELA.

t-ISABELA Compression Rate

Table 4.2 shows the compression rate of t-ISABELA resulting for k_{MOD}, $k_{\mathrm{D}} = 1,\ 2,\ 4,\ 8,\ 16$. For $k_{\mathrm{MOD}} = 16$ and $k_{\mathrm{D}} = 1$, the compression rate is best at ~11.9 $\%_{32b}$, which is a ~9.5 $\%_{32b}$ improvement compared to d-ISABELA. When decreasing the amount of DFs using k_{MOD}, or when increasing the temporal resolution using k_{D}, the compression performance decreases. For $e_{\text{MAX}} = 1.00\,\%$, t-ISABELA achieves a compression rate between ~11.9–42.4 $\%_{\text{32b}}$ depending on k_{MOD} and k_{D}.



[$\%_{\text{32b}}$] k_{D}	1	2	4	8	16
$k_{\text{MOD}} = 1$	42.4	42.4	42.4	42.4	42.4
$k_{\text{MOD}} = 2$	26.1	27.2	28.8	30.7	32.8
$k_{\text{MOD}} = 4$	18.0	19.5	21.9	24.9	28.0
$k_{\text{MOD}} = 8$	13.9	15.7	18.5	22.0	25.6
$k_{\text{MOD}} = 16$	11.9	13.8	16.8	20.5	24.4

Table 4.2: t-ISABELA compression rate for data set `CUBC` with k_{MOD}, $k_{\text{D}} = 1$, 2, 4, 8, 16 and input stream size $N = 16^3$. Compression is carried out using `zstd` on level 2 and error bound $e_{\text{MAX}} = 1.00\,\%$.

4.4.3 t-SBD Compression

t-SBD also exploits similarities between the sorted data of consecutive time steps. As the sort order changes smoothly, SBD yields similar sequences of LUT indices between consecutive time steps. Particularly, t-SBD determines and stores the LUT repeatedly in every time step, and encodes differences between sequences of LUT indices in order to improve the compression performance.

t-SBD Difference Encoding

Fig. 4.21 (a) shows a sequence of LUT indices from an input stream taken from a KF of the `CUBC` data set. Progressing into the next DF, the resulting sequence of LUT indices keeps the rough shape of the previous sequence, as can be seen in Fig. 4.21 (b). The differences between the LUT indices of the DF and its previous frame are small, and increase depending on $k_{\text{D}} = 1$, 4, 16. As can be seen, differences constitute repetitive patterns and fluctuate within a narrow range. Therefore, they can be compressed more efficiently than LUT indices using a GPLC.

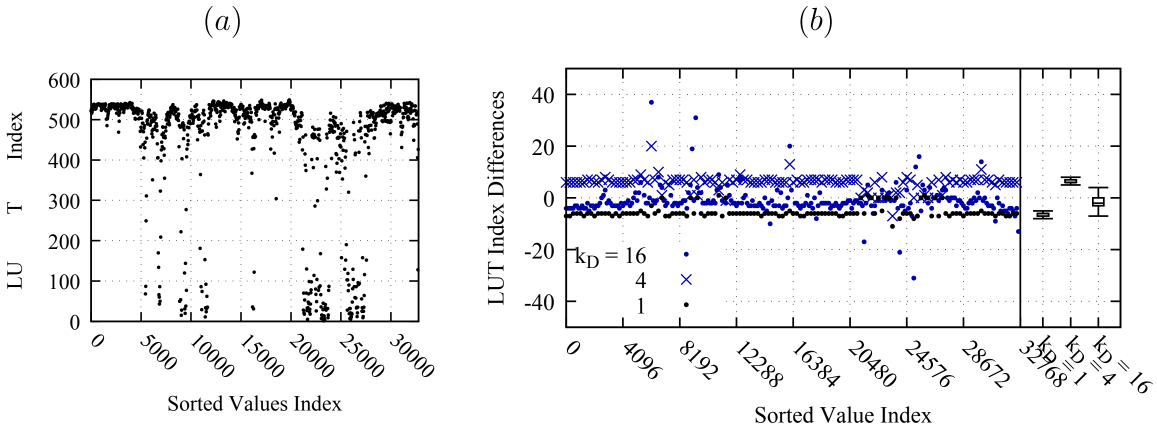


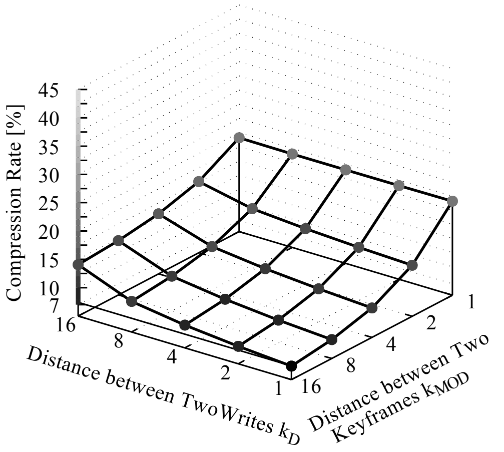
Figure 4.21: t-SBD internal data. (a) Sequence of LUT indices obtained from a KF in the `CUBC` data set. (b) Differences between LUT indices from the KF and the DF. The boxplots show distribution of differences for $k_{\mathrm{D}} = 1$, 4, 16.

t-SBD Compression Rate

Table 4.3 shows the compression rate of t-SBD resulting for k_{MOD}, $k_{\mathrm{D}} = 1$, 2, 4, 8, 16. For $k_{\mathrm{MOD}} = 16$ and $k_{\mathrm{D}} = 1$, the compression rate is best at ~7.4 ‰$_{32b}$, which is better than t-ISABELA and d-ISABELA. By decreasing the amount of DF through k_{MOD}, or by increasing the temporal resolution through k_{D}, the compression performance of t-SBD decreases. For $e_{\mathrm{MAX}} = 1.00\,\%$, t-SBD achieves a compression rate between ~7.4–24.4 ‰$_{32b}$ depending on k_{MOD} and k_{D}.

t-SBD Compression Artifacts

The t-SBD algorithm achieves a better compression rate than d-ISABELA and t-ISABELA. However, the compression procedure of SBD introduces a systematic error into the decompressed data. The construction of the LUT from a sorted sequence in every time step causes visually noticeable artifacts, if the temporal resolution is too high and changes in the data are too small. As can be seen in Fig. 4.22, the decompressed data fluctuates randomly within the error bound if the temporal resolution is too high and data changes are too small.



[%$_{32b}$] k_D	1	2	4	8	16
$k_{MOD} = 1$	24.5	24.5	24.4	24.4	24.4
$k_{MOD} = 2$	14.0	14.3	14.8	15.5	16.6
$k_{MOD} = 4$	10.1	10.7	11.5	12.6	14.2
$k_{MOD} = 8$	8.3	8.9	9.8	11.1	12.9
$k_{MOD} = 16$	7.4	8.0	9.0	10.3	12.3

Table 4.3: t-SBD compression rate for data set `CUBC` with k_{MOD}, $k_D = 1, 2, 4, 8, 16$ and input stream size $N = 64^3$. Compression is carried out using `zstd` on level 6 and error bound $e_{MAX} = 1.00\,\%$.

The problem arises from SBD version 1, as implemented in Alg. 3.2. When sorting the data, the smallest value of the sorted sequence usually fluctuates stronger than values in the center of the sorted sequence. Therefore, the calculation of the LUT values is not stable w.r.t. the LUT of the previous time step. Fig. 4.23 (1)–(3) shows original and decompressed data or three consecutive time steps of the `CUBC` data set. The red rectangle illustrates the fluctuations of LUT values for fast velocity regions on high-resolution temporal data.

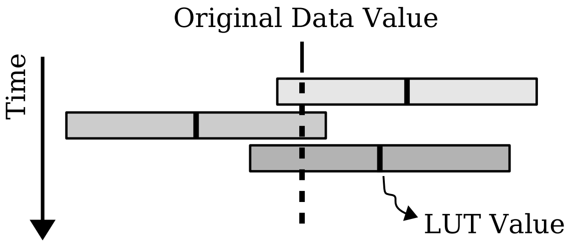


Figure 4.22: t-SBD fluctuating LUT values. The LUT values may fluctuate around the original data value in a random way. If the changes between consecutive time steps are too small, the fluctuations become visible.

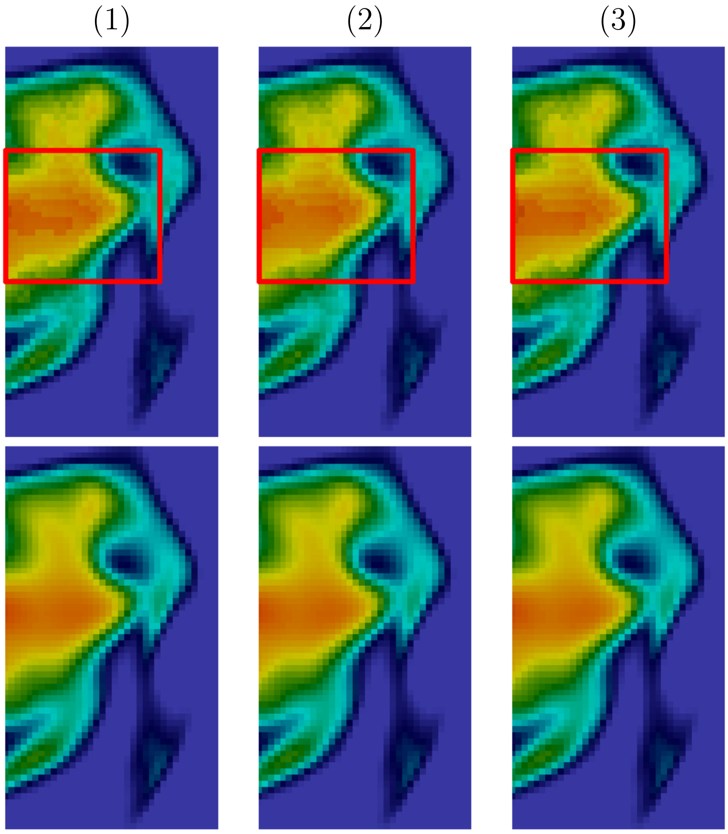


Figure 4.23: t-SBD compression artifacts. The plot shows the velocity magnitude of the CUBC data set for (TOP) t-SBD decompressed data using $e_{\text{MAX}} = 1.00\,\%$, and (BOTTOM) uncompressed original data (*Blue* corresponds to slow, *Orange* corresponds to fast). Red rectangles indicate high velocity regions with a strong fluctuation within the error bound.

4.4.4 t-GLATE Compression

As compared to LUT values in t-SBD, t-GLATE encodes small differences on a global step function, which does not require stabilization. Unlike SBD, the global step function employed for quantization in GLATE only depends on the error bound e_{MAX} and not on the data. Therefore, the t-GLATE difference encoding can handle high-resolution temporal data, and the decompressed values do not fluctuate within the error bound.

t-GLATE Difference Encoding

Fig. 4.24 (a) shows the signed mLUT indices from a KF in the t-GLATE compression procedure. As already mentioned in Section 3.4.3, the signed mLUT indices fluctuate in a bounded range $[0, 2 \cdot n_{\mathrm{M}})$, which size is determined by the error bound e_{MAX} only. Fig. 4.24 (b) shows the signed/differential mLUT indices computed in the next DF for $k_{\mathrm{D}} = 1$, 4, 16 using $e_{\mathrm{MAX}} = 1.00\,\%$.

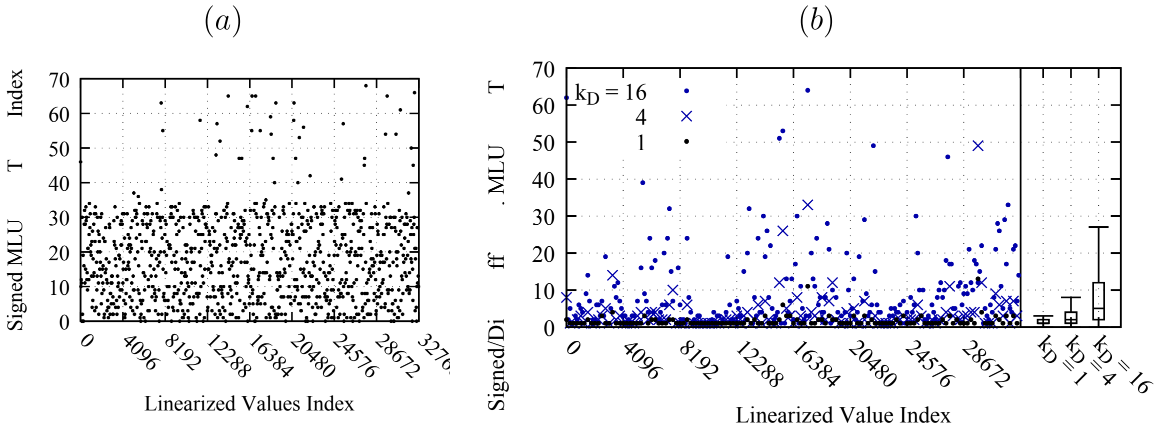
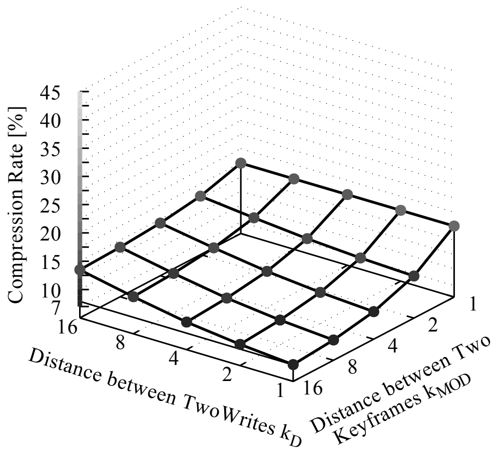

(a) (b)

Figure 4.24: t-GLATE internal data. (a) Stream of mantissa indices in KFs using $n_{\mathrm{M}} = 35$ quantized mantissas in the range $[0, 2 \cdot n_{\mathrm{M}})$. (b) Stream of differences between mantissa indices in a DF in the range $[0, n_{\mathrm{M}})$. Data is generated using $e_{\mathrm{MAX}} = 1.00\,\%$

The exponents and mLUT indices are encoded according to Eq. (4.1) and Eq. (4.2), p. 104. As can be seen, the signed/differetial mLUT indices in the DFs are reasonably smaller than the signed mLUT indices in the KF resulting in more efficient compression.

t-GLATE Compression Rate

Table 4.4 shows the compression rate of t-GLATE resulting for k_{MOD}, $k_{\text{D}} = 1, 2, 4, 8, 16$. For $k_{\text{MOD}} = 16$ and $k_{\text{D}} = 1$, the compression rate is best at ~7.9 $\%_{32b}$, which outperforms t-SBD, t-ISABELA and d-ISABELA. Further, in KFs, the optimized t-GLATE compression procedure improved the compression rate by ~2.0–3.5 $\%_{32b}$, as can be seen in Fig. 4.25. As t-GLATE performs quantization on a global step function, the compression artifacts of t-SBD are prevented. t-GLATE establishes a trade-off between the compression rate, the amount of DF through k_{MOD}, and the temporal resolution through k_{D}. For $e_{\text{MAX}} = 1.00\,\%$, t-GLATE achieves a compression rate of ~7.9–17.4 $\%_{32b}$ depending on k_{MOD}, k_{D} and constitutes the best temporal compression performance.



[$\%_{32b}$] k_{D}	1	2	4	8	16
$k_{\text{MOD}} = 1$	17.5	17.5	17.4	17.4	17.4
$k_{\text{MOD}} = 2$	12.4	12.8	13.4	14.3	15.3
$k_{\text{MOD}} = 4$	9.8	10.4	11.3	12.7	14.2
$k_{\text{MOD}} = 8$	8.5	9.3	10.3	11.9	13.7
$k_{\text{MOD}} = 16$	7.9	8.7	9.8	11.5	13.4

Table 4.4: t-GLATE compression rate for data set `CUBC` with k_{MOD}, $k_{\text{D}} = 1, 2, 4, 8, 16$ and input stream size $N = 64^3$. Compression is carried out `zstd` on level 2, `tpfor` with block size 128, and error bound $e_{\text{MAX}} = 1.00\,\%$. The optimized t-GLATE compression procedure for KFs and DFs is used.

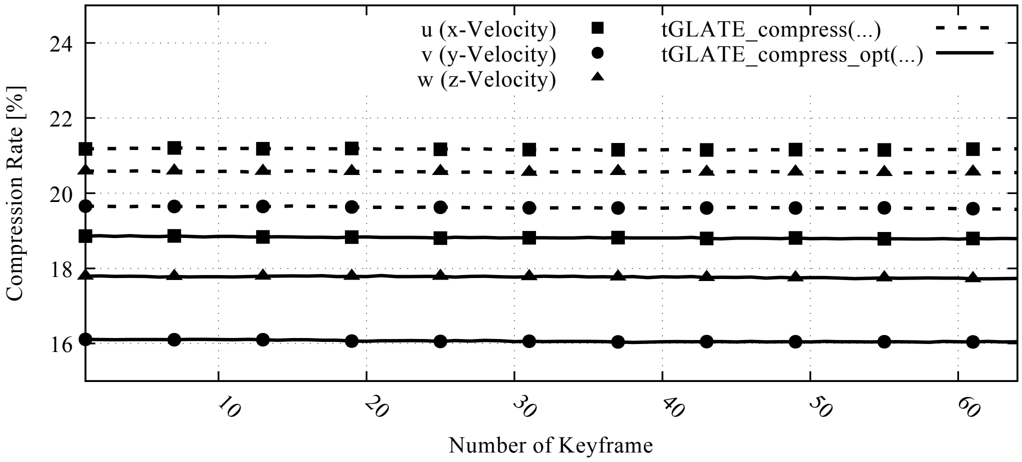


Figure 4.25: Compression rate improvement through optimized t-GLATE compression procedure in KFs. Plot shows the compression rate for the velocity vector (u, v, w) in the `CUBC` data set. Dashed line shows the compression rate obtained by t-GLATE compression procedure tGLATE_ compress (...). Solid line shows the compression rate obtained by the optimized t-GLATE compression procedure tGLATE_ compress_ opt (...).

4.5 Comparison to ZFP Lossy Floating Point Compressor

During the development of t-GLATE, the lossy floating point compressor `ZFP` [63] was published. Whereas t-GLATE bounds the maximum relative error of decompressed data using scalar quantization, the `ZFP` algorithm is inspired by fixed-rate texture compression which is used in graphics hardware. `ZFP` takes into account the spatial coherence of $4 \times 4 \times 4 = 64$ values in three dimensional uniform grids by employing a orthogonal block transform and embedded coding [63].

From user perspective, `ZFP` allows for different modes of compression, i.e. near-lossless, restrict absolute error, and restrict precision of decompressed data. In the context of t-GLATE compression, which restricts the relative error, the `ZFP` precision level is most interesting for this comparison. Specifically, `ZFP` allows for the configuration of a precision level in the range of 1, ..., 32 for `FLOAT` data, which is related to the relative error of decompressed data e_{MAX} used in t-GLATE. The precision level is related to the number of bits of agreement between original data and decompressed data [63].

t-GLATE and ZFP Compression Test

In order to outline the performance of *t*-GLATE temporal in-situ compression w.r.t. the compression in `ZFP`, the compression rate and the compression error are determined through compression tests on *y*-velocity component of 60 time steps (= 960, …, 1020) in the temporal end state of the high-resolution temporal data set `CUBC`. Specifically, *t*-GLATE uses the error bounds $e_{\mathrm{MAX}} = 1.00, 0.50, 0.25, 0.10, 0.05, 0.02, 0.01\,\%$. For `ZFP`, the compression of the `CUBC` data set is carried out using the precision level 9, …, 20. Furthermore, `ZFP` is applied in three dimensional compression mode, i.e. `ZFP` knows about the size of the underlying uniform grid and performs on the most competitive compression mode.

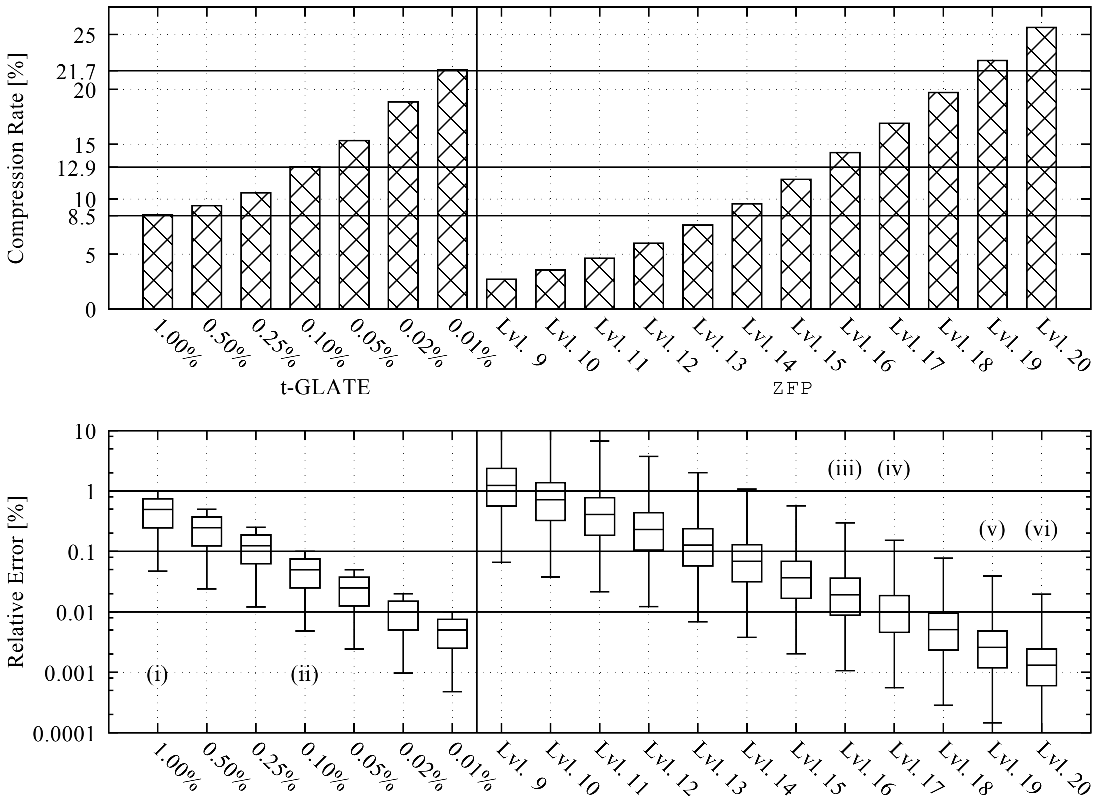
Compression Performance for Uniform Grid Data

Fig. 4.26 (TOP) shows the compression rate resulting from the aforementioned compression test for *t*-GLATE and `ZFP`. As can be seen, both algorithms reach compression rates of ~25 %$_{32b}$ and better while establishing a trade-off between compression rate and quality of decompressed data. `ZFP` reaches a better compression rate than *t*-GLATE for precision level 9, …, 13. However, as can be seen in Fig. 4.26 (CENTER), the mean error of decompressed data varies between $e_{\mathrm{MAX}} = 1.00, \ldots, 0.1\,\%$ for `ZFP`, whereas *t*-GLATE restricts the maximum error. The `ZFP` error varies by more than one order of magnitude, whereas *t*-GLATE bounds the maximum error for similar compression rates.

Concluding, *t*-GLATE and `ZFP` yield similar performance w.r.t. different error-specific compression tasks, i.e. `ZFP` restricts the mean error, whereas *t*-GLATE restricts the maximum decompression error. However, regarding the policy for the restriction of the maximum relative error proposed in Section 3.2, p. 30, *t*-GLATE achieves an improved compression rate using scalar quantization and temporal compression, as elaborated in the following.

Overview of Quality of Decompressed Data

Fig. 4.26 (BOTTOM) shows the percentage $q_{\mathrm{MAX}}^{1.00\,\%}$, $q_{\mathrm{MAX}}^{0.10\,\%}$, $q_{\mathrm{MAX}}^{0.01\,\%}$ of decompressed values, which *exceed* the error bound $e_{\mathrm{MAX}} = 1.00, 0.10, 0.01\,\%$, i.e. the user-defined error bound is violated. The value range of the *y*-velocity component used for testing spans $[-0.03, +0.07]$. In order to count only the relevant decompressed data values, which



[%]	t-GLATE 1.00%	0.50%	0.25%	0.10%	0.05%	0.02%	0.01%	ZFP Lvl. 9	Lvl. 10	Lvl. 11	Lvl. 12	Lvl. 13	Lvl. 14	Lvl. 15	Lvl. 16	Lvl. 17	Lvl. 18	Lvl. 19	Lvl. 20
$q_{MAX}^{1.00\%}=$	0.0	0.0	0.0	0.0	0.0	0.0	0.0	57.9	36.8	17.0	6.7	2.8	1.3	0.5	0.2	0.0	0.0	0.0	0.0
$q_{MAX}^{0.10\%}=$	89.6	79.6	59.6	0.0	0.0	0.0	0.0	95.2	91.8	86.0	75.8	58.6	34.7	13.6	4.6	1.9	0.8	0.3	0.1
$q_{MAX}^{0.01\%}=$	98.7	97.7	95.6	89.6	79.6	49.3	0.0	99.2	98.9	98.3	97.2	95.2	91.5	84.4	71.3	49.3	22.6	7.1	2.6

Figure 4.26: Overview of compression performance for t-GLATE and ZFP on uniform grid data set. t-GLATE ($k_{\mathrm{D}} = 1$, $k_{\mathrm{MOD}} = 16$) is shown for $e_{\mathrm{MAX}} = 1.00$, 0.50, 0.25, 0.10, 0.05, 0.02, 0.01 %, and ZFP using precision level 9, …, 20. Compression performance is shown w.r.t. (TOP) compression rate, and (CENTER) decompression error. (BOTTOM) The table shows the amount of values $|x| > 1\text{E}{-4}$ exceeding the error bound e_{MAX} using the y-velocity component of the CUBC data set. The optimized t-GLATE compression procedure in KFs and DFs is used. ZFP is operated in three dimensional mode. Grayed out cells, correspond to configurations which do not restrict the requested error bound. ZFP with precision level 16 approximately restricts the error to at most ~1.00 %.

violate the maximum decompression error, small numbers $|x| \leq$ 1E–4 are excluded from the statistics. Therefore, the values $q_{\text{MAX}}^{1.00\,\%}$, $q_{\text{MAX}}^{0.10\,\%}$, $q_{\text{MAX}}^{0.01\,\%}$ in percent correspond to the fraction of values $|x| >$ 1E–4 for which the decompression error exceeds $e_{\text{MAX}} = 1.00$, 0.10, 0.01 % respectively.

Taking into consideration the trend of the `ZFP` mean error only, `ZFP` is ~5 $\%_{32b}$ better than t-GLATE but has a wider error distribution which also effects larger values $|x| >$ 1E–4. This behavior also occurs for lower error bounds, e.g. for `ZFP` precision level 16, an amount of ~4.6 % of decompressed values exhibit an error greater than $e_{\text{MAX}} = 0.10\,\%$, and for precision level 19, an amount of ~7.1 % of decompressed values exceed an error of $e_{\text{MAX}} = 0.01\,\%$.

The following examples illustrate the t-GLATE and `ZFP` compression performance w.r.t. to the compression rate and the decompression error.

(*i*) t-GLATE compression rate ~8.5 $\%_{32b}$ for $e_{\text{MAX}} = 1.00\,\%$, $q_{\text{MAX}}^{1.00\,\%} = 0\,\%$.

(*ii*) t-GLATE compression rate ~12.9 $\%_{32b}$ for $e_{\text{MAX}} = 0.10\,\%$, $q_{\text{MAX}}^{0.10\,\%} = 0\,\%$.

t-GLATE compression rate ~21.7 $\%_{32b}$ for $e_{\text{MAX}} = 0.01\,\%$, $q_{\text{MAX}}^{0.01\,\%} = 0\,\%$.

`ZFP` compression rate ~7.6 $\%_{32b}$ for precision level 13, $q_{\text{MAX}}^{1.00\,\%} = 2.8\,\%$.

`ZFP` compression rate ~9.5 $\%_{32b}$ for precision level 14, $q_{\text{MAX}}^{1.00\,\%} = 1.3\,\%$.

(*iii*) `ZFP` compression rate ~14.2 $\%_{32b}$ for precision level 16, $q_{\text{MAX}}^{0.10\,\%} = 4.6\,\%$.

(*iv*) `ZFP` compression rate ~16.8 $\%_{32b}$ for precision level 17, $q_{\text{MAX}}^{0.10\,\%} = 1.9\,\%$.

(*v*) `ZFP` compression rate ~22.6 $\%_{32b}$ for precision level 19, $q_{\text{MAX}}^{0.01\,\%} = 7.1\,\%$.

(*vi*) `ZFP` compression rate ~25.6 $\%_{32b}$ for precision level 20, $q_{\text{MAX}}^{0.01\,\%} = 2.6\,\%$.

The examples denoted as (*i*)–(*vi*) correspond to the boxplots in Fig. 4.26 (CENTER) and to the error scatter plots in Fig. 4.27 (*i*)–(*vi*). The plots support the error statistics, e.g. for `ZFP` precision level 13, …, 14, about ~1.4–2.8 % of values $|x| >$ 1E–4 exhibit decompression error greater than $e_{\text{MAX}} = 1.00\,\%$.

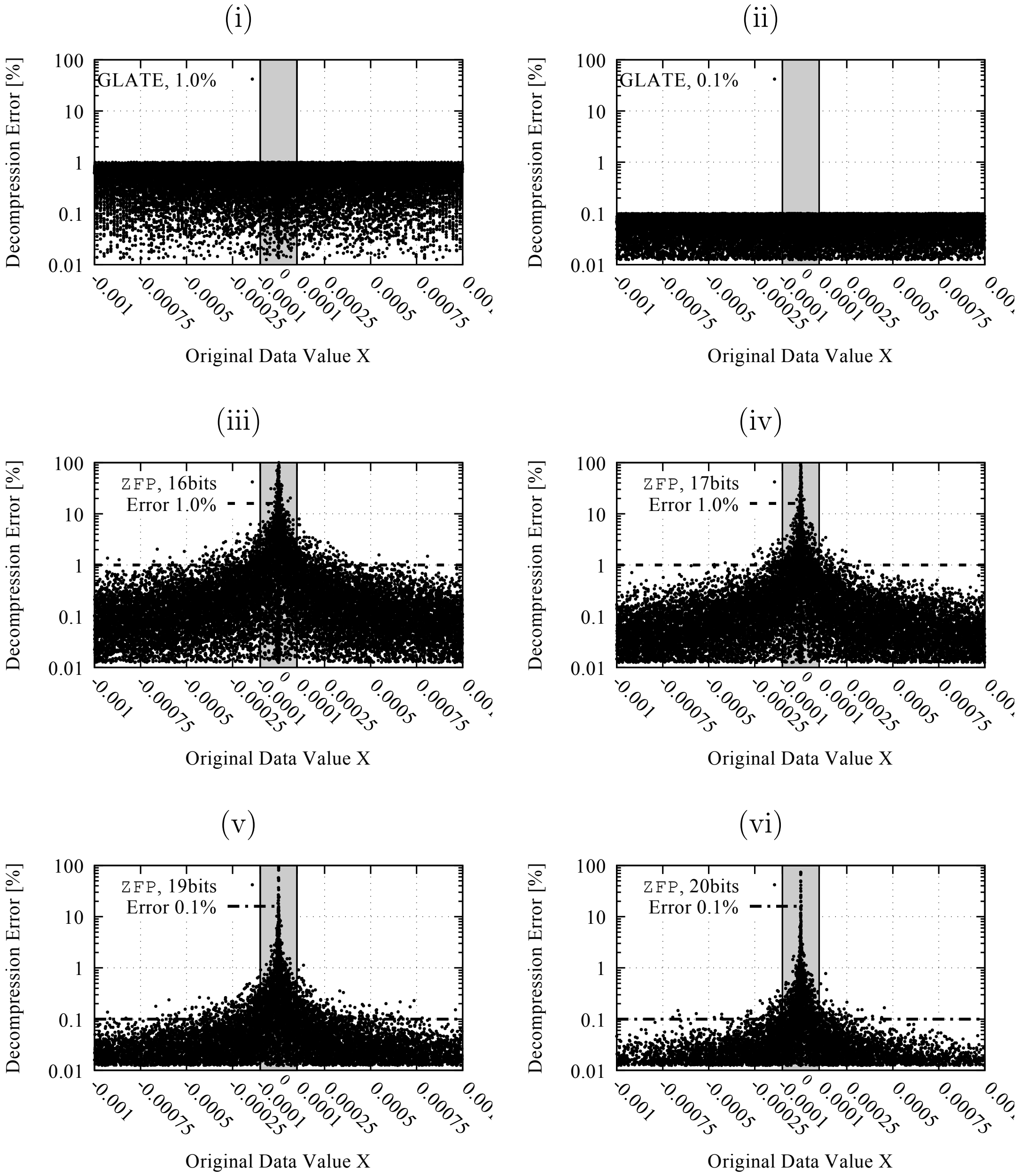


Figure 4.27: Scatter plot of relative error for t-GLATE and `ZFP` decompressed data. The error is plotted against original data value. (i)–(ii) Relative error of t-GLATE using $e_{\text{MAX}} = 1.00\,\%$, and $0.10\,\%$. (iii)–(vi) Relative error of `ZFP` using precision level 16, 17, and 19, 20. The compression rate and the error distributions for t-GLATE and `ZFP` are shown in Fig. 4.26. Compression rate for t-GLATE and `ZFP` is shown in Fig. 4.26 (TOP). The boxplots corresponding to the error scatter plots are shown in Fig. 4.26 (CENTER) denoted as (i)–(vi). The amount of values $|x| > 1\text{E}{-}4$ exceeding the error bound e_{MAX} is shown in Fig. 4.26 (BOTTOM), whereas the gray area corresponds to small values $|x| \leq 1\text{E}{-}4$ which are excluded from the statistics.

Linearization and Compression of Non-Uniform CFD Grids

Specifically, the development of t-GLATE has been tested through the compression of uniform voxel-based grids using linearization curves, which conserve the spatial coherence of neighboring grid cells. However, other grid types are widely used for CFD simulations e.g. unstructured grids have been used for the simulation of *Air Flow in Human Vocal Fold* [81], as shown in Fig. 4.28.

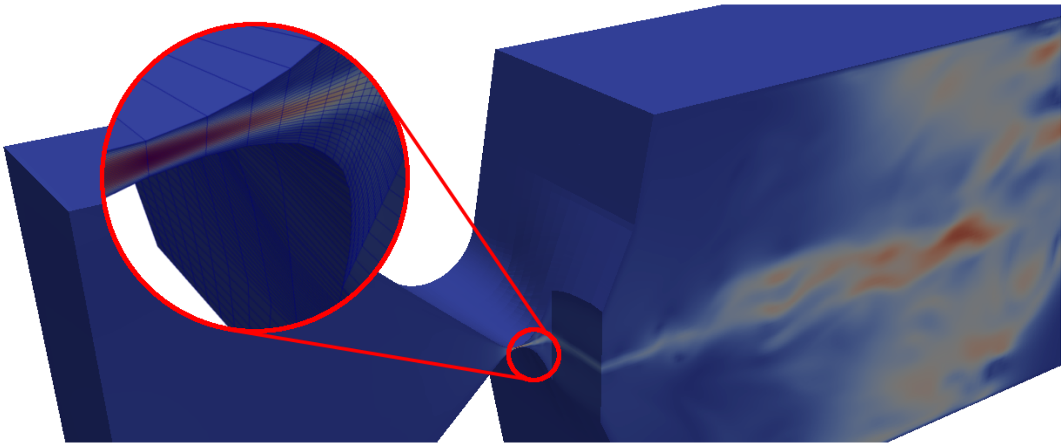


Figure 4.28: CFD simulation of moving vocal cords using unstructured grid with dynamic mesh points. The image shows the color plot of the y-velocity component. The simulation was conducted using the *OpenFOAM* CFD toolkit and uses a non-static mesh for the simulation of moving vocal cords during a 440 Hz vibration corresponding to the tone A.

The grid models the dynamic shape of the vocal cords during the creation of the tone A in human phonetics. In contrast to uniform grids, which represent regular domains, unstructured grids allow for modeling of complex domains, e.g. involving scales at different orders of magnitude, or even moving mesh parts, as shown in the red circle in Fig. 4.28.

Compression Rate for Unstructured Grid Data

Since any linearization scheme can be used in t-GLATE, the compression is not limited to uniform grids. By employing a different linearization scheme, an arbitrary grid type can be fed into t-GLATE, and fast lossy in-situ compression and temporal compression can be used. Referring to the unstructured grid shown in Fig. 4.28, the performance of t-GLATE and ZFP is evaluated on the y-velocity component of *Air Flow in Human Vocal Fold* [81].

Instead of employing a special linearization scheme for unstructured grids, the velocity field is simply fed into t-GLATE in the order of mesh points provided by the mesh topology, which often arranges mesh points close to each other and exhibits spatial coherence [36]. As ZFP is specifically crafted for the compression of three dimensional uniform grids, ZFP cannot be applied in the three dimensional compression mode when compressing the unstructured grid data set. Therefore, ZFP is applied in the linear compression mode resulting in a weaker compression performance.

Fig. 4.29 shows the compression rate of t-GLATE and ZFP for the compression of a linear data stream coming from the y-velocity component of *Air Flow in Human Vocal Fold* [81]. Similar to Fig. 4.26, t-GLATE uses error bound $e_{\text{MAX}} = 1.00, 0.50, 0.25, 0.10, 0.05, 0.02, 0.01\,\%$, and ZFP uses precision level 9, . . ., 20. As can be seen, in comparison to Fig. 4.26, the compression performance of t-GLATE and ZFP diverges w.r.t. comparable quality of decompressed data e.g. ZFP using precision level 13 already yields a poor compression rate of ~27.2 %, whereas t-GLATE compression rate is %11.1 %$_{32b}$ for $e_{\text{MAX}} = 1.00\,\%$. However, the resulting error for ZFP precision level 13 exceeds $e_{\text{MAX}} = 1.00\,\%$, whereas t-GLATE achieves the same compression rate of ~27.2 %$_{32b}$ for a maximum decompression error of $e_{\text{MAX}} = 0.01\,\%$. As elaborated for uniform grids, using ZFP with precision level 16 corresponds to approximately $e_{\text{MAX}} = 1.00\,\%$.

Concluding, due to the algorithmic structure of t-GLATE, which uses grid linearization as front end and lossless compression techniques as back end for scalar quantization, t-GLATE constitutes a powerful tool for lossy in-situ compression and temporal compression of CFD data sets composed of arbitrary grid types.

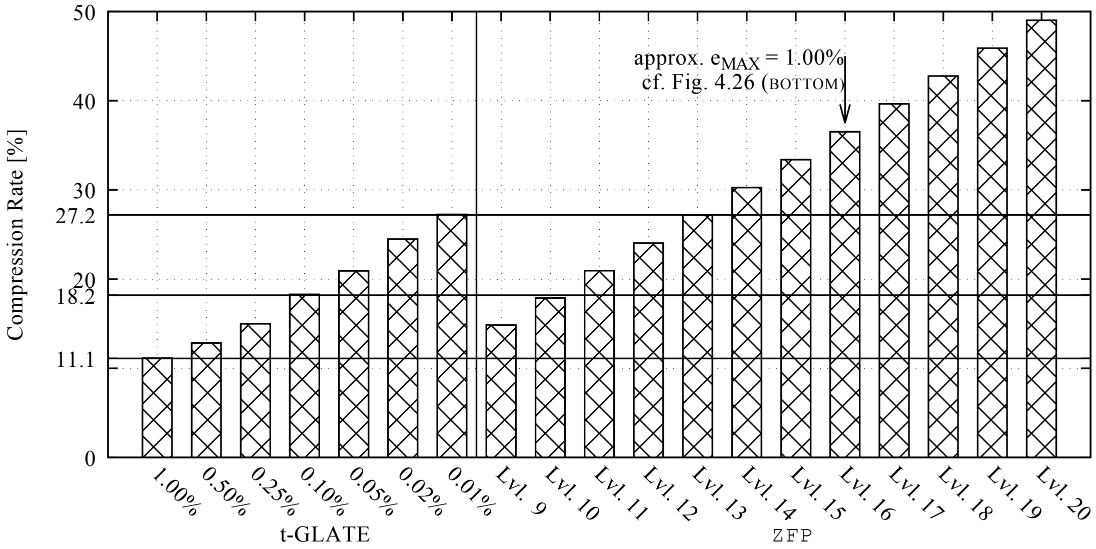


Figure 4.29: Overview of compression performance for t-GLATE and `ZFP` on unstructured grid data set. t-GLATE ($k_{\mathrm{D}} = 1$, $k_{\mathrm{MOD}} = 16$) uses error bound $e_{\mathrm{MAX}} = 1.00$, 0.50, 0.25, 0.10, 0.05, 0.02, 0.01 %, and `ZFP` uses precision level 9, …, 20. Data is compressed in the order of mesh points, as provided by the topology of the unstructured grid. Therefore, `ZFP` is operated in one dimensional mode. According to Fig. 4.26 (BOTTOM), `ZFP` with precision level 16 approximately restricts the error to ~1.00 %. However, for `ZFP` with level 16 the compression rate exceeds 30 $\%_{32\mathrm{b}}$, whereas t-GLATE for error bound $e_{\mathrm{MAX}} = 0.01, \ldots, 1.00\,\%$ achieves a compression rate of ~11.1–27.2 $\%_{32\mathrm{b}}$.

4.6 Summary

In this chapter, three schemes *t*-ISABELA, *t*-SBD and *t*-GLATE for lossy temporal in-situ compression of SFPD were developed, implemented and tested. The algorithms improve the compression rate of the lossy in-situ compressors ISABELA, SBD and GLATE presented in Chapter 3 and constitute new approaches for temporal compression based on the idea of the existing temporal compression procedure *d*-ISABELA [45]. The algorithms compress sequences of CFD data sets using an algorithm-specific difference encoding and achieve an improvement of the compression rate.

t-ISABELA Algorithm

For $e_{\mathrm{MAX}} = 1.00\,\%$, *t*-ISABELA achieves a compression rate of ~11.9–40 $\%_{32b}$, which is an improvement of ~10.5 $\%_{32b}$, as compared to *d*-ISABELA with ~22.4 $\%_{32b}$ minimum. *t*-ISABELA avoids fixed start-up costs, i.e. the storage of the sort order. At the end of each DF, the sort order and the B-spline are updated based on the decompressed values in order to update the prediction for the next data. In contrast to *d*-ISABELA, *t*-ISABELA decreases the *t*-ISABELA start-up costs by reusing the sort order and the B-spline from the previous time steps.

t-SBD Algorithm

The application of *t*-SBD results in a further improvement of the compression rate by ~4.5 $\%_{32b}$, as compared to *t*-ISABELA. For $e_{\mathrm{MAX}} = 1.00\,\%$, *t*-SBD achieves a compression rate of ~7.4–24.4 $\%_{32b}$. Particularly, *t*-SBD compresses the differences of LUT indices between time steps. Since SFPD changes smoothly, a difference encoding of LUT indices improves the compression rate. However, for high-resolution temporal CFD data, *t*-SBD shows compression artifacts because decompressed data randomly jumps within the error bound e_{MAX}, as shown in Fig. 4.23. In order to apply *t*-SBD, additional effort has to be spent on e.g. stabilization of LUT values between consecutive time steps.

t-GLATE Algorithm

The *t*-GLATE algorithm achieved the best temporal compression performance and the best trade-off between temporal resolution of decompressed data and the resulting compression rate. Furthermore, as t-GLATE operates on a global step function, no compression artifacts are present like for t-SBD. The application of t-GLATE to temporal sequences of CFD data sets yields a compression rate of ~7.9–17.4 %$_{32b}$ for $e_{\mathrm{MAX}} = 1.00\,\%$. Specifically, t-GLATE encodes the data differences between time steps on a global step function. Furthermore, the global step function is aligned with floating point exponents and allows for encoding of global differences while differentiating between smooth exponents and noisy signs/mantissas. The approach for difference encoding automatically restricts the maximum difference through fallback to an absolute encoding in the case differences become too large. Therefore, no noticeable decline of compression rate was introduced through the application of temporal compression.

Compression Rate Overview

Fig. 4.30 shows the compression rate of d-ISABELA, t-ISABELA, t-SBD and t-GLATE for data set `CUBC` using error bound $e_{\mathrm{MAX}} = 1.00,\ 0.50,\ 0.25,\ 0.12\,\%$. d-ISABELA and t-ISABELA use `zstd` level 6 for the compression of sort order differences and quantized errors, t-SBD uses `zstd` on level 2 and 6 for compression of differences between LUT indices, and t-GLATE uses `zstd` on level 2 and 6 for the compression of the exponents, and `tpfor` with block size 128 for signed/differential mLUT indices.

Originally developed for `DOUBLE` data [45], the d-ISABELA extension for temporal compression yields the worst compression rate on `FLOAT` data. For $e_{\mathrm{MAX}} = 1.00\,\%$, d-ISABELA is outperformed by t-ISABELA, t-SBD and t-GLATE, which achieve a compression rate between ~7.4–11.9 %$_{32b}$. However, as t-SBD cannot be used with high-resolution temporal data, t-GLATE yields the best trade-off between compression rate and decompression error. For $e_{\mathrm{MAX}} = 1.00,\ 0.50,\ 0.25,\ 0.12\,\%$, t-GLATE achieves a compression rate betwen ~7.9–11.9 %$_{32b}$.

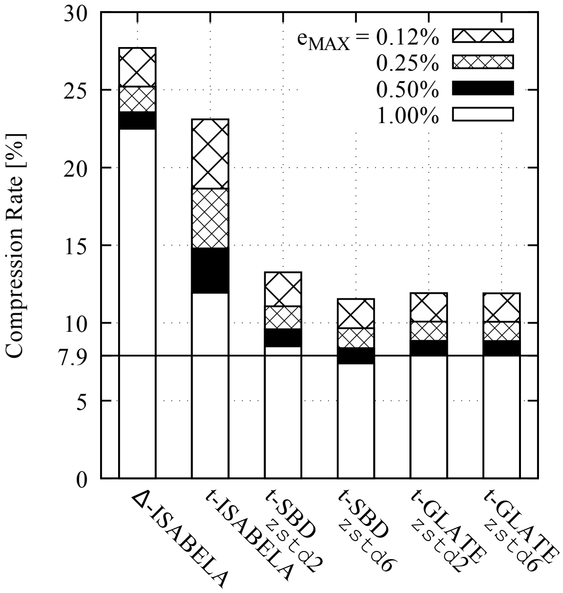


Figure 4.30: Overview of compression rate for lossy temporal compression of data set `CUBC`. Compression is carried out using error bound $e_{\mathrm{MAX}} = 1.00, 0.50, 0.25, 0.12\,\%$. The data set is compressed using d-ISABELA and t-ISABELA with `zstd` level 2, t-SBD with `zstd` level 2 and 6, and t-GLATE with `zstd` level 2 and 6. For GLATE, signed/differential MLUT indices are compressed with `tpfor` block size 128 always, and `zstd` is used to compress the stream of exponents.

Compression Run-Time Overview

Fig. 4.31 (*a*)–(*b*) shows the run-time for compression and decompression using the algorithms *d*-ISABELA, *t*-ISABELA, *t*-SBD and *t*-GLATE for the compression of the data set `CUBC`. While *t*-ISABELA yields an improvement of compression speed compared to *d*-ISABELA, *t*-SBD and *t*-GLATE are the faster algorithms. Where *t*-ISABELA uses a rather complicated procedure, which requires sorting the decompressed data and updating the B-spline. *t*-SBD and *t*-GLATE in contrast employ simple transformations of the data and use carefully chosen lossless compressors for high compression speed.

As can be seen in Fig. 4.31 (*a*), *t*-SBD with `zstd` on level 2 and 6 take much longer than *t*-GLATE. As the data complexity of LUT indices is much higher than the complexity of *t*-GLATE exponents, `zstd` operates much faster in *t*-GLATE than in *t*-SBD. Further, noisy MLUT indices in *t*-GLATE are compressed using the high-speed compressor `tpfor`. During decompression, *t*-GLATE and *t*-SBD yield a similar speed during decompression, as can be seen in Fig. 4.31 (*b*).

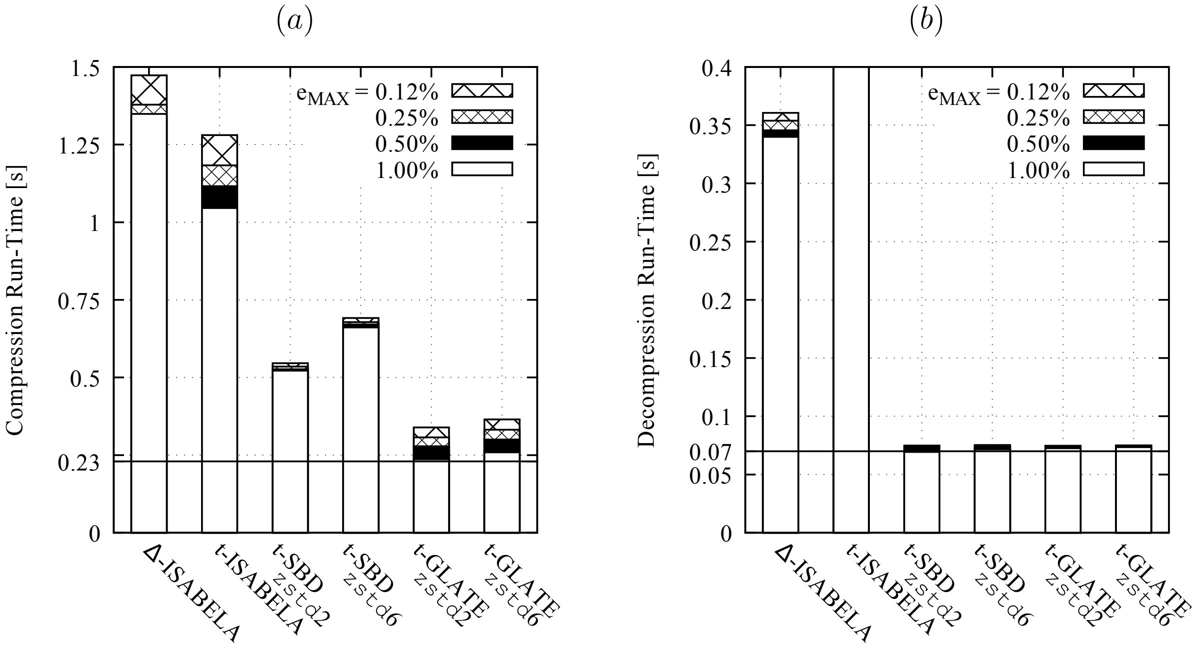


Figure 4.31: Overview of compression speed for lossy temporal compression of data set `CUBC`. Compression is carried out using error bound $e_{\mathrm{MAX}} = 1.00$, 0.50, 0.25, 0.12 %, showing (a) compression run-time and (b) decompression run-time. The data set is compressed using d-ISABELA and t-ISABELA with `zstd` level 2, t-SBD with `zstd` level 2 and 6, and t-GLATE with `zstd` level 2 and 6. For GLATE, signed/differential mLUT indices are compressed with `tpfor` block size 128 always, and `zstd` is used to compress the stream of exponents.

Closing Statement

Summarizing, temporal compression improves the performance of lossy in-situ compression for temporal sequences of CFD data sets. In particular, *t*-ISABELA, *t*-SBD and *t*-GLATE deliver improved compression performance, while providing similar speed as the non-temporal implementation described in Chapter 3. Whereas, *d*-ISABELA and *t*-ISABELA yield moderate improvement of the compression rate, the *t*-SBD and *t*-GLATE algorithm offer large improvements in compression rate and compression speed. However, as the *t*-SBD algorithm showed compression artifacts for high-resolution temporal data, the *t*-GLATE algorithm constitutes the best trade-off between compression rate, compression speed and error of decompressed data. Further, by using `zstd` and `tpfor` as back end for lossless compression, *t*-GLATE maintains a very good compression rate at high compression speeds. For $e_{\mathrm{MAX}} = 1.00, \ldots, 0.01\,\%$, *t*-GLATE achieves a compression rate between ~7.9–11.9 $\%_{\text{32b}}$. Particularly, for $e_{\mathrm{MAX}} = 1.00\,\%$, *t*-GLATE encodes `FLOAT` values with an average of ~2.56 bit per value.

Concluding, *t*-GLATE lossy temporal in-situ compression decreases the bandwidth requirements for transmission of sequences of CFD data sets and helps to reduce the I/O bottleneck in the scientific workflow, which renders *t*-GLATE a practical tool for reduction of temporal sequences of SFPD inside CFD simulations.

5 Application of In-Situ Compression in a Computational Fluid Dynamics Simulation

In this chapter, lossy in-situ compression is evaluated inside a CFD simulation in a HPC environment aiming towards the improvement of the *Time-to-Analysis*. First, lossy in-situ compression is motivated within the context of a HPC environment with powerful compute nodes and a parallel Lustre$^{(R)}$ file system. Second, a CFD simulation for metal melt filtration inside a complex filter structure is described briefly. Third, different data layouts for storing uncompressed data and compressed data are presented. Fourth, the compression performance of GLATE and t-GLATE, as well as, I/O times for writing compressed full/low-resolution data are evaluated inside a parallel CFD simulation and compared to handling uncompressed data. Fifth, based on the evaluation results, the in-situ applicability of t-GLATE and the improvement of the *Time-to-Analysis* in the scientific workflow are discussed.

5.1 Motivation for Lossy In-Situ Compression

With scientific computations and simulations being able to generate virtually any amount of data, the I/O bottleneck is more than evident. Therefore, lossy in-situ compression for large-scale CFD simulations is researched because it helps to scale up applications by reducing the I/O bottleneck. Although there are very powerful network file systems, which can be scaled up expensively by hardware upgrades, the I/O bottleneck dominates for data-intensive tasks [4]. Since FLOPS develop reasonably faster than the speed and capacity of storage technology, moving data between main memory and storage likely will be the limiting factor for applications of the future [50].

Data-intenstive HPC Applications

In the present CFD domain, lossy GLATE compression was shown to achieve a data reduction by ~4–5× on offline data sets, while offering a trade-off between precision and compression rate, as described in Section 3.4.3. For temporal compression, *t*-GLATE encodes differences and achieves a data reduction by ~6–12×, while outperforming the other approaches ISABELA, SBD, *d*-ISABELA, *t*-ISABELA and *t*-SBD, which have been developed and tested in Chapters 3 and 4.

Data compression achieves a decrease in *Time-to-Analysis*, if *"simulation, compression, storage, transmission & decompression"* using *t*-GLATE works faster than the *"simulation, storage & transmission"* of uncompressed `FLOAT` data. However, in parallel storage systems, the sole reduction of the memory footprint does not imply a direct decrease of write time, as it is the case for moving data into e.g. local hard disks. The optimization of the I/O performance in parallel file systems is a complicated task and often involves multiple strategies, e.g. distribution patterns of processes among compute nodes, coordinated I/O procedures, data aggregation, temporary local storage [43, 51, 71, 78, 79].

Reduction of Time-to-Analysis in Scientific Workflow

As illustrated in Fig. 5.1, the transfer of full-size data slows down the scientific workflow. Especially if data is collected into temporary scratch storage, data movement is inevitable and can take a long time, e.g. results usually are moved away from scratch storage using a slower network connection.

Traditional approaches, which store the full uncompressed solutions have outlived their feasibility due to the I/O limitations in HPC systems [86]. Particularly, *t*-GLATE tackles the storage problem by lossy compression of full-resolution data, tackles data loss with error-bounded decompression, and offers a reduction of storage cost by ~4–12×. In practice, collective I/O takes a considerable amount of the wall-clock time of data-intensive HPC applications. Therefore, early data reduction during the simulation while using "cheap" CPU time, allows to circumvent the I/O bottleneck by moving data sets with strongly decreased memory footprint during analysis, i.e. improving the *Time-to-Analysis* for analysis of full-resolution data.

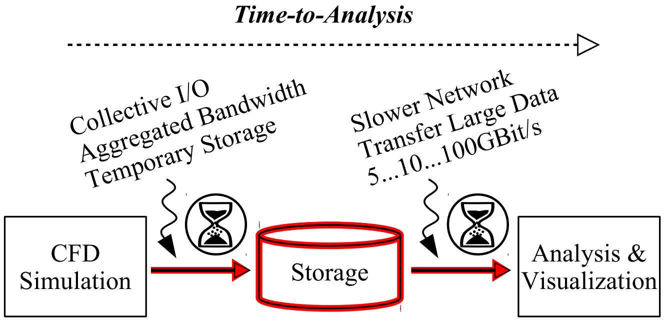


Figure 5.1: Time-to-Analysis of data-intensive HPC applications. Collective I/O in modern HPC systems offers large bandwidth but optimization of I/O performance is non-trivial. Data transfer over slow networks into analysis and visualization workstations directly benefits from very high data reduction by temporal in-situ compression.

Image is adapted from [44].

5.2 Simulation of Metal Melt Flow in a Complex Filter Structure

In the CRC920, numerical models based on the LBM are conducted for the simulation of the filtration process inside a whole filter structure for analysis of flow field and particle dynamics, as well as, for the estimation of the filtration efficiency [85].

LBM for CFD Simulations

The LBM is a powerful method for the simulation of fluid flow using distribution functions of particles undergoing so-called streaming and collision steps on a discrete lattice [5, 90]. The amount of different directions for streaming determines the quality of the discretization. Typical LBM implementations use 19 directions c_1, c_2, c_3, $\ldots$, c_{19}, as shown in Fig. 5.2. Local pressure and velocity are given by the moments of the particle distribution function. For the numerical implementation, the physical domain is discretized into a uniform grid of voxel cells resulting in a staircase approximation of the geometry. The LBM allows for efficient parallelization in HPC systems [85], and thus allows the simulation of filters with complex geometries using high-resolution uniform grids for discretization.

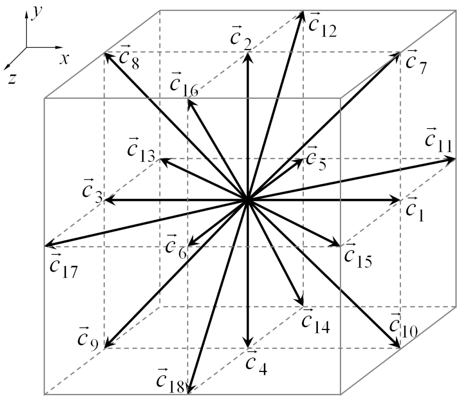


Figure 5.2: Streaming directions in discrete lattice in the LBM. The so-called D3Q19 scheme has 19 streaming directions c_1, c_2, c_3, …, c_{19}. Increasing the number of directions increases the quality of the discretization and the complexity of the LBM.

Simulation of Aluminum inside a Complex Filter Structure

Using the LBM as described above, the transient flow of liquid aluminum inside a complex filter structure is simulated assuming a melt temperature of 730 °C and a superficial velocity of 8.72 cm/s. Fig. 5.3 (*a*) shows, the dicretized porous foam-like filter, which is generated algorithmically and ensures properties of filters from the real-world production processes [2]. Using algorithmically generated filters allows for the prescription of periodic boundary condition, and the control of the porosity, which amounts to 90 % in the present case. The simulation has a Reynolds number of $Re = 90$ w.r.t. to the strut width and a temporal step width of $\Delta t = 9.13\,\mu\text{s}$. The filter geometry comprises 216 pores inside a physical domain of $17.7 \times 17.7 \times 17.7$ mm, which is discretized using $512 \times 512 \times 512 = 134{,}217{,}728$ voxels with a spatial resolution of 34.5 µm in each direction. In order to offer a preview and quick browsing of the data set $\mathcal{D}$ during post-processing, for each variable an additional low-resolution grid is computed. The low-resolution grid consists of $^1/_8$ of the resolution, i.e. $256 \times 256 \times 256$ voxels, as shown in Fig. 5.3 (*b*)–(*c*).

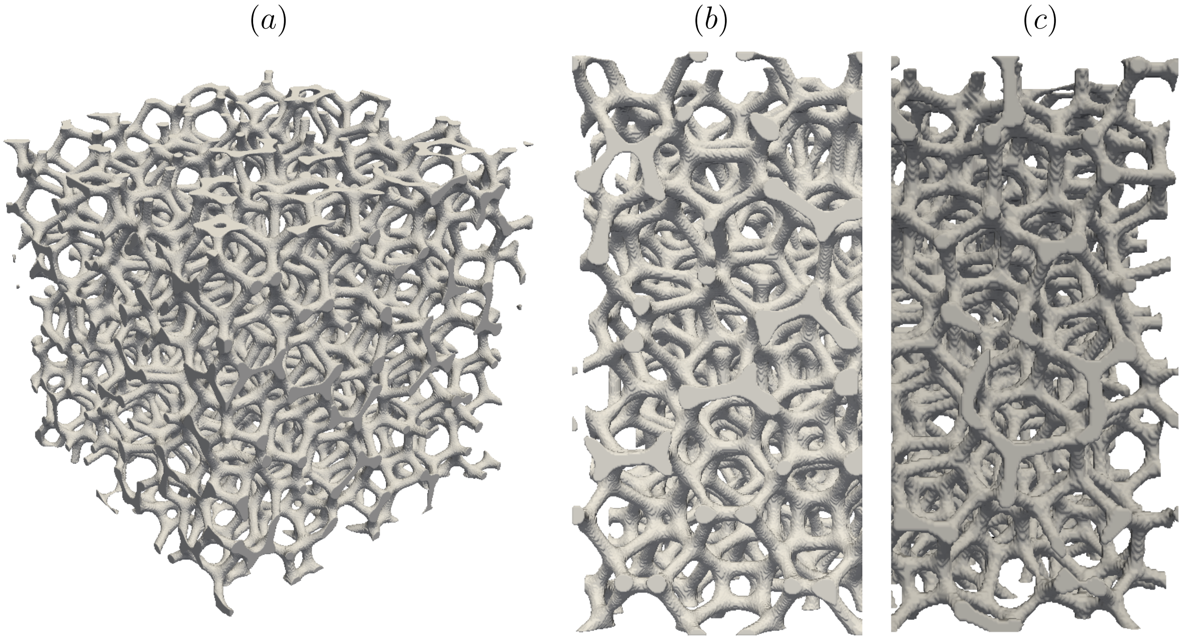


Figure 5.3: Discretization of complex filter structure with 216 pores in a uniform grid. (a) The filter structure is discretized by $512 \times 512 \times 512$ voxels. The geometry is periodic and can be stacked. The filter was designed according to properties of real-world production of ceramic filters and has a porosity of 90 %. (*b*)–(*c*) Comparison of full-resolution and low-resolution grid. Low-resolution grid consists of $256 \times 256 \times 256$ voxels and is computed during the CFD simulation in order to provide a quick preview to the data set.

Production of Temporal Sequences of CFD Data

During the LBM simulation a typical CFD data set $\mathcal{D}$ is generated. The data set contains the melt flow field (u, v, w), as well as, two flow field properties: the velocity magnitude $M = \sqrt{u^2 + v^2 + w^2}$ describing the speed of the flow, and the Q-criterion Q[1] describing vortex characteristics of the flowfield. The variables u, v, w, M, Q are calculated, compressed and written to the storage inside parallel CFD simulation processes. The velocity field (u, v, w) allows for analysis of the melt flow in a post-hoc fashion e.g. using stream lines, path lines, tracer particles, etc. As direct access to the velocity magnitudes and vortex characteristics is convenient for exploration of the flow field, the variables M and Q are computed during the simulation and written out in compressed form. Further, the low-resolution grid for each variable is computed, compressed and stored in all parallel processes.

The space requirements of a high-resolution simulation grid imposed by using the LBM can be reduced dramatically using lossy in-situ temporal compression. For example, the grid containing the complex filter structure consisting of 512^3 voxels requires 512 MB for each 32 bit `FLOAT` variable to be stored. Recording five variables requires 2.5 GB per time step, 100 GB for a sequence of 40 time steps, and 1 TB for 4000 time steps to be stored in the file system. The direct visualization of flow dynamics from long temporal sequences of CFD data sets is a very data-intensive task. Using lossy in-situ temporal compression, the amount of data to be written out can be reduced by a factor of ~4–12×, and even by a factor of ~32× ($\simeq {}^1/_8 \times 25\,\%$) for compressed low-resolution data.

5.3 Storage of In-Situ Compressed Data

This section summarizes strategies for data writing using collective I/O in HPC environments. Different storage procedures for uncompressed full-size data and compressed full/low-resolution data are presented and implemented using *MPI Input/Ouput* (MPI-I/O). At the end of this chapter, the storage procedures are used for the evaluation of I/O times for merging compressed data from 512 parallel processes into large files.

---

[1] $Q = \frac{1}{2}(||\Omega||^2 - ||S||^2)$, S and Ω are the symmetric and the anti-symmetric components of the velocity gradient tensor [19]. They are also known as the strain rate tensor and the rotation tensor respectively.

5.3.1 Collective I/O in parallel HPC Environments

As described in Chapters 3 and 4, error-bounded lossy in-situ compression offers fast and efficient data reduction for scientific data sets. However, in parallel environments the grid is partitioned across numerous processes, which execute a collective algorithm and typically use MPI for message passing and synchronization [24]. MPI is extensively developed by the research community and the industry and is able to run on a large variety of HPC systems. Leveraging the scalable algorithms of MPI, e.g. broadcast, scatter, gather, MPI allows for up-scaling of algorithms and cluster systems to large numbers of nodes and cores [51]. Further, MPI allows for collective I/O using the MPI-I/O interface [16, 24].

MPI-I/O for Collective Storage Procedures

While MPI was developed for data transport between processes, the MPI-I/O interface [16, 24] was developed for coordinating and improving the performance of I/O in parallel environments. Collective I/O is needed, as standard I/O interfaces do not provide features for parallel I/O, and uncoordinated I/O from many parallel processes introduces a lot of overhead into the file system [79]. Instead of opening a seperate file for each MPI process, data organization and collection is accomplished using special routines, which collect metadata about the upcoming write operation and allow for the optimization of the data flow between compute nodes and the nodes of the storage system [78].

MPI-I/O for Writing Shared Files

Different patterns for parallel I/O have emerged during the development of MPI-I/O, e.g. two-phase I/O [78] and data sieving [79]. Two-phase I/O coordinates many parallel MPI processes, which access contiguous blocks of one shared file. Data sieving reuses MPI data types for the definition of non-contiguous collective write operations, e.g. for writing out uniform grids in a global column-major data layout into one large file. In contrast to writing one file per process, which usually demonstrates poor scalability for the management of file system metadata [51], using MPI-I/O yields better performance and constitutes an important component for optimized I/O in HPC systems [51].

```
int rank, size, status;
FILE* file_handle = NULL;
char* file_path = ”/storage/time001”;
MPI_Init(&argc, &argv); // initialize mpi
MPI_Comm_rank(MPI_COMM_WORLD, &rank); // get process id
MPI_Comm_size(MPI_COMM_WORLD, &size); // get num procs
MPI_File_open(MPI_COMM_WORLD, file_path, // open file
  MPI_MODE_WRONLY, MPI_INFO_NULL, &file_handle);
/*** Execute Collective Write Operations Here ***/
MPI_File_close(&file_handle); // close file
```

Algorithm 5.1: Opening a shared file using MPI-I/O. The file is opened using the `MPI_File_Open()` statement. All MPI processes are involved in the collective I/O operations according to the MPI communicator `MPI_COMM_WORLD`. `MPI_File_Close()` closes the file. The example illustrates the MPI-I/O library calls.

Implementation of Collective Write into Shared Files

For writing data which is fragmented across many parallel processes into a shared file, MPI-I/O offers methods for the coordination of write operations, i.e. `MPI_File_Open()`, `MPI_File_write_at_all()` and `MPI_File_Close()`. Specifically, as described in Alg. 5.1, the `MPI_File_Open()` statement opens a shared file and communicates the active set of MPI processes participating in the collective write operation. The set is specified using a MPI communicator, e.g. `MPI_COMM_WORLD` for all processes writing into the file. Accordingly `MPI_File_Close()` is used for closing the shared file. Often, the underlying storage system, e.g. PanasasFS, Lustre, GPFS, GFS, provides a specialized implementation for the MPI-I/O interface and offers methods for optimized distribution of data from/to parallel MPI processes.

5.3.2 Column-Major Storage Layout for Uncompressed Grid Data

In the simulation setup, as described in Section 5.2, a uniform LBM grid composed of $512 \times 512 \times 512$ voxels is produced in each time step of the CFD simulation. The grid is partitioned into 512 MPI processes, each containing one subgrid of size $64 \times 64 \times 64 =$ 262,144 voxels. After one time step has been computed, the data is written into the file system before the simulation moves on to the next time step. If the uniform LBM grid is written out without compression, it typically is stored in a global column-major data layout in one large file. However, since the subgrids are distributed among many MPI processes, the simple concatenation of raw memory blocks from MPI processes does not yield the desired data layout in the shared file.

Column-Major Global Grid Topology

Fig. 5.4 (*a*) shows a two dimensional data grid composed of 16 × 16 cells, which is partitioned among four processes each containing 4 × 4 cells. As can be seen, the concatenation of the raw memory blocks does not yield the column major ordering, as shown in Fig. 5.4 (*b*). Instead, following the column-major ordering produces the data linearization as shown in Fig. 5.4 (*c*), where local data chunks from processes 1&2 and 3&4 are interlaced.

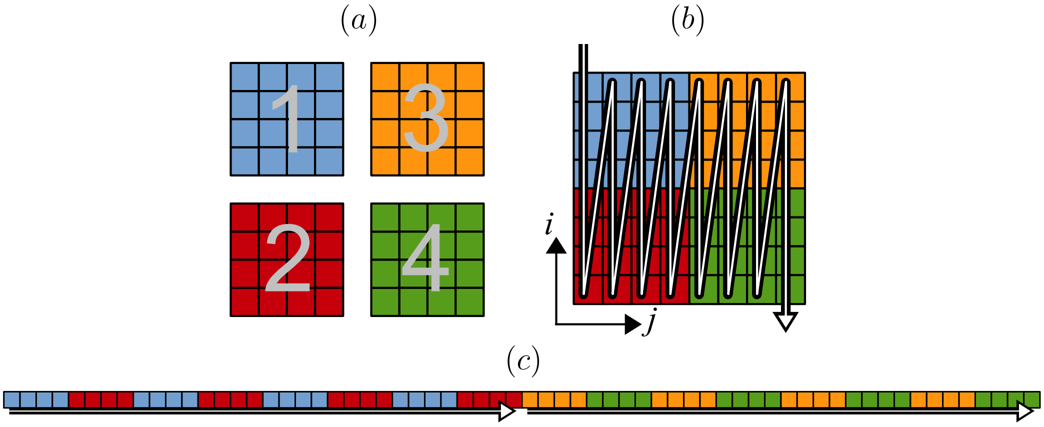


Figure 5.4: Global column-major data layout. (a) Two dimensional grid consisting of 16 × 16 cells distributed among four MPI processes. (b) Column-major linearization of global grid. (c) data layout in shared file according to the column-major ordering shown in (b). As the linearization crosses the boundaries of the subgrids, data chunks of MPI processes are interlaced.

MPI-I/O File Views for Global Uniform Grids

The column-major storage procedure for uniform grids is conveniently implemented in the MPI-I/O interface itself. In particular, MPI data types are leveraged for the representation of data distribution patterns describing the mapping of process-local data into global positions of a shared file. Using MPI data types, local contiguous data is copied into a shared file and distributed non-contiguously using strides [79]. Thereby, the MPI data types describe the striding pattern and are called *File Views*.

File views allow each process to configure its own striding pattern according to the interlacing for the global column-major data layout, as shown in Fig. 5.5 (1)–(4). Bytes that are written by the local process contain data, and bytes that are written by other processes are skipped. Therefore, file views constitute a powerful tool for the representation of contiguous and non-contiguous access patterns, which further allows for integrated optimizations in MPI-I/O implementations.

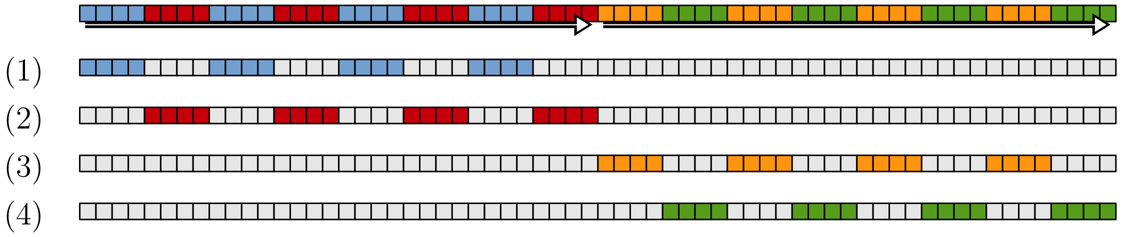


Figure 5.5: File views for global column-major data layout. (TOP) non-contiguous interlaced striding pattern resulting from the global column-major data layout according to Fig. 5.4 (*c*). (1)–(4) file view as generated on MPI process 1–4. The partition of the global grid into subgrids for MPI processes is shown in Fig. 5.4 (*b*).

Specifically, the file views are used for carrying out an optimized two-phase I/O procedure available in popular MPI-I/O implementations [78]. Since I/O cost is much higher than communication cost for message passing, typically a subset of the MPI processes collectc data chunks into larger contiguous data blocks and then performc a few large I/O requests with increased performance instead of many concurrent small writes which would reduce the performance. The negotiation, the communication and the buffering of data is handled by the MPI-I/O implementation in the background.

Implementation of Collective Column-Major Storage Procedure

File views allow for the construction of arbitrary striding patterns, hence they constitute a powerful tool for merging fragmented data. However, for the write of a uniform grid in the column-major data layout, the MPI *Subarray* data type already supplies a conforming MPI file view. As shown in Alg. 5.2, the `MPI_Type_create_subarray()` and the `MPI_File_set_view()` statements are used for the configuration and activation of the

```
int size_global[3] = {512, 512, 512}; // global grid size
int size_local [3] = { 64,  64,  64}; // size of local subgrid
int subgrid_origin[3] = {...};        // origin of subgrid
float data_local[64][64][64];         // computed by simulation
MPI_Datatype file_view;               // data type handle
MPI_Type_create_subarray(3, size_global, size_local, // create view
  subgrid_origin, MPI_ORDER_C, MPI_FLOAT, &file_view);
MPI_File_set_view(file_handle, 0, MPI_FLOAT, file_view); // set view
MPI_File_write_all(file_handle, data_local, size_local, // write data
  MPI_FLOAT, &status);
```

Algorithm 5.2: Collective write of uniform grid with global column-major data layout using MPI data types and MPI file views. The data is written according to the column-major linearization used for the uniform grid in the LBM. The `MPI_Type_create_subarray()` and `MPI_File_set_view()` statements create and setup file views, and `MPI_File_write_all()` initiates the collective write operation. The example illustrates the MPI-I/O library calls.

MPI file view, and the `MPI_File_write_all()` statement is used for the execution of the write operation. For each MPI process, the call to `MPI_Type_create_subarray()` results in a different view for distribution of process-local data in the global grid depending on the parameters `size_local` and `subgrid_origin`. The write operation creates one file per time step per variable and results in the following file structure:

```
/scratch/archive/time001_variable001
/scratch/archive/time001_variable002
/scratch/archive/...
/scratch/archive/time002_variable001
/scratch/archive/...
```

5.3.3 Sequential Storage Layout for Compressed Grid Data

The application of lossy in-situ compression creates binary blobs inside each node of a parallel CFD simulation. The topology of the grid is no longer available after compression. Additionally, the compression yields different data size on each subgrid, and each process writes a data block of arbitrary size into the shared file. The data blocks constitute spatial regions in the global grid, which can be decompressed independently.

Global Topology for Compressed Sequential Data

Fig. 5.6 (*a*) shows the partition of the global grid consisting of $512 \times 512 \times 512$ voxels into 512 subgrids composed of $64 \times 64 \times 64$ voxels each. Each MPI process receives a unique id, the so-called *rank* $r = 1, 2, 3, \ldots, 512$. As each MPI process compresses its data independently, the collective write operations result in a global sequence concatenating the compressed blocks from 512 single processes.

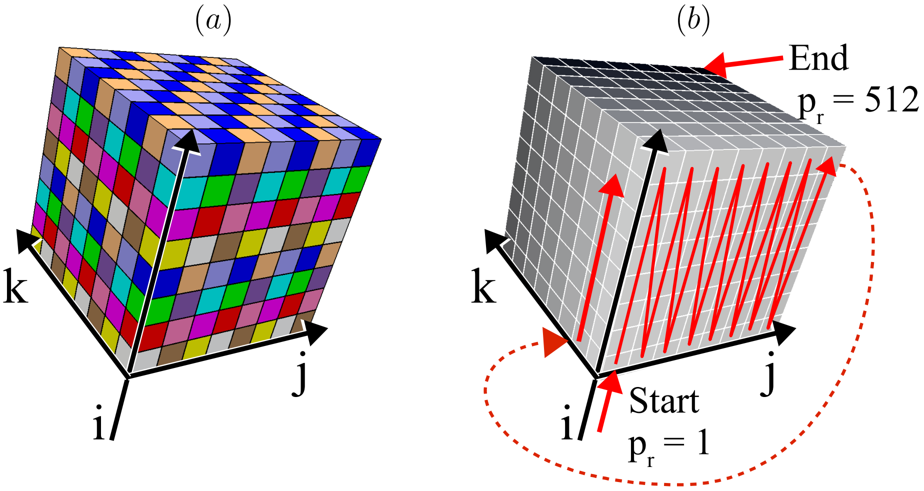


Figure 5.6: Partition of global simulation grid used for the LBM. (a) A global grid composed of $512 \times 512 \times 512 = 134{,}217{,}728$ voxels is partitioned into 512 subgrids of size $64 \times 64 \times 64 = 262{,}144$ voxels each. (b) Global column-major ordering of subgrids L_G used for writing the linear sequence of compressed blocks into the shared file.

In general, MPI process ranks r are assigned in an internal manner and cannot be assumed to order the data in a beneficial way, e.g. conserving locality of nearby compressed blocks. Hence, a global column-major ordering L_G is employed for the linearization of compressed binary blobs into the shared file. Each MPI process compresses its local subgrid and determines the starting position for writing the compressed format according to L_G. Using the global column-major ordering of subgrids, each MPI process r receives a unique position $p_r = 1, 2, 3, \ldots, 512$ in the sequence of subgrids, as shown in Fig. 5.6 (*b*).

MPI-I/O File Offsets for Writing Global Sequence

After process with rank r at position p_r has compressed its subgrid into a memory block of size A_{p_r} bytes, the tuples $(p_r, A_{p_r})_r$ are distributed among all MPI processes using the `MPI_Allgather()` statement. During the distribution, each MPI process constructs a matrix which contains all positions p_r and all sizes A_{p_r} of the compressed subgrids for $r = 1, 2, 3, \ldots, 512$. After the distribution, the matrix has the following structure on each process:

$$\begin{pmatrix} p_1 & p_2 & p_3 & p_4 & \ldots & p_{512} \\ A_{p_1} & A_{p_2} & A_{p_3} & A_{p_4} & \ldots & A_{p_{512}} \end{pmatrix} \tag{5.1}$$

Since `MPI_Allgather()` arranges the data in the order of the MPI rank r, the columns of the matrix are sorted w.r.t. the first row. The first row contains the position p_r of the subgrid r in the global ordering L_G of subgrids. Consequently, the elements of the second row in the matrix are rearranged according to the order of subgrids in L_G. After the sorting is finished, the second row of the matrix contains the sequence A_1, A_2, A_3, $\ldots$, A_{512} which now describes the size of the compressed data blocks in the final order on disk, as shown in Fig. 5.7. The sorted matrix has the following structure:

$$\begin{pmatrix} 1 & 2 & 3 & 4 & \ldots & 512 \\ A_1 & A_2 & A_3 & A_4 & \ldots & A_{512} \end{pmatrix} \tag{5.2}$$

Lastly, before the data can be written, each MPI process r needs to compute its starting offset B_{p_r} in the shared file according to the sorted data sizes A_1, A_2, A_3, $\ldots$, A_{512}. The offsets B_{p_r} are calculated as partial sums using the following equation:

$$B_{p_r} = \sum_{i=1}^{p_r} A_i \tag{5.3}$$

As further shown in Fig. 5.7, the partial sums B_1, B_2, B_3, $\ldots$, B_{512} directly describe the file offsets required for placing the compressed data in the global sequence of subgrids while taking into account the varying data size resulting from the compression. It has to be noted that in order to extract single compression blocks from the file, the offsets B_1, B_2, B_3, $\ldots$, B_{512} have to be stored in the header of the shared file. Therefore, the process with rank r at position $p_r = 1$ writes a header at the very beginning of the file before writing its own compressed data.

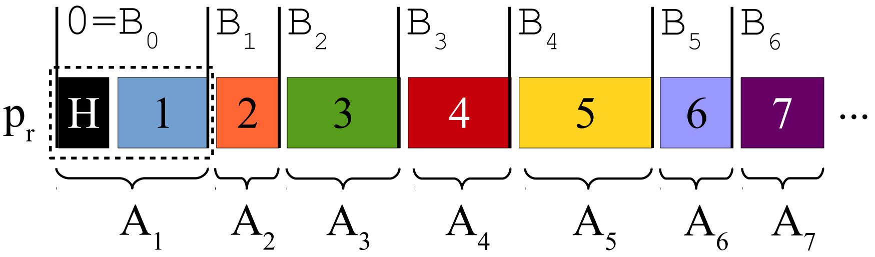


Figure 5.7: Global sequential layout for compressed data on disk. Each MPI process with rank r writes one compressed data block into a shared file. The data size A_{p_r} of compressed subgrids is numbered according to the global column-major ordering L_G ($p_r = 1, 2, 3, \ldots, 512$), as shown in Fig. 5.6 (*b*). The partial sums B_{p_r} of the data size A_{p_r} correspond to the starting offsets for each process during the collective write operation. As denoted by the dashed rectangle, the process at position $p_r = 1$ writes a header containing the offsets $B_1, B_2, B_3, \ldots, B_{512}$.

Implementation of Collective Sequential Storage Procedure

Given the file offsets $B_1, B_2, B_3, \ldots, B_{512}$, where MPI processes r at position p_r start to write their data, the `MPI_File_write_at_all()` statement is used for the collective write of the global sequential data layout. As shown in Alg. 5.3, each MPI process invokes the statement with a different offset B_{p_r}, as well as, a different data size A_{p_r}. The write operation creates one file per time step per variable and results in the following file structure:

```
/scratch/archive/time001_variable001
/scratch/archive/time001_variable002
/scratch/archive/...
/scratch/archive/time002_variable001
/scratch/archive/...
```

```
int rank, size, status;
FILE* file_handle = NULL;
char* file_path = "/archive/time1_variable1";
char* buffer = new char[MAX_BUFFER_SIZE];
// initialize mpi
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank); // get process id
MPI_Comm_size(MPI_COMM_WORLD, &size); // get num procs
// compute $p_r$, $A_{p_r}$, gather $p_*$, $A_*$
// compress data into buffer[] and merge header if $p_r$ = 1
// sort tuples ($p_*$, $A_*$), compute offsets $B_*$
MPI_File_open(MPI_COMM_WORLD, file_path, // open file
  MPI_MODE_WRONLY, MPI_INFO_NULL, &file_handle);
MPI_File_write_at_all(file_handle, $b_{p_r}$, // write data
  &buffer[0], $a_{p_r}$, MPI_BYTE, &status);
MPI_File_close(&file_handle); // close file
MPI_Finalize(); // finalize mpi
```

Algorithm 5.3: Collective write of length-varying compressed data using MPI-I/O. Compressed blocks are linearized into a sequence and written using `MPI_File_Open()`, `MPI_File_write_at_all()` and `MPI_File_Close()` statements. The example illustrates the MPI-I/O library calls.

5.3.4 Storage Layout for Compressed Low-Resolution Data

The sequential data layout described in the previous section offers individual access to spatial regions, which have been compressed independently. Therefore, data access based on the original grid topology requires decompression, and only decompression of single full-resolution regions is possible.

Low-resolution Data for Visualization

As direct rendering of low-resolution preview data is not possible using compressed full-resolution data, the low-resolution data is computed additionally and stored separately. Fig. 5.8 (a) shows a visualization of a melt data set at two different levels of resolution, i.e. at full resolution with $64 \times 64 \times 64$ voxels per subgrid, and at low resolution with $32 \times 32 \times 32$ voxels per subgrid. As can be seen in Fig. 5.8 (b), low-resolution data is sufficient for e.g. overview browsing, the visualization of trends, content delivery for mobile devices/web browsers etc. [53, 56]. Many post-processing scenarios which involve visual analysis and exploratory visualization can use low-resolution data for e.g. offering preview and quick browsing, or creating a context next to full-resolution data. As compressed low-resolution data exhibits an extremely small memory footprint, it can be loaded quickly during interactive applications, or it can be rendered quickly during iterative adjustment of visualization parameters.

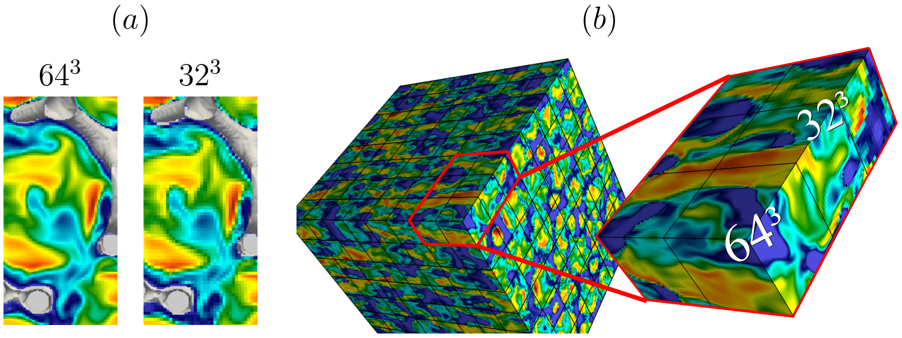


Figure 5.8: Visualization of melt data set in two different levels of resolution. (a),(LEFT) full-resolution data with $64^3 = 262{,}144$ voxels per subgrid. (a),(RIGHT) low-resolution data with $32^3 = 32{,}768$ voxels per subgrid. (b) visualization using full-resolution data for regions of interest and low-resolution data for context.

Computation of Low-Resolution Data

Each MPI process computes the low-resolution data by using an octree averaging operation. An octree is a tree data structure for the spatial organization of three dimensional uniform grid data, where each node has eight children. Nodes partition the space into octants, and leaves correspond to voxels [53]. However, instead of spanning a full octree over the whole data set, each MPI process executes the averaging operation on its own local subgrid for only one iteration, as shown in Fig. 5.9.

One uncompressed subgrid of size $32 \times 32 \times 32$ requires $128\,\mathrm{kB} = {}^{1}/_{8} \cdot 1\,\mathrm{MB}$ of the full-resolution data. Therefore, writing out compressed multi-resolution data further decreases the data size per core, which may complicate collective writes in HPC applications. According to the compression tests conducted for GLATE in Section 3.4.3, p. 66, GLATE yields a stable compression performance also on shorter streams consisting of $N = 8^3$, 16^3, 32^3 values. Therefore, GLATE is a reasonable choice for the compression of low-resolution data, and the memory footprint of the compressed low-resolution data is expected be in the order of $128\,\mathrm{kB} \times 25\,\%_{\mathrm{32b}} = {}^{1}/_{32} \cdot 1\,\mathrm{MB} = 32\,\mathrm{kB}$.

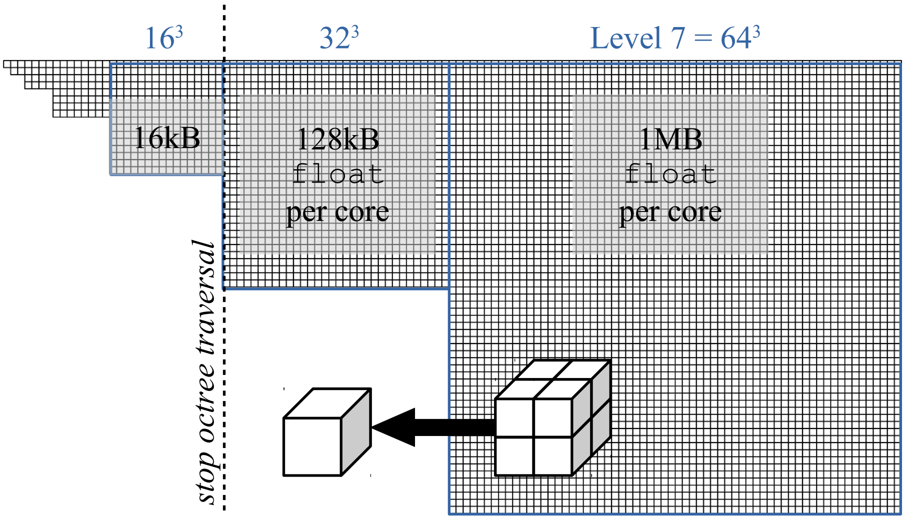


Figure 5.9: Octree averaging operation for computation of low-resolution data. Eight voxels are merged into one using averaging. One averaging operation transforms a subgrid into a grid with 8× less voxels, i.e. from $64 \times 64 \times 64$ voxels to $32 \times 32 \times 32$ voxels.

Concretely, each MPI process computes the averaging operation for levels 7 and 6, i.e. $64^3 \rightarrow 32^3$ voxels. The averaging is carried out using the following equation for $i, j, k = 1, 2, 3, \ldots, 32$:

$$\begin{aligned} x^6(i,j,k) \; = \; {}^1/_8 \cdot [\; & x^7(\; 2i{+}0 \;,\; 2j{+}0 \;,\; 2k{+}0 \;) \;+ \\ & x^7(\; 2i{+}1 \;,\; 2j{+}0 \;,\; 2k{+}0 \;) \;+ \\ & x^7(\; 2i{+}0 \;,\; 2j{+}1 \;,\; 2k{+}0 \;) \;+ \\ & x^7(\; 2i{+}1 \;,\; 2j{+}1 \;,\; 2k{+}0 \;) \;+ \\ & x^7(\; 2i{+}0 \;,\; 2j{+}0 \;,\; 2k{+}1 \;) \;+ \\ & x^7(\; 2i{+}1 \;,\; 2j{+}0 \;,\; 2k{+}1 \;) \;+ \\ & x^7(\; 2i{+}0 \;,\; 2j{+}1 \;,\; 2k{+}1 \;) \;+ \\ & x^7(\; 2i{+}1 \;,\; 2j{+}1 \;,\; 2k{+}1 \;) \;] \end{aligned} \tag{5.4}$$

$x^6(i,j,k)$ and $x^7(i,j,k)$ correspond to the data of variable x stored in the voxel (i,j,k) on the level of resolution 6 and 7 respectively.

Writing Low-resolution Data using Sequential Storage Layout

Before writing data to the storage system, the low-resolution subgrids are computed and cached temporarily inside each MPI process. Both full-resolution and low-resolution grids are compressed and ordered into the global sequential data layout, as described in the previous section. In contrast to the sequential data layout for full-resolution data, which creates one file per variable, the sequential data layout for low-resolution data creates two files per variable, i.e. one for low-resolution and one for full-resolution grids. In each time step of the CFD simulation, the procedure creates the following file structure.

```
/scratch/archive/time001_level001_variable001
/scratch/archive/time001_level001_variable002
/scratch/archive/...
/scratch/archive/time001_level002_variable001
/scratch/archive/time001_level002_variable002
/scratch/archive/...
/scratch/archive/time002_level001_variable001
/scratch/archive/...
```

5.4 In-Situ Tests in the Taurus HPC Cluster

As described in Section 5.2, the evaluation of GLATE and t-GLATE is carried out on the *Taurus* HPC *Cluster*, where the LBM is conducted for a CFD simulation of metal melt in a complex filter structure. The LBM code is used as a vehicle for mass data production and execution of in-situ data compression. The storage procedures, as explained in Section 5.3, are integrated into the MPI-based LBM solver [85] and applied during the running CFD simulation in order to record the I/O times for writing uncompressed and compressed full/low-resolution data.

Compute Hardware in Taurus HPC Cluster

The tests are carried out on the *Taurus* HPC *Cluster* located in the *Centre for Information Services and High-Performance Computing* in Dresden, Germany. By the end of 2017, the cluster comprised ~340 cores with large main memory for shared memory parallelization, ~340 GPUs, as well as, ~38,000 cores housed in more than 2,000 nodes.

Specifically for the compression tests, nodes with two 12 core Intel$^{(R)}$ CPUs (Xeon$^{(R)}$ E5-2680 at 2.50 GHz, MultiThreading disabled) are used. The data products are written to a parallel Lustre$^{(R)}$ [75] file system.

Architecture of the Lustre$^{(R)}$ Parallel File System

Lustre$^{(R)}$ is a distributed file system for HPC clusters based on object storage principles, which can handle large amounts of storage devices distributed over many storage nodes. A Lustre storage system consists of a set of storage nodes, so-called *Object Storage Server*s (OSSs), used for spanning a distributed storage across many devices, so-called *Object Storage Target*s (OSTs), typically a set of hard drives in a RAID configuration. One node, the so-called *Metadata Server* (MDS), is dedicated to the management of the metadata stored on the so-called *Metadata Target* (MDT), and maps data blocks to a file system hierarchy [75]. Fig. 5.10 shows the architecture of a typical Lustre distributed file system.

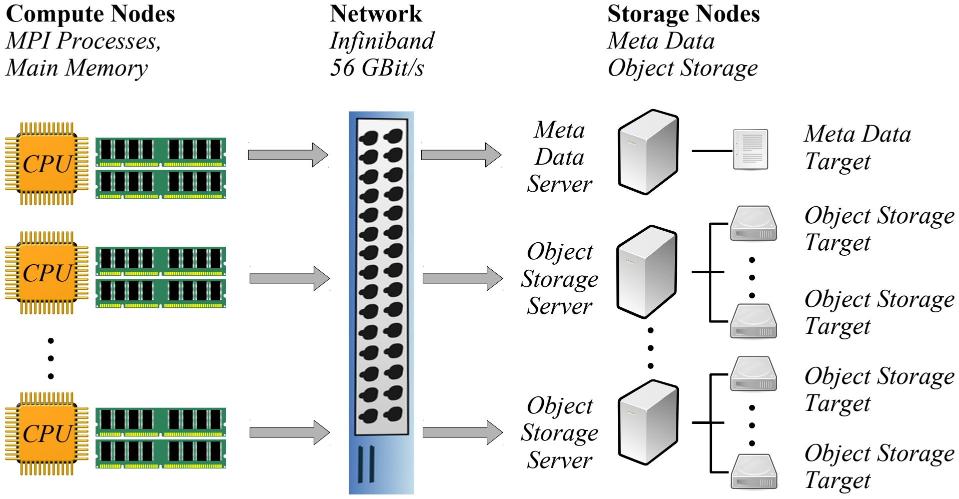


Figure 5.10: Architecture of a Lustre$^{(R)}$ distributed file system. The data throughput is limited by the speed of storage devices and network connections. Therefore, Lustre allows for the aggregation of I/O bandwidth, i.e. MPI processes can use designated channels for data transfer.

Image credits: `https://www.nics.tennessee.edu/files/images/lustre-components.jpg`

In total, by the end of 2017, the storage system comprised 192 OSTs in 16 OSS providing ~5.2 PB of temporary scratch storage for results of HPC computations. The parallel file system and the compute nodes are connected using a 56 Gbit/s Infiniband$^{(R)}$

link. Although the file system offers a huge storage, a large difference in the total amount of compute nodes and storage nodes is present, i.e. more than 35,000 CPU cores and several hundred GPUs are available for data production in more than 2,000 nodes, whereas 16 nodes are designated for data storage.

File Striping in the Lustre$^{(R)}$ Parallel File System

One of the most important features of the Lustre file system is *File Striping*, i.e. large fragmented files can be distributed across many OSTs. A so-called *Striped File* is separated into constant sized chunks, so-called *Stripes*, which are distributed over several OSTs. The main advantage of striping is that many I/O operations can be performed on multiple OSSs/OSTs in parallel, i.e. a single shared file can be read or written using the aggregate I/O bandwidth of multiple OSTs in multiple OSSs. The amount of OSTs and OSSs participating in the write operation are both controlled by the so-called *Stripe Count* and configured per directory using the `lfs` utility. Files created automatically inherit the present striping configuration of the directory. For the *Stripe Size*, the default configuration of the Lustre file system of size 1 MB is used. In order to create a directory where files are striped accross e.g. 4 OSTs, the following command is invoked.

```
lfs  setstripe  -c 4  /scratch/archive/
```

Writing Stripe-Aligned Uncompressed Data

As the data throughput is limited by the speed of storage devices and network connections, the aggregation of I/O bandwidth and network bandwidth allows for collective write of fragmented data into shared files at very high speeds. In the extreme case, each MPI process moves its data into one OST. There, each MPI process can use an exclusive channel for data transfer similar to a local hard drive. Usually a lower stripe count is used and stripes are distributed across OSTs in a round robin fashion.

For the highest performance, the MPI processes need to ensure a *Stripe-Aligned Data Fragmentation*, i.e. the size of the data fragments of the MPI processes is aligned with the *Stripe Size* [38, 62]. In the case each MPI process has a designated set of targets, unneeded message passing is avoided, and Lustre can achieve an effective data throughput in the order of giga byte per second [75]. Accordingly, for writing the two dimensional 16×16 grid shown in Fig. 5.4 (*c*), the stripe count of 4 is used, which results in a stripe-aligned data fragmentation, as shown in Fig. 5.11.

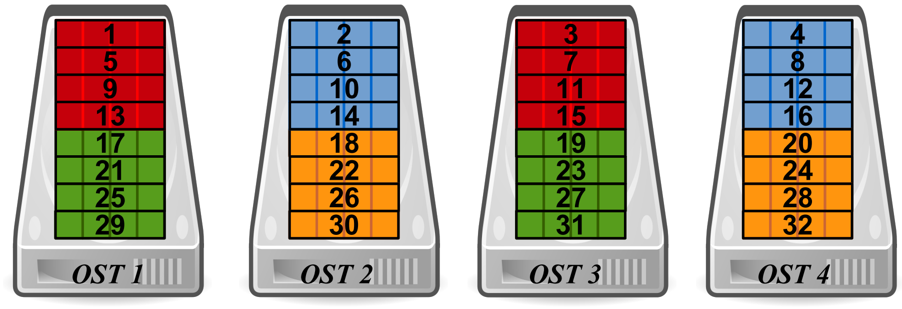


Figure 5.11: Distribution of stripe-aligned data fragments across four OSTs. Low cost for communication and data distribution.

As the data chunks of MPI processes are interlaced, the linearization of the global column-major grid, as described for the column-major storage layout in Section 5.3.2, directly results in a stripe-aligned data fragmentation. Hence, the stripe count is selected for single MPI processes to access only a small number of OSTs, i.e. one stripe is written by at most one MPI process, and one MPI process accesses only two different OSTs. According to the example in Fig. 5.5, p. 146, MPI processes 1&3 perform stripe-aligned writes to OSTs 1&3, and MPI processes 2&4 to OSTs 2&4.

Writing Size-Varying Compressed Data

As noted above, stripes in the Lustre file system are data chunks of constant size. Therefore, the amount of message passing required for the distribution of the data accross OSSs and OSTs increases if the size of the data chunks is not aligned with the stripe size. In general, compression results in *Size-Varying Data Fragments*, which are merged into a shared file with increased message passing overhead, as shown in Fig. 5.12. Consequently, the collective write operation is not stripe-aligned and the I/O time of the data write is not expected to scale according to the reduction of data size due to compression.

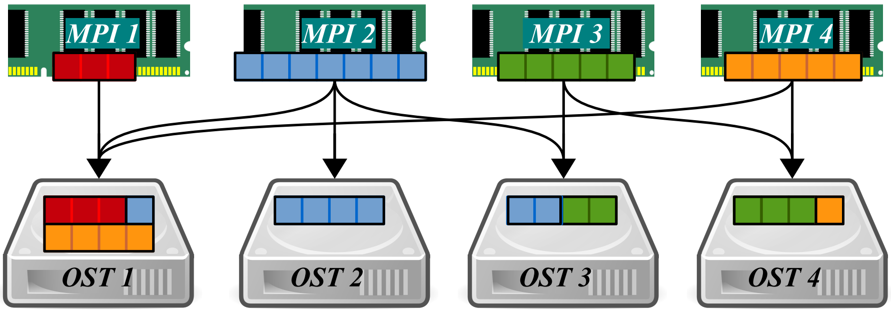


Figure 5.12: Distribution of size-varying data fragments across four OSTs. Increased cost for communication and data distribution.

5.4.1 In-Situ Compression using GLATE

The run-time performance of GLATE and the I/O performance for writing size-varying compressed full/low-resolution data into the Lustre$^{(R)}$ file system is evaluated in a parallel CFD simulation with 512 processes. Further, the I/O times are compared to the write of stripe-aligned uncompressed data, as well as, compressed data generated by the `ZFP` lossy floating point compressor. Whereas the comparison of I/O times for compressed and uncompressed data highlights the overhead in message passing for joining unevenly fragmented data, the comparison to `ZFP` highlights the run-time performance of the current implementation of GLATE.

In-Situ Compression Test

The tests carried out on the Taurus cluster were not executed with exclusive access to the file system. Therefore, during each simulation 128 time steps are stored in order to report the average I/O times for writing one time step of the CFD data set. Additionally, in order to lower the influence of the cluster usage by other users, the jobs are started e.g. at the end of the month, in the nights, and on the weekends. Each simulation is spawned on 512 MPI processes using 22 physical nodes exclusively, which corresponds to the most dense packing of MPI processes on compute nodes possible.

As explained in Section 5.2, the flow field (u, v, w) is computed along with the flow field properties velocity magnitude M and vortex characteristics Q. All CFD simulations are conducted inside the same filter structure, as shown in Fig. 5.3, p. 141, and are started from the same checkpoint solution, hence producing the same data. During the simulation, the five variables u, v, w, M, Q are computed in full/low-resolution and stored without compression as `FLOAT`, and with compression using GLATE and `ZFP`. For the storage of the uncompressed data and the compressed full/low-resolution data, the global column-major data layout and the sequential data layout are used respectively, as explained in Sections 5.3.2 to 5.3.4.

Compression Rate in CFD Simulation

Each one of the 512 MPI processes computes the flow field for $64 \times 64 \times 64$ voxels, which accounts for 1 MB of uncompressed `FLOAT` per variable per time step. For evaluation of the compression rate, the GLATE compressor is configured to use the error bound $e_{\mathrm{MAX}} = 1.00\,\%$, and the optimized compression procedure tGLATE_ compress_ opt (...) is used, as given in Alg. 4.9, p. 112. The execution of the temporal compression procedure using $k_{\mathrm{MOD}} = 1$ results in non-temporal compression, as only KFs are inserted. `ZFP` is executed with precision level 15 corresponding approximately to a point-wise maximum error of $e_{\mathrm{MAX}} = 1.00\,\%$, i.e. 0.5 % of all values greater than 1E–4 exhibit decompression error greater than 1.00 %, according to $q_{\mathrm{MAX}}^{1.00\,\%}$ shown in Fig. 4.26 (BOTTOM), p. 125. Fig. 5.13 (a) shows the average compression rate for all simulation variables u, v, w, M, Q resulting from the application of `ZFP` and GLATE in the CFD simulation.

The storage of uncompressed `FLOAT` data is denoted as `FLOAT` *Grid* and accounts to 100 $\%_{32b}$. `ZFP` and GLATE show a similar average compression rate w.r.t. the error-specific compression task, as discussed in Section 4.5, i.e. `ZFP` achieves ~16.4 $\%_{32b}$ and GLATE achieves ~18.7 $\%_{32b}$. GLATE in non-temporal compression using $e_{\mathrm{MAX}} = 1.00\,\%$ is slightly worse by ~2.5 $\%_{32b}$, as compared to `ZFP`. However, unlike `ZFP`, GLATE guarantees the error bound for all values. By showing a more than five fold data reduction on 32 bit `FLOAT` data, GLATE constitutes a powerful scalar quantization method while restricting the maximum error of decompressed data. The lowest compression rate of ~14.1 $\%_{32b}$ is achieved for the variable M by both `ZFP` and GLATE.

As shown in Fig. 5.13 (b), using `ZFP` and GLATE for the compression of low-resolution data introduces at most ~3.0 $\%_{32b}$ of additional storage overhead on top of the compressed full-resolution data. In contrast, storing additional uncompressed low resolution data would demand for 12.5 $\%_{32b}$ additional storage.

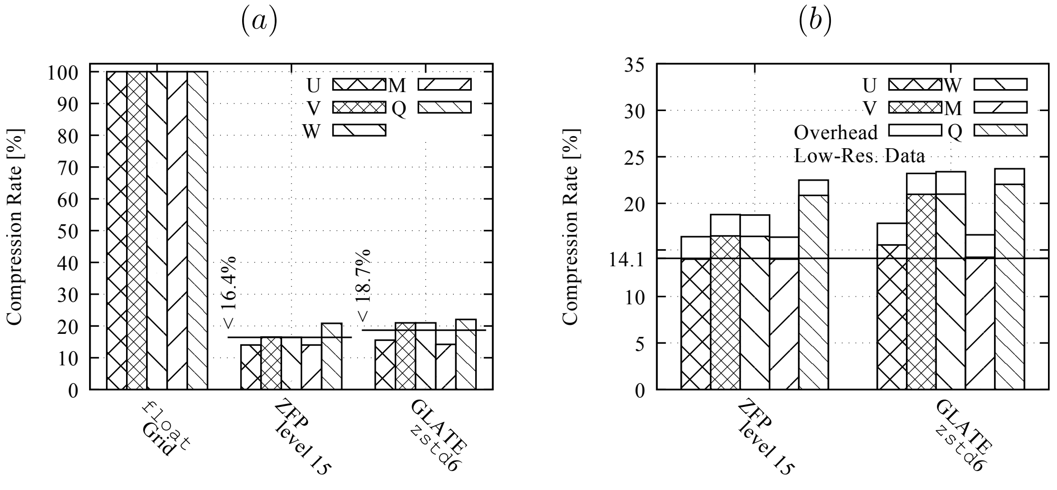


Figure 5.13: Compression rate of `ZFP` and GLATE on full-resolution and low-resolution data for simulation variables u, v, w, M, Q. (a) Compression of full-resolution data. The average compression rate for `ZFP` is ~16.4 $\%_{32b}$, and for t-GLATE is ~18.7 $\%_{32b}$. (b) Additional compression of low-resolution data introduces at most ~3 $\%_{32b}$ overhead. `FLOAT` *Grid* corresponds to writing uncompressed data in global column-major layout. t-GLATE is applied with $k_{\mathrm{MOD}} = 1$ and $e_{\mathrm{MAX}} = 1.00\,\%$. `ZFP` is applied with precision level 15.

Run-Time of CFD Simulation

Fig. 5.14 shows the simulation run-time when writing uncompressed data and for writing compressed data from `ZFP` and GLATE. The reported times include the run-time of the LBM solver. Each test has been executed using an increasing stripe count of 4, 8, 16, 32, 64, 128. Using 4, 8, 16, 32 stripes, the average run-time per iteration decreases constantly for writing uncompressed data. Since 1 MB `FLOAT` data per MPI process corresponds to the default stripe size, the write of uncompressed data results in a stripe-aligned data fragmentation. Therefore, an optimized I/O performance is given, which is improved by increasing the stripe count, as more aggregate bandwidth is available.

As the 512 MPI processes are spawned on 22 physical nodes, several MPI processes share the bandwidth of one compute node and no further improvement was observed for larger stripe counts of 64, 128 stripes. When writing compressed data with reduced memory footprint, the total simulation run-time including data writes decreases for lower stripe counts of 4, 8 stripes. Using lower stripe counts for writing compressed data allows for the acceleration of the simulation, as less aggregated bandwidth between compute nodes and storage nodes is available. Using the GLATE implementation given

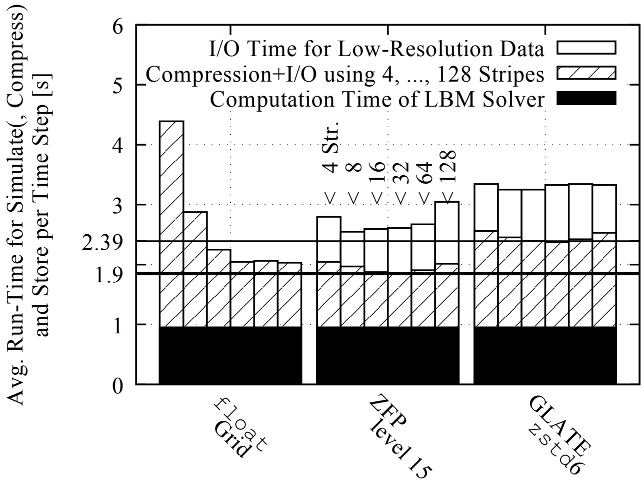


Figure 5.14: Simulation run-time when writing uncompressed and compressed data into the Lustre$^{(R)}$ file system using different data layouts. `FLOAT` *Grid* corresponds to using the global column-major layout with uncompressed data. The plots show the overall simulation run-time including the run-time of the LBM solver and the I/O time for data writing. The simulation run-time decreases strongly for an increasing stripe count of 4, 8, 16, 32 stripes when uncompressed data is written. The simulation which uses compression has an increased cost for message passing and distribution of size-varying compressed data, and the improvement does not scale according to the data reduction. Writing low-resolution data with extremely small memory foot print introduces a nearly constant run-time penalty. Fastest simulation run when writing full-resolution data for `ZFP` is ~1.9 s, and for GLATE is ~2.39 s.

in Alg. 4.9, p. 112, the fastest simulation run was ~2.39 s, which introduces about ~0.49 s run-time overhead per time step, as compared to `ZFP` with ~1.9 s, i.e. slightly increased I/O time due to ~2.5 $\%_{32b}$ larger data size, and additional time spent for data preparation, `FLOAT` decomposition and quantization.

For writing out additional low-resolution data, the effect intensifies even more as compressed low resolution data has an even smaller memory footprint. Particularly in this case, the GLATE compression reduces the data size from 1 MB to less than ~205 kB $\simeq$ 1 MB × 20.0 $\%_{32b}$ per variable per MPI process for full-resolution data, and to less than 26 kB $\simeq$ 128 kB × 20.0 $\%_{32b}$ per variable per MPI process. For such small data per core, the overhead in message passing dominates and results in a nearly constant run-time penalty for writing the low-resolution data for all stripe counts 4, 8, 16, 32, 64, 128 stripes. Write-out of uncompressed low-resolution `FLOAT` data was not conducted.

I/O Times of Collective Write Operations

Fig. 5.15 shows the I/O times for writing uncompressed `FLOAT` data and compressed full/low-resolution data from `ZFP` and GLATE *without* run-time of the LBM solver. The fastest write operation of stripe-aligned full-size data lasted ~1.05 s using 32 stripes. The same time was required for writing GLATE compressed data without taking into account the time spent in tGLATE_ compress_ opt (...) for data preparation, `FLOAT` decomposition and quantization.

The current implementation of the non-temporal GLATE compression procedure using $k_{\mathrm{MOD}} = 1$ introduces a run-time penalty of ~0.27 s. Compared to the run-time of `tpfor` and `zstd`, the implementation of tGLATE_ compress_ opt (...), as given in Alg. 4.9, p. 112, consumes a disproportionate amount of run-time due to an unoptimized implementation.

According to the compression rate of `ZFP`, which is ~2.5 $\%_{32b}$ ahead of GLATE, the time for writing the compressed data slightly decreases to ~0.9 s. Because of the optimized implementations, the run-time of the `ZFP` algorithm itself and the run-time for the lossless compression using `zstd`, `tpfor` inside of GLATE only take a negligible fraction of $\leq$0.08 s. The fastest write of compressed data occurs using 32 stripes and took ~0.9 s for `ZFP` and ~1.05 s for GLATE, i.e. compressed data was written at least as fast as stripe-aligned uncompressed `FLOAT` data. As OSTs are accessed concurrently, the increased overhead of message passing and distribution of data fragments over OSTs/OSSs hinders

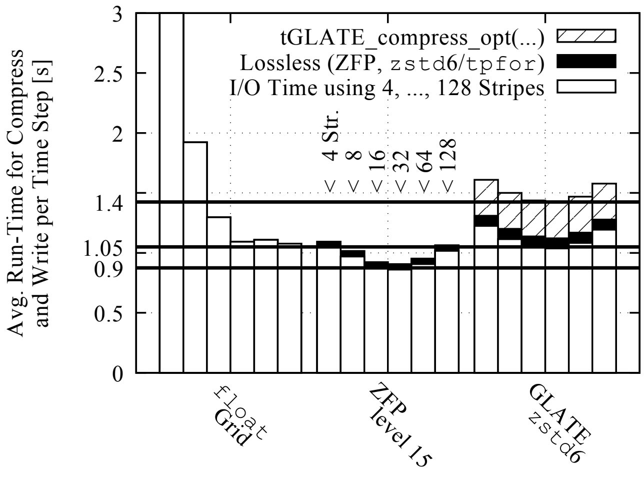


Figure 5.15: I/O times for writing uncompressed and compressed data into the Lustre$^{(\mathrm{R})}$ file system using different data layouts. `FLOAT` *Grid* corresponds to writing uncompressed data in global column-major layout. Fastest I/O time for GLATE compressed data is ~1.05 s, which is ~0.15 s slower than `ZFP` compressed data with ~0.9 s because of the ~2.5 %$_{\text{32b}}$ larger data size. The application of `zstd`, `tpfor` and `ZFP` only introduce a negligible run-time penalty. The current GLATE implementation introduces a run-time penalty of ~0.27 s for data preparation, `FLOAT` decomposition and quantization.

a further speed-up. However, for low stripe counts of 4, 8 stripes, the improvement of the I/O times is clearly visible, as compared to writing uncompressed data.

5.4.2 Temporal In-Situ Compression using t-GLATE

As shown in the previous section, `ZFP` and GLATE yield more than a fivefold data reduction of 32 bit `FLOAT` data for the flow field computed in the CFD simulation. Using 16, 32 stripes, the compressed full-resolution data was moved into the Lustre$^{(\mathrm{R})}$ file system at least as fast as moving stripe-aligned full-size data.

Using t-GLATE during the CFD simulation allows for further reduction of the size of full-resolution data using temporal compression, as explained in Section 4.3.4, p. 102. The run-time performance of t-GLATE temporal compression using $k_{\mathrm{MOD}} > 1$ and the I/O performance for merging temporal compressed full-resolution data into large files stored in the Lustre$^{(\mathrm{R})}$ file system is evaluated in the same CFD simulation with 512 processes. Low-resolution data is not compressed in this test.

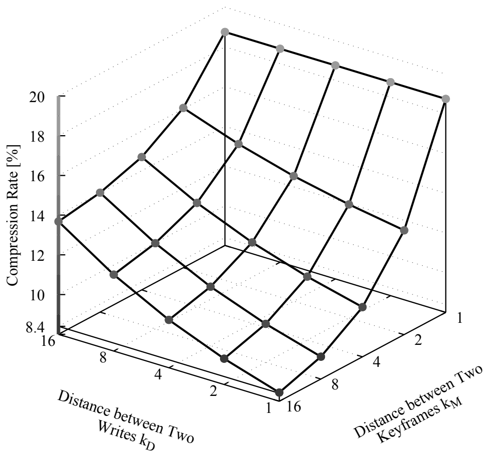
Temporal In-Situ Compression Test

The evaluation of the run-time and compression rate of t-GLATE temporal compression takes place through the variation of the parameters k_{D} and k_{MOD}, which control the trade-off between temporal resolution, amount of KFs, and the compression rate, as explained in Section 4.2, p. 82. When decreasing the temporal resolution using $k_{\mathrm{D}} > 1$, the computation time for the LBM solver increases relative to the run-time of compression and data writing, as time steps are skipped in the temporal compression procedure. As the parameter $k_{\mathrm{MOD}} > 1$ determines how many DFs are inserted before the next KF, $k_{\mathrm{MOD}} = 1$ corresponds to non-temporal GLATE compression.

Specifically, for each combination of k_{D}, $k_{\mathrm{MOD}} = 1, 2, 4, 8, 16$, one simulation run is conducted, in which temporal compressed data is written into the Lustre$^{(\mathrm{R})}$ file system using a stripe count of 16 stripes. In each run, a total of 128 time steps are compressed and written in order to report the average I/O time for moving one compressed full-resolution data set to storage. Depending on $k_{\mathrm{D}} = 1, 2, 4, 8, 16$ the actual number of iterations in the CFD simulation run varies between 128, 256, 512, 1024, 2048 ($= k_{\mathrm{D}} \cdot 128$).

Compression Rate in CFD Simulation

Table 5.1 shows the average compression rate for all five simulation variables u, v, w, M, Q resulting for temporal compression depending on k_{D} and k_{MOD}. For $k_{\mathrm{MOD}} = 1$, the compression performance of GLATE is reproduced, as shown in the previous section. For temporal compression using $k_{\mathrm{MOD}} > 1$, the compression rate ranges from ~8.4–16.0 $\%_{32b}$. As can be seen, temporal compression improves the compression rate of t-GLATE twofold when using $k_{\mathrm{MOD}} = 16$, as compared to non-temporal GLATE and `ZFP`. This confirms the excellent data reduction capabilities of the t-GLATE temporal compression algorithm which ensures a point-wise error of at most $e_{\mathrm{MAX}} = 1.00\,\%$ for the decompressed data.



[$\%_{32b}$] k_{D}	1	2	4	8	16
$k_{\mathrm{MOD}} = 1$	18.7	18.7	18.7	18.7	18.7
$k_{\mathrm{MOD}} = 2$	13.2	13.7	14.3	15.0	16.0
$k_{\mathrm{MOD}} = 4$	10.5	11.1	12.0	13.2	14.6
$k_{\mathrm{MOD}} = 8$	9.1	9.9	10.9	12.3	14.0
$k_{\mathrm{MOD}} = 16$	8.4	9.3	10.4	11.8	13.6

Table 5.1: t-GLATE compression rate in parallel CFD simulation with k_{MOD}, $k_{\mathrm{D}} = 1$, 2, 4, 8, 16, input stream size $N = 64^3$, and maximum error $e_{\mathrm{MAX}} = 1.00\,\%$. t-GLATE is applied using `zstd` on level 6 and `tpfor` with block size 128. Depending on k_{D} the temporal resolution of the decompressed data decreases, as only every k_{D}-th iteration is compressed.

CFD Simulation Run-Time and I/O Performance

Fig. 5.16 shows the run-time of the CFD simulation including the run-time of the LBM solver. The simulation run-time is shown for $k_{\mathrm{MOD}} = 2$, 16 and $k_{\mathrm{D}} = 1$, 2, 4, 8, 16, which corresponds to the second and fifth row of Table 5.1. The compression rate is lower than ~16.4 $\%_{32b}$ of `ZFP` using precision level 15, i.e. the t-GLATE compression rate varies between ~13.2–16.0 $\%_{0\,\text{32b}}$ for $k_{\mathrm{MOD}} = 2$, and between ~8.4–13.6 $\%_{0\,\text{32b}}$ for $k_{\mathrm{MOD}} = 16$. As can be seen, the run-time of the LBM solver increases for $k_{\mathrm{D}} = 2, 4, 8, 16$, whereas the time required for compression and collective write stays the same.

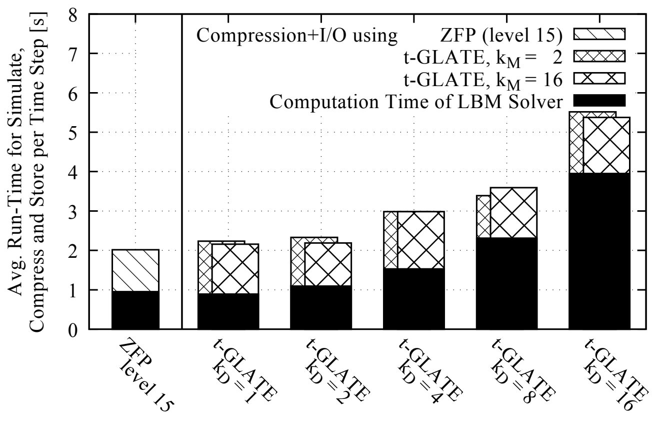


Figure 5.16: Varying run-time time of CFD simulation during temporal compression procedure. The plot shows the simulation run-time including the time required for compression and data writing. The CFD simulation is carried out with 512 MPI processes on 22 nodes. Temporal compression is applied using a low/high number of KFs for $k_{\mathrm{MOD}} = 2, 16$. Depending on $k_{\mathrm{D}} = 2, 4, 8, 16$, the simulation is run for 256, 512, 1024, 2048 iterations. The simulations using $k_{\mathrm{D}} = 1$ or `ZFP` on level 15 are run for 128 iterations.

The temporal compression yields a compression rate of ~8.4–16.0 $\%_{0\,\text{32b}}$. Therefore, the data per MPI process to be moved into the shared file fluctuates between ~85 kB and ~165 kB, which constitutes a further reduction of data size, as compared to non-temporal compression. Fig. 5.17 shows the average run-time per write of one CFD data set during the temporal compression procedure using $k_{\mathrm{MOD}} = 2, 16$ and $k_{\mathrm{D}} = 1, 2, 4, 8, 16$ *without* run-time of the LBM solver. Although *t*-GLATE achieves a further reduction in data size, the I/O times fluctuate around the same range of ~0.9–1.05 s as observed for non-temporal compression in the previous section, i.e. time required for collective writing does not reflect the reduction of data through temporal compression. As limited cluster access and the available time make it difficult to produce further significant results, no further analysis of the I/O time is carried out.

Similar to non-temporal compression, the current implementation of the *t*-GLATE temporal compression procedure introduces a run-time penalty of ~0.29 s using $k_{\mathrm{MOD}} > 1$. The run-time of `ZFP` and the lossless compression back end using `tpfor` and `zstd`

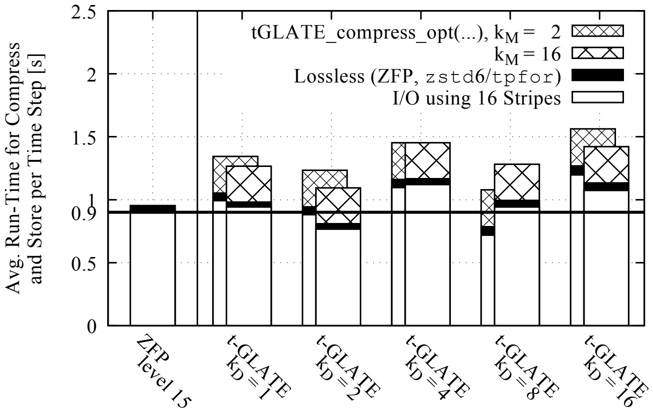


Figure 5.17: I/O time for writing temporal compressed data into the Lustre$^{(R)}$ file system during the CFD simulation. The collective write operation uses 16 stripes and incorporates 512 MPI processes on 22 nodes. Although the data size is further reduced by using temporal compression, the I/O times do not decrease according to the reduction in data size and fluctuate around the I/O times ~0.9–1.05 s for non-temporal compressed data. The current GLATE implementation introduces a run-time penalty of ~0.29 s for data preparation, `FLOAT` decomposition, quantization and caching.

consume a negligible amount of time ≤ 0.08 s.

5.5 Summary

In this chapter, GLATE compression and t-GLATE temporal compression were applied in a large-scale CFD simulation using 512 MPI processes on the *Taurus* HPC *Cluster*. For different configurations of the compression procedure, the run-time and compression rate of GLATE and t-GLATE were compared to `ZFP`. Further, the I/O performance for writing compressed full/low-resolution data into the Lustre$^{(R)}$ file system was compared to writing full-size `FLOAT` data. To this end, GLATE and t-GLATE were integrated into an in-house LBM solver written in FORTRAN, which is executed for the computation of the test data, i.e. the flow field (u, v, w), and two flow field properties M, Q.

Overview of Compression Rate

Fig. 5.18 summarizes the compression rate achieved by `ZFP`, GLATE and t-GLATE for the compression of full/low-resolution data during the CFD simulation. For the compression of temporal sequences of CFD data sets, t-GLATE is applied using k_{D}, $k_{\mathrm{MOD}} = 1, 2, 4, 8, 16$. The non-temporal GLATE compression corresponds to the application of t-GLATE using $k_{\mathrm{MOD}} = 1$. The parameter k_{D} controls the temporal resolution of the decompressed data, as only every k_{D}-th iteration is included into the temporal compression procedure. As can be seen, t-GLATE temporal compression reduces the data size between ~8.4–16.0 %$_{\text{32b}}$ and restricts the decompression error to at most $e_{\mathrm{MAX}} = 1.00\,\%$. Therefore, while improving the data reduction for temporal sequences of full-resolution CFD data sets, t-GLATE creates a trade-off between the compression rate and

k_{MOD}	the amount of inserted KFs,
k_{D}	the temporal resolution of decompressed data,
e_{MAX}	the maximum error of decompressed data.

Further, the compressed low-resolution data takes up at most ~3.0 % to provide a global low-resolution preview for the full data set. The extremely small memory footprint of compressed low-resolution data allows for fast transfer into visualization and analysis applications.

Collective Write of Compressed Data

Considering the small memory footprint of the compressed low-resolution data, as well as, the high reduction of temporal compressed data, the data size per MPI process to be moved from main memory to storage falls far below the default stripe size of 1 MB used in the Lustre$^{(\mathrm{R})}$ file system. The compressed data is small and not stripe-aligned, which introduces overhead for message passing between compute nodes and storage nodes when using collective I/O procedures. As compared to writing stripe-aligned full-size `FLOAT` data using the global column-major data layout, the collective write of compressed data does not accelerate the process in the same manner as compression reduces the data size.

Moving compressed full-resolution data into the Lustre$^{(\mathrm{R})}$ file system has however been at least as fast as moving stripe-aligned full-size data. In all cases for non-temporal `ZFP` and GLATE compression, as well as, temporal t-GLATE compression, the time required for the write operation using 16 stripes fluctuates around the time ~0.9–1.05 s required

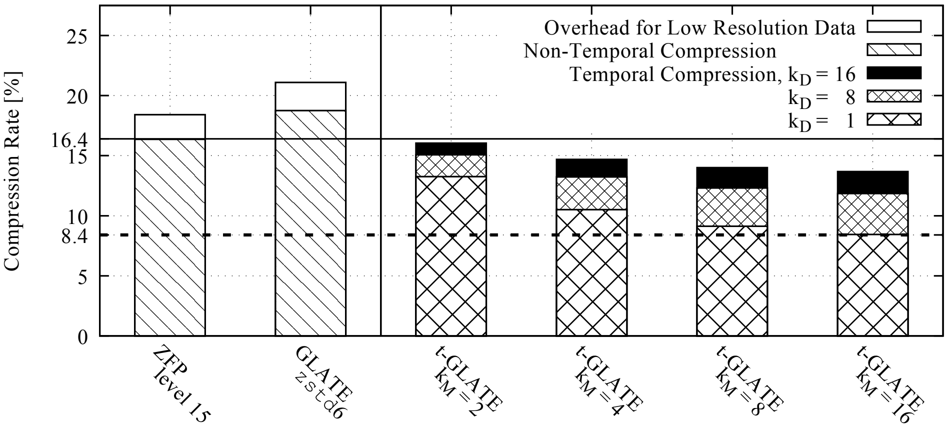


Figure 5.18: Overview of compression rate of `ZFP` and t-GLATE in CFD simulation. `ZFP` operates on precision level 15. t-GLATE uses $k_{\mathrm{D}} = 1, 8, 16$, $k_{\mathrm{MOD}} = 2, 4, 8, 16$, and $e_{\mathrm{MAX}} = 1.00\,\%$. GLATE corresponds to the application of t-GLATE using $k_{\mathrm{MOD}} = 1$. $k_{\mathrm{MOD}} > 1$ corresponds to temporal compression. For $k_{\mathrm{MOD}} \geq 2$ and all values of $k_{\mathrm{D}} = 1, 2, 4, 8, 16$, t-GLATE improves the compression rate, as compared to `ZFP` which achieves ~16.4 $\%_{32b}$. Low-resolution is stored with extremely small memory footprint of at most ~3.0 %

to move uncompressed data. For low stripe counts of 4, 8 stripes, the improvement of the I/O times compared to writing uncompressed data is clearly visible, as less aggregate bandwidth is available for the data transfer. In the extreme case using only one stripe, the I/O performance resembles a constant data throughput similar to e.g. data transfer using a slower network connection, or data movement into a local hard drive.

t-GLATE Implementation State

The implementation of the temporal compression procedure tGLATE_compress_opt (...) in its current state introduces a run-time penalty of ~0.27–0.29 s, as explained in Sections 5.4.1 and 5.4.2. Including the time ~0.08 s for the application of `zstd` and `tpfor`, the run-time for t-GLATE compression takes $p_G = 0.37$ s in the worst case. According to Alg. 4.9, p. 112, the t-GLATE implementation uses a lot of conditional branching, i.e. `if`-statements inside of `for`-loops, which produce a bad run-time performance [39]. Further, the run-time performance of the binary search for quantization of the mantissa is to be tested against the analytical determination based on the inverse step function, as shown in Fig. 3.11, p. 48. Nevertheless, the current t-GLATE implementation offers potential for optimizations, and, as motivated in Section 4.5, constitutes a promising approach for fast compression, which is, unlike `ZFP`, not limited to the compression of uniform grids. Further improvement of the current t-GLATE implementation appears possible. Hence, t-GLATE is a promising approach for fast compression of `FLOAT` data supporting temporal compression. Unlike `ZFP`, the t-GLATE algorithm can compress arbitrary grid types through employing linearization schemes or indices of the mesh topology, as motivated in Section 4.5, p. 123.

Data Transfer during Post-Processing

Despite the absent improvement of I/O times for writing compressed data in the Lustre$^{\text{(R)}}$ file system using high stripe counts, the high data reduction rates improve the scientific work flow if data is moved from the HPC environment to the analysis workstations. In many modern storage systems for HPC clusters, which employ e.g. temporary scratch storage during mass data production, or node-local/rack-local storage as burst-buffer for data-intensive applications, data movement is inevitable [43]. There, a strong reduction of the memory footprint using *Temporal Lossy In-Situ Compression* accelerates the scientific workflow in the case data has to be moved at least one time after its creation. Unlike the mass production of data during the CFD simulation, the access patterns during visualization tasks are often problem-specific and hard-to-predict [26, 52]. Here, early data preparation and compression enhance the accessibility of large data sets e.g. for iterative loading of subregions, or by fast transfer into remote visualization workstations.

Improvement of Time-to-Analysis

The improvement of the *Time-to-Analysis* in the scientific workflow is estimated through the transfer time savings for moving compressed data. As shown in Fig. 5.15, p. 163 and Fig. 5.17, p. 167, the I/O times for writing out GLATE and t-GLATE compressed data fluctuate around the write times for the stripe-aligned full-size `FLOAT` data. The storage of strongly compressed data using collective I/O routines did not improve the write performance according to the reduction of the data size. Further, as explained above, the current implementation of t-GLATE introduces a run-time penalty of ~0.27–0.29 s per write operation. Specifically, the following relation describes the condition under which an improvement is possible: The time for "*simulate & store* (t_S) and *transfer*

(t_T) uncompressed `FLOAT` data" is to be greater than the time for "*simulate, compress & store* (t'_S) and *transfer & decompress* (t'_T) using t-GLATE".

$$\begin{pmatrix} \text{Run Simulation,} \\ \text{Store } \texttt{FLOAT}, \\ \text{Transfer } \texttt{FLOAT} \end{pmatrix} > \begin{pmatrix} \text{Simulate \& Compress,} \\ \text{Store Compressed,} \\ \text{Transfer \& Decompress} \end{pmatrix} \tag{5.5}$$

$$t_S + t_T \quad > \quad t'_S + t'_T$$

The time t'_S is estimated through the tests which have been conducted for temporal compression using t-GLATE, as described in Section 5.4.2. Assuming a nearly tenfold data reduction using $k_\text{D} = 4$, $k_\text{MOD} = 16$, which results in a compression rate of $c_G = {\sim}10.4\,\%_{\text{32b}}$ according to Table 5.1, the time required for *simulate, compress & store* equals $t'_S = 3.0\,\text{s}$ according to Fig. 5.16. Further, according to Fig. 4.31 (a)–(b), p. 135, the t-GLATE decompression consumes $d_G = {}^{0.07\,\text{s}}/_{0.23\,\text{s}} \simeq 30.4\,\%$ of the compression run-time. Assuming the t-GLATE compression run-time to be of at most $p_G = 0.37\,\text{s}$, including `zstd` and `tpfor` run-time, the times t_S, t'_S, t'_T can be estimated as follows:

t'_S *Simulate, compress & store* using t-GLATE:
$t'_S = 3.0\,\text{s}$, cf. Fig. 5.16, p. 166

c_G *Compression rate* using t-GLATE:
$c_G = 10.4\,\%_{\text{32b}}$ for $k_\text{D} = 4$, $k_\text{MOD} = 16$, cf. Table 5.1, p. 165

p_G *Compress* using t-GLATE:
$p_G = 0.37\,\text{s}$
tGLATE_compress_opt(...) run-time ~0.27–0.29 s
`zstd` and `tpfor` run-time ~0.08 s

$p_G \cdot d_G$ *Decompress* using t-GLATE:
$d_G = {}^{0.07\,\text{s}}/_{0.23\,\text{s}} \simeq 30.4\,\%$, cf. Fig. 4.31 ($a$)–($b$), p. 135
$p_G \cdot d_G = 0.37\,\text{s} \times 30.4\,\% = 0.11\,\text{s}$

t_S *Simulate & store* uncompressed `FLOAT` data:
$t_S = t'_S - p_G = 2.63\,\text{s}$
(Assuming I/O time uncompressed `FLOAT` = I/O time compressed data.)

t'_T *Transfer & decompress* using t-GLATE:
$t'_T = c_G \cdot t_T + p_G \cdot d_G$

t_T *Transfer* uncompressed `FLOAT` data:
Estimated by speed of network connection.

For the estimation of the data transfer time t'_T between HPC scratch storage and local visualization workstations, a constant network speed is assumed, i.e. the data transfer time scales according to the t-GLATE compression rate c_G. Substitution of t_S, t'_S, t'_T in Eq. (5.5) yields the following relation for the estimation of the improvement of the *Time-to-Analysis*.

$$\begin{array}{ccc} \cancel{t_S} + t_T & > & \cancel{t_S} + p_G + d_G \cdot p_G + c_G \cdot t_T \\ (1 - c_G) \cdot t_T & > & (1 + d_G) \cdot p_G \\ t_T \times 89.6\,\% & > & 0.48\,\mathrm{s} \\ \left(\begin{array}{c} \text{Transfer Time} \\ \text{Savings} \end{array} \right) & > & \left(\begin{array}{c} t\text{-GLATE} \\ \text{Run-Time} \end{array} \right) \end{array} \tag{5.6}$$

In conclusion, as long as the time saved through transferring the compressed data exceeds the run-time required for the invocation of the t-GLATE compression and decompression procedure, the *Time-to-Analysis* in the scientific workflow is improved by using *Temporal Lossy In-Situ Compression*. Specifically for $k_\mathrm{D} = 4$, $k_\mathrm{MOD} = 16$, the *Time-to-Analysis* is improved, if ~89.6 % of the transfer time of uncompressed `FLOAT` data exceed ~0.48 s.

Speed-Up of the Scientific Workflow

The aforementioned improvement of the *Time-to-Analysis*, as described by Eq. (5.6), is quantified using the practical CFD simulation scenario, as described in Section 5.2. After the CFD simulation has been used for the production of the flow field and its properties, the data needs to be transferred into local visualization workstations. In order to transfer one time step of the data set $\mathcal{D}$ containing u, v, w, M, Q, a total amount of $a_D = 5 \cdot 512\,\mathrm{MB} = 2{,}560\,\mathrm{MB}$ of uncompressed `FLOAT` data needs to be moved. Although, for $k_\mathrm{D} = 4$ the data is *simulated & stored* in only $t_S = 2.63\,\mathrm{s}$, as explained above, assuming a constant network transfer speed of b_D [MB/s], the data needs $t_T = a_D / b_D$ [s] for transmission. Further, t-GLATE requires $(1 + d_G) \cdot p_G =$

130.4 % × 0.37 ≃ 0.48 s run-time for *compression & decompression* per time step, and for $k_{\mathrm{D}} = 4$, $k_{\mathrm{MOD}} = 16$, *t*-GLATE achieves a compression rate of $c_G = 10.4\,\%_{32b}$ using $e_{\mathrm{MAX}} = 1.00\,\%$. Inserting the numbers into Eq. (5.6) yields the following relation:

$$\begin{aligned} (1 - c_G) \cdot \frac{a_D}{b_D} &> (1 + d_G) \cdot p_G \\ \frac{2{,}293.76\,\mathrm{MB}}{b_D} &> 0.48\,\mathrm{s} \end{aligned} \tag{5.7}$$

Assuming a network transfer speed of $b_D = 1$ Gbit/s ≃ 128 MB/s, the transfer time savings of ~17.92 s significantly exceed the *t*-GLATE run-time of ~0.48 s. Therefore, for each time step which is transferred, the data is ready for analysis ~17.44 s = 17.92 s − 0.48 s earlier than for transfer of uncompressed `FLOAT` data. Consequently, for transfer of 1,000 time steps, the data is ready ~4.8 hours earlier, and for 10,000 time steps, ~2 days earlier. Fig. 5.19 (*a*) summarizes the transfer time savings for network connections between 1 Gbit/s, 2.5 Gbit/s, . . ., 10 Gbit/s transfer speed.

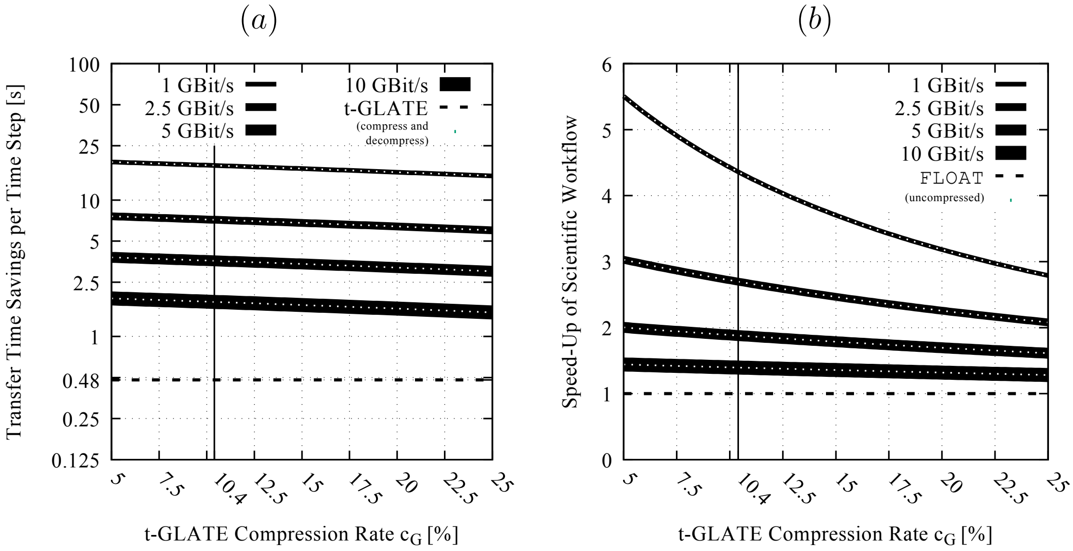


Figure 5.19: Transfer time savings and speed-up of the scientific workflow by using *t*-GLATE in-situ compression. The usage of temporal lossy in-situ compression in the scientific method cycle, as shown in Fig. 5.1, p. 139, improves the *Time-to-Analysis*, i.e. (a) transfer time savings according to Eqs. (5.6) and (5.7), and (b) speed-up of the scientific workflow according to Eq. (5.8) for network connections with maximum speed of 1 Gbit/s, 2.5 Gbit/s, . . ., 10 Gbit/s.

The speed-up of the scientific workflow describes how many time steps can be *"simulated, compressed, transferred & decompressed"* using t-GLATE in the same time in which one uncompressed `FLOAT` data set is *"simulated, stored & transferred"*. The speed-up is given by the following equation:

$$
\begin{aligned}
S &= \frac{\text{Time for Simulate, Store, Transfer Uncompressed } \texttt{FLOAT}}{\text{Time for Simulate, Store, Transfer Compressed} + \text{Compression \& Decompression}} \\
S &= \frac{t_S + t_T}{t'_S + t'_T} = \frac{t_S + {}^{a_D}/_{b_D}}{t_S + c_G \cdot {}^{a_D}/_{b_D} + (1 + d_G) \cdot p_G}
\end{aligned}
\tag{5.8}
$$

Assuming the same network transfer speed of $b_D = 1\,\text{Gbit/s} \simeq 128\,\text{MB/s}$, a total amount of ~4.3× more time steps are ready for analysis in the same time, as compared to using uncompressed data. Fig. 5.19 (*b*) summarizes the speed-up of the scientific workflow when using t-GLATE for in-situ compression and a 1 Gbit/s, 2.5 Gbit/s, …, 10 Gbit/s network connection for data transfer. As can be seen, for the t-GLATE compression rate of $c_G = 10.4\,\%_{32\mathrm{b}}$ using $k_{\mathrm{D}} = 4$, $k_{\mathrm{MOD}} = 16$, an amount of ~1.4–4.3× more time steps can be transferred in the same time as one uncompressed `FLOAT` data set, i.e. the analysis can happen quicker.

Closing Statement

The in-situ applicability of t-GLATE was evaluated using a large-scale CFD simulation for aluminum metal melt in a complex filter structure, as described in Section 5.4.1. Further, the improvement of the *Time-to-Analysis* in the scientific method cycle was discussed based on the evaluation results. All measurements of I/O times are carried out for writing stripe-aligned uncompressed `FLOAT` data, and by applying GLATE, t-GLATE, as well as, `ZFP` for compression directly inside of the LBM solver.

For high stripe counts of 16, 32, the I/O performance when writing out fragmented size-varying compressed full-resolution data does not scale according to the compression rate ~8.4–18.7 $\%_{32\mathrm{b}}$ achieved by GLATE and t-GLATE. Even for a tenfold data reduction, the I/O times of t-GLATE compressed data fluctuate around the I/O times for stripe-aligned uncompressed `FLOAT` data due to increased message passing for distribution of size-varying data chunks. In contrast, for low stripe counts 4, 8, the improvement of the data throughput is clearly visible, as less aggregate bandwidth to storage nodes is available. Further, as the memory footprint of compressed low-resolution data decreases even more, the storage of additional multi-resolution data introduces a significant overhead in I/O times.

In order to improve the *Time-to-Analysis* w.r.t. *storage & transfer* of uncompressed `FLOAT` data, the time saved through transfer of compressed data must be at least as large as the run-time required for compression and decompression. Depending on the speed of the network connection, which is used for movement of the data away from HPC scratch storage, a speed-up of the scientific workflow by a factor of at least ~1.4–4.3× is achieved when using a network connection between 1 Gbit/s, 2.5 Gbit/s, …, 10 Gbit/s transfer speed and $k_{\mathrm{D}} = 4$, $k_{\mathrm{MOD}} = 16$ for temporal compression. Even larger improvements are expected to be possible when using a run-time optimized implementation of t-GLATE.

As a concrete example, for the application scenario motivated in Sections 5.2 and 5.4, the simulation results of metal melt casting simulations are moved away from HPC scratch storage in *Dresden* into local visualization workstations in *Freiberg* using 1 Gbit/s. According to Fig. 5.19 (*b*), moving data using a 1 Gbit/s $\simeq$ 128 MB/s connection allows for the *"simulation, compression, storage, transfer & decompression"* of ~4.3× more time steps, as compared to *"simulation, storage & transfer"* of uncompressed `FLOAT` data, i.e. for 1,000 time steps 1.4 hours instead of 6.2 hours, and for 10,000 time steps 14.4 hours instead of 2.6 days when using t-GLATE temporal in-situ compression with $k_{\mathrm{D}} = 4$, $k_{\mathrm{MOD}} = 16$. Concluding, t-GLATE is a viable option for decreasing the *Time-to-Analysis* for data-intensive HPC applications through *Temporal Lossy In-Situ Compression*.

6 Conclusion and Future Work

In this chapter, first, a brief summary of the research in this dissertation is given. Second, the contributions of this dissertation are summarized, and third, future research directions are proposed, which relate to the integration of *Lossy In-Situ Compression* and *Temporal Compression* into future *Data Management* and *Data Processing* applications.

6.1 Research Summary

During the research for this dissertation, the I/O bottleneck in the scientific workflow was reduced by employing *Lossy In-Situ Data Compression* and *Temporal Compression* inside a large-scale CFD simulation run on the *Taurus* HPC *Cluster*. The newly developed compression methods achieved a high data reduction during the running CFD simulation, and the data was stored in the parallel Lustre$^{(\mathrm{R})}$ file system with a much lower memory footprint. As the data is compressed, transferred and decompressed faster than the time required for sole transfer of full-size data, the scientific workflow is accelerated.

Development and Implementation of New Algorithms

The newly developed GLATE algorithm as well as the temporal compression procedure t-GLATE allow for fast compression of time series of CFD data sets. GLATE achieved a fourfold to fivefold data reduction during non-temporal compression, and t-GLATE achieved a sixfold to twelvefold data reduction during temporal compression of flow fields from a metal melt casting process. In order to ensure the data quality for scientific applications, GLATE and t-GLATE restrict the maximum decompression error e_{MAX} in percent, as described in Section 3.2. Further, GLATE and t-GLATE establish a trade-off between the maximum allowed decompression error, the temporal resolution of the decompressed data, and the resulting compression rate.

Specifically, GLATE and *t*-GLATE use scalar quantization and are developed based on the ideas of the existing compression algorithms ISABELA, *d*-ISABELA [45] and SBD [36]. Two other temporal compression procedures for ISABELA and SBD, namely *t*-ISABELA and *t*-SBD have been developed and implemented before *t*-GLATE. However, the *t*-GLATE temporal compression procedure outperformed the other implementations w.r.t. compression rate and compression speed. As compared to ISABELA and SBD, GLATE decomposes `FLOAT` values into two streams containing smoothly fluctuating exponents and noisy signs and mantissas. By employing different real-time lossless compression techniques, *t*-GLATE achieves a higher data reduction for SFPD.

Collective Write of Compressed Data

The application scenario is motivated by the CRC920, where large-scale CFD simulations are conducted for analysis and visualization of the flow field dynamics in complex porous filter structures. During the parallel CFD simulation the performance of data writes into the parallel Lustre$^{(\mathrm{R})}$ file system depends on the stripe count used for the creation of shared files, i.e. the effective bandwidth for moving the data from MPI processes into OSTs is the aggregate of many designated transfer channels. Hence, parallel storage systems resemble a scaling of throughput according to the reduction of the data size only if the data is stripe-aligned. For high stripe counts of 16, 32 stripes, the storage of uncompressed `FLOAT` data is accelerated according to the available aggregate bandwidth. In contrast, moving *t*-GLATE compressed data with high stripe counts did not yield a decrease in I/O times, as merging of size-varying data fragments resulted in increased message passing overhead for distribution of data across storage nodes. However, for lower stripe counts of 4, 8 stripes, the write of compressed data was faster than the write of stripe-aligned full-size data.

Improvement of Time-to-Analysis

Based on the evaluation results, the improvement of the *Time-to-Analysis* in the scientific method cycle was discussed and quantified. As shown in Fig. 6.1, the data sets have been stored with a reduced memory footprint of ~8.4–16.0 %$_{\mathrm{32b}}$ using *t*-GLATE temporal compression during the CFD simulation. As stated above, using high stripe counts did not decrease the data throughput as compared to uncompressed `FLOAT` data. However, when the results are moved away from temporary HPC scratch storage into local storage, compressed data is moved ~6.25–12× times faster than uncompressed `FLOAT` data. Using a 1 Gbit/s ≃ 128 MB/s network connection, the scientific workflow is accelerated by a factor of about ~4.3×, i.e. *"simulate, compress, store, transfer & decompress"* using t-GLATE happenes faster than *"simulate, store & transfer"* of full-size `FLOAT` data.

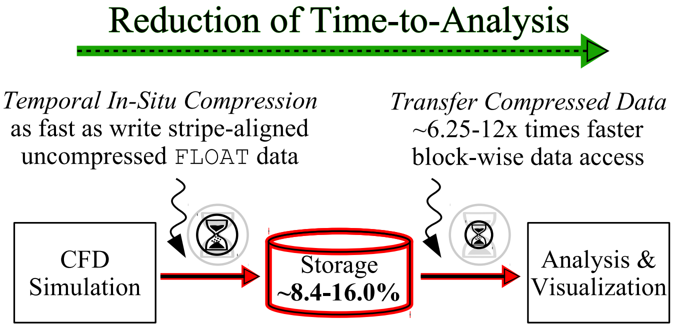


Figure 6.1: Improvement of *Time-to-Analysis* in the scientific workflow. Fast *Temporal Lossy In-situ Compression* reduces the memory footprint during the simulation. The I/O times for writing compressed data were similar to writing stripe-aligned uncompressed `FLOAT` data into the Lustre$^{(R)}$ paralell file system. Temporal compressed data allows for faster transfer and decompression using t-GLATE, as compared to transfer of uncompressed `FLOAT` data.

Comparison to Lossy Float Compressor ZFP

During the development of GLATE and t-GLATE, the lossy float compressor `ZFP` [63] has been published, which is optimized for the compression of uniform grids. `ZFP` allows for near-lossless compression, and for restriction of the mean relative/absolute error of decompressed data. Concerning the error policy, as motivated in Section 3.2, p. 30, the compression performance of t-GLATE is compared to `ZFP` w.r.t. to the same error-specific compression task, i.e. the restriction of the maximum relative error.

For compression tests on uniform grids, t-GLATE achieved a better compression rate through temporal compression, as shown in Fig. 4.26, p. 125, e.g. for maximum decompression error of $e_{\text{MAX}} = 1.00\,\%$, `ZFP` on precision level 16 yields ~14.2 $\%_{32b}$, whereas t-GLATE yields an improved compression rate of ~8.5 $\%_{32b}$ through temporal compression.

Further, for the compression of data stored in non-uniform grids like unstructured grids, it was shown exemplarily, that t-GLATE achieves a better compression rate than

`ZFP`, as shown in Fig. 4.29, p. 130. For the compression of unstructured grid data t-GLATE achieved a compression rate of ~11.1 $\%_{32\mathrm{b}}$ for $e_{\mathrm{MAX}} = 1.00\,\%$, whereas `ZFP`, without knowledge of the underlying grid topology, yielded a worse comprerssion rate of $\geq$30 $\%_{32\mathrm{b}}$ at worse error rates using precision level 14.

Conclusion

The evaluation of t-GLATE in a modern HPC environment showed that lossy temporal in-situ compression decreases the *Time-to-Analysis* for the scientific workflow within the CFD setup described in Section 5.2. Although the parallel file system makes it difficult to speed up writing of compressed data when using high stripe counts, the transmission of the temporally compressed data from the cluster into workstations occurs much faster than transmission of full-size raw `FLOAT` data, i.e. data can be analyzed earlier.

Further, the architecture of t-GLATE allows for the easy adoption of arbitrary grid types. If the grid linearization conserves the spatial coherence of the data, GLATE achieves a very good compression performance through using carefully selected real-time lossless compression algorithms for compression of internal data.

Although the current implementation of t-GLATE introduces a small run-time penalty, t-GLATE yields very good compression rates while restricting the maximum decompression error. Summarizing, t-GLATE constitutes a powerful compression algorithm for temporal sequences of SFPD generated during CFD simulations.

6.2 Research Contributions

The objective of this research is the reduction of the *Time-to-Analysis* in the scientific workflow through employing *Lossy In-Situ Data Compression*. Specifically, the research focuses on the development of new schemes for *Temporal Compression* for time series of CFD simulation data. Using the new GLATE and t-GLATE algorithms, the spatial and temporal coherence being inherently present in continuous CFD processes has been exploited for successful improvement of the compression rate. In this regard, the main contribution of this work is the new scalar quantization algorithm GLATE and the temporal compression procedure t-GLATE, which allow for fast compression at very good compression rates while restricting the maximum decompression error e_{MAX} for SFPD.

The contributions of this work are summarized in the following:

Algorithmic Contributions

1. Development and implementation[1] of the *Lossy In-Situ Compressor* GLATE based on the ideas of ISABELA [45] and SBD [36]. GLATE decomposes `FLOAT` values into smoothly fluctuating exponents and noisy signs/mantissas. Through the application of different real-time lossless compressors on the internal data, i.e. `zstd` for exponents and `tpfor` for noisy mantissas, GLATE achieves very good compression rates at high compression speeds and outperforms ISABELA and SBD.

2. Development and implementation[2] of the *Temporal Compression Procedures* t-ISABELA, t-SBD and t-GLATE based on the idea of d-ISABELA [45]. t-GLATE achieves a the best compression rate at high compression speed by employing a difference encoding on a global step function and outperforms d-ISABELA, t-ISABELA and t-SBD.

3. Development and implementation[3] of the *Optimized Compression Procedure* for KFs based on the idea of the t-GLATE difference encoding used in DFs. The optimized t-GLATE algorithm for KFs takes into account spatial coherence between consecutive values in the linearized input stream by employing the same difference encoding in KFs. The compression rate is improved by ~2.0–3.5 %$_{\text{32b}}$.

Evaluation of Lossless Compression Back End

Design decisions for the development of new in-situ compressors GLATE and t-GLATE were informed by comprehensive performance evaluations of state-of-the-art lossless compressors in the CFD context.

1. Evaluation of the compression performance of different GPLCs on three different `FLOAT` CFD data sets using different grid linearization schemes. Particularly, data

---

[1] The pseudo codes for ISABELA, SBD and GLATE are given in Alg. 3.1, p. 39, Alg. 3.2, p. 44 and Alg. 3.3, p. 54.

[2] The d-ISABELA, t-ISABELA, t-SBD, t-GLATE pseudo codes for the compression procedure in KFs are given in Alg. 4.1, p. 88, Alg. 4.3, p. 95, Alg. 4.5, p. 101, Alg. 4.7, p. 108, and in DFs are given in Alg. 4.2, p. 89, Alg. 4.4, p. 96, Alg. 4.6, p. 102, Alg. 4.8, p. 109.

[3] The pseudo code for the optimized t-GLATE compression procedure in KFs is given in Alg. 4.9, p. 112.

was linearized using the column/row-major linearization, the Z-curve, and the Hilbert-curve, and was compressed using `zstd`, `bzip`, `lzma` and `zstd`. The usage of the Hilbert-curve yields an improvement of the compression rate by ~3 %$_{\text{32b}}$.

2. Comprehensive evaluation of compression performance for currently available state-of-the-art GPLCs w.r.t. to the application as back end for lossy in-situ compression in the CFD context. `zlib`, `bzip`, `lzma`, `lz4`, `snappy`, `tpfor`, `zstd`, `bsc`, `snappy`, `tpfor` have been tested on signed/unsigned integer data w.r.t. the application as back end for lossy `FLOAT` compression algorithms. `zstd` on level 2 and 6, `lz4` on level 4 and 6, `snappy` and `tpfor` have been selected as candidates for the lossless compression of internal data of lossy in-situ compressors.

3. Evaluation of different real-time GPLCs for compression of internal data of ISABELA, SBD and GLATE. Using `tpfor` and `zstd` on level 6, GLATE yields the best compression rate of ~21.6–22.2 %$_{\text{32b}}$ for $e_{\text{MAX}} = 1.00\,\%$ on three CFD data sets and outperforms ISABELA and SBD w.r.t. compression rate and compression run-time. Using `zstd` on level 2 or `snappy` for compression of exponents increases the compression speed of GLATE while imposing a ~0.5–3.0 %$_{\text{32b}}$ penalty on the compression rate.

4. Evaluation of the *Temporal Compression Procedures* t-ISABELA, t-SBD and t-GLATE on a high-resolution temporal CFD data set. Using `zstd` and `tpfor`, t-GLATE temporal compression yields the best compression rate of ~7.9–15.3 %$_{\text{32b}}$ for $e_{\text{MAX}} = 1.00\,\%$ and outperforms d-ISABELA, t-ISABELA and t-SBD w.r.t. compression rate and compression speed.

Comparison to ZFP Lossy Float Compressor

`ZFP` is a recent state-of-the art lossy in-situ compressor optimized for the compression of data in uniform grids. Unlike GLATE and t-GLATE, `ZFP` ensures the mean relative error of decompressed data, instead of the maximum relative error.

1. Evaluation of compression performance of `ZFP` [63] on *Uniform Grid* and comparison to t-GLATE w.r.t. to compression rate and the maximum relative error of decompressed data. `ZFP` is operated in the optimized mode for three dimensional uniform grid data. Using precision level 16, `ZFP` achieves a compression rate of

~14.2 $\%_{32b}$, where ~0.2 % of data values $|x| > 1E{-}4$ exhibit a decompression error greater than 1.00 %. Using temporal compression, t-GLATE achieves an improved compression rate of ~8.5–12.9 $\%_{32b}$ for $e_{\text{MAX}} = 0.10, \ldots, 1.00\,\%$ while guaranteeing the error for all data values.

2. Evaluation of compression performance of `ZFP` [63] on *Unstructured Grid* and comparison to t-GLATE w.r.t. to compression rate and the maximum error of decompressed data. `ZFP` is operated on a one dimensional sequence of data without knowledge of the underlying grid. `ZFP` using precision level 15 achieves a compression rate of ~32.9 $\%_{32b}$, whereas t-GLATE achieves a better compression rate of ~11.1–18.2 $\%_{32b}$ for $e_{\text{MAX}} = 0.10, \ldots, 1.00\,\%$ while guaranteeing the error for all data values.

In-Situ Evaluation of t-GLATE in Taurus HPC Cluster

1. *In-Situ Application* of t-GLATE in large-scale LBM simulation using 512 MPI processes. t-GLATE yields the best compression rate of ~8.4–16.0 $\%_{32b}$. `ZFP` using precision level 15 yields a compression rate of ~16.7 $\%_{32b}$. The current implementation of t-GLATE introduces a small run-time penalty for decomposition of `FLOAT` data into two streams for exponents and signs/mantissas.

2. Evaluation of the speed of writing data from 512 MPI processes into the Lustre$^{(R)}$ parallel file system using collective I/O with high and low stripe counts. Using high stripe counts of 16, 32 stripes, the stripe-aligned uncompressed `FLOAT` data is written out with a high aggregate bandwidth as fast as compressed data. Using low stripe counts of 4, 8 stripes, the write of compressed data is faster.

3. Discussion of the improvement of the *Time-to-Analysis* through the application of t-GLATE in the scientific method cycle. Based on the evaluation results, the worst case is assumed, i.e. no faster write due to using high stripe counts and a small run-time penalty for t-GLATE compression. Using the current implementation of t-GLATE, *"simulate, compress, store, transfer & decompress"* happened ~4.3× faster than *"simulate, store & transfer"* of uncompressed `FLOAT` data when using a 1 Gbit/s $\simeq$ 128 MB/s network connection for moving away data from temporary HPC storage.

6.3 Future Work

t-GLATE lossy in-situ compression accelerates the scientific workflow through the reduction of storage costs for CFD data sets, and through decreasing transfer times of compressed data sets, as explained in Section 5.5, p. 167. In the following, two directions for further integration of *t*-GLATE into the scientific workflow are proposed. First, the integration of temporal compression into popular scientific data formats and I/O libraries, and second, the improvement of data access mechanisms by using an in-situ generated index for selective decompression [46].

6.3.1 Tool Integration for Temporal Compression

Temporal lossy in-situ compression yields very good compression rates and directly contributes to the feasibility of data capture in high-resolution CFD simulations. Especially if the acquisition of additional storage resources is too expensive, temporal in-situ compression is a viable option for reducing storage requirements. Therefore, it is desirable to provide fault tolerant and transparent implementation of temporal compression which simplifies the integration into existing applications.

Different scientific file formats, e.g. the *HDF5* [33] file format and the formats of the *Open-Source Visualization Toolkit* (VTK) [91], as well as, different middlewares for scientific applications and collective I/O, e.g. *ADIOS* [64] and *MLOC* [26], offer plug-in mechanisms for integrated compression of `FLOAT` and `DOUBLE` data stored in simulation grids. The tool integration of temporal compression would allow for usage of metadata from file formats which is associated with e.g. the underlying numerical grid, or the time step of the data, while leveraging optimized algorithms for data processing and data management.

6.3.2 Selective Decompression using In-situ generated Index

As described in Section 5.3.4, the support for compression into a multi-resolution data format facilitates access mechanisms with direct control of the amount of data loaded for visualization tasks. The amount of data is controlled by the pre-defined levels of resolution in which the data is stored. However, if the locations to be loaded are unknown, then the spatial grid locations accommodating the values of interest have to be determined by using expensive linear search in the entire full-resolution data set. If the data set is compressed, the whole data set needs to be decompressed in order to execute linear search. Such workflows may greatly benefit from an *In-Situ Generated Index* allowing for *Selective Decompression* during post-processing.

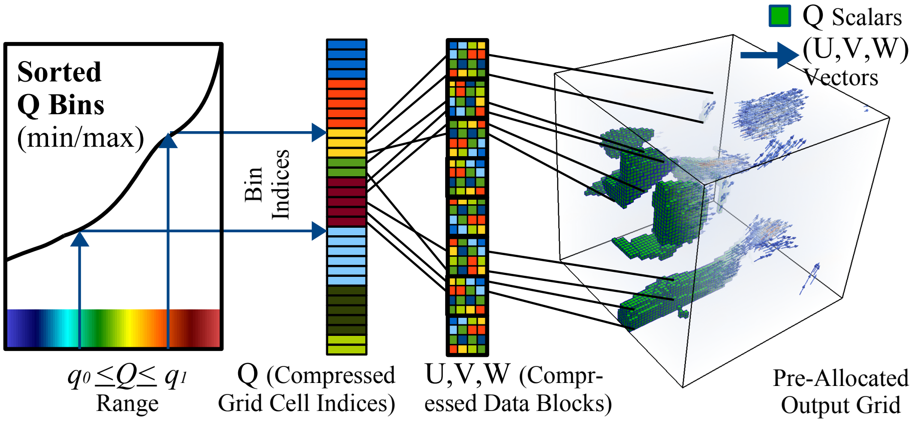

Sorted
Q Bins
(min/max)
Bin Indices
$q_0 \leq Q \leq q_1$
Range
Q (Compressed Grid Cell Indices)
U,V,W (Compressed Data Blocks)
Q Scalars
(U,V,W) Vectors
Pre-Allocated Output Grid

Figure 6.2: Selective decompression by using an in-situ generated index. The GLATE quantization on the global step function can be employed for binning during generation of the *Inverted Index*. Using *Range Queries* on e.g. the Q variable describing vortex characteristics, the indices of grid cells containing the data of interest are loaded and used to steer the decompression of flow vectors for analysis of the flow field.

By employing a so-called *Inverted Index* [46], a compressed *Write-Once/Read-Often* database for scientific data can be realized, as illustrated in Fig. 6.2. The t-GLATE global quantization mechanism can be employed for error-bounded binning and collecting of grid locations with same data values. Similar to SBD with *Differential Encoding* [36], the index structure can be compressed efficiently using `tpfor`. Using *Range Querys*, e.g. $q_0 \leq Q \leq q_1$ for indicating vortexes using the Q-criterion, the index allows for the fast selection of grid locations, which can be used to steer the decompression of t-GLATE compressed contents during post-processing.

Summarizing, the combination of an in-situ generated index with temporal in-situ compression allows for database-like access to compressed contents. By employing selective decompression, the *Time-to-Analysis* can further be reduced through minimizing the amount of data to be loaded and decompressed for analysis.

Bibliography

[1] 7Zip. *LZMA - general-purpose data compression software with a high compression ratio*. URL: `http://7-zip.org/sdk.html`.

[2] Martin Abendroth et al. “An approach towards numerical investigation of the mechanical behaviour of ceramic foams during metal melt filtration processes”. In: *Advanced Engineering Materials* 19 (2017).

[3] J. Ahrens et al. “An Image-Based Approach to Extreme Scale in Situ Visualization and Analysis”. In: *High Performance Computing, Networking, Storage and Analysis, SC14: International Conference for*. Nov. 2014, pp. 424–434.

[4] James Ahrens, David Rogers, and Becky Springmeyer. “Visualization and Data Analysis at the Exascale”. In: National Nuclear Security Administration (NNSA) Accelerated Strategic Computing (ASC) Exascale Environment Planning Process, 2011.

[5] Cyrus K Aidun and Jonathan R Clausen. “Lattice-Boltzmann Method for Complex Flows”. In: *Annual Review of Fluid Mechanics* 42 (2010), pp. 439–472.

[6] F. Benboubker, F. Abdi, and A. Ahaitouf. “Shape-Adaptive Motion Estimation Algorithm for MPEG-4 Video Coding”. In: *CoRR* abs/1002.1168 (2010). arXiv: `1002.1168`.

[7] Janine C. Bennett et al. “Combining In-situ and In-transit Processing to Enable Extreme-scale Scientific Analysis”. In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC ’12. Salt Lake City, Utah: IEEE Computer Society Press, 2012, 49:1–49:9. ISBN: 978-1-4673-0804-5.

[8] Janine C. Bennett et al. “Trigger detection for adaptive scientific workflows using percentile sampling”. In: *CoRR* abs/1506.08258 (2015).

[9] Martin Burtscher and Paruj Ratanaworabhan. “FPC: A High-Speed Compressor for Double-Precision Floating-Point Data”. In: *IEEE Trans. Comput.* 58.1 (Jan. 2009), pp. 18–31. ISSN: 0018-9340.

[10] Hamid Buzidi. *TurboPFOR - fastest integer compression.* URL: `http://github.com/powturbo/TurboPFor`.

[11] Y. M. Camacho et al. “Development of a robust and efficient biogas processor for hydrogen production. Part 1: Modelling and simulation”. In: *International Journal of Hydrogen Energy* 42 (2017), pp. 22841–22855.

[12] Chun-Hung Cheng, Ada Waichee Fu, and Yi Zhang. “Entropy-based subspace clustering for mining numerical data”. In: *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining.* KDD ’99. San Diego, California, United States: ACM, 1999, pp. 84–93. ISBN: 1-58113-143-7.

[13] Wai-Ki Ching and Michael K. Ng. *Markov Chains: Models, Algorithms and Applications (International Series in Operations Research & Management Science).* 1st ed. Springer, Dec. 2005. ISBN: 0387293353. URL: `http://www.worldcat.org/isbn/0387293353`.

[14] Jerry Chou, Kesheng Wu, and Prabhat. “FastQuery: A Parallel Indexing System for Scientific Data”. In: *2013 IEEE International Conference on Cluster Computing (CLUSTER)* 0 (2011), pp. 455–464.

[15] Yann Collet. *LZ4 - fast lossless compression algorithm with an extremely fast decoder.* URL: `http://lz4.github.io/lz4/`.

[16] Peter Corbett et al. “Overview of the MPI-IO Parallel I/O Interface”. In: *Proceedings of the IPPS ’95 Workshop on Input/Output in Parallel and Distributed Systems.* 1995, pp. 1–15.

[17] Thomas M. Cover and Joy A. Thomas. *Elements of information theory.* New York, NY, USA: Wiley-Interscience, 1991. ISBN: 0-471-06259-6.

[18] L. Peter Deutsch and Jean-Loup Gailly. *ZLIB Compressed Data Format Specification version 3.3.* Internet RFC 1950.

[19] Yinlin Dong, Yong Yang, and Chaoqun Liu. *DNS Study on Three Vortex Identification Methods.* Tech. rep. Department of Mathematics, University of Texas at Arlington, July 2016.

[20] Soumya Dutta et al. “In Situ Distribution Guided Analysis and Visualization of Transonic Jet Engine Simulations”. In: *IEEE Transactions on Visualization & Computer Graphics* 23.1 (2017), pp. 811–820. ISSN: 1077-2626.

[21] Hervé Duval et al. “Simulation of Aluminum Filtration including Lubrication Effect in three-dimensional Foam Microstructures”. In: *Light Metals 2007*. Warrendale, PA, USA: The Minerals, Metals & Materials Society, 2007, pp. 645–650.

[22] Facebook. *ZStd - real-time compression algorithm providing high compression ratios and very fast decoder*. URL: `http://facebook.github.io/zstd/`.

[23] Thomas Fogal et al. “Freeprocessing: Transparent in situ Visualization via Data Interception”. In: *Eurographics Symposium on Parallel Graphics and Visualization*. Ed. by Margarita Amor and Markus Hadwiger. The Eurographics Association, 2014. ISBN: 978-3-905674-59-0.

[24] Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*. Tech. rep. 2003.

[25] Tim Golla and Reinhard Klein. “Real-time Point Cloud Compression”. In: *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Congress Center Hamburg, Germany, Oct. 2015.

[26] Zhenhuan Gong et al. “MLOC: Multi-level Layout Optimization Framework for Compressed Scientific Data Exploration with Heterogeneous Access Patterns”. In: *ICPP*. IEEE Computer Society, 2012, pp. 239–248. ISBN: 978-1-4673-2508-0.

[27] Google. *Brotli - generic-purpose lossless compression algorithm that compresses data using a combination of a modern variant of the LZ77 algorithm, Huffman coding and 2nd order context modeling*. URL: `https://github.com/google/brotli`.

[28] Google. *Snappy - very fast data compression and decompression library*. URL: `http://google.github.io/snappy/`.

[29] Google. *Zopfli - encoder with more dense compression for existing zlib or deflate libraries*. URL: `https://github.com/google/zopfli`.

[30] Ilya Grebnov. *libbsc - high performance block-sorting data compression library*. URL: `http://libbsc.com/`.

[31] Reid Gryder and Kerry Hake. *SURVEY OF DATA COMPRESSION TECHNIQUES*. Tech. rep. OAK RIDGE NATIONAL LABORATORY, Oak Ridge, Tennessee 37831, 1991.

[32] B. G. Haskell and A. Puri. “MPEG Video Compression Basics”. In: *The MPEG Representation of Digital Media*. Ed. by Leonardo Chiariglione. New York, NY: Springer New York, 2012. ISBN: 978-1-4419-6184-6.

[33] HDF5. *Hierarchical Data Format*. URL: `http://www.hdfgroup.org/HDF5/`.

[34] Mohammad Hosseini. *A Survey of Data Compression Algorithms and their Applications*. Tech. rep. School of Computing Science, Simon Fraser University, Canada, 2012.

[35] David Huffman. “A method for the construction of minimum-redundancy codes”. In: *Resonance* 11.2 (Feb. 2006), pp. 91–99. ISSN: 0971-8044.

[36] Jeremy Iverson, Chandrika Kamath, and George Karypis. “Fast and effective lossy compression algorithms for scientific datasets”. In: *Proceedings of the 18th international conference on Parallel Processing*. Euro-Par’12. Rhodes Island, Greece: Springer-Verlag, 2012, pp. 843–856. ISBN: 978-3-642-32819-0.

[37] John Jenkins et al. “ALACRITY: Analytics-Driven Lossless Data Compression for Rapid In-Situ Indexing, Storing, and Querying”. In: *Transactions on Large-Scale Data- and Knowledge-Centered Systems X: Special Issue on Database- and Expert-Systems Applications*. Ed. by Abdelkader Hameurlain et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 95–114. ISBN: 978-3-642-41221-9.

[38] S. Jian, L. Zhan-huai, and Z. Xiao. “The performance optimization of Lustre file system”. In: *2012 7th International Conference on Computer Science Education (ICCSE)*. 2012, pp. 214–217.

[39] Chae Jubb. *Loop Optimizations in Modern C Compilers*. Tech. rep. Columbia Engineering, Computer Science Department, New York, 2014.

[40] Jinoh Kim et al. “Parallel in situ indexing for data-intensive computing”. In: *IEEE Symposium on Large Data Analysis and Visualization, LDAV 2011, Providence, Rhode Island, USA, 23-24 October, 2011*. IEEE, 2011, pp. 65–72.

[41] Jusub Kim and Joseph Jaja. *A Novel Information-Aware Octree for the Visualization of Large Scale Time-varying Data*. Tech. rep. 2006.

[42] James Kress et al. “Loosely Coupled In Situ Visualization: A Perspective on Why It’s Here to Stay”. In: *Proceedings of the First Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*. ISAV2015. Austin, TX, USA: ACM, 2015, pp. 1–6. ISBN: 978-1-4503-4003-8.

[43] Julian Kunkel, Michael Kuhn, and Thomas Ludwig. “Exascale Storage Systems - An Analytical Study of Expenses”. In: *Supercomput. Front. Innov.: Int. J.* 1.1 (Apr. 2014), pp. 116–134. ISSN: 2409-6008.

[44] S. Lakshminarasimhan et al. “ISABELA-QA: Query-driven analytics with ISABELA-compressed extreme-scale scientific data”. In: *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*. Nov. 2011, pp. 1 –11.

[45] Sriram Lakshminarasimhan et al. “Compressing the Incompressible with ISABELA: In-situ Reduction of Spatio-temporal Data.” In: *Euro-Par (1)*. Ed. by Emmanuel Jeannot, Raymond Namyst, and Jean Roman. Vol. 6852. Lecture Notes in Computer Science. Springer, 2011, pp. 366–379. ISBN: 978-3-642-23399-9.

[46] Sriram Lakshminarasimhan et al. “DIRAQ: scalable in situ data- and resource-aware indexing for optimized query performance”. In: *Cluster Computing* 17.4 (2014), pp. 1101–1119. ISSN: 1573-7543.

[47] Sriram Lakshminarasimhan et al. “ISABELA for Effective In-situ Compression of Scientific Data”. In: *Concurrency and Computation: Practice and Experience* (2012). Invited paper.

[48] Sriram Lakshminarasimhan et al. “Scalable in Situ Scientific Data Encoding for Analytical Query Processing”. In: *Proceedings of the 22Nd International Symposium on High-performance Parallel and Distributed Computing*. HPDC ’13. New York, New York, USA: ACM, 2013, pp. 1–12. ISBN: 978-1-4503-1910-2.

[49] Aaditya G. Landge et al. “In-situ Feature Extraction of Large Scale Combustion Simulations Using Segmented Merge Trees”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC ’14. New Orleans, Louisana: IEEE Press, 2014, pp. 1020–1031. ISBN: 978-1-4799-5500-8.

[50] Daniel Laney et al. “Assessing the Effects of Data Compression in Simulations Using Physically Motivated Metrics”. In: *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis.* SC ’13. Denver, Colorado: ACM, 2013, 76:1–76:12. ISBN: 978-1-4503-2378-9.

[51] Robert Latham, Robert B. Ross, and Rajeev Thakur. “The Impact of File Systems on MPI-IO Scalability”. In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 11th European PVM/MPI Users’ Group Meeting, Budapest, Hungary, September 19-22, 2004, Proceedings.* 2004, pp. 87–96.

[52] Henry Lehmann, Katja Fiedler, and Bernhard Jung. “Processing In-Situ Compressed Large Data Sets in VR-based Flow Analysis”. In: *Virtuelle und Erweiterte Realität - 9. Workshop der GI-Fachgruppe VR/AR 2012, Düsseldorf, Deutschland, September 19-20, 2012. Proceedings.* Ed. by Christian Geiger, Jens Herder, and Tom Vierjahn. Shaker, 2012, pp. 61–70. ISBN: 978-3-8440-1309-2.

[53] Henry Lehmann and Bernard Jung. “Applying In-Situ Compression to Hierarchical Scientific Voxel Data”. In: vol. 1. EDIS - Publishing Institution of the University of Zilina, 2012, pp. 1953–1958. ISBN: 978-80-554-0606-0.

[54] Henry Lehmann and Bernard Jung. “In-situ data compression of Flow in pourous media”. In: *PDPTA’12 - 18th The International Conference on Parallel and Distributed Processing Techniques and Applications.* 2012.

[55] Henry Lehmann and Bernhard Jung. “In-situ multi-resolution and temporal data compression for visual exploration of large-scale scientific simulations”. In: *4th IEEE Symposium on Large Data Analysis and Visualization, LDAV 2014, Paris, France, November 9-10, 2014.* 2014, pp. 51–58.

[56] Henry Lehmann, Matthias Lenk, and Bernhard Jung. “Adding Interlacing to In-Situ Data Compression for Multi-Resolution Visualization of Large-Scale Scientific Simulations”. In: *ICAT-EGVE 2014 - Posters and Demos.* Ed. by Yuki Hashimoto et al. The Eurographics Association, 2014. ISBN: 978-3-905674-77-4.

[57] Henry Lehmann, Eric Werzner, and Christian Degenkolb. “Optimizing In-situ Data Compression for Large-scale Scientific Simulations”. In: *Proceedings of the 24th High Performance Computing Symposium.* HPC ’16. Pasadena, California: Society for Computer Simulation International, 2016, 5:1–5:8. ISBN: 978-1-5108-2318-1.

[58] Henry Lehmann et al. “In Situ Data Compression Algorithm for Detailed Numerical Simulation of Liquid Metal Filtration through Regularly Structured Porous Media”. In: *Advanced Engineering Materials* 15.12 (2013), pp. 1260–1269. ISSN: 1527-2648.

[59] Daniel Lemire and Leonid Boytsov. “Decoding billions of integers per second through vectorization”. In: *CoRR* abs/1209.2137 (2012). arXiv: `1209.2137`. URL: `http://arxiv.org/abs/1209.2137`.

[60] Daniel Lemire, Leonid Boytsov, and Nathan Kurz. “SIMD Compression and the Intersection of Sorted Integers”. In: *CoRR* abs/1401.6399 (2014).

[61] Yi Li et al. *A public turbulence database cluster and applications to study lagrangian evolution of velocity increments in turbulence*. 2008.

[62] Kuan-Wu Lin et al. “Optimizing Fastquery Performance on Lustre File System”. In: *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*. SSDBM. Baltimore, Maryland, USA: ACM, 2013, 29:1–29:12. ISBN: 978-1-4503-1921-8.

[63] P. Lindstrom. “Fixed-Rate Compressed Floating-Point Arrays”. In: *IEEE Transactions on Visualization and Computer Graphics* 20.12 (2014), pp. 2674–2683. ISSN: 1077-2626.

[64] Jay F. Lofstead et al. “Flexible IO and Integration for Scientific Codes Through the Adaptable IO System (ADIOS)”. In: *Proceedings of the 6th International Workshop on Challenges of Large Applications in Distributed Environments*. CLADE ’08. Boston, MA, USA: ACM, 2008, pp. 15–24. ISBN: 978-1-60558-156-9.

[65] Kwan-Liu Ma et al. “In-situ processing and visualization for ultrascale simulations”. In: *Journal of Physics: Conference Series* 78.1 (July 2007), pp. 012043+. ISSN: 1742-6596.

[66] Matt Mahoney. *ZPAQ - incremental journaling backup utility and archiver*. URL: `http://mattmahoney.net/dc/wwwzpaq2015.html`.

[67] Miguel A. A. Mendes et al. “Measurement and simplified numerical prediction of effective thermal conductivity of open-cell ceramic foams at high temperature”. In: *Int. J. Heat Mass Tran.* 102 (2016), pp. 396–406.

[68] A. Moffat and R. Isal. “Word-based text compression using the Burrows-Wheeler transform”. In: *Information Processing & Management* 41.5 (Sept. 2005), pp. 1175–1192. ISSN: 03064573.

[69] Bongki Moon et al. “Analysis of the clustering properties of the Hilbert space-filling curve”. In: *IEEE Transactions on Knowledge and Data Engineering* 13 (2001), p. 2001.

[70] Markus Oberhumer. *LZO - general purpose compression with higher compression and decompression speed*. URL: `http://www.oberhumer.com/opensource/lzo/`.

[71] Jean-Pierre Prost et al. “MPI-IO/GPFS, an Optimized Implementation of MPI-IO on Top of GPFS”. In: *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*. SC ’01. Denver, Colorado: ACM, 2001, pp. 17–17. ISBN: 1-58113-293-X.

[72] Tom Richardson and Ruediger Urbanke. *Modern Coding Theory*. Cambridge University Press, Mar. 2008. ISBN: 0521852293. URL: `http://www.worldcat.org/isbn/0521852293`.

[73] Marzia Rivia et al. *In-situ Visualization: State-of-the-art and Some Use Cases*. Whitepaper. 2012.

[74] Steven J. Ross. “The Spreadsort High-performance General-case Sorting Algorithm”. In: *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA ’02, June 24 - 27, 2002, Las Vegas, Nevada, USA, Volume 3*. 2002, pp. 1100–1106.

[75] Philip Schwan. “Lustre: Building a File System for 1,000-node Clusters”. In: *PROCEEDINGS OF THE LINUX SYMPOSIUM*. 2003, p. 9.

[76] Julian Seward. *The bzip2 compression library*. URL: `http://www.bzip.org/`.

[77] Charles Alexander Smith. *A Survey of Various Data Compression Techniques*. Tech. rep. 2010.

[78] Rajeev Thakur and Alok Choudhary. “An Extended Two-Phase Method for Accessing Sections of Out-of-Core Arrays”. In: *Scientific Programming* 5 (1996), pp. 301–317.

[79] Rajeev Thakur, William Gropp, and Ewing Lusk. “A Case for Using MPI’s Derived Datatypes to Improve I/O Performance”. In: *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing.* SC ’98. San Jose, CA: IEEE Computer Society, 1998, pp. 1–10. ISBN: 0-89791-984-X.

[80] Yuan Tian et al. “neCODEC: nearline data compression for scientific applications”. In: *Cluster Computing* 17.2 (2013), pp. 475–486. ISSN: 1573-7543.

[81] Michael Triep et al. “Up-scaled Dynamical Model of the Human Vocal Folds”. In: *Proceedings in Applied Mathematics and Mechanics (PAMM)* 8 (2008), pp. 10643–10644.

[82] Gaurav Vijayvargiya, Sanjay Silakari, and Rajeev Pandey. “A Survey: Various Techniques of Image Compression.” In: *CoRR* abs/1311.6877 (2013).

[83] Yang Wang, Hongfeng Yu, and Kwan-Liu Ma. “Scalable Parallel Feature Extraction and Tracking for Large Time-varying 3D Volume Data”. In: *EGPGV.* 2013, pp. 17–24.

[84] T. A. Welch. “A Technique for High-Performance Data Compression”. In: *Computer* 17.6 (June 1984), pp. 8–19. ISSN: 0018-9162.

[85] E. Werzner et al. “Numerical Investigation on the Depth Filtration of Liquid Metals: Influence of Process Conditions and Inclusion Properties”. In: *Journal of Advanced Engineering Materials* 15 (2013), pp. 1307–1314.

[86] Pak Chung Wong et al. “The Top 10 Challenges in Extreme-Scale Visual Analytics”. In: *IEEE Computer Graphics and Applications* 32.4 (2012), pp. 63–67. ISSN: 0272-1716.

[87] Yucong Ye, Robert Miller, and Kwan-Liu Ma. “In Situ Pathtube Visualization with Explorable Images”. In: *EGPGV.* 2013, pp. 9–16.

[88] C. S. Yoo et al. “Direct numerical simulation of a turbulent lifted hydrogen/air jet flame in heated coflow”. In: *In Computational Combustion, ECCOMAS Thematic Conference.* 2007.

[89] Sung-Eui Yoon and Peter Lindstrom. “Mesh Layouts for Block-Based Caches”. In: *IEEE Transactions on Visualization & Computer Graphics* 12 (2006), pp. 1213–1220. ISSN: 1077-2626.

[90] Dazhi Yu et al. “Viscous flow computations with the method of lattice Boltzmann equation”. In: *Progress in Aerospace Sciences* 39 (2003), pp. 329–367.

[91] George Zagaris, Charles Law, and Berk Geveci. “Visualization and Analysis of AMR Datasets”. In: *Kitware Source* 18 (July 2011), pp. 2–6.

[92] Fan Zhang et al. “In-situ Feature-Based Objects Tracking for Large-Scale Scientific Simulations”. In: *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:* 2012, pp. 736–740.

[93] J. Ziv and A. Lempel. “A universal algorithm for sequential data compression”. In: *IEEE Transactions on Information Theory* 23.3 (1977), pp. 337–343. ISSN: 0018-9448.