ABSTRACT

SCHENDEL, ERIC RICHARD. Preconditioner-based In Situ Data Reduction for End-to-End Throughput Optimization. (Under the direction of Dr. Nagiza F. Samatova.)

Efficient handling of large volumes of data is a necessity for future extreme-scale scientific applications and database systems. To address the growing storage and throughput imbalance between the data production on such systems and their I/O subsystems, reduction of the handled data volume by *compression* is a reasonable approach. However, quite often many scientific data sets compress poorly, referred to as *hard-to-compress* datasets, due to the negative impact of highly entropic information represented within the data. Lossless compression efforts on such datasets typically do not yield more than a 20% reduction in size when exact reproduction of the original data is required. Moreover, modern applications of compression for hard-to-compress scientific datasets hinder end-to-end throughput performance due to overhead timing costs of data analysis, compression, and reorganization. When overhead costs of applying compression are greater than end-to-end performance gains obtained by the data reduction, utilization of a compressor has no practical benefit for scientific systems.

A difficult problem in lossless compression for improving scientific data reduction efficiency and throughput performance is to identify the hard-to-compress information and subsequently optimize the compression techniques. To address this challenge, we introduce the *In Situ Orthogonal Byte Aggregate Reduction Compression* (ISOBAR-compress) methodology as a *preconditioner* of lossless compression to identify and optimize the compression efficiency and throughput of hard-to-compress datasets. Out of 24 scientific datasets from both the public domain and peta-scale simulations, ISOBAR-compress accurately identified the 19 that were hard-to-compress. Additionally, ISOBAR-compress improved data reduction by an average of 19% and increased compression and decompression throughput by an average speedup of 24.1 and 33.6, respectively.

Additionally, dataset preconditioning for lossless compression is a promising approach for reducing disk and network I/O activity to address the problem of limited I/O bandwidth in current analytic frameworks. Hence, we also introduce a *hybrid compression-I/O* methodology for interleaving I/O activity with data compression to improve end-to-end throughput performance along with the reduced dataset size. We evaluate several interleaving strategies, present theoretical models, and evaluate the efficiency and scalability of the approach through comparative analysis. The hybrid method when applied to 19 hard-to-compress scientific datasets demonstrates a 12% to 46% increase in end-to-end throughput. At the reported peak bandwidth of 60 GB/s of uncompressed data for a current, leadership-class parallel I/O system, this translates into an effective gain of 7 to 28 GB/s in aggregate throughput.

Lastly, it is important that scientific applications further streamline their end-to-end throughput performance beyond only preconditioning datasets for compression. The concept of applying a preconditioner is generalizable for other techniques that allow optimizing performance by data analysis and reorganization. For example in present-day scientific simulations, there is a drive to optimize in situ processing performance by inspecting the layout structure of a generated dataset and then restructuring the content. Typically, these simulations interleave dataset variables in memory during their calculation phase to improve computational performance, but deinterleave the data for subsequent storage and analysis. As a result, an efficient preconditioner for data deinterleaving is critical since common deinterleaving methods provide inefficient throughput and energy performance. To address this problem, we present a deinterleaving method that is high performance, energy efficient, and generic to any data type. When evaluated against conventional deinterleaving methods on 105 STREAM standard micro-benchmarks, our method always improved throughput and throughput/watt. In the best case, our deinterleaving method improved throughput up to 26.2x and throughput/watt up to 7.8x.

Preconditioner-based In Situ Data Reduction for End-to-End Throughput Optimization

by
Eric Richard Schendel

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2014

APPROVED BY:

Dr. Rada Y. Chirkova	Dr. Christopher G. Healey
Dr. Frank Mueller	Dr. Nagiza F. Samatova Chair of Advisory Committee

DEDICATION

To my loving wife, Bushra Iftikhar Schendel,
whose inspirations and enduring sacrifices are beyond measure.

BIOGRAPHY

Eric R. Schendel began his Doctor of Philosophy in Computer Science at North Carolina State University in fall of 2010 with Dr. Nagiza Samatova as his advisor. He earned a Bachelor of Science degree in Computer Engineering, Electrical Engineering track, with Mathematics minor from Texas A&M University, College Station, TX in 2001. In addition, he received a Master of Science in Computer Science from Texas A&M University at Corpus Christi in 2010. Prior to starting his doctorate program, Eric had extensive industry experience as a software architect and engineer working at companies such as Advanced Micro Devices, Hewlett Packard, and IBM. Moreover, he had a research aide appointment at Argonne National Laboratory during his doctoral study.

ACKNOWLEDGEMENTS

Completing this dissertation is realizable due to the support of many people and institutions. Foremost, I am forever thankful to my advisor Dr. Nagiza Samatova for mentoring my academic, professional, and personal growth. Her wisdom, intellect, and caring nature have been awe-inspiring and motivation to reach a potential of myself beyond initial awareness.

In addition, I am grateful to my PhD committee members for their support and constructive criticisms during the course of my work: Professors Rada Chirkova, Frank Mueller, Christopher Healey, Xiaosong Ma, and Peng Ning. I am further thankful to Dr. Douglas Reeves and Andrew Sleeth for their advice and wisdom necessary to complete the PhD program at North Carolina State University.

During the course of completing this dissertation, I was fortunate enough to collaborate with exceptional people at notable research institutions. I am grateful to Venkat Vishwanath, Michael Papka, Robert Ross, and Robert Latham for their invaluable insights and allocation of leadership-class computing resources at Argonne National Laboratory. In addition, leadership-class computing resources at Oak Ridge National Laboratory were made available during research collaboration with Scott Klasky. I am also thankful to Qing Lu at Oak Ridge National Laboratory, Jackie Chen and Hemanth Kolla at Sandia National Laboratory, C.S Chang and Stephane Ethier at Princeton Plasma Physics Laboratory, and Seung-Hoe Ku at New York University for collaborative access to their profound expert knowledge.

There are many other people I interacted with during the course of my PhD program, and I now call them life-long friends. If it was not for them, my continuous growth and sanity would have not been possible: John Jenkin, David (Drew) Boyuka, Saurabh Pendse, Neil Shah, Steven Harenberg, Ye Jin, Kanchana Padmanabhan, Sriram Lakshminarasimhan, Xiaocheng Zou, Isha Arkatkar, Ramona Seay, Zhenhuan Gong, Houjun Tang, and Terry Rogers.

The work within this dissertation was funded by the U.S. Department of Energy, Office of Science (SciDAC SDM Center), and the U.S. National Science Foundation (Expeditions in Computing and EAGER programs).

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

As exascale computing comes closer to becoming reality and more powerful High Performance Computing (HPC) systems become available, the complexity of scientific simulations and analyses has grown commensurately. Unfortunately, the data I/O subsystems offered by these systems have not kept up, which is leading to a serious data bottleneck in read and write throughput performance of HPC applications.

Utilization of I/O staging, optimized analytic frameworks, and compression is becoming commonplace to help cope with the growing gap between computing power and I/O bandwidth in current petascale HPC environments. While providing increased and more consistent computational performance, the sheer scale of the data necessitates data reduction methodologies. This prompts the technical challenge of identifying effective and low-overhead data reduction methods for boosting I/O throughput performance for large-scale systems.

In this thesis, we investigate the challenges and introduce methods for improving end-to-end data throughput required for the future of exascale computing. Traditionally, compression is inappropriate for improving I/O throughput of data generated by HPC simulations due to timing costs added that negatively affect overall throughput performance. However, we argue that the goals of reducing the data size footprint and improving end-to-end throughput for both storage and analytics can in fact be complementary.

1.1 Hypothesis

By our motivation to have compression and I/O performance work in concert, we present the following hypothesis to support our thesis:

Application of preconditioners for lossless compression and layout reorganization optimizes data reduction without loss of fidelity and improves end-to-end throughput for data storage, retrieval, and analysis.

1.2 Proposed Approaches

1.2.1 ISOBAR Preconditioner of Lossless Compression

In the last ten years of High Performance Computing (HPC), there has been an increasing imbalance between the amount of data being produced by high-speed processors and the file system bandwidth [58, 74]. This imbalance necessitates the need to reduce the data before it is written to the file system; but due to the increasing complexity of the scientific data from many simulations, standard lossy and lossless compression techniques often become limited in their usefulness [25]. Improving the identification of difficult to compress content within scientific data can help optimize general lossless compression techniques.

Problems and Challenges

Scientific datasets typically comprise of single- and double-precision floating-point values. Compression of these scientific datasets is complicated by the fact that scientists cannot sacrifice simulation fidelity, especially when saving simulation checkpoints. This excludes the possibility of using lossy compression as a viable data reduction method for simulations. On the other hand, lossless compression techniques typically offer no more than 20% reduction on many single- and double-precision floating-point scientific datasets [12].

There are a number of technical challenges in utilizing lossless compression in the scientific computing environment. Scientific datasets are typically considered *hard-to-compress* due to the minute gains obtained from the use of lossless compression processes to reduce the data size. Also, these techniques do not alleviate the constraints on file system bandwidth. For example, FPC and bzip2 [11, 75] do not provide enough compression throughput to justify the data reduction, while techniques such as zlib and RCFile [29, 38] improve compression throughput but sacrifice the compression ratio. Moreover, the low throughput of data compression and decompression makes these techniques hardly suitable for in situ processing (in place processing of the data during a simulation run), which is required for applications such as those with simulation checkpoint and restart data.

Approach and Results

To bridge the gap between data size reduction and compression throughput performance, we introduce a novel method that enables fast, effective, and high-throughput reduction of single- and double-precision floating-point scientific data. The intuition behind this method arises from the use of *preconditioners* for improving the convergence rate of iterative solvers in linear algebra, such as Algebraic Multigrid (AMG) [23], Quasi-Minimal Residual (QMR) [26], LDL solver [7], and others. Although preconditioners are widely used by applications including aeronautics and

fluid dynamics, preconditioning techniques to optimize input for solvers have not been used in the lossless data-compression realm to our knowledge.

The lossless compression preconditioner we introduce is called In Situ Orthogonal Byte Aggregate Reduction Compression (ISOBAR-compress). ISOBAR-compress allows fast analysis of data and is capable of identifying characteristics that result in poor compression ratio and compression/decompression throughput. The preconditioner analyzes the compressibility of the data and creates the appropriate pipelines for compression. Specifically, it decides how to partition data into compressible and incompressible segments, how to linearize multi-dimensional data, and which compressor should be used to optimize compression performance. This enhances the compression efficiency in terms of the compression throughput as well as the compression ratio.

ISOBAR-compress was evaluated against 24 scientific datasets generated by real-world simulations and those available from the public for evaluating compression technologies. These scientific datasets are from various disciplines including combustion, physics, and astrophysics. Through empirical evaluations by applying various compression techniques [12, 53], 19 of the 24 were determined to be hard-to-compress. ISOBAR-compress was able to accurately identify all 19 of these hard-to-compress datasets along with isolate the segments to be considered incompressible. Furthermore, after ISOBAR-compress preconditioned the dataset based on its findings, it improved data reduction by an average of 18.6% and increased compression and decompression throughput by an average speedup of 24.1 and 33.6, respectively.

1.2.2 ISOBAR Hybrid Compression-I/O Interleaving Method

Using preconditioners of lossless compression to reduce the amount of data placed on the file systems only partially solves the problem of the growing gap between computing power and I/O bandwidth in current peta-scale HPC environments. Limitations on disk I/O performance of HPC systems have lead to a serious bottleneck in read and write performance in scientific applications [9, 64]. In addition, increasing frequency of checkpoint operations performed by scientific applications further adds to the I/O overload [82]. Ideally, the solution to the I/O bottleneck will involve both data reduction and parallel I/O access pattern optimization.

Problems and Challenges

The prevailing strategy to improve I/O performance in current peta-scale systems is to offload the burden of I/O transactions to dedicated I/O nodes in the system [3]. This data staging architecture allows minimal idle time on the compute nodes while the I/O nodes handle the rate-limiting disk writes and reads collectively. However, given the trend of ever-increasing simulation data sizes, combined with the need to checkpoint simulation state to minimize data

loss in the face of node failure, the I/O offloading approach alone cannot keep up with the computational throughput available [57].

Unfortunately, data compression and parallel I/O performance optimization strategies have traditionally been in conflict. State-of-the-art I/O middleware solutions, such as ADIOS [55], HDF5 [83], and PnetCDF [52], have no native support for write compression in a parallel context. This is due to the complexity of handling the compressed, non-uniform data, which requires synchronization between all nodes performing shared-file I/O. Additionally, we observed that addressing the integration of compression with I/O strategies still leaves the technical challenge of absorbing the costs of compression and overhead in parallel I/O performance.

Approach and Results

By applying ISOBAR preconditioner techniques to a data staging architecture and incorporating the popular ADIOS I/O framework as the I/O backend, we present the ISOBAR Hybrid Compression-I/O methodology for data reduction and I/O optimization. This is a novel compression-I/O interleaving strategy for effective parallel compression with state-of-the-art I/O performance.

The ISOBAR Hybrid Compression-I/O strategy simultaneously processes the compressible and incompressible components of the data as identified by the ISOBAR preconditioner. The input data is partitioned into two streams: compressible and incompressible data. The compressible stream is compressed and then written to disk, while the incompressible stream can be immediately written to disk. There is no dependency between the two streams, so operations on the streams are independent. Thus, the hybrid method first issues an asynchronous write of the incompressible stream, and then it begins compressing the compressible stream. This enables immediate asynchronous transfer and writing of incompressible data while compression is applied concurrently to the remainder before it is written. This technique hides the compression costs and fully utilizes all compute, network, and I/O resources of the data staging architecture.

By applying the ISOBAR hybrid compression-I/O methodology against 19 real-world hard-to-compress scientific datasets, it exhibited read and write performance gains proportional to the degree of data reduction, which ranges as high as 46% on scientific datasets. This would translate into an effective increase of 28 GB/s bandwidth over the peak aggregate throughput of 60 GB/s of uncompressed data offered by the leadership-class Lustre parallel filesystem at Oak Ridge National Labs [57]. Even under worst-case conditions, where the dataset is highly entropic and difficult to compress, we show that our system still maintains a gain in throughput over the state-of-the-art.

Since it is impossible to evaluate the performance gains of this methodology on every possible cluster configuration, it is also desirable to have an analytical performance model. A performance

model allows prediction of performance characteristics on new hardware and software, and aids application developers in configuration choices. Therefore,we developed a performance model for this methodology, which demonstrates a high degree of accuracy through validation against empirical data.

1.2.3 High-performance Deinterleaving Method for Scientific Data

Currently, a major restriction placed on the effectiveness of high-performance compression methods is the cost of data reorganization and multi-pass operations, which is against the spirit of in situ processing. Even in present-day scientific simulations, there is a similar drive to improve in situ performance due to lack of effective data management operations especially when transitioning between their calculation and subsequent analysis and storage phases. In our thesis, we explored these challenges and provide a generic method for improving throughput performance and reducing energy utilization of start-of-the-art compression and analysis methodologies leveraged by modern HPC systems.

Problems and Challenges

Compression, analysis, and indexing methods within the HPC community typically require reorganization of scientific data at varying data type granularities [8, 41], such as bytes, single- and double-precision floating-points, etc. A common data reorganization technique utilized by scientific simulations for improving analysis performance is the deinterleaving of data. *Deinterleaving* is the transformation of interleaved variables generated during the calculation phase of a simulation into a grouping of individual variables to be contiguous in memory and storage. After deinterleaving data at the byte level, compression, precision-level layout operations, and compression-based indexing technologies are more effective in terms of throughput performance and space reduction.

Commonly applied deinterleaving methods for compression and analysis in the scientific community were found to be ineffective for in situ processing due to poor processor cache efficiency and energy performance since they were geared more towards post-processing analysis. Alternative and better performing deinterleaving methods were identified but unfortunately were found to be overly specialized (such as square matrices of size $2^n \times 2^n$) and not fitting for general application on modern systems.

Approach and Results

To address the challenge of improving state-of-the-art compression and analysis performance for in situ processing, our focus was on creating an out-of-place deinterleaving method (OPD) that is high performance, energy efficient, and generic to any variable data type. We created such

a deinterleaving method that inherently exploits hardware cache prefetching, reduces memory accesses, and optimizes use of complete cache line reads and writes.

For evaluation, we use the accepted STREAM benchmark, which is useful for evaluating memory throughput performance of single- and multi-core I/O-intensive functions that are sensitive to system architecture characteristics. We collect throughput performance metrics on 105 STREAM micro-benchmarks and test across both AMD and Intel systems. We deinterleave combinations of multiple data types (bytes, single-precision, double-precision), columns (up to 16 variables), and input buffer sizes. In addition, we compare the performance against general deinterleaving methods such as standard (column-based) and strided (row-based). OPD always out performs the general deinterleaving methods and when compared to the next best deinterleaving method, our method demonstrates a performance improvement in throughput up to 26.2x and energy effectiveness (throughput/watt) up to 7.8x.

Chapter 2

ISOBAR Preconditioner for Effective and High-throughput Lossless Data Compression

2.1 Introduction

In the last ten years of High Performance Computing (HPC), we have seen an increasing imbalance between the high speed processors of the machine and the file system bandwidth. This imbalance necessitates the need to reduce the data before it is written to the file system; but due to the increasing complexity of the data from many simulations, standard compression techniques often become limited in their usefulness. *Lossless* compression techniques offer no more than 20% reduction on many single and double-precision floating-point scientific datasets that we have tested on. These datasets are considered *hard-to-compress* due to the minute gains obtained from the use of such compression processes to reduce the data size. Moreover, the low throughput of data compression and decompression makes these techniques hardly suitable for *in situ processing* (in place processing of the data during simulation run), which is required for applications, such as those with simulation checkpoint and restart data.

To bridge this gap, we introduce a strategy that enables fast, effective, and high-throughput reduction of single- and double-precision floating-point scientific data. The intuition behind this method arises from the use of *preconditioners* for improving the convergence rate of iterative *solvers* in linear algebra, such as Algebraic Multigrid (AMG) [23], Quasi-Minimal Residual (QMR) [26], LDL solver [7], and others. Other preconditioning processes such as matrix factorization (e.g., QR factorization) are widely used by applications including aeronautics and fluid dynamics. To the best of our knowledge, such preconditioning techniques optimizing input for solvers have not been used in the lossless data-compression realm.

Table 2.1: Characteristics of Simulation Output Datasets from Seven Applications

Applications	Research Area	Variable(s)	Data Type	Reference
GTS	Fusion Plasma Core	density, potential	double	[78]
XGC	Fusion Plasma Edge	igid, iphase	double, integer	[46]
S3D	Combustion	temperature, vmagnitude	float	[16]
FLASH	Astrophysics	velocity	double	[28]
MSG	NAS Parallel Benchmark (NPB) and ASCI Purple	bt, lu, sp, sppm, sweep3d	double	[11]
NUM	Numeric Simulations	brain, comet, control, plasma	double	[13, 67]
OBS	Measurements of Satellite	error, info, spitzer, temp	double	[12]

The lossless compression preconditioner introduced in this paper is called *In Situ Orthogonal Byte Aggregate Reduction Compression* (ISOBAR-compress). ISOBAR-compress allows fast analysis of data and is capable of identifying characteristics in data that result in poor compression ratio and compression/decompression throughput. Our preconditioner analyzes the compressibility of data and creates the appropriate pipelines for compression of various datasets. Specifically, it decides how to partition data into compressible and incompressible segments, how to linearize multi-dimensional data, and also which compressor should be used to optimize compression performance in terms of storage or speed (user-specified). ISOBAR-compress essentially circumvents the additional complexities presented by multi-dimensional data by performing roughly the same for original data linearized in different means. Upon testing ISOBAR-compress on 24 scientific datasets of 7 applications summarized in Table 2.1 (see Appendix A for more details), we found that 19 of them were identified as ISOBAR-compress improvable hard-to-compress datasets. On these datasets, ISOBAR-compress provided both higher throughput (varying from 100MB to 450MB per second) and improved compression ratios than those obtained without its use; 2 datasets had about 40% increase in compression ratio (ΔCR, Eq. 2.2), 13 of 19 had at least a 20% increase in the same, and the other 6 datasets experienced compression enhancements in the range of $[5\%, 10\%]$. In addition, ISOBAR offered a multi-fold increase in compression and decompression throughputs. [See Table 2.2 for some examples of ISOBAR-compress performance and Results section for details.]

$$CR = \frac{Original\ Data\ Size}{Compressed\ Data\ Size} \tag{2.1}$$

$$\Delta CR = (\frac{CR_{ISOBAR}}{CR_{Standard}} - 1) \times 100\% \tag{2.2}$$

$$Sp = \frac{Throughput\ of\ ISOBAR\text{-}compress}{Throughput\ of\ Standard\ (De\text{-})Compressor} \tag{2.3}$$

We aim to optimize the solver (i.e., compressor) portion of the data reduction pipeline by

Table 2.2: ISOBAR-compress Performance Summary

Dataset	Δ CR (%)[1]	TP_C[2]	Sp_C[3]	TP_D[4]	Sp_D[5]
GTS	10.15	111.7	8.05	551.90	5.01
XGC	14.09	76.83	21.17	388.87	51.92
S3D	32.56	104.73	31.45	424.79	63.12
FLASH	17.52	455.83	35.89	1617.02	14.19

[1] Δ CR (%): Percentage improvement of compression ratio (see Equation 2.2) comparing to the best alternative

[2] TP_C: Compression throughput in MB (megabyte) per second

[3] Sp_C: Speed-up of compression (see Equation 2.3)

[4] TP_D: Decompression throughput in MB per second

[5] Sp_D: Speed-up of decompression (see Equation 2.3)

enabling our preconditioner to function with any type of general-purpose compressors. Thus, a user can specify a preference in compressor to use with little to no change to our preconditioning method. We commonly use `zlib` and `bzlib2` as solvers, but could just as easily use `fpzip`, `FPC`, and various other tools (each may provide a different tradeoff in terms of throughput and compression ratio).

2.2 Method

To understand what makes datasets hard-to-compress, we analyzed several double-precision floating-point datasets (64-bit) at the bit-level for their probability distributions (see Figure 2.1). When a bit position has a probability distribution of 1.0, it means that there is an absolute guarantee that the bit position value will be either 0 or 1 for all the values in the entire dataset. On the other hand, a probability distribution of 0.5 means the bit value for a given bit position has an equal probability of either being 0 or 1 for all the values. Based on this observation and experimentation, *xgc_igid*, *gts_zeon*, and *flash_gamc* (see Figure 2.1) are considered hard-to-compress datasets, whereas *msg_sppm* is not. Often, the first two bytes have high probabilities due to the representation of double-precision values as exponent and mantissa segments. Exponent values are often close together due to locality of data. Mantissa bytes can, in some cases have high probability based on degree of approximation and amount of precision needed, but are typically not predictable. The assumption is that the 0.5 probability distribution bits made the dataset hard-to-compress due to lack of predictability; this lack of predictability, or presence of randomness can be considered as *noise* in the data. In signal processing, filtering or denoising methods are widely used to improve the *compression efficiency* (a compressor's applied performance on a dataset) of a signal by discarding the noise [62]. Inspired by this idea,

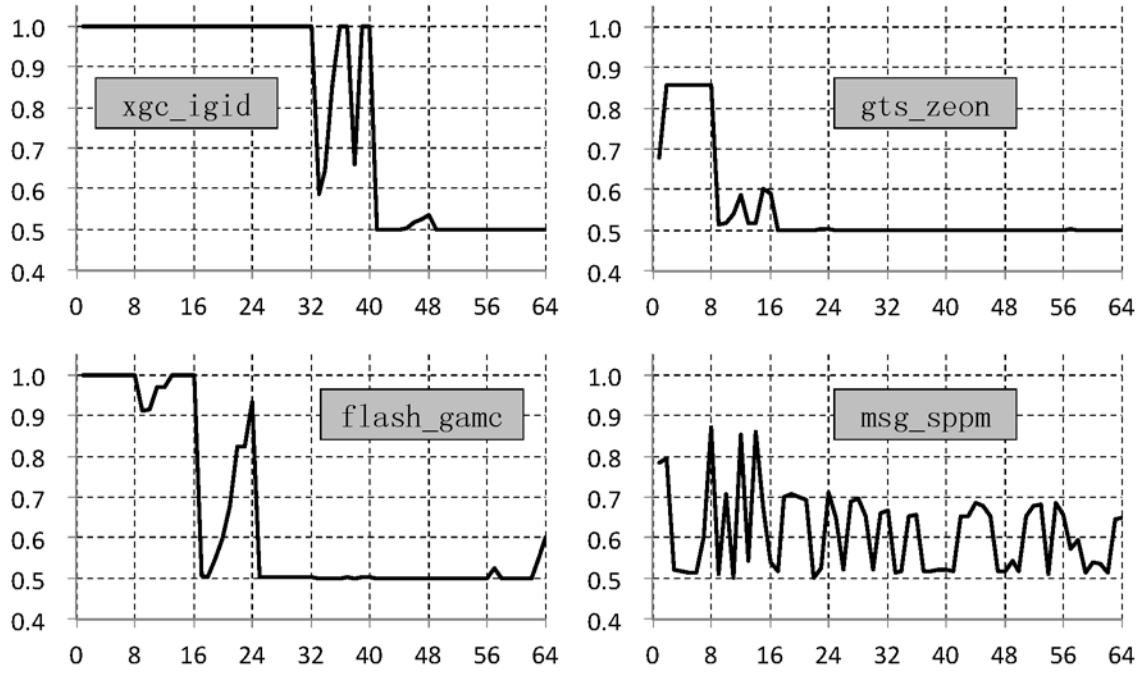


Figure 2.1: Bit frequencies of 4 representative datasets; x-axis represents bit position (1 to 64), y-axis represents the probability distribution ranging from 0.5 to 1.0 of the more common bit values at that position (0 or 1)

we worked with the notion that by identifying and extracting "noisy" data and only compressing the remaining "signal" data, we will obtain a better compression efficiency and throughput for any given general lossless compressor.

The following subsections explore, in further detail, the central components that compose the In Situ Orthogonal Byte Aggregate Reduction Compression (ISOBAR-compress) preconditioner. The preconditioner's objective is to identify the noise-like properties within a dataset that negatively impact compression efficiency and reduce the burden on the compressor from processing such noise. The preconditioner workflow is illustrated in Figure 2.2. There are two main components that make up ISOBAR-compress: (1) ISOBAR-analyzer, which is responsible for identifying the "high complexity" data [63] (noise) that makes a dataset hard-to-compress, and (2) the ISOBAR-partitioner, which is responsible for segmenting out the noise that is considered hard-to-compress from the signal-like data, thus improving the compression efficiency [65]. The remaining components (EUPA-selector, general lossless compressor, and merger) of ISOBAR-compress are important for successfully implementing the workflow and are also discussed in the next few subsections.

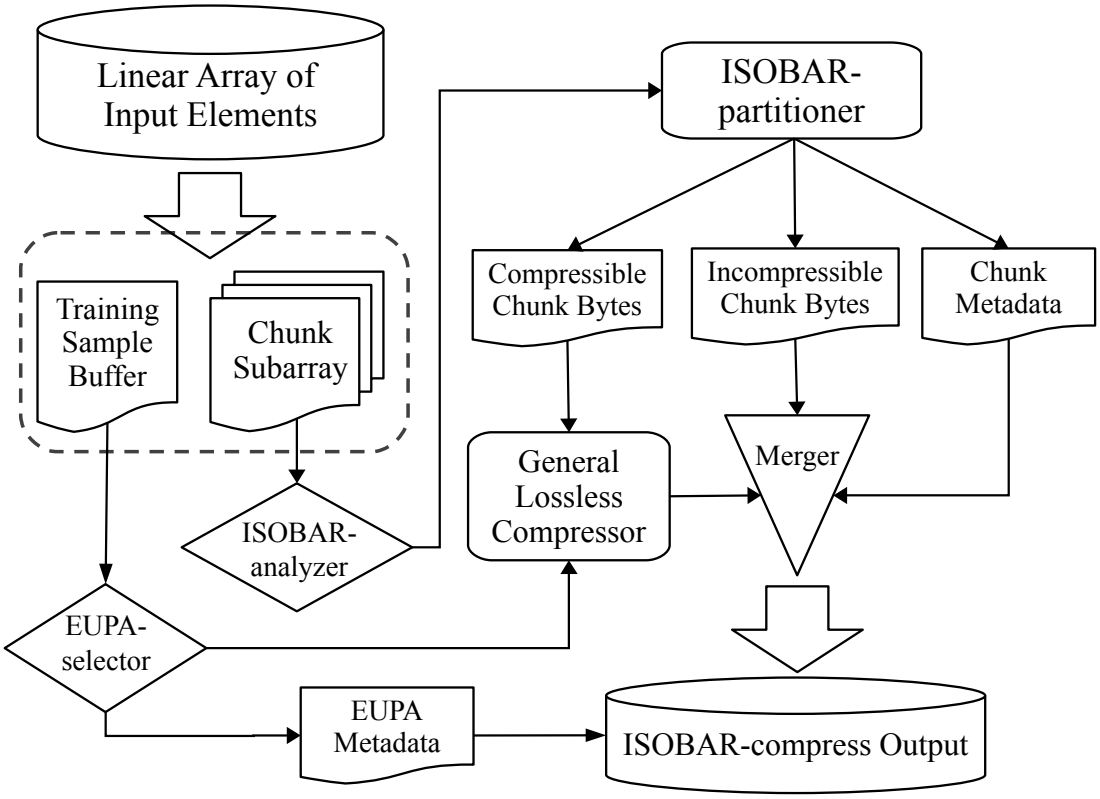


Figure 2.2: ISOBAR-compress preconditioner workflow

$$\begin{pmatrix} Element_1 \\ Element_2 \\ \vdots \\ Element_{N-1} \\ Element_N \end{pmatrix} \Leftrightarrow \begin{pmatrix} Byte_{1,1} & Byte_{1,2} & \cdots & Byte_{1,\omega} \\ Byte_{2,1} & Byte_{2,2} & \cdots & Byte_{2,\omega} \\ \vdots & \vdots & \ddots & \vdots \\ Byte_{N-1,1} & Byte_{N-1,2} & \cdots & Byte_{N-1,\omega} \\ Byte_{N,1} & Byte_{N,2} & \cdots & Byte_{N,\omega} \end{pmatrix}$$

Figure 2.3: Element and byte-level representation of an input array

2.2.1 ISOBAR-analyzer

The ISOBAR-analyzer is primarily responsible for analyzing an input dataset consisting of elements of the same type for clusters of bytes that negatively influence compressibility throughout the dataset. This is accomplished by first determining if the input array of elements is a candidate for lossless compression improvement at the *byte level*. After completing the improvement identification phase, the ISOBAR-analyzer will promote appropriate decisions to other processes in the ISOBAR-compress workflow (see Figure 2.2).

There are two main reasons that the ISOBAR-analyzer operates at a byte level: granularity support of general lossless compressors, better entropic complexity [73] identification and computational performance. Firstly, all the general lossless compressors (such as `zlib` and `bzlib2`) operate at the byte level when doing entropy encoding [75, 29, 81, 10]. Thus, it is important for the ISOBAR-compress workflow to process input data in a manner that promotes pipelining of bytes to a compressor. Secondly, ISOBAR-analyzer can make more accurate and quicker identification decisions in regards to compressibility at the byte-level due to the greater variance of entropy [45] as compared to analysis strictly at the *bit level*.

When ISOBAR-analyzer receives a dataset of N elements, it starts analyzing each element as an aggregate of ω bytes, where each byte value is in the $[0, 255]$ range. In order to store all N elements of an input dataset, $N \cdot \omega$ bytes are required, for example, each double-precision floating point number needs $\omega = 8$ bytes to store it, storing $N = 1024$ of them will take 8192 bytes on the disk. The values of all required bytes, $Byte_{i,j}$, where $1 \leq i \leq N$ and $1 \leq j \leq \omega$, are represented by the right matrix in Figure 2.3. Conceptually, the layout for the set of elements can be illustrated as a matrix of bytes, where each row represents an element of ω bytes and each column represents an orthogonal placement of N bytes (one from each element), called a *byte-column*.

ISOBAR-analyzer will begin its identification phase after receiving an input dataset. The identification phase starts by processing each byte-column separately looking for hard-to-compress properties (illustrated in Figure 2.4). In order to process all the byte-columns, ω byte-value

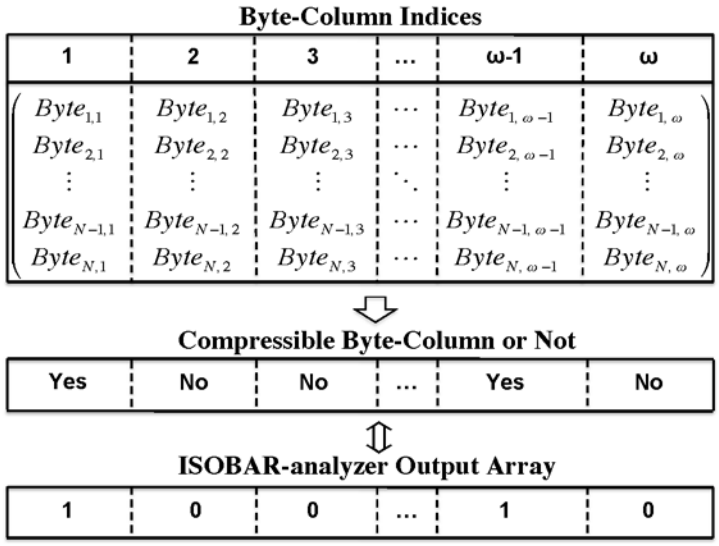


Figure 2.4: Byte-column reduction selection example

frequency counters are required. A frequency counter is necessary for generating a frequency distribution of the possible 256 values for a given byte-column. The frequency distribution helps determine the possible reduction of byte sets that negatively influence overall compressibility. If the distribution for each N byte value within a byte-column is less than or equal to $N/256$, then that column-based data set will be considered incompressible for entropy encoding. Byte-columns with incompressible properties negatively affect the compressibility of the original input dataset [73] at the byte level. Thus, eliminating the incompressible set of bytes represented by a column will improve potential compressibility of the remaining byte-columns as a whole.

After the ISOBAR-analyzer calculates the byte value frequency distribution for each byte-column, the process then selects the columns of bytes for reduction, identifiable as an incompressible byte set. The byte-column selection is based on a frequency distribution tolerance level that is a configurable property of the ISOBAR-analyzer process. If all byte values [0, 255] in a byte-column frequency distribution are below the tolerance level, then that byte-column will be selected as incompressible. For the sake of utilizing ISOBAR-compress as a general solution, the frequency distribution tolerance level is defined as $\tau \cdot N/256$ for N elements, where τ is a value between 1 (always incompressible) and 256 (always compressible). We fixed $\tau = 1.42$ based on our experiments and the reason is because when τ varies in the range of [1.4, 1.5], the compression ratio's improvement keeps stable for each tested dataset.

2.2.2 ISOBAR-partitioner

Once the ISOBAR-analyzer identifies all compressible byte-columns for reduction, the next step is to determine whether the input of elements qualifies as a candidate for lossless compression improvements. These improvements are readily available through the ISOBAR-compress workflow's remaining processes. Identification types for compressibility improvement for the entire dataset fall under two categories: improvable, or undetermined. This determination is handled by the ISOBAR-partitioner process. If none or all of the columns are selected for reduction, then the input dataset is considered undetermined, where the entire dataset is then passed to the compressor process. If the dataset is identified as improvable, then the ISOBAR-analysis selected byte-columns are compressed, while the remaining byte-columns are not. Algorithm 1 provides the operational data flow combining the ISOBAR-analysis and ISOBAR-partitioner processes.

Algorithm 1: ISOBAR-compress

Input:
X—Set of input elements to be compressed
E—End user's preference: throughput or ratio
Output:
X'—Compressed array
Data:
S—ISOBAR-analyzer output array
C—Compressible bytes of input elements
C'—Compressed compressible bytes of input elements
I—Incompressible bytes of input elements
M—Metadata

```
1 S← ISOBAR-analyzer(X)
2 if S = {0, 0, ..., 0} or S = {1, 1, ..., 1} then
3     X' ← compressor(X, E)
4 else
5     {C, I, M} ← ISOBAR-partitioner(S, X)
6     C' ← compressor(C, E)
7     X' ← {C', I, M}
8 return X'
```

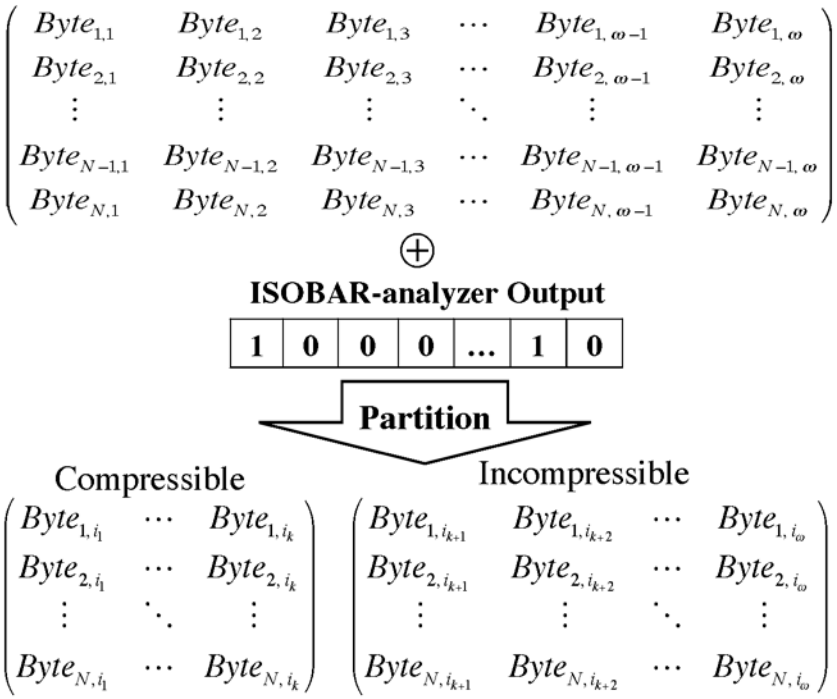


Figure 2.5: Example of ISOBAR-partitioner operation

Based on the output array of ISOBAR-analyzer, the original dataset is partitioned into two segments: the improvable (compressible) byte-columns, and the hard-to-compress (incompressible) byte-columns (see Figure 2.5). Following the information passed from the End User's Preference Adaptive Selector (EUPA-selector, see Section 2.2.3, designed to heuristically choose the linearization strategy and lossless compression technique), the compressible columns will be realigned with the chosen linearization strategy. For example, if we encounter a dataset of 64-bit double-precision floating-point numbers, the EUPA-selector will decide to use `zlib` as the standard compression method (the corresponding row-linearization and metadata provided by the ISOBAR-analyzer is 10000010). The ISOBAR-partitioner will then partition and row-wise linearize the 1^{st} and 7^{th} columns and identify them as the input of the `zlib` standard compression process. In doing so, we only pass the compression algorithm 2 bytes rather than all 8 bytes of the double-precision number as one row in the matrix. This guarantees an increase in compression throughput.

2.2.3 EUPA-selector

Since ISOBAR-compress is designed as a preconditioner to improve performance of all *solvers* (lossless compression techniques), the question that arose with this intuition was, "Which type

of *performance* is the most desired by the end-user?" While some users may only want to save disk space and hence desire the highest compression ratio rather than compression throughput, most others, including scientists running peta-scale simulation codes (XGC, GTS, etc.), would desire a technique that provides the highest compression throughput with reasonably acceptable (but perhaps not the best) compression ratio. To preserve end-user flexibility in regards to these choices, we designed the End User's Preference Adaptive Selector (EUPA-selector).

The EUPA-selector is a deterministic process that selects the most suitable lossless compression framework for applying to all the compressible bytes selected by ISOBAR-analysis during the compression process of the workflow. The selector makes a decision based on the evaluation of an input training sample acquired from the input dataset, end-user criterion, and lossless compressor evaluations.

For the purpose of this paper, the ISOBAR-compress workflow will be utilized as a black box solution where the most commonly known compression algorithms (`zlib` and `bzlib2`) are applied by the EUPA-selector. The selector is designed to make a decision on which standard compression method and byte-level linearization will provide the best performance for the end-user's preference. For example, although the EUPA-selector inherently chooses the technique that provides the best compression ratio, the user can instruct the selector to choose a faster method as long as the compression ratio is above a certain, specified threshold. The selector is implemented by first testing each combination of the standard compression method and linearization strategy on sample sets of random elements from the input dataset. Based on the results from the corresponding combinations, the EUPA-selector will make a decision to use either `bzlib2` or `zlib` as the standard compression method and apply either row-based or column-based linearization. Regardless of whether an input dataset to ISOBAR-analysis is identified as improvable or not, the EUPA-selector will choose the optimal standard compression method (`zlib` or `bzlib2`) and linearization strategy for end-users. Based on our experimentations, datasets identified as improvable will have a better compression ratio whether `zlib` or `bzlib2` is used as the standard compression method.

2.2.4 Input Array Chunking and Output Merging

Scientific information generated from extreme-scale simulations can easily expand beyond terabytes of archival data, which is impractical for a lossless compressor to handle at a post-processing stage. Compressing an extreme-scale generated dataset as a single input array of elements is problematic due to a single-pass timing cost along with system memory limits. Therefore, it is practical for a lossless compressor to segment an input array into chunks (see Figure 2.6) of manageable data for pipelining to an in situ process of a workflow.

A minimum chunk size is required by the ISOBAR-compress workflow in order for the

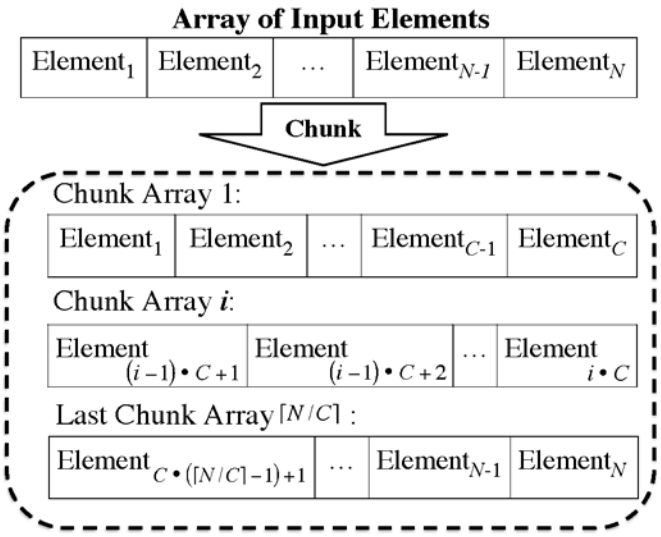


Figure 2.6: Chunking a dataset (array of input elements)

ISOBAR-analysis process to function optimally. The reason is because ISOBAR-analysis requires statistical evaluation of all the bytes-columns within an input chunk of elements. This is similar to as why general compressors require minimum block sizes, such as 4 MB used in RCFile [38], and 3 MB demonstrated in the paper [81]. The appropriate dataset chunk size, C, for ISOBAR-compress to efficiently operate is covered in the performance evaluation section of this chapter.

At the end of the ISOBAR-compress workflow, an output array of bytes is generated by merging the overall metadata created by EUPA-selector, each chunk's metadata generated by ISOBAR-partitioner, and the incompressible byte-columns along with the compressed byte-columns of each chunk (see Figure 2.7).

2.3 Performance Evaluation

To demonstrate that ISOBAR-compress has the ability to not only identify hard-to-compress datasets, but also improve the compression efficiency, we apply two major metrics in our tests: Compression Ratio (CR, see Equation 2.1) and Compression/Decompression Speed-up (Sp, see Equation 2.3).

All metrics where collected on the Lens Linux cluster at Oak Ridge National Laboratory. The Lens cluster is primarily for data analysis and high-end visualization. Each cluster node consists of four quad-core 2.3 GHz AMD Opteron processessors and 64 GB of memory. All the results for the following experiments were done utilizing only a single core of one processor.

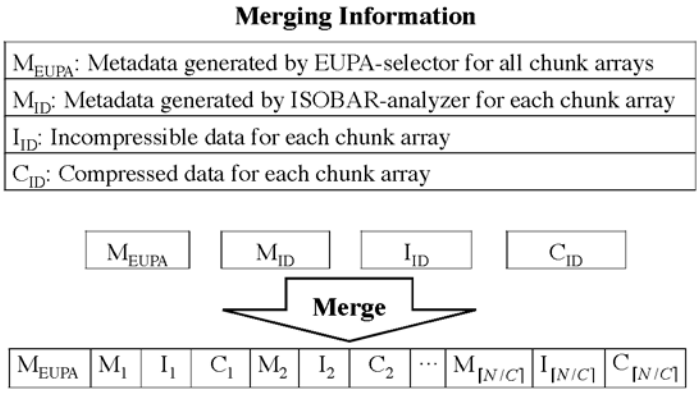


Figure 2.7: ISOBAR-compress merged output

2.3.1 Datasets

To calculate these metrics, we tested the ISOBAR-compress preconditioner on 24 scientific datasets of different data types, including single-precision and double-precision floating-point values and 64-bit integers, from various scientific disciplines including combustion, physics, astrophysics, and astronomy.

To briefly introduce those datasets, Table 2.1 is provided giving the name of simulation codes generating the datasets (or the capitalized prefix of their name), their data types, applications, variables and referencing materials. Detailed description for each dataset is in Appendix A. All tested datasets' statistical characteristics are portrayed in Table 2.3. Most of the datasets consist of unique values (see Equation 2.4). V is the original vector, V_{Unique} is the array composed of elements from V, with duplicates removed. The datasets also have high degrees of random entropy (see Equation 2.5); $H(V)$ is the Shannon entropy [76] (see Equation 2.6) of the input vector, where p_i is the probability of a given element in the vector V, and $Random(|V|)$ is a randomly generated vector consisting of all unique elements with the same size of the original V. The randomness value reflects how close the Shannon entropy of the scientific data is to that of a truly 100% random dataset with the same number of double elements [12].

$$Unique\ Value = \frac{|V_{Unique}|}{|V|} \times 100\% \tag{2.4}$$

$$Randomness = \frac{H(V)}{H(Random(|V|))} \times 100\% \tag{2.5}$$

$$H(V) = -\sum_{i=1}^{N} p_i log_2 p_i \tag{2.6}$$

Table 2.3: Statistical Information about Test Datasets

Dataset	Data Type	Set Size (MB)	Number of Elements (millions)	Unique Value (percent)	Shannon Entropy	Randomness (percent)
gts_phi_l	double	42	5.5	99.9	12.05	99.9
gts_phi_nl	double	42	5.5	99.9	12.05	99.9
gts_chkp_zeon	double	18	2.4	99.9	14.68	99.9
gts_chkp_zion	double	18	2.4	99.9	15.12	99.9
xgc_igid	64-bit integer	146	19.2	22.6	13.81	100.0
xgc_iphase	8 doubles	1170	153.4	7.7	12.32	76.4
s3d_temp	single	77	20.2	45.9	12.21	95.4
s3d_vmag	single	77	20.2	49.9	12.81	99.9
flash_velx	double	520	68.1	100	24.34	100
flash_vely	double	520	68.1	100	25.74	100
flash_gamc	double	520	68.1	100	11.26	100
msg_bt	double	254	33.3	92.9	23.67	94.7
msg_lu	double	185	24.2	99.2	24.47	99.7
msg_sp	double	276	36.2	98.9	25.03	99.7
msg_sppm	double	266	34.8	10.2	11.24	44.9
msg_sweep3d	double	119	15.7	89.8	23.41	97.9
num_brain	double	135	17.7	94.9	23.97	99.5
num_comet	double	102	13.4	88.9	22.04	93.1
num_control	double	152	19.9	98.5	24.14	99.6
num_plasma	double	33	4.4	0.3	13.65	61.9
obs_error	double	59	7.7	18.0	17.80	77.8
obs_info	double	18	2.3	23.9	18.07	85.3
obs_spitzer	double	189	24.7	5.7	17.36	70.7
obs_temp	double	38	4.9	100.0	22.25	100.0

Table 2.4: ISOBAR-analyzer's Predictions

Dataset	HTC†?	HTC Bytes(%)	Improvable?
gts_chkp_zeon	Yes	75%	Yes
gts_chkp_zion	Yes	75%	Yes
gts_phi_l	Yes	75%	Yes
gts_phi_nl	Yes	75%	Yes
xgc_igid	Yes	37.5%	Yes
xgc_iphase	Yes	75%	Yes
s3d_temp	Yes	25%	Yes
s3d_vmag	Yes	50%	Yes
flash_gamc	Yes	62.5%	Yes
flash_velx	Yes	75%	Yes
flash_vely	Yes	75%	Yes
msg_bt	No	0%	No
msg_lu	Yes	75%	Yes
msg_sp	Yes	62.5%	Yes
msg_sppm	No	0%	No
msg_sweep3d	Yes	50%	Yes
num_brain	Yes	75%	Yes
num_comet	Yes	37.5%	Yes
num_control	Yes	75%	Yes
num_plasma	No	0%	No
obs_error	No	0%	No
obs_info	Yes	75%	Yes
obs_spitzer	No	0%	No
obs_temp	Yes	75%	Yes

† Hard-to-compress dataset

2.3.2 Overall Performance of Identification and Improvement

Table 2.4 and 2.5 show that 19 out of 24 (79%) datasets were classified as hard-to-compress, 19 of the 19 (100%) were identified by ISOBAR-compress as improvable, and all datasets showed improvement of both compression ratio and compression/decompression throughput (speed) with ISOBAR-compress.

One such example is the *flash_velx* dataset. While using standard compression methods such as `zlib` and `bzlib2` to reduce this data, we achieved compression ratios (CRs) of 1.113 with

Table 2.5: Performance Comparison

Dataset	zlib		bzlib2		TP_A[3]	ISOBAR-CR Preference		ISOBAR-Sp Preference	
	CR[1]	TP_C (MB/s)[2]	CR	TP_C (MB/s)	(MB/s)	CR	TP_C (MB/s)	CR	TP_C (MB/s)
gts_chkp_zeon	1.040	14.10	1.022	3.55	500.40	**1.182**	**24.35**	**1.140**	**104.99**
gts_chkp_zion	1.044	13.87	1.027	3.57	501.60	**1.187**	**24.60**	**1.150**	**111.66**
gts_phi_l	1.041	14.20	1.020	3.55	501.83	**1.186**	**14.92**	**1.160**	**66.36**
gts_phi_nl	1.045	14.11	1.018	3.57	501.26	**1.180**	**15.41**	**1.157**	**65.65**
xgc_igid	3.003	1.12	3.120	5.99	505.33	**3.368**	**5.34**	**2.962**	**100.91**
xgc_iphase	1.362	6.71	1.377	3.63	501.79	**1.589**	**4.21**	**1.571**	**76.83**
s3d_temp†	1.336	7.25	1.452	3.25	513.15	**2.063**	**8.95**	**1.831**	**53.14**
s3d_vmag†	1.190	11.12	1.210	3.33	516.75	**1.774**	**8.50**	**1.604**	**104.73**
flash_gamc	1.289	19.50	1.281	3.87	503.05	**1.557**	**16.40**	**1.532**	**245.19**
flash_velx	1.113	12.70	1.084	3.61	501.34	**1.319**	**17.30**	**1.308**	**455.83**
flash_vely	1.135	12.01	1.091	3.62	501.57	**1.319**	**60.12**	**1.307**	**444.75**
msg_bt	1.131	13.54	1.102	3.79	495.98	*NI*[*]	*NI*	*NI*	*NI*
msg_lu	1.057	14.52	1.021	3.55	499.92	**1.298**	**20.19**	**1.246**	**235.21**
msg_sp	1.112	17.52	1.075	3.66	502.61	**1.330**	**5.16**	**1.304**	**106.65**
msg_sppm	7.436	9.14	6.932	1.35	495.02	*NI*	*NI*	*NI*	*NI*
msg_sweep3d	1.093	13.46	1.277	3.21	501.72	**1.344**	**6.61**	**1.287**	**78.86**
num_brain	1.064	14.01	1.042	3.13	503.51	**1.276**	**10.08**	**1.238**	**226.51**
num_comet	1.160	13.78	1.172	3.66	496.75	**1.236**	**4.83**	**1.215**	**21.13**
num_control	1.057	14.78	1.029	3.11	501.90	**1.143**	**12.52**	**1.126**	**65.10**
num_plasma	1.608	19.50	5.789	0.48	503.44	*NI*	*NI*	*NI*	*NI*
obs_error	1.448	8.79	1.338	3.58	502.51	*NI*	*NI*	*NI*	*NI*
obs_info	1.157	15.42	1.213	3.41	503.47	**1.292**	**5.28**	**1.249**	**228.91**
obs_spitzer	1.228	10.42	1.721	4.14	503.25	*NI*	*NI*	*NI*	*NI*
obs_temp	1.035	14.70	1.024	3.03	502.95	**1.142**	**22.89**	**1.125**	**96.62**

† Single-precision floating-point dataset

[*] *NI*: Not Identified—the dataset is identified as non-improvable by ISOBAR-compress

[1] CR: Compression Ratio (see Equation 2.1)

[2] TP_C (MB/s): Compression throughput in MB (megabyte) per second

[3] TP_A (MB/s): ISOBAR-analysis throughput in MB per second

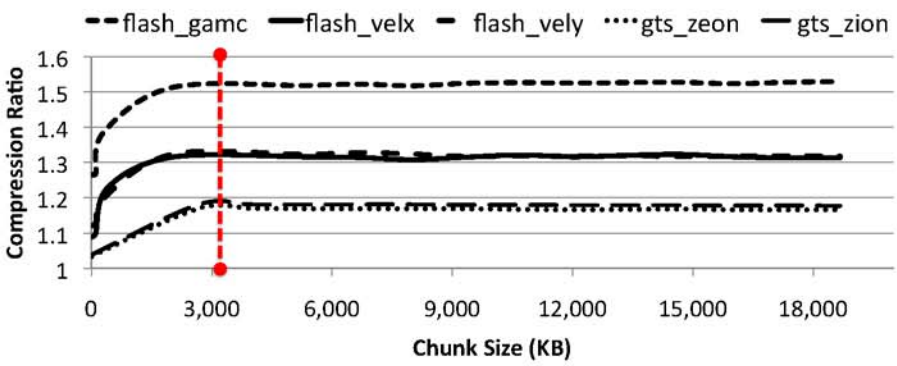


Figure 2.8: Chunking size for settled compression ratios

`zlib` and 1.084 with `bzlib2`. However, when compressing the same data with ISOBAR-compress, the CRs for `zlib` and `bzlib2` were enhanced by 17.52% with 35.89 times compression speed-up and 20.7% with 126.27 times compression speed-up, respectively.

2.3.3 Effect of ISOBAR-analyzer

Compression efficiency is sensitive to message length and input chunk size for most lossless compression techniques that adapt based on calculated statistics of the subject data [81]. We did experiments on five datasets and conclude that the proper chunk (or block) size is about 375,000 elements (doubles), which is about 3 MB (shown in Figure 2.8), which is consistent with previous conclusions that 3 MB is statistically reasonable [81, 38]. The compression ratios given by ISOBAR-compress start being consistent with the chunking size at this point. With the chunk size guaranteeing consistent compression ratio, the ISOBAR-analyzer identified 19 of 24 datasets as "improvable hard-to-compress" datasets in terms of CR and compression/decompression speed-up by the ISOBAR-compress method. These results are shown in Table 2.5.

By using ISOBAR-compress with `bzlib2`, comparing to standard `bzlib2`, all 19 datasets experienced an improved compression ratio and throughput. The improvement of compression ratio ($\Delta CR(\%)$) was in the range of [5.24%, 46.6%], while the improvement in speed-up (Sp) was in the range of [1.319, 16.607] times faster (see Equation 2.3).

Using ISOBAR-compress with `zlib`, comparing to standard `zlib`, all 19 datasets showed better compression ratios, and all 19 had gain in throughput (speed). The $\Delta CR(\%)$ was in the range of [4.7%, 37.0%] and the Sp was in the range of [1.533, 35.89].

2.3.4 Optimization with EUPA-selector

As shown in Table 2.6 with the speed preference chosen and Table 2.7 with the CR preference chosen, combining ISOBAR-compress with general-purpose lossless compression such as `zlib` and `bzlib2` will improve the compression efficiency yielded by using these standard compressors as standalone utilities. Since the EUPA-selector boasts superb flexibility of use, despite inconsistencies in the relative performances of `bzlib2` and `zlib`, end-users will be able to get the performance of a (relatively) speedy compression routine with a satisfactory compression ratio. For example, if an end user wishes to compress the *msg_lu* dataset, prioritizing the compression/decompression throughput (speed) above a higher compression ratio will cause EUPA-selector to choose `zlib` with byte-level column-linearization (the fastest technique, while still offering a compression ratio superior to standard compression). In the worst case scenario, if the dataset to be compressed is not identified as improvable, the EUPA-selector will offer the optimal choice of the standard compression method and linearization scheme. To extend to the user complete flexibility of use, explicit specification of input parameters (fixing the compression method and the linearization strategy) is also permitted by EUPA-selector.

2.3.5 Single-Precision Data Compression

Although scientific simulations often produce double-precision floating-point data (8-byte elements), for archival and community sharing purposes, the datasets often get converted to single-precision (4-byte elements) format similar to the "s3d" datasets in our experiments. Moreover, in some research activities, such as hurricane prediction in climate studies, original raw datasets for analysis only consist of single-precision floating-point values.

To support our claim that ISOBAR-compress can be used for data other than double-precision floating-point values, we tested our methodology on 2 single-precision data sets (see Table 2.8). Both data sets were identified as improvable by ISOBAR-compress. As shown in Table 2.8, both compression ratio and compression/decompression throughput (speed) were enhanced via ISOBAR-compress. For example, while using ISOBAR-compress with CR (compression ratio) preference compressing for the "s3d" temperature dataset, compared to the compressor with better compression ratio, $\Delta CR = 42.08\%$, and compression speed-up ($Sp = 2.758$). If ISOBAR-Sp is chosen, compared to the better compression throughput compressor, for the same "s3d" dataset, $\Delta CR = 37.05\%$ and $Sp = 7.329$.

2.3.6 Consistent Improvement over the Entire Simulation

Typically, scientific simulations generate many intermediate datasets to allow trend analysis and prediction. For example, a single run of the GTS simulation [78] will generate spatial data for approximately $300,000$ time steps. To show that both the EUPA-selector and ISOBAR-analyzer

Table 2.6: Improvement of ISOBAR-Sp Preference

Dataset*	LS[1]	ΔCR(%)[2]	Sp[3]
gts_chkp_zeon	Row	9.62	7.447
gts_chkp_zion	Row	10.15	8.050
gts_phi_l	Row	11.43	4.673
gts_phi_nl	Row	10.72	4.653
xgc_iphase	Column	15.35	11.450
flash_gamc	Row	18.85	12.576
flash_velx	Row	17.52	35.899
flash_vely	Row	15.15	37.032
msg_lu	Column	17.88	16.199
msg_sp	Column	17.267	6.087
msg_sweep3d	Column	17.75	5.859
num_brain	Row	16.35	16.168
num_comet	Row	4.74	1.533
num_control	Row	6.53	4.405
obs_info	Row	7.95	14.845
obs_temp	Row	8.70	6.573

[1] LS: Linearization Strategy

[2] ΔCR(%): Percentage improvement of compression ratio (Eq. 2.2) compared to the alternative with the highest compression throughput

[3] Sp: Compression speed-up (Eq. 2.3)

* EUPA-selector selected zlib as the better lossless compression technique for all datasets above

Table 2.7: Improvement of ISOBAR-CR Preference

Dataset*	LS[1]	ΔCR(%)[2]	Sp[3]
gts_chkp_zeon	Row	13.65	1.727
gts_chkp_zion	Row	13.69	1.774
gts_phi_l	Row	13.93	1.051
gts_phi_nl	Row	12.92	1.092
xgc_iphase	Column	15.39	1.160
flash_gamc	Row	20.79	0.841
flash_velx	Row	18.51	1.362
flash_vely	Row	16.21	5.006
msg_lu	Column	22.80	1.390
msg_sp	Column	19.60	0.295
msg_sweep3d	Column	5.24	1.410
num_brain	Row	19.92	0.719
num_comet	Row	5.46	1.319
num_control	Row	8.13	0.847
obs_info	Row	6.512	1.548
obs_temp	Row	10.34	1.557

[1] LS: Linearization Strategy

[2] ΔCR(%): Percentage improvement of compression ratio (Eq. 2.2) compared to the alternative with the best compression ratio

[3] Sp: Compression speed-up (Eq. 2.3)

[*] EUPA-selector selected bzlib2 as the better lossless compression technique for all datasets above

Table 2.8: Performance on Single-precision Datasets

	Dataset*	LS[3]	ΔCR (%)	Sp
ISOBAR-CR[1]	s3d_temp	Column	42.08	2.758
	s3d_vmag	Row	46.67	2.552
ISOBAR-Sp[2]	s3d_temp	Column	37.05	7.329
	s3d_vmag	Row	34.79	9.418

[1] Performance of ISOBAR-CR preference

[2] Performance of ISOBAR-Sp preference

[3] LS: Linearization Strategy selected by EUPA-selector

[*] EUPA-selector selected bzlib2 for ISOBAR-CR and zlib for ISOBAR-Sp as the better lossless compression technique

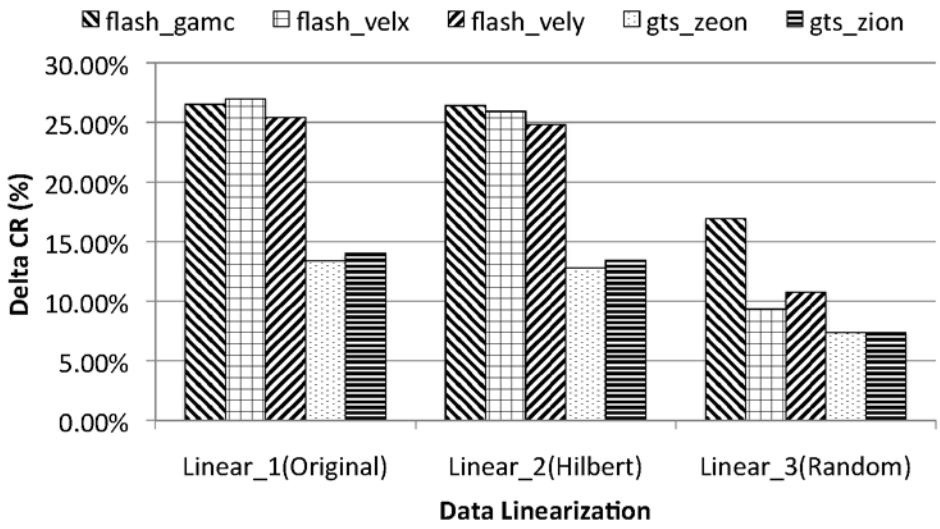


Figure 2.9: Compression ratio improvement ($\Delta CR(\%)$, Eq. 2.2) for multiple datasets with different linearization schemes: original order, Hilbert-linearized order, and random order

perform consistently well throughout an entire simulation, we tested our technique on GTS simulation data.

Results of the experiments for linear and nonlinear potential fluctuation values show that not only all choices for time steps given by EUPA-selector are the same (using `bzlib2` with row-linearization), but also that all time step datasets are identified as improvable by ISOBAR-compress. For the *linear* regime of potential fluctuation values, the average ΔCR was 14.4% with a standard deviation of 1.8% and the average Sp was 5.952 with a standard deviation of 0.065; for the *nonlinear* regime of potential fluctuation values, the average ΔCR was 13.4% with a standard deviation of 2.7% and the average Sp was 3.749 with a standard deviation of 0.053.

2.3.7 Robustness with Different Data Linearization

Data is constantly linearized. When data is generated by simulation codes, it is linearized in memory by the application. For example, XGC fusion simulations have multiple means of linearization over the number of variables; particles are mapped onto a mesh and characterized by radial, poloidal, and toroidal dimensions. On disk, data is linearized in various ways, such as Hilbert Space-Filling curves (used in querying multidimensional data) [51], to improve the efficiency of data retrieval.

We argue that by applying ISOBAR-compress as a preconditioner to the general-purpose lossless compression schemes, the performance does not significantly vary based on means of

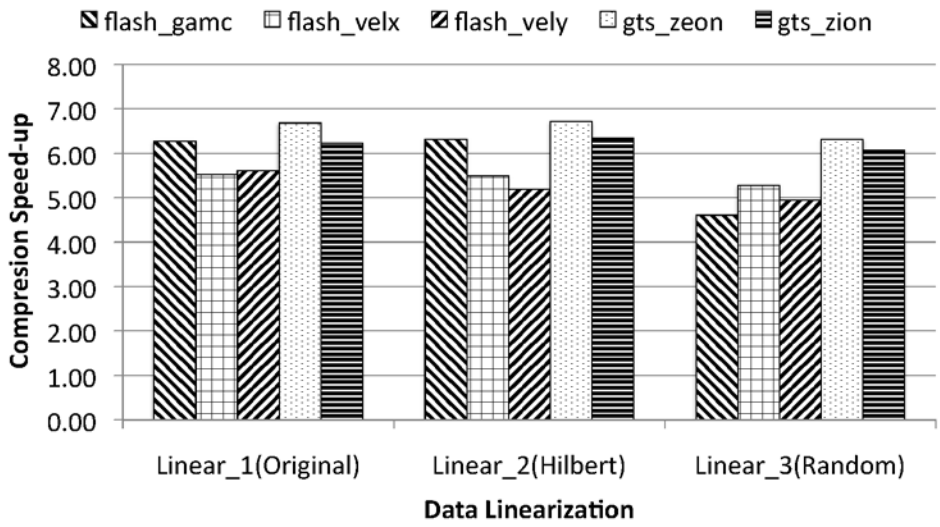


Figure 2.10: Compression speed-up (Sp) for multiple datasets with different linearization schemes: original order, Hilbert-linearized order, and random order

data linearization. As shown in Figure 2.9, improvement of compression ratios stays almost constant for the data linearized in a specific linearization similar to Hilbert Curve [51] datasets; even in the worst case scenario (where data is totally random), ISOBAR-compress still enhances the compression ratio by approximately 10% as compared to standard compression. Moreover, Figure 2.10 shows that the compression throughput (speed) is also consistent with various data linearizations.

2.3.8 Faster Decompression Throughput

Besides compression ratio and compression throughput (speed), ISOBAR-compress also improves the throughput (speed) of the decompression process. Decompression throughputs of both standard compressor and ISOBAR-compress are listed in Table 2.9, also the speed-ups of ISOBAR-compress, comparing to the faster one of either standard `bzlib2` or `zlib`, is included. In Table 2.9, two datasets' throughputs increase from about 100 MB/s to more than 1400 MB/s, which means the speed-ups are about 14.0, 15 of 19 had speed-ups greater than 3.0, all the other datasets' decompression throughputs are improved. Test runs on permuted data sets yield approximately the same improvement as the run on the original, non-permuted data.

Table 2.9: Decompression Throughput Comparison

Dataset	zlib (MB/s)	bzlib2 (MB/s)	ISOBAR[1] (MB/s)	Sp[2]
gts_chkp_zeon	115.22	10.48	**517.89**	4.5
gts_chkp_zion	110.38	10.57	**551.90**	5.0
gts_phi_l	114.41	10.00	**366.25**	3.2
gts_phi_nl	117.97	9.90	**358.21**	3.0
xgc_igid	177.69	21.08	**341.50**	1.9
xgc_iphase	138.99	7.49	**388.87**	2.8
s3d_temp†	113.80	6.26	**250.46**	2.2
s3d_vmag†	103.69	6.73	**424.79**	4.1
flash_velx	113.95	10.51	**1617.02**	14.2
flash_vely	112.03	10.53	**1538.98**	13.7
flash_gamc	113.41	12.02	**940.91**	8.3
msg_lu	112.51	10.51	**866.21**	7.7
msg_sp	106.77	10.68	**527.18**	4.9
msg_sweep3d	114.43	6.89	**446.49**	3.9
num_brain	114.47	6.55	**908.65**	7.9
num_comet	123.08	7.69	**145.73**	1.2
num_control	122.13	7.28	**373.63**	3.1
obs_info	118.61	7.27	**910.12**	7.7
obs_temp	114.10	6.59	**511.98**	4.5

† Single-precision floating-point dataset

[1] ISOBAR: Decompression throughput using ISOBAR-compress with speed preference

[2] Sp: Decompression speed-up comparing ISOBAR throughput to the faster alternative of either bzlib2 or zlib

Table 2.10: Comparison among ISOBAR-compress, FPC, and fpzip

Data Set	ISOBAR-Sp			FPC			fpzip		
	CR[1]	TP_C[2]	TP_D[3]	CR	TP_C	TP_D	CR	TP_C	TP_D
gts_chkp_zeon	1.140	104.99	517.89	1.018	38.22	39.24	1.096	35.80	29.48
gts_chkp_zion	1.150	111.66	551.90	1.025	38.42	38.98	1.100	36.47	30.20
gts_phi_l	1.160	66.36	366.25	1.077	24.06	23.93	1.182	38.66	31.44
gts_phi_nl	1.157	65.65	358.21	1.072	24.10	24.06	1.177	38.38	31.23
xgc_igid	2.962	100.91	341.50	1.960	87.84	87.85	2.736	13.63	12.84
xgc_iphase	1.571	76.83	388.87	1.360	17.66	17.43	1.535	44.89	36.12
flash_gamc	1.532	245.19	940.91	1.416	91.64	90.06	1.620	38.70	31.72
flash_velx	1.308	455.83	1617.02	1.265	49.61	49.70	1.342	36.77	31.39
flash_vely	1.307	444.75	1538.98	1.294	53.80	53.37	1.435	38.47	32.11
mean	1.476	185.80	735.73	1.276	47.26	47.18	1.469	35.75	29.61

[1] CR: Compression Ratio (see Equation 2.1)

[2] TP_C: Compression throughput tested on Lens system (MB per second)

[3] TP_D: Decompression throughput tested on Lens system (MB per second)

2.4 Related Work

While the idea of preconditioning data for standard compression routines is relatively unexplored (as far as we know), there are many instances in which lossless compression is required in the information storage and retrieval community. For this reason, there is a wealth of related work on the development of other lossless reduction tools. In this section, we acknowledge and address the relevance and comparative performance (in terms of compression ratio and throughput) of these standalone utilities to those obtained via the ISOBAR-compress workflow.

One such tool is PFOR [85], which aims to reduce the I/O bottleneck in the compression routine by extracting maximum instructions per cycle from modern CPUs. PFOR [85] (and some similar variants including PDICT [85] and PFOR-DELTA [85]) are specifically designed with the aim of eliminating *if-then-else* constructs and value dependencies in prediction and encoding of data. This strategy makes the data being compressed or decompressed fully loop-pipelineable by modern compilers and allows for out-of-order execution on modern CPUs while achieving high Instructions Per Cycle efficiency. Based on the experimental results provided in the paper, PFOR performs approximately 4 times faster than `zlib` and `bzlib2` for most data sets tested, though its compression ratios hardly beat those obtained with `zlib` and `bzlib2` (in some cases, the ratio is even 3 times worse). ISOBAR-compress was designed to improve both compression efficiency and throughput for general purpose lossless compression techniques, such as `zlib` and `bzlib2`. In the results section, we showed that ISOBAR-compress's performance almost

universally surpassed the performance of these compressors in terms of compression ratio and compression/decompression throughput.

General purpose lossless compression tools are also used with MapReduce techniques. For example, RCFile [38] is another reduction utility focusing on efficient storage of data produced from Hadoop MapReduce [18] based applications. RCFile compresses data tables sequentially column-wise, but uses no intelligent technique to determine how the input data should be optimally compressed, defaulting to `zlib` for all data columns. For this reason, compression yielded with RCFile varies in effectiveness, because it does not consider data-type of columns and blindly relies on `zlib` compression and a particular linearization, whereas other combinations of these two factors may be more fruitful in compression ratio and throughput. ISOBAR-compress's EUPA-selector takes care of this problem, and the workflow involves applying general-purpose lossless compression techniques, particularly after the ISOBAR-analyzer phase because of the varying sizes of values in the data columns (for example, a column storing numerical data may occupy 3 bytes per element in a MySQL database, whereas some string type columns may require approximately 100 bytes per element).

Tools, such as `fpzip` [53] and `FPC` [12] are also widely used in scientific database compression applications. Although `fpzip` and `FPC` are tools designed only for compression of 32 (`fpzip`) or 64 (`fpzip` and `FPC`) bit floating point data, their performance is also competitive (when compared to other lossless compression techniques) on other types of variously-sized scientific data. Hence, it is worthwhile introducing and comparing these utilities with ISOBAR-compress. Both `fpzip` and `FPC` are based on context modeling and prediction. `Fpzip` traverses data in a coherent order and then uses the corresponding n-dimensional (where n is the dimensionality of the data) Lorenzo predictor [39] to predict the subsequent values. It next maps the predicted values and actual values to their integer representations, and encodes the XORd́ residual between these values. Similarly, `FPC` first predicts values sequentially using two predictors (`fcm` [84] and `dfcm` [33]), and subsequently selects the closer predicted value to the actual. Lastly, `FPC` XORs the selected predicted value with the actual value, and compresses the leading-zero result. We compared the performance of ISOBAR-compress, `fpzip`, and `FPC` on identified scientific datasets in Table 2.10.

It is briefly worth addressing that there has also been a wealth of research done on the lossy compression front. Methods using `DCT` (Discrete Cosine Transform) [79] and wavelets [14] have been actively researched over the last few decades in the lossy compression realm. These works have primarily been applied in the context of visualization and geometric modeling applications. Lakshminarasimhan *et al.* explored the use of B-spline modeling to exploit monotonic properties in sorted data and reduce size by as much as 85% of the original [48]. Multi-dimensional histogram binning has also been used in the lossy reduction of simulation data. However, these techniques are not applicable in the reduction of certain types of simulation data and checkpoint/restart

data as inaccuracies in these values can completely throw off a simulation run and yield incorrect results.

2.5 Conclusion

In this paper, we proposed a new technique called ISOBAR-compress, which is a *preconditioner* to identify hard-to-compress datasets and improve compression efficiency for all general-purpose lossless compression *solvers*.

Given a dataset requiring lossless compression, our system first applies the ISOBAR-analyzer to determine whether ISOBAR-compress will improve the performance of lossless compression algorithms such as `zlib`. If performance can be improved, then the ISOBAR-partitioner will segment the dataset into two pieces: compressible and incompressible data. EUPA-selector will choose the optimal combination of standard lossless compression methods and proper multi-dimensional data linearization to meet the end-user's preference in terms of better compression ratio vs. much higher compression speed. Finally, ISOBAR-compress will merge the metadata, compressed data, and incompressible data into an output file. When tested on 24 scientific datasets, ISOBAR demonstrated high compression and decompression throughput as well as an improved compression ratio upon standard compressors.

Chapter 3

ISOBAR Hybrid Compression Interleaving for Large-scale Parallel I/O Optimization

3.1 Introduction

As exascale computing comes closer to becoming reality and more powerful High Performance Computing (HPC) systems become available, the complexity of scientific simulations and analyses has grown commensurately. Unfortunately, the level of disk I/O performance offered by these systems has not kept up, leading to a serious bottleneck in read and write performance in these applications. This problem is exacerbated by the increasing frequency of checkpoint operations performed by such computations due to their increasing vulnerability to node failures at this scale, which further adds to the I/O overload.

Ideally, the solution to the I/O bottleneck will involve both data reduction and parallel I/O access pattern optimization. Unfortunately, these two optimization methods have traditionally been in conflict. State-of-the-art I/O middleware solutions, such as ADIOS [55], HDF5 [83], and PnetCDF [52], have no native support for write compression in a parallel context, due to the complexity of handling the resultant non-uniform data, which requires synchronization between all nodes performing shared-file I/O. Further constraining the problem is the fact that scientists cannot sacrifice simulation fidelity, especially at checkpoints, which rules out lossy compression as a viable data reduction method. And yet, typical lossless compression techniques are ineffective on hard-to-compress floating-point data generally produced by such simulations.

However, we argue that these goals can in fact be complementary. Our key insight is that, by dynamically identifying a subset of highly-compressible data to process while asynchronously writing the remainder to storage, we can effectively hide the cost of compression and I/O synchro-

nization behind this transfer, thus rendering parallel write compression viable. This interleaving method is a natural fit for data staging architectures, where various data transformations can occur while data is "in transit" or in situ, from compute nodes to disk. Traditionally, staging has been used to compute statistical analyses or perform indexing operations [1, 2]. With interleaved compression and I/O, however, we can augment this functionality by performing compression as an in-situ transfer and storage optimization, as well.

The problem of identifying a highly-compressible subset of the original data is itself quite difficult, as scientific data is usually hard-to-compress with typical compression libraries. We argue that this is because I/O libraries optimized for scientific applications tend to view multi-byte data elements, such as floating-point values, as atomic units of data. Instead, by relaxing this notion and utilizing *byte-level analysis* of the scientific data, better results can be obtained. ISOBAR, or In Situ Orthogonal Byte Aggregate Reduction, enables exactly this sort of analysis for lossless compression, and has been shown to be effective on such datasets [70]. By modifying ISOBAR's analysis methods to partition the data into compressible and incompressible byte streams, we form an effective basis for interleaving the usage of computing resources. By transmitting the incompressible data over the network immediately, we can hide the cost of compressing the remainder and synchronizing for non-uniform I/O to write it to disk.

Therefore, we present a *hybrid compression-I/O* methodology for data reduction and I/O optimization. By employing ISOBAR analysis within a data staging architecture and incorporating the popular ADIOS I/O framework as our I/O backend, we implement effective parallel compression with state-of-the-art I/O performance. Furthermore, we develop a resource interleaving strategy to process the compressible and incompressible components of the data simultaneously, as identified by ISOBAR analysis. This enables immediate asynchronous transfer and writing of incompressible data while compression is applied concurrently to the remainder. Additionally, we develop a performance model for our methodology, which we demonstrate to have a high degree of accuracy through validation against empirical data.

Our system exhibits read and write performance gains proportional to the degree of data reduction, which ranges as high as 46% on scientific datasets. This would translate into an effective increase of 28 GB/s bandwidth over the peak aggregate throughput of 60 GB/s of uncompressed data offered by the leadership-class Lustre parallel filesystem at Oak Ridge National Labs [57]. Even under worst-case conditions, where the dataset is highly entropic and difficult to compress, we show that our system still maintains a gain in throughput over the state-of-the-art.

Although we demonstrate performance gains when using our methodology in a leadership-class HPC system (the Cray XK6 Titan cluster), the constantly changing architectural characteristics as well as the diversity of configurations of current and future HPC systems make it impossible to evaluate our methodology on every possible cluster configuration. Hence, it is critically

important to provide an analytical performance model, which would allow both robust prediction of performance characteristics on new hardware and aid in configuration choices for application developers. Our model predicts system performance on the Titan cluster within an average 0.53% error on the read/decompress side and 0.97% error on the write/compress side.

Our methodology and related components are organized as follows. First, precursory information on ISOBAR compression, the compression system for which we base our interleaving method on, as well as ADIOS, the parallel I/O middleware used for evaluation, are discussed in Section 3.2. Our hybrid compression-I/O method is discussed in earnest in Section 3.3: Section 3.3.1 describes the general interleaving strategy, Section 3.3.2 describes the ISOBAR analysis method as well as deduction of key ISOBAR parameters using information-theoretic analysis (as opposed to empirical bootstrapping used in previous work [70]), and Section 3.3.3 discusses integration of ISOBAR, ADIOS, and the interleaving methodology. Given the components in Section 3.3, we present the analytical performance model in Section 3.4. Finally, a detailed evaluation of our methodology and performance model for numerous real-world datasets are shown in Section 3.5, followed by related work and concluding remarks in Sections 3.6 and 3.7, respectively.

3.2 Background

3.2.1 ISOBAR Compression

ISOBAR is a lossless compression method that we built specifically for data that varies in compressibility on a byte-by-byte basis [70]. An ubiquitous example of such data is scientific floating-point data, where the exponent bits can be highly similar while the significand bits are highly entropic. To this end, ISOBAR first performs a *preconditioner* on the linearized input data, selecting data to compress based on its expected degree of compressibility. This is performed by the ISOBAR-analyzer. The analyzer's objective is to identify high-entropic content within a dataset that negatively impacts the compression efficiency and reduces the burden on the compressor from processing the components with low compression potential. This enhances the compression algorithm in both the compression throughput and the compression ratio.

The input data is considered as a matrix of bytes, where each row is an input value (e.g., a double-precision floating-point) and each column is an individual byte of the input value. The preconditioner counts the frequency of each byte on a column basis and marks that column as compressible if the distribution appears to be non-random. Once these columns are identified, any general purpose compressor may be used, but ISOBAR automatically chooses the best one by user preference (compression ratio vs. speed).

In the previous chapter, the threshold for determining compressibility was determined using empirical analysis based on dataset subsampling, placing the burden on the user. Section 3.3.2

describes a more robust analytical deduction of the threshold.

3.2.2 ADIOS

High-performance computing applications increasingly leverage I/O libraries, such as HDF5 (Hierarchical Data Format), ADIOS (Adaptable I/O System), and PnetCDF (Parallel Network Common Data Form), that allow scientists to easily describe the data to be written out and analyzed. These I/O libraries provide a high-performance I/O abstraction, efficiently handling collective I/O operations, synchronization and meta-data generation during shared-file writes. We choose to utilize ADIOS, since it has been shown to deliver performance improvements of up to 300% at scale [57, 66] on the Cray Jaguar leadership-class computing facilities at ORNL.

ADIOS provides an efficient componentization of the HPC I/O stack. Through an XML file, it provides the option to describe the data, and to choose optimal *transport methods* such as POSIX and MPI-IO, without the need to recompile the application codes. The data written using ADIOS is in the form of a native Binary Packed (BP) file, comprising of *process groups* – sets of variables described in XML configuration, typically tagged according to their functionality. For example, checkpoint and restart data is written under a single process group, as is analysis data.

3.3 Hybrid Compression-I/O for Data Staging Architectures

The prevailing I/O strategy in current peta-scale systems is to offload the burden of I/O to dedicated *staging nodes*, as shown in Figure 3.1. This allows minimal idle time on the compute nodes allocated for the simulation, as the I/O nodes handle the rate-limiting disk writes and reads collectively and network bandwidth is an order of magnitude faster than disk bandwidth. However, given the trend of ever-increasing simulation data sizes, combined with the need to checkpoint simulation state to minimize data loss in the face of node failure, the I/O offloading approach alone cannot keep up with the computational throughput available.

3.3.1 Method

A promising approach to aid in mitigating rate-limiting I/O is to write compressed data to the disk [82]. The aggregate reduction of data has the potential of partially alleviating the I/O bottleneck. However, there are a number of technical challenges in utilizing compression in the scientific computing environment. Most state-of-the-art compression algorithms do not provide enough compression throughput to justify the data reduction, and those that do sacrifice the compression ratio. Moreover, the target simulation data is notoriously "hard-to-compress;" traditional compression algorithms provide meager compression ratios [70]. In

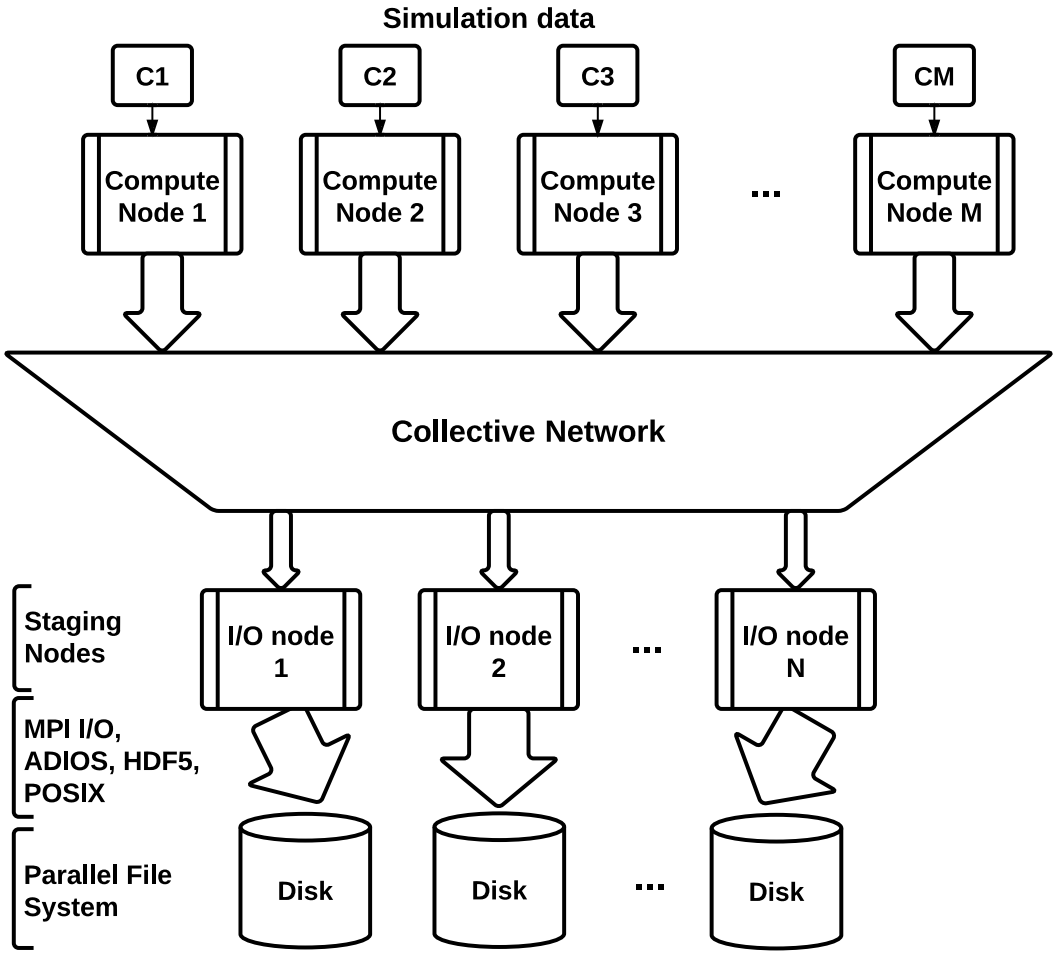


Figure 3.1: A peta-scale computing system with staging environment

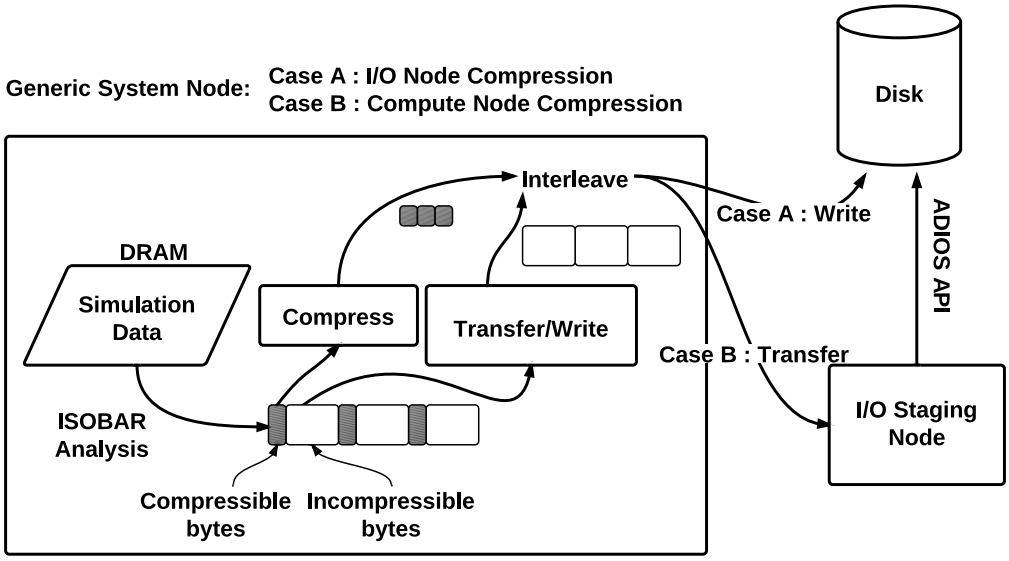
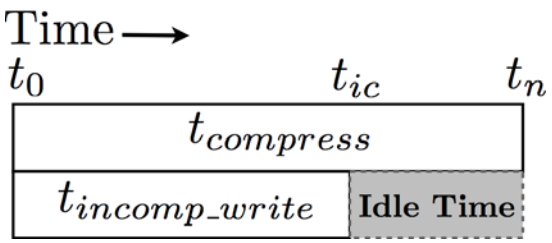


Figure 3.2: The hybrid compression-I/O method; interleaved compression may occur at the compute nodes or the I/O nodes

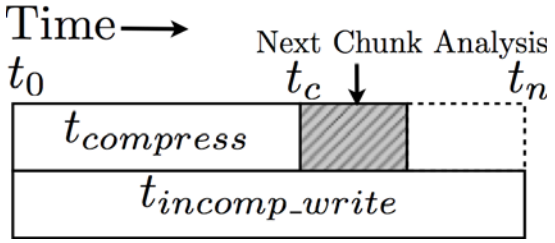
addition, compression introduces the non-trivial issue of managing the I/O of variable-sized chunks of data. This mandates devising a strategy to efficiently handle parallel I/O coordination, taking into account data organization and writer node synchronization while also keeping the overhead imposed by the associated metadata in check.

Current state-of-the art I/O frameworks such as ADIOS and HDF5 have no capability to compress and store simulation data when performing parallel filesystem writes. In this work, we utilize the data staging model, based on the ISOBAR technology, to write and simultaneously compress simulation data. In addition to using the data staging paradigm, interleaving compression and I/O can also be directly integrated into I/O frameworks. Especially important to our interleaving methodology is the fact that we can use ISOBAR to produce multiple streams of data that, once defined by the analysis portion of ISOBAR, can be operated on independently. This presents the perfect opportunity to hide the compression costs by asynchronously writing streams (the incompressible byte streams) while operating on the remaining streams (compressing the compressible byte streams). We call this the *hybrid compression-I/O* approach. The interleaving of compression and I/O helps to hide the compression costs, while the reduction in data size reduces disk costs.

Figure 3.2 illustrates our generic hybrid approach. ISOBAR analysis categorizes the data into two streams: compressible and incompressible bytes, and incorporates a small, constant-sized metadata block to each (containing the buffer size and the analysis array, a bitfield marking compressible byte columns). The compressible stream is compressed and then written to disk,



(a) Compression is the rate limiting factor.



(b) Compression interleaving results in free time usable for the analysis of the next chunk.

Figure 3.3: Possible scenarios for compression and incompressible data write times. $t_{compress}$ is time to compress the compressible byte stream and t_{incomp_write} is time to write the incompressible byte stream to disk

while the incompressible stream can be immediately written to disk. There is no dependency between the two streams, so we may choose to order the operations to our advantage. Thus, we compress the compressible stream while asynchronously writing the incompressible stream, followed by writing the compressible stream. This strategy has numerous benefits: we maximize resource utilization by performing network and I/O operations while compressing, and since ISOBAR compression throughput tends to be much higher than I/O bandwidth, it is possible to eliminate the compression costs entirely.

Two performance scenarios of interleaving arise based on the individual performance of compression and file writing, which are shown in Figure 3.3. The first scenario, shown in Figure 3.3a occurs when the uncompressed data is written before the remainder of the data is compressed. In this scenario, the compression time is the bottleneck, and so is only partially hidden by the writing of the incompressible data to disk. However, this case generally occurs when most of the data is deemed compressible, in which case the increased compression cost directly translates into substantial data reduction; thus, overall time-to-disk may still be reduced due to a higher compression ratio.

The second scenario, shown in Figure 3.3b, occurs if data compression finishes before the incompressible data has been fully written, and must wait to write the compressed data stream. In this scenario, compression time is completely hidden, and can be considered a “free” operation with respect to time. Also, the idle time can be used for other activities, such as running ISOBAR analysis on another chunk if available. A related scenario to this is when data compression and incompressible data writing finish at approximately the same time resulting in full utilization of all resources. This case also completely hides the compression costs.

3.3.2 ISOBAR Analysis

The viability of the hybrid compression-I/O approach is based on the ISOBAR analysis process, which identifies and divides compressible and incompressible byte streams. As mentioned in Section 3.2, the threshold parameter involved in identifying a byte-column as compressible is crucial for the compression methodology to work. Hence, we provide an in-depth discussion behind the ISOBAR analysis technique, its effectiveness with hard-to-compress datasets, and an analytic methodology for determining entropy-based compressibility within our method.

Lossless compression of hard-to-compress datasets is burdened by its highly entropic content embedded within. In information theory, Shannon entropy is a measure of information uncertainty, given by

$$H(X) = -\sum_{i=1}^{n} p(x_i) log_b p(x_i), \tag{3.1}$$

where X is a discrete random variable with the possible values $\{x1, \ldots, x_n\}$ and $p(X)$ is the probability mass function. Let X be a set of bits, so the corresponding base b is two. Figure 3.4 illustrates the probability that the bit value within the set is 1 with its corresponding entropy value; entropy is maximized for equal outcome probabilities ($H(X) = 1.0$ when $p(X = 1)$ is 50%) and minimized for fixed outcome probabilities ($H(X) = 0.0$ when $p(X = 1)$ is 0% or 100%).

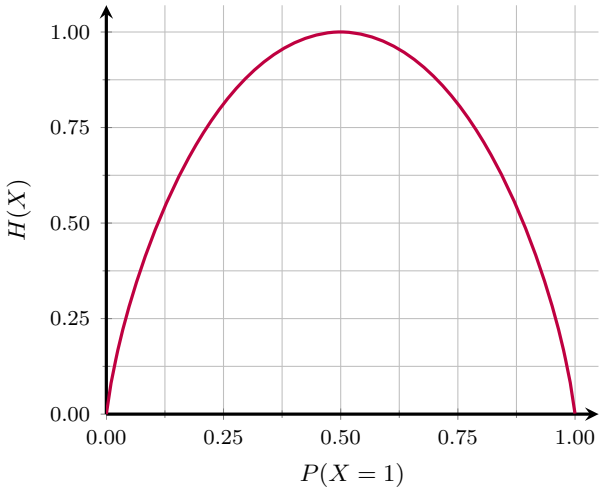
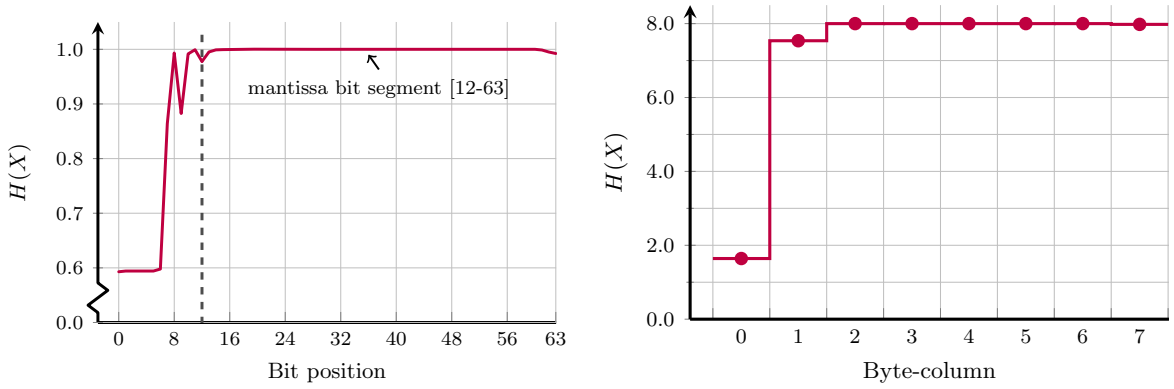


Figure 3.4: Relationship between entropy $H(X)$ and probability $P(X = 1)$ of bit value within set X is 1

Figure 3.5a shows the evaluation of entropy at each significant bit for the double-precision floating-point zion variable of a GTS simulation dataset, which is considered hard to compress.



(a) Bit position entropy.

(b) Byte-column entropy.

Figure 3.5: Calculated entropy of a zion double-precision floating-point variable dataset from a GTS simulation run at bit and byte-column granularity

Significant bit positions 18 to 60 of this variable, representing all but nine bits of the mantissa component of the floating-point number, have an entropy of 0.9999 ± 10^{-4}. Without a priori dataset knowledge, these bit positions can be considered random, with nearly maximal entropy. If we move to the significant byte level (byte-column), which is the granularity of most lossless compressors, we see similar patterns, as shown in Figure 3.5b (since there are $2^8 = 256$ possible values, the maximum entropy is 8).

Given that general lossless compressors work at the byte level and ineffectively perform on high-entropy ingress data, while scientific data (especially floating-point) has the characteristic of byte-column regularity rather than adjacent-byte regularity, ISOBAR analysis performs entropy evaluation at the byte-column level to determine compressibility. To do so efficiently, the relative frequency, or probability, is calculated for each byte-column. Then, the maximal relative frequency is compared against a threshold to determine compressibility. For example the relative frequency of each byte value is 0.39% for a uniform distribution (100%/256). In Figure 3.6, the maximum relative frequencies of byte-columns 1 and 2 of the GTS zion dataset are 1.47% and 0.41%, respectively. Entropy encoders work well for the non-uniform byte probabilities for byte-column 1, but not for the near-uniform byte probabilities for byte-column 2.

Thresholding is used to determine at what maximal relative frequency the byte-column is determined compressible. To determine a proper frequency threshold, we evaluate entropy encoders based on *codebooks*, which assign small-sized code words to variables with high-frequency and large-sized code words to variables with low-frequency. An example of such encoder is the Huffman prefix-free encoder, the most efficient of this type of encoder [81]. Since the probability mass function must be equal to 1, any relative frequency increase of x% for a given byte value

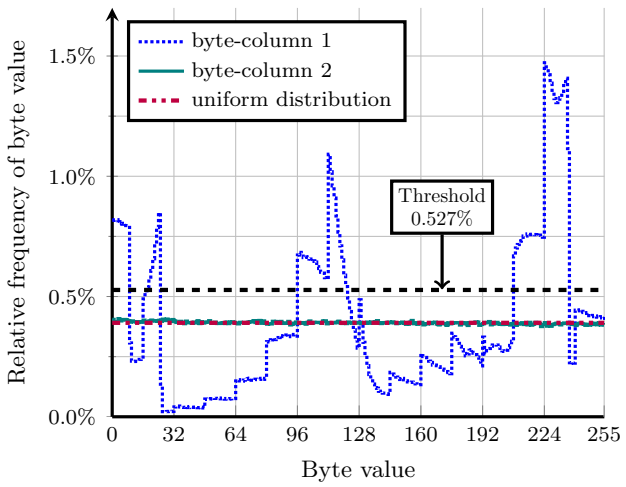


Figure 3.6: Relative frequency of each byte value from a sample set of byte-columns

must be offset by an accumulative relative frequency decrease of $x\%$ for the remaining byte values. Thus, by examining the span for half of the byte value's relative frequencies, we calculated that the relative frequency of 1.35/256 (0.527%) causes codebooks to begin assigning 7-bit code words to some of the higher frequency values, rather than 8. Therefore, we apply the threshold of 1.35 for ISOBAR analysis to determine byte-column compressibility.

3.3.3 Data Layout in ADIOS

The layout of our compressed and uncompressed data on disk is managed by ADIOS's self-describing file format (.bp), which is specifically designed to attain scalable high performance I/O, to support delayed consistency and data characterization, and to maintain compatibility with standard file formats. Also, since this file format is log-based [66], new data can be appended without incurring any additional overhead, irrespective of the number of processes or timesteps. As a result, the performance improvements we report using our hybrid framework over a single timestep can be maintained over multiple timesteps.

For our system, we define two ADIOS groups (or "data containers"): one for the compressed data, and one for the uncompressed data. Every writing process submits data to each of the two groups. Depending on the transport method chosen in the configuration file, ADIOS can store data from all the writing processes into a single shared file using collective MPI-IO or multiple files using POSIX I/O with a separate file handle for each writing process. Fast access to data for a specific process or timestep is supported via footer indexes that avoid the known limitation of header-based formats [68], where any change to the length of the file data requires moving the index.

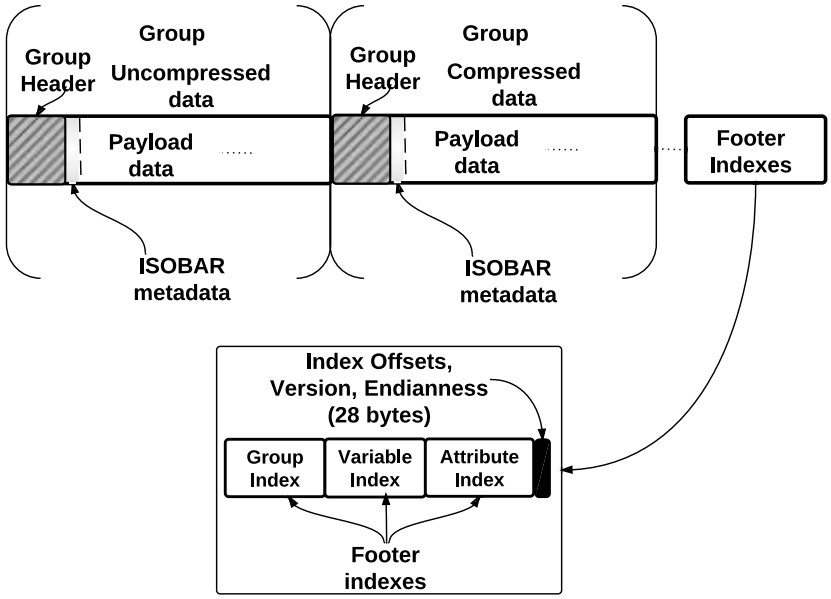


Figure 3.7: Data layout (with associated metadata) for a single timestep

Figure 3.7 shows the ADIOS data layout specific to our application for a single timestep. ADIOS handles the data organization among groups via local group headers and indexes. The ISOBAR metadata is stored first within each group, followed by the payload data. The metadata size is dependent on the ADIOS system configuration, the chunk size, and the compute-I/O node ratio. The relative ordering of groups is arbitrary, dependent on the order in which the processes submit data and on synchronization among them (via a coordination token) [56]. The global footer indexes shown are used for query-driven data retrieval (accessing a specific timestep or process output). In this work, we confine our focus to data layout for the write-all/read-all paradigm, which is ubiquitous in HPC computing, especially in checkpoint and restart operations. From the standpoint of future work, however, our framework can be adapted for the WORM paradigm by leveraging previous work [35] to optimize the data layout on disk. In addition, support for compression schemes [48, 47, 43] which offer up to 7 times reduction in datasizes can be extended. Together, these possibilities provide avenues for increasing throughput gains whilst maintaining read performance.

The metadata required to support this format is very small, requiring less than 0.01% overhead in the worst case, as summarized in Table 3.1. If applied to 1 GB of data, this translates into less than 100 KB of overhead. As shown in the table, the metadata can be split into two categories: that maintained specifically for ISOBAR, and that required by ADIOS. The exact amount of metadata is dependent on the transport method used (POSIX or MPI-IO) as well as the location of the compressor (compute node or I/O node). The overhead is minimal in

Table 3.1: Average Metadata Overhead for Different Interleaving Strategies

Test Case	**Average Metadata Overhead** (%)			
	ISOBAR		ADIOS	
	POSIX	MPI	POSIX	MPI
Base	0.00002	0.00002	0.00321	0.00148
ISOBAR at I/O nodes	0.00004	0.00004	0.00902	0.00417
ISOBAR at compute nodes	0.00007	0.00007	0.00978	0.00471
Serial ISOBAR	0.00004	0.00004	0.00806	0.00358

the base case (no compression), as expected, and maximal when compressing at the compute nodes, which generates the most individual streams of data. The metadata cost is also higher for the POSIX transport method, since this allocates one file per writing process, as opposed to the single shared file maintained by MPI-IO.

3.4 Performance Modeling

While we demonstrate that our hybrid compression-I/O methodology provides improved I/O performance in one testing environment, there are many supercomputing systems, each with widely-varying performance characteristics. Since our optimization algorithms exhibit a strong dependence on hardware parameters, it is important to devise an accurate performance model, so that we can generalize the results collected in Section 3.5 to other systems. This will enable application designers to estimate the benefit of compression given their particular hardware configuration, problem characteristics, etc. We therefore develop such a performance model, which we then validate against empirical data, as reported in Section 3.5.4.

3.4.1 Model Preliminaries

Given our target cluster architecture, we make some underlying assumptions in our model. We assume a fixed compute node to I/O node ratio, ρ, consistent with the majority of I/O staging frameworks currently in use. Furthermore, on each compute node, we assume fixed-size input *chunks* of size (C), which are all written following the bulk-synchronous parallel I/O model. This is a common mode of operation when writing checkpoint and restart data, which synchronously flushes the simulation state to file, then continues with the simulation. We also assume that the I/O staging framework (e.g., ADIOS) and the network architecture provide a relatively consistent I/O and transfer rate, respectively. As shown in numerous experiments

Table 3.2: Input Symbols for the Performance Models

Input Symbol	**Description**
C	The chunk size
ρ	Compute to I/O node ratio
θ	Throughput of the collective network between the compute and I/O nodes
δ	Size of the metadata
μ_r	Throughput of the disk reads
μ_w	Throughput of the disk writes
α	Fraction of the chunk that is compressible
σ	Compression ratio (compressed vs original)
T_{prec}	Throughput of the ISOBAR preconditioner
T_{comp}	Compression throughput (orig. size / time)
T_{decomp}	Decompression throughput (compressed size / time)

with ADIOS [55, 56], this is a reasonable assumption to make. Finally, we require some *a priori* information about the compression performance in order to accurately predict overall system performance. Fortunately, this can be gathered easily by running the ISOBAR analysis stage on a small, representative set of data to predict overall performance [70].

To provide a complete model, there are three cases of writing from compute nodes to disk that we wish to consider for comparative purposes. The first is when no compression is performed, and data is written directly to disk (through the I/O nodes). This forms our base case to compare the other compression methods against, and allows us to check the sanity of our model before looking at the more complex compression models. The remaining cases use ISOBAR hybrid compression-I/O method, but in different locations. The second case compresses at the compute-node level. If time-to-disk is our sole optimization metric, then we expect the second method of compute-node compression to perform best, exhibiting the greatest aggregate compression throughput due to the large number of compute cores utilized (by constrast, compressing at the I/O nodes yields less aggregate compression throughput by a factor of ρ, which is 8 in our experiments, but can be much higher). Our third and final case is compressing at the I/O-node level. This case is important as future staging architectures shift toward dedicating compute nodes strictly to simulation work [54], relying on asynchronous RDMA to offload data to I/O nodes and prevent simulation stalls.

In this work, we refer to the I/O node as a staging node on the system that receives the

Table 3.3: Output Symbols for the Performance Models

Output Symbol	**Description**
t_{prec}	Time to run the ISOBAR preconditioner on the data
$t_{compress}$	Time to compress the data (algorithm dependent)
$t_{transfer}$	Total transfer time [1]
$t_{comp\_transfer}$	Transfer time for compressible data [1]
$t_{incomp\_transfer}$	Transfer time for incompressible data [1]
t_{write}	Time to write data to the disk
t_{comp_write}	Time to write compressible data to the disk
t_{incomp_write}	Time to write incompressible data to the disk
t_{write_depend}	Time for all the dependencies to complete before writing the compressible byte stream
$t_{decompress}$	Time to decompress the data (algorithm dependent)
t_{incomp_read}	Time to read the incompressible data from the disk
t_{comp_read}	Time to read the compressible data from the disk
$t_{combine}$	Time to reconstruct data from compressible and incompressible portions
$t_{combine\_depend}$	Time until all data is ready to be recombined into the original chunk
t_{comp_ion}	Intermediate processing time for handling compressible data at I/O nodes
t_{incomp_cn}	Intermediate processing time for handling incompressible data at compute nodes
t_{total}	Total end-to-end data transfer time
τ	Aggregate throughput

[1]Interpretation of transfer direction based on context of usage i.e., compute to I/O for writes and I/O to compute for reads.

data from the compute nodes and sends it to the OSS (Object Storage Servers), which manage writing of the data to the Lustre File System. We do not operate at these file system nodes.

We will build the models in increasing order of complexity: the base case of no compression, compression at the I/O nodes, and compression at the compute nodes. Tables 3.2 and 3.3 summarize the symbols for parameters and output variables used in the model. In all scenarios, the aggregate I/O node throughput τ is given by

$$\tau = \frac{\rho \cdot C}{t_{total}}, \tag{3.2}$$

where ρ is the number of compute nodes and C is the chunk size.

3.4.2 Base Case: No Compression

In this scenario, we simply transfer the simulation data from the compute nodes to the I/O nodes ($t_{transfer}$), followed by writing the data to file (t_{write}), in a synchronous manner. The end-to-end transfer time for a chunk of data from the compute nodes to the disk ($t_{total} = t_{transfer} + t_{write}$) is similarly simple, given our assumptions on aggregate network and disk bandwidths:

$$t_{transfer} = \frac{C}{\theta} + \frac{C}{\theta}\rho \tag{3.3}$$

$$t_{write} = \frac{C\cdot}{\mu_w}\rho \tag{3.4}$$

$$t_{total} = \frac{C}{\theta}(1+\rho) + \frac{C}{\mu_w}\rho \tag{3.5}$$

To reload the data from disk, the same operations occur in reverse, except with read throughputs instead of write throughputs.

3.4.3 I/O Node Compression Case

An overview of the compression-I/O workflow is shown in Figure 3.8. The I/O nodes implement the ISOBAR preconditioning analysis and interleave the disk writes of the incompressible byte-columns with compression.

The compute nodes merely forward the raw data to the I/O nodes. In our design, we issue the I/O operation for the incompressible bytes asynchronously, allowing us to concurrently perform compression. Thus, the writing of the compressed data waits (if necessary) on the incompressible stream writing to complete. This can be captured more intuitively with the task dependency graph shown in Figure 3.9. Each vertex represents a task and directed edges represent task dependencies. If each edge is weighted to be the completion time of the originating task, then the longest path from the head vertex to the tail vertex, also known as the *critical path*, gives

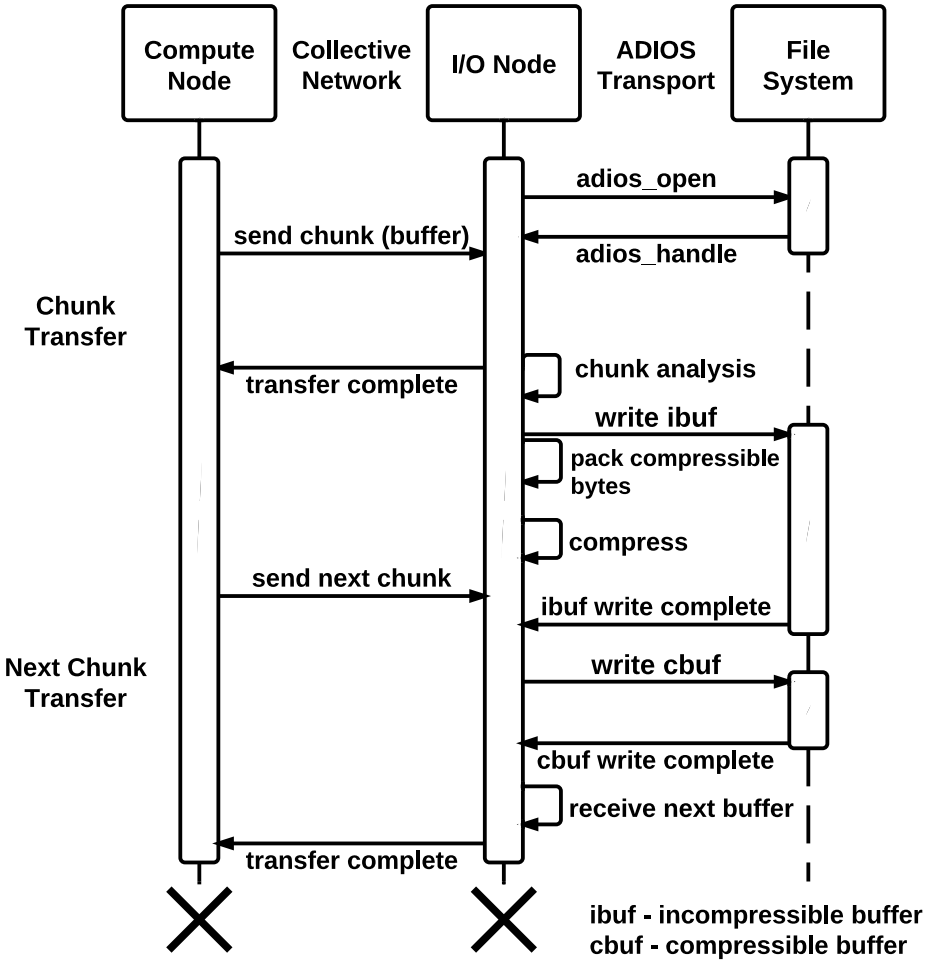


Figure 3.8: Compute-I/O interleaving strategy with compression at the I/O nodes

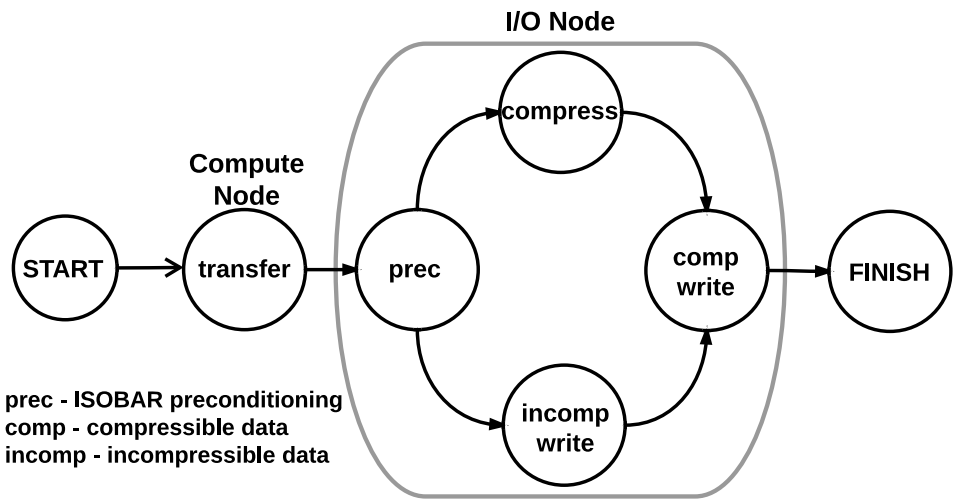


Figure 3.9: Task dependencies for the I/O node compression case

the overall run time of the interleaved process. Using this diagram, we can capture the overall runtime in a single equation. First, we define the cost of each individual task.

The total preconditioning time in this case is ρ times the preconditioning time of a single chunk. Moreover, the partitioning has to be handled by the I/O node. The partitioning throughput is approximately equal to the preconditioner throughput. Thus, the total preconditioning time is given by:

$$t_{prec} = 2\left(\frac{\rho C}{T_{prec}}\right) \tag{3.6}$$

The transfer time is the same as for the base case. Since the compression takes place at the I/O node, the total amount of compressed data is ρ times the compressible part of the chunk from every compute node:

$$t_{compress} = \frac{\alpha C}{T_{comp}}\rho \tag{3.7}$$

By the same logic, the total compressible and incompressible write times are given by:

$$t_{incomp\_write} = \frac{(1-\alpha)C}{\mu_w}\rho \tag{3.8}$$

$$t_{comp\_write} = \frac{(\alpha\sigma C + \delta)}{\mu_w}\rho \tag{3.9}$$

Looking at Figure 3.9, we see that incompressible write (t_{incomp_write}) and compression

($t_{compress}$) are interleaved. Thus, the length of the critical path, and therefore the overall writing time, can be calculated as

$$\begin{aligned} t_{total} &= t_{transfer} + t_{prec} \\ &\quad + \max(t_{compress}, t_{incomp\_write}) \\ &\quad + t_{comp\_write}. \end{aligned} \tag{3.10}$$

The reading stage of restoring the chunks into memory from disk requires the dependency graph to be inverted (that is, each directed edge reversed). This maintains the interleaving property of decompression and reading of incompressible byte-columns. The preconditioner task is replaced with reconstruction, which reorders the decompressed byte streams and the incompressible byte stream to their original locations in memory. This is captured in the following model:

$$t_{comp\_read} = \frac{(\alpha\sigma C + \delta)}{\mu_r}\rho \tag{3.11}$$

$$t_{incomp\_read} = \frac{(1-\alpha)C}{\mu_r}\rho \tag{3.12}$$

$$t_{decompress} = \frac{\alpha\sigma C}{T_{decomp}} \tag{3.13}$$

$$t_{transfer} = \frac{(1+\rho)C}{\theta} \tag{3.14}$$

The overall reading time, assuming constant chunk combination time ($t_{combine}$) for fixed sized chunks can be calculated as

$$\begin{aligned} t_{total} &= t_{comp\_read} \\ &\quad + \max(t_{decompress}, t_{incomp\_read}) \\ &\quad + t_{combine} + t_{transfer}. \end{aligned} \tag{3.15}$$

The equations 3.10 and 3.15 allow us to model the end-to-end times for the write and the read scenarios respectively for the I/O node compression case.

3.4.4 Compute Node Compression Case

The integration of the ISOBAR hybrid compression-I/O method into the compute nodes is a more nuanced task. Figure 3.10 shows the general workflow of the interleaved compression and network-I/O, and Figure 3.11 shows the corresponding task-dependency graph for this scenario.

After ISOBAR analysis preconditioning, the incompressible byte-columns are sent asyn-

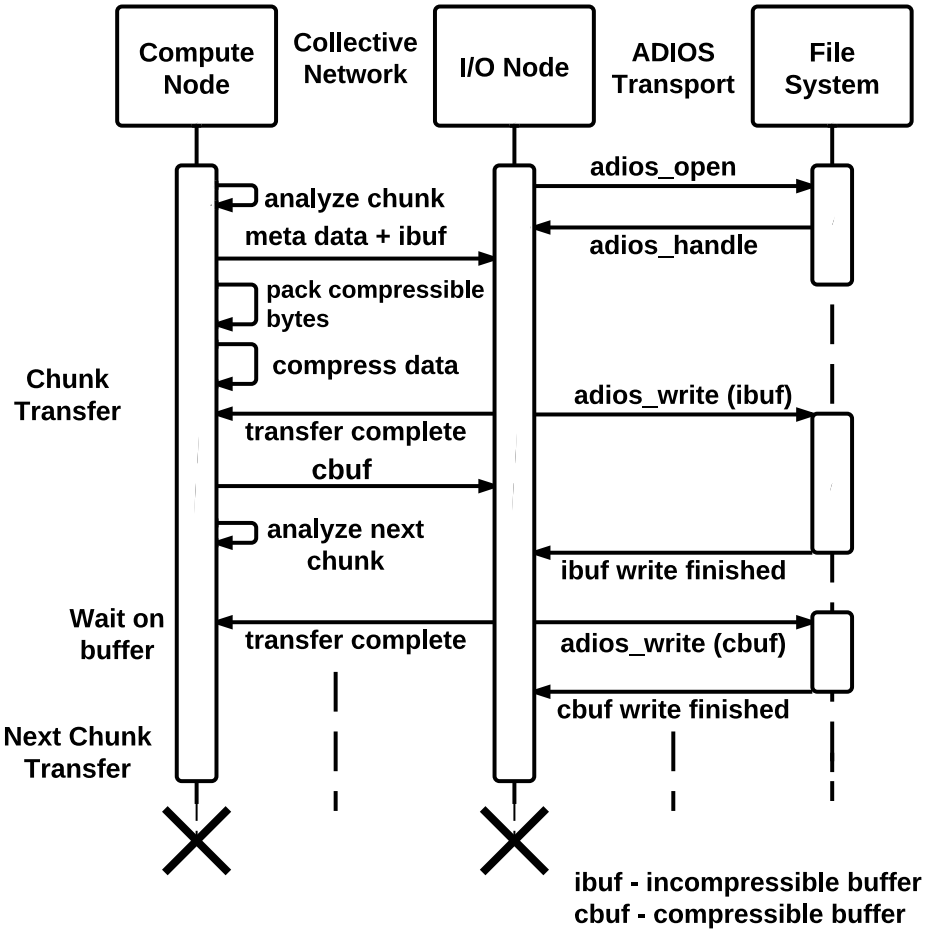


Figure 3.10: Compute-I/O interleaving strategy with compression at the compute nodes

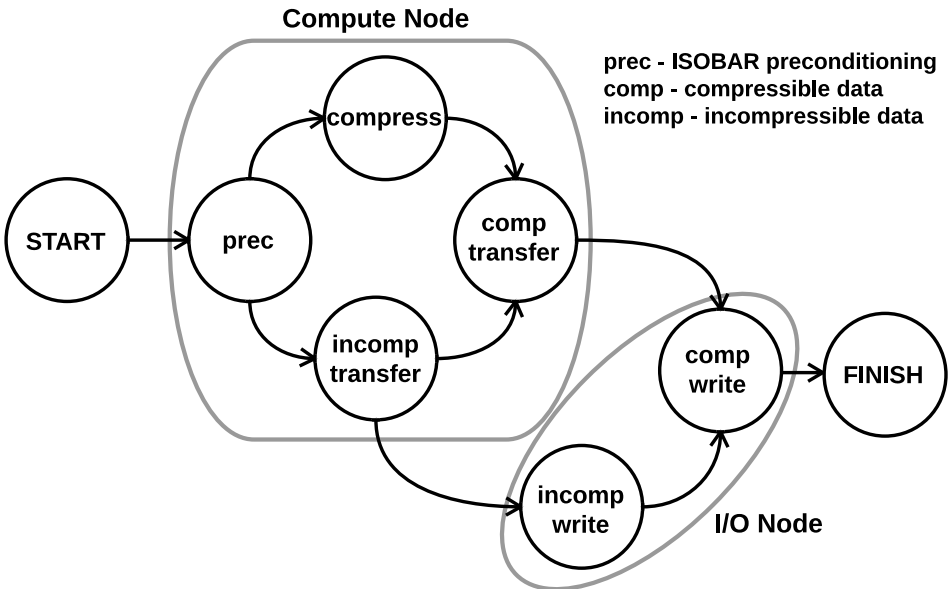


Figure 3.11: Task dependencies for the compute node compression case

chronously while the remaining byte-columns are compressed. Once the incompressible byte-columns are sent, the I/O nodes may immediately issue its asynchronous writing operation. Once the compression is complete, the compressed stream must wait for the incompressible network transfer to complete (if necessary) to transfer its results. Finally, once the incompressible bytes are written to disk, the compressed stream may be written.

The preconditioning and compression operations take place locally at every compute node, and these are given by:

$$t_{prec} = \frac{C}{T_{prec}} \tag{3.16}$$

$$t_{compress} = \frac{\alpha C}{T_{comp}} \tag{3.17}$$

The transfer times of the compressible and incompressible parts of each chunk are given by:

$$t_{incomp\_transfer} = \frac{(1-\alpha)C}{\theta}(1+\rho) \tag{3.18}$$

$$t_{comp\_transfer} = \frac{(\alpha\sigma C + \delta)}{\theta}(1+\rho) \tag{3.19}$$

Once the compressible and incompressible parts are aggregated at the I/O node, the disk

write times are given by:

$$t_{incomp\_write} = \frac{(1-\alpha)C}{\mu_w}\rho \tag{3.20}$$

$$t_{comp\_write} = \frac{(\alpha\sigma C + \delta)}{\mu_w}\rho \tag{3.21}$$

As discussed in Section 3.3 and seen in the dependency graph in Figure 3.11, interleaving is achieved through the compression and transfer/writing of network data. Additionally, there may be interleaving of the transfer of compressed byte-columns and the incompressible byte writing. Given the structure of the graph and the residence of the tasks on different nodes, we may define the critical path using the following two quantities:

$$\begin{aligned} t_{comp\_ion} &= \max(t_{compress}, t_{incomp\_transfer}) \\ &\quad + t_{comp\_transfer} \end{aligned} \tag{3.22}$$

$$\begin{aligned} t_{write\_depend} &= \max(t_{comp\_ion}, \\ &\quad t_{incomp\_transfer} + t_{incomp\_write}), \end{aligned} \tag{3.23}$$

where t_{comp_ion} represents the time taken to compress and send the compressible byte-columns to the I/O nodes, accounting for stalls caused by a longer incompressible byte-column transfer. Furthermore, t_{write_depend} represents the time taken for all dependencies to clear before writing the compressible byte stream, including possible stalls at the second network-I/O interleaving level. Adding in the preconditioner and the compressible byte-column writing, the total time-to-disk can be defined as

$$t_{total} = t_{prec} + t_{write\_depend} + t_{comp\_write}. \tag{3.24}$$

Once again, the reading of hybrid-compressed data chunks causes an inversion of the dependency graph, except that write operations are replaced with read operations, the transfers are reversed, the compressed byte-columns are decompressed, and the preconditioner task is replaced with the reconstruction task. In fact, in this particular instance, the inverted task graph is isomorphic to the original task graph, thus simplifying building the read model. The compressible byte-columns are read and then asynchronously sent to the compute nodes while the incompressible byte-columns are read. Afterwards, the incompressible byte-columns are transferred asynchronously while the decompression process begins. Finally, the compressible

and incompressible columns are recombined. This is captured in the following model:

$$t_{comp\_read} = \frac{(\alpha\sigma C + \delta)}{\mu_r}\rho \tag{3.25}$$

$$t_{comp\_transfer} = \frac{\alpha\sigma C + \delta}{\theta} \tag{3.26}$$

$$t_{decompress} = \frac{\alpha\sigma C}{T_{decomp}} \tag{3.27}$$

$$t_{incomp\_read} = \frac{(1-\alpha)C}{\mu_r}\rho \tag{3.28}$$

$$t_{incomp\_transfer} = \frac{(1-\alpha)C}{\theta} \tag{3.29}$$

The reconstruction time is assumed to be constant for fixed sized data chunks. Similar to the write case, the critical paths are defined as follows:

$$\begin{aligned} t_{incomp\_cn} &= \max(t_{incomp\_read}, t_{comp\_transfer}) \\ &\quad + t_{incomp\_transfer} \end{aligned} \tag{3.30}$$

$$\begin{aligned} t_{combine\_depend} &= \max(t_{incomp\_cn}, \\ &\quad t_{comp\_transfer} + t_{decompress}), \end{aligned} \tag{3.31}$$

where t_{incomp_cn} represents the time taken for the incompressible byte stream to reach the compute nodes, taking into account stalls as a result of the compressible byte stream being sent first, and $t_{combine\_depend}$ represents the time until all data is ready to be recombined into the original chunk of data. Thus, the total time to restore the data to its original state is

$$t_{total} = t_{comp\_read} + t_{combine\_depend} + t_{combine}. \tag{3.32}$$

The equations 3.24 and 3.32 allow us to model the end-to-end times for the write and read scenarios respectively for the compute node compression case.

3.5 Experiments and Results

In this section, we present the empirical evaluations of our framework via a set of micro-benchmarks to evaluate the throughput performance for the writes as well as the reads. We report the percentage improvement in performance obtained using the hybrid compression-I/O framework (at the compute as well as the I/O nodes) over the base case without compression. We also report theoretical evaluations for the interleaving strategies discussed in Section 3.4 via performance model simulations. Lastly, we specify the parameters used for the simulations and

present a comparison between the predicted and actual system performance.

3.5.1 Experimental Setup

Our experiments were conducted on the Cray XK6 Titan cluster at the Oak Ridge Leadership Computing Facility (OCLF). It consists of 18,688 compute systems, each containing a 16-core 2.2 GHz AMD Opteron 6724 processor and 32 GB of RAM. It uses the Lustre [72] file system for parallel I/O and a high performance Gemini interconnect for communication. The compute-I/O node ratio for all experiments is kept fixed at 8 : 1. The definitive choice of a single optimal ratio is non-trivial, since it depends on the size of the data being moved, the degree of inter-node communication, as well as the memory requirement of the staging nodes. The study in this realm is the subject of future work.

We evaluated the system characteristics using a set of micro-benchmarks to measure the network and disk I/O throughputs. The aggregate network throughput for our experiment cases was measured to be 530 MB/s on average, while the I/O read and write throughputs were measured to be 62.6 MB/s and 15.6 MB/s per node, respectively. It should be noted that for all our experiments, we refer to the term "node" as a processing core on the Titan system.

We use the `s3d_temp`, `flash_velx`, `msg_sweep3d`, and `gts_chkp_zion` datasets discussed in [16, 28, 70, 78, 11] for our analyses. The datasets are chosen to reflect the entire compressibility spectrum across a range of scientific datasets.

The S3D dataset consists of about 20.2 million double precision values of the temperature variable with 46% unique values. It is relatively less hard-to-compress in comparision with the other datasets.

The FLASH dataset, on the other hand, consists of about 68.1 million double precision values of the velocity variable with entirely unique values. It is hard-to-compress and exhibits compressibility characteristics similar to most scientific datasets discussed in [70]. Therefore, it is a good representative for a large number of scientific datasets under consideration.

The MSG dataset consists of about 15.7 million double precision values of numeric messages sent by a node in a parallel system running the NAS Parallel Benchmark (NPB) and ASCI Purple applications with 90% unique values. It exhibits compressibility characteristics similar to the FLASH dataset.

The GTS dataset consists of about 2.4 million double precision values of the zion variable's checkpoint and restart data for each 10th timestep of the GTS simulation. It consists of entirely unique values. It has a high degree of apparent randomness and is one of the most "hard-to-compress" scientific dataset. Note that this is only *apparent* randomness; in reality, the dataset contains patterns that are non-trivial to isolate, preventing general-purpose compressors from leveraging them.

The ISOBAR framework supports the use of any general purpose byte-level compression algorithm. However, for our evaluations, we use `zlib` [29], designed by Jean-loup Gailly and Mark Adler. It is a lossless compression /decompression algorithm that uses an LZ77 algorithm variant to compress the data in a block sequence. In addition to scoring high marks in general-purpose compression rate and compression throughput, the memory usage of `zlib` is independent of any input data.

In addition to the three interleaving strategies discussed in Section 3.4, we also include a serial compression case for completeness sake, wherein we apply ISOBAR compression serially, i.e., we do not interleave compressible and incompressible data processing. This is essentially an extension of the base case, allowing us to directly evaluate the impact of interleaving.

3.5.2 Write Performance

Figure 3.12 shows the results gathered from write micro-benchmarks. For each dataset, the results are reported in terms of the percent improvement in the write throughput relative to the base case, measured for each of the four scenarios (i.e. the base case, interleaving at compute nodes, and interleaving at I/O nodes, and the serial compression), versus the number of compute nodes.

We observe that both interleaved approaches (compute node and I/O node compression) yield an improvement in performance over state-of-the-art I/O middleware framework without compression (the base case). As expected, interleaving using compute node compression results in the highest performance gain, from around 12% over the base case for the GTS dataset and to as high as 46% over the base case for the S3D dataset. Improvements for the FLASH and MSG datasets are about 31% over the base case. On the other hand, interleaving using I/O node compression yields improvements in the range of about 8% for the GTS dataset, 37% for the S3D dataset, and 25% for the FLASH and MSG dataset. Using ISOBAR serially yields a modest 1% (GTS) to 16% (S3D) gain in throughput performance. Note that for the MSG dataset, using ISOBAR serially results in a 4% reduction in performance over the base case. These results clearly suggest that a significant portion of the performance boost comes from our compression-I/O interleaving strategy, affirming the efficacy of this approach.

The experiments are conducted with weak scaling up to 2048 nodes on the Titan system. The stability of the results over a varying number of cores suggests that the framework is, indeed, scalable.

3.5.3 Read Performance

We carried out equivalent read micro-benchmark tests on the disk data, evaluating the base case (without decompression) and each of the two interleaved decompression strategies (decompression

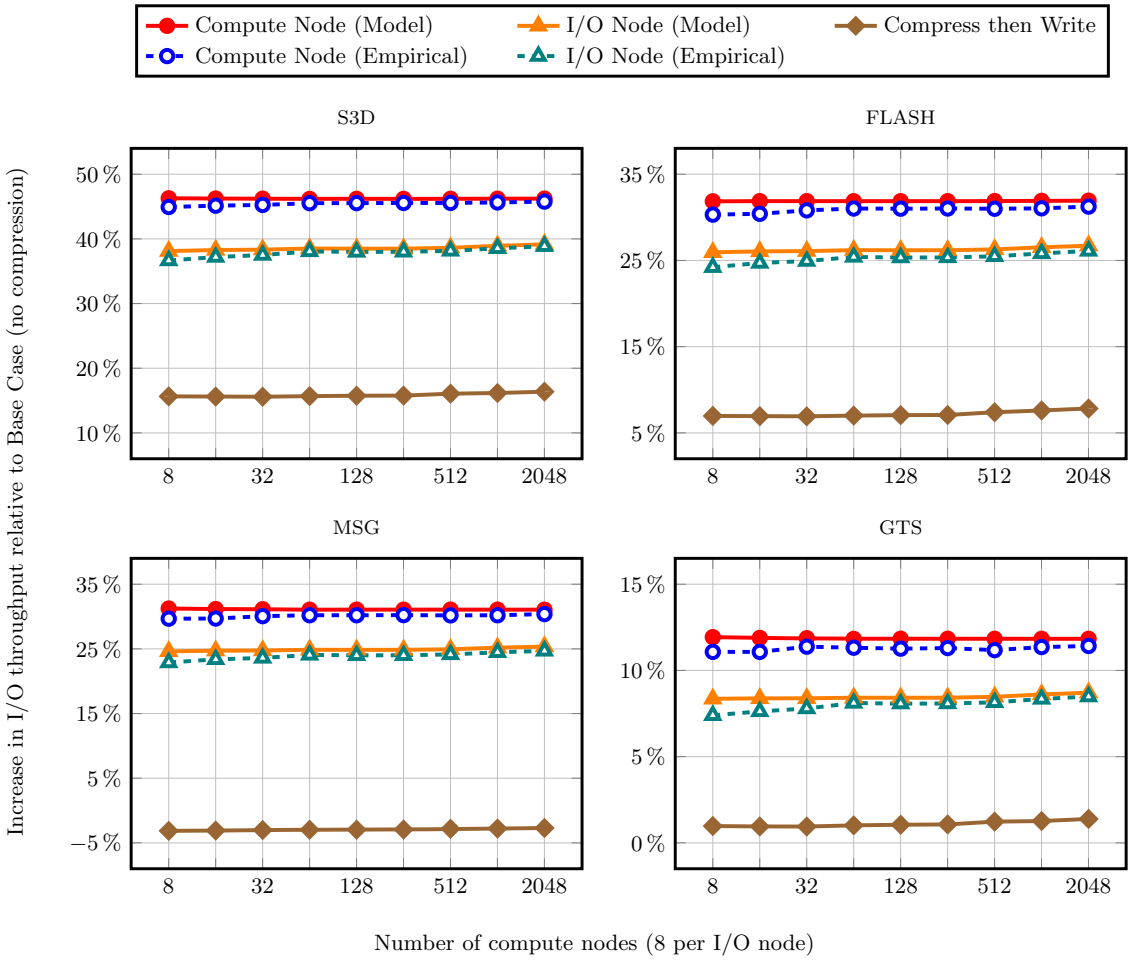


Figure 3.12: Model and empirical end-to-end write throughput versus number of compute nodes (weak scaling)

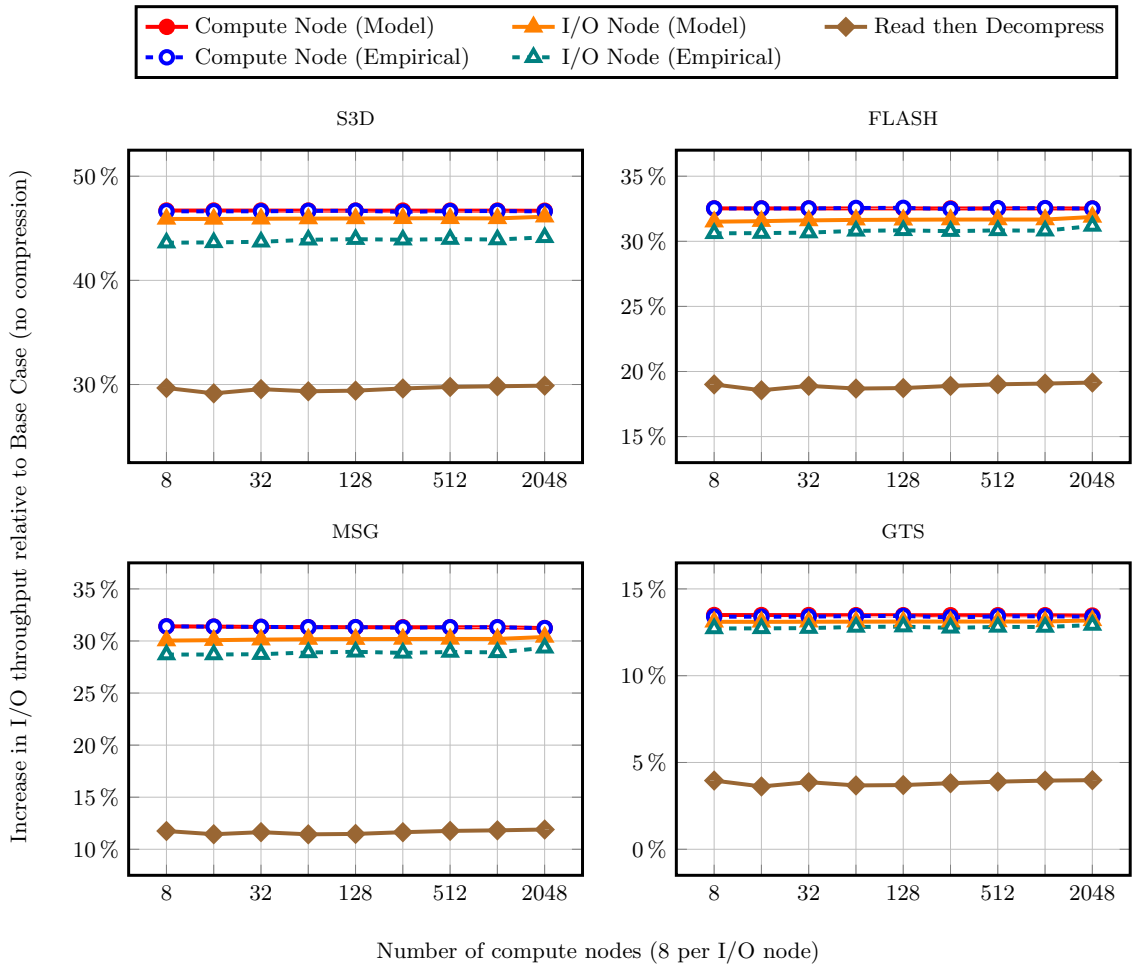


Figure 3.13: Model and empirical end-to-end read throughput versus number of compute nodes (weak scaling)

at the compute nodes and at the I/O nodes).

The experimentation results of the read micro-benchmarks are shown in Figure 3.13 in the form of percent improvements over the base case (i.e., direct reads without decompression). Both the interleaved approaches for the reads exhibit performance gains of the same order as reported for the writes for all the datasets. This suggests that the hybrid framework is symmetric with respect to reads as well as writes.

In order to support asynchronous processing of the compressible and incompressible portions of the data for the interleaved scenarios, we used a separate file per ADIOS group. The reason for this is that ADIOS currently reads data from all the groups upfront when using a single file and this operation is inherently blocking, i.e., a request for the read of only the compressed buffer

Table 3.4: Dataset Evaluations for the Best Strategy of Interleaving at Compute Nodes

Application	α	σ	$\mathbf{T_{comp}}$	$\mathbf{T_{decomp}}$	**Avg. write gain (%)***	**Avg. read gain (%)***
GTS †	0.25	0.527	90	414	11.26	13.40
FLASH †	0.25	0.019	149	127	30.87	32.53
S3D †	0.375	0.152	186	457	45.43	46.64
MSG †	0.375	0.361	140	560	31.61	31.84
OBS	0.25	0.203	43	172	24.55	24.99
NUM	0.25	0.231	163	652	23.51	23.93

†Validated by experimental results. *Adjusted for model bias.

requires the entire data to be read. The two-file-approach does not affect the write performance.

3.5.4 Performance Modeling

The performance models for evaluation were setup to use the following parameter values. The compute-I/O node ratio was chosen as $\rho = 8 : 1$. Compression efficiency is sensitive to the chunk size for most lossless compression techniques that adapt based on calculated statistics of the subject data [81, 38]. We chose a chunk size $C = 3$ MB taking into account the sensitivity of most lossless compression techniques to the input chunk size [81]. The ISOBAR preconditioner operates at an approximate throughput of $T_{prec} = 500$ MB/s. Other ISOBAR specific parameters were chosen based on the statistical analyses of 24 different scientific datasets [70], 19 of which were "hard-to-compress." The average values of these parameters based on the application type (i.e., α, σ, T_{comp}, T_{decomp} from Table 3.2) are shown in Table 3.4.

The predicted performance for the data writes and reads for all the test scenarios and evaluation datasets are shown in Figure 3.12 and 3.13, respectively. It is evident that the theoretical performance improvements are generally consistent with the empirical results. Though some small overestimating bias is visible, it is itself relatively consistent, and can therefore be readily factored out (as has been done in Table 3.4). Additionally, the fact that the trends exhibited by the model predictions are equivalent to those of the measured results points to a mismeasured system parameter as the likely culprit for the minor discrepency that exists. Thus, the performance model can be used to closely approximate the true behavior of the system.

In addition, we also performed theoretical evaluations of our framework on other scientific datasets from various application domains. These include the OBS [12], and NUM [13, 67] simulations, both of which normally produce a reasonable amount of data per process. The results for the best strategy (interleaving at the compute nodes) for reads as well as writes are

shown in the Table 3.4.

We observe that the expected theoretical performance gains for these more typical datasets are as high as 25%, even after accounting for the observed model bias. This shows that our framework improves significantly on less harder to compress datasets and that the performance gains are directly proportional to the compression ratio of datasets.

3.6 Related Work

The ISOBAR hybrid compression framework utilizes the I/O forwarding paradigm [3], which is a common technique for alleviating the I/O bottleneck in super-computing environments, and is the subject of active research. The multithreaded ZOID architecture [40], developed under the ZeptoOS [5] project on the IBM BlueGene/P, is a state-of-the-art data staging system. LambdaRAM [77] is an asynchronous data staging system which mitigates WAN latency via dedicated staging nodes. IODC [64] is a portable MPI-IO layer implementing a caching framework wherein certain tasks, such as file caching, consistency control, and collective I/O optimization, are delegated to a small set of I/O delegate nodes. Recent work in ADIOS includes the DataStager component [2], which focuses on I/O performance through data staging via network rate limiting and I/O phase prediction, and JITStaging [1], which provides a framework for placing data filter, analysis and organization code in the data pipeline to reduce overall time-to-data.

SCR [9] and PLFS [6] are well-known middleware approaches designed specifically for single (N-N) and shared (N-1) checkpointing, respectively. While SCR provides efficient checkpoints and improves system reliability by shifting checkpoint I/O workload to hardware better suited for the job, it is not suitable for applications that need process-global access to checkpoint files. Moreover, hardware and file system support is required to cache checkpoint files. PLFS, on the other hand, transparently rearranges shared checkpoint patterns into single patterns, thereby decreasing the checkpoint time by taking advantage of the increased bandwidth. However, this requires managing the overwhelming pressure resulting from the simultaneous creation of thousands of files within a single directory. Also, since PLFS is specifically a checkpoint file system and not a general purpose file system, certain usage patterns may suffer a significant performance hit [6].

Our approach differs in that we focus on optimizing data staging I/O throughput via compression and resource interleaving. Some recent work has examined compression in a data staging context [82]; however, only traditional compression algorithms are explored, which do not function well on the hard-to-compress scientific data we consider, and resource interleaving is not used to hide the compression and I/O synchronization costs.

Previous work on data deduplication can also be considered a form of compression, detecting

and eliminating duplication in data with a goal to improving disk utilization. State-of-the-art deduplication systems include HYDRAstor [21], MAD2 [80], and others [37]. Deduplication can operate at either sub-file or whole-file scale; the relative merits of these aproaches have been explored [60]. Although these systems are scalable, provide a good deduplication efficiency, and attain near-optimal throughput for common filesystem data, they are unfortunately less effective when dealing with peta-scale scientific data. Unlike typical file system data, scientific data exhibits very few duplicate non-contiguous patterns, nullifying much of the effectiveness of the deduplication approach. Furthermore, the possibility of running compression *in situ* remains desirable for performance reasons, which is not possible with filesystem-bound algorithms such as deduplication.

3.7 Conclusion

The I/O staging paradigm has arisen to cope with the growing gap between computing power and I/O bandwidth in current peta-scale HPC environments. While providing increased and more consistent performance, the sheer scale of the data necessitates lossless compression as a data reduction methodology, leaving the technical challenge of absorbing the costs of compression and overhead in parallel I/O performance on nonuniform chunk sizes.

To meet these challenges, we presented the ISOBAR hybrid compression-I/O framework. The ISOBAR preconditioner allows us to separate the high-entropy components of the data from the low-entropy components, forming independent streams that may be interleaved. The high-entropy components are sent across the network and to disk asynchronously while the low-entropy data is compressed, hiding the compression costs and fully utilizing all compute, network, and I/O resources. Placement of the compression routine itself is an important issue, so we implement a hybrid approach where the compression phase may be placed either on the compute nodes or the I/O nodes, trading off between aggregate compression throughput and leaving the compute nodes free to run the application at hand. Finally, each of the implementations are accurately modeled by a set of performance metrics, allowing a generalization of our methodology's performance past the experimental environment.

We demonstrated the efficiency of compression-I/O interleaving, improving end-to-end I/O throughput by predicted values ranging from 12 to 46% on datasets that are considered particularly hard-to-compress. We therefore believe that data compression can be used effectively within an HPC environment to help bridge the computational and I/O performance gap.

Chapter 4

Generic High-performance Method for Deinterleaving Scientific Data

4.1 Introduction

Emerging extreme-scale high performance computing (HPC) systems enable high fidelity scientific simulations that generate data at an increasing rate [58]. Yet, these HPC systems and data-intensive applications they support consume energy at an ever-increasing amount [50, 74]. Thus, the need for performance and energy efficient data management applications is of utmost importance to maximize throughput/watt while achieving improved scalability and sustainability [32].

To improve performance during scientific data analysis, which is critical for gaining insights from the simulations, simulations often have to *deinterleave* data variables. Upon deinterleaving, the data set for each variable of the simulation is contiguous in memory and storage. This deinterleaved layout is beneficial since most data analyses span multiple time steps of a particular variable [49]. In contrast, most simulations perform calculations using instances of many variables from a current/previous time step. Hence, an *interleaved* layout in memory provides better data locality during simulation runs by keeping each group of variables together in memory for the active time steps, see Figure 4.1.

Deinterleaving data is frequently necessary after the completion of a simulation step before data analysis and storage. For example, simulations such as FLASH [28], S3D [16] and Nek5000 [24] have variables that are interleaved in memory while most storage and analysis, such as data compression [70, 71] and variable precision analytics [42], are performed using a deinterleaved layout. Through performing numerous micro-benchmarks, we found that common deinterleaving methods have poor throughput and energy performance.

To address this problem, we propose a deinterleaving method that is high performance,

ρ_0	P_0	T_0	ρ_1	P_1	T_1	...	ρ_m	P_m	T_m

(a) variables interleaved in memory

ρ_0	ρ_1	...	ρ_m	P_0	P_1	...	P_m	T_0	T_1	...	T_m

(b) variables deinterleaved in memory

ρ_0	P_0	T_0
ρ_1	P_1	T_1
$\vdots$	$\vdots$	$\vdots$
ρ_m	P_m	T_m

(c) interleaved matrix format

Figure 4.1: FLASH data in interleaved and deinterleaved layouts; each ρ_f, P_f, and T_f for $f = 0$ to m refers to the value of ρ, P, and T of the simulation at the f^{th} matrix row

energy efficient, and generic to any variable data type. To the best of our knowledge, this is the first deinterleaving method that 1) exploits data cache prefetching, 2) reduces memory accesses, and 3) optimizes the use of complete cache line writes. As a result, our method increases the throughput performance, reduces memory latency, and improves energy utilization.

Specifically, we compare the throughput performance and energy utilization of our deinterleaving method to two common deinterleaving methods. We assessed our method with 105 STREAM standard micro-benchmarks including 84 throughput and 21 energy performance test cases of varying input sizes and data types. In all cases tested, our method achieved better throughput and energy performance than the other two methods. In the best case, our method improved throughput up to 26.2x and throughput/watt up to 7.8x, when compared to the next best deinterleaving method.

4.2 Background

Simulations such as FLASH, S3D, and Nek5000 have variables that are interleaved in memory. These interleaved variables can be thought of as a matrix of data stored in row major format where each column corresponds to a particular variable. For multidimensional variables, each dimension has a separate column. Consider an example of FLASH simulation data with a sample of three variables ρ, P, and T corresponding to gas density, pressure, and temperature, respectively. The interleaved layout of these variables in memory can be seen in Figure 4.1a. Representing this data in matrix form would give an $m \times 3$ matrix where the three columns correspond to the three variables and the rows correspond to different steps of the simulation, see Figure 4.1c. With this interpretation, deinterleaving the data is equivalent to performing a matrix transposition, which would change the layout of the variables in memory, see Figure 4.1b.

There are two common techniques for deinterleaving data by performing an out-of-place matrix transposition. We refer to these techniques as *standard transposition* and *strided transposition*. These two techniques, along with our proposed method in the following section, are considered *out-of-place* due to the use of an output memory space equal to the size of the

original matrix where the elements are copied. In contrast, *in-place* transposition methods use a bounded amount of memory space and, in some cases, can slightly outperform out-of-place methods. However, in-place methods are often complex and can be performance constraining for simulations requiring variable interleaving, such as FLASH, S3D and Nek5000, to continue from where it left off in the calculation phase.

The standard and strided out-of-place transposition methods differ from each other in how they copy elements into an output memory buffer. The standard transposition method uses two loops to iterate row-wise and writes out the elements in a strided manner [27]. Alternatively, the strided transposition method uses two loops to iterate column-wise and writes out the elements contiguously.

4.3 Method

Our deinterleaving method performs an out-of-place transposition to transform a matrix of data stored in row major format to one stored in column major format. During the transposition process, our method combines the strength of both the standard transposition and strided transposition techniques.

In this section, we describe our deinterleaving method in detail. The method section is divided into three subsections corresponding to the three major components of our method: 1) cache prefetching on blocks of data, 2) using the registers as a vector transposition buffer, and 3) optimizing for full cache line writes. In addition, we provide a simple example for clarity.

4.3.1 Cache Prefetching on Blocks of Data

The benefit of cache prefetching is to hide latency time sinks associated with accessing main memory [30]. The standard transposition method, as discussed in Section 4.2, is able to take advantage of these benefits due to the sequential data reads inherent in its method. In contrast, the major weakness of the strided transposition method is that cache prefetching is not guaranteed and its effectiveness is dependent on the input buffer size. The cache prefetching benefits of the standard transposition method were the motivation for performing cache prefetching in our method.

Given an $m \times n$ (m rows and n columns) matrix of elements, A, stored in row major format, the first step of our deinterleaving method is to partition A into a block matrix where the blocks correspond to submatrices of A that will be consecutively prefetched into cache. As illustrated in Figure 4.2, matrix A is partitioned as an $M \times 1$ block matrix where each block is of size $m_b \times n$. Partitioning A in this manner creates M blocks each of which we label as B_k for $k = 1$ to M. The number of rows in each block, denoted m_b, is chosen so a block column can fill the

$$A = \begin{pmatrix} \begin{pmatrix} e_{1,1} & e_{1,2} & \cdots & e_{1,n} \\ e_{2,1} & e_{2,2} & \cdots & e_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ e_{m_b,1} & e_{m_b,2} & \cdots & e_{m_b,n} \end{pmatrix} \\ \begin{pmatrix} e_{(m_b+1),1} & e_{(m_b+1),2} & \cdots & e_{(m_b+1),n} \\ e_{(m_b+2),1} & e_{(m_b+2),2} & \cdots & e_{(m_b+2),n} \\ \vdots & \vdots & \ddots & \vdots \\ e_{2m_b,1} & e_{2m_b,2} & \cdots & e_{2m_b,n} \end{pmatrix} \\ \vdots \\ \begin{pmatrix} e_{(M-1)m_b+1,1} & e_{(M-1)m_b+1,2} & \cdots & e_{(M-1)m_b+1,n} \\ e_{(M-1)m_b+2,1} & e_{(M-1)m_b+2,2} & \cdots & e_{(M-1)m_b+2,n} \\ \vdots & \vdots & \ddots & \vdots \\ e_{Mm_b,1} & e_{Mm_b,2} & \cdots & e_{Mm_b,n} \end{pmatrix} \end{pmatrix} = \begin{pmatrix} B_1 \\ B_2 \\ \vdots \\ B_M \end{pmatrix}$$

Figure 4.2: Matrix A being partitioned into M blocks of size $m_b \times n$

entire cache line, discussed in Section 4.3.3.

For example, suppose the cache line is size of C bytes, which on most modern architectures is 64 or 128 bytes [17]. Suppose the elements of matrix A are each β bytes. Then, for m_b elements to fill the cache line as full as possible we want $m_b\beta = C$, and therefore make $m_b = \lfloor C/\beta \rfloor$. It is plausible that the last block will have fewer elements than the other blocks because m_b may not evenly divide the m elements. In this case, $M = \lceil m/m_b \rceil$. To process the smaller block, the matrix can be padded with values that will be disregarded [19].

The blocks, B_k for $k = 1$ to M, correspond to the submatrices of A that will be consecutively prefetched into the cache. Block of data B_{k+1} will be prefetched into cache while the block B_k is being further processed, as described in the following subsections. By prefetching blocks of elements in this manner, our method can reduce memory latency associated with loading blocks from memory.

4.3.2 Using the Registers as a Vector Transposition Buffer

Each block B_k can further be partitioned into submatrices using the columns as dividers, making B_k into a $1 \times n$ block matrix, referred to as a *column vector*, as seen in Figure 4.3a. With both partitions applied, matrix A can be viewed as a matrix of column vectors as shown in Figure 4.3b. Each column vector of B_k, which we denote as $V^c_{k,j}$ for $j = 1$ to n, consists of elements that are currently non-contiguous in memory due to the row major storage format of A.

The goal of our deinterleaving method is the elements of the column vectors to be contiguous in memory or, equivalently, the elements to belong to the same row in the matrix. To make the

$$B_k = \begin{pmatrix} \begin{bmatrix} e_{(k-1)m_b+1,1} \\ e_{(k-1)m_b+2,1} \\ \vdots \\ e_{km_b,1} \end{bmatrix} \begin{bmatrix} e_{(k-1)m_b+1,2} \\ e_{(k-1)m_b+2,2} \\ \vdots \\ e_{km_b,2} \end{bmatrix} \cdots \begin{bmatrix} e_{(k-1)m_b+1,n} \\ e_{(k-1)m_b+2,n} \\ \vdots \\ e_{km_b,n} \end{bmatrix} \end{pmatrix}$$

$$= \begin{pmatrix} V_{k,1}^C & V_{k,2}^C & \cdots & V_{k,n}^C \end{pmatrix}$$

(a) Partitioning of block B_k into column vectors $V_{k,j}^C$ for $j = 1$ to n

$$A = \begin{pmatrix} B_1 \\ B_2 \\ \vdots \\ B_M \end{pmatrix} = \begin{pmatrix} V_{1,1}^C & V_{1,2}^C & \cdots & V_{1,n}^C \\ V_{2,1}^C & V_{2,2}^C & \cdots & V_{2,n}^C \\ \vdots & \vdots & \ddots & \vdots \\ V_{M,1}^C & V_{M,2}^C & \cdots & V_{M,n}^C \end{pmatrix}$$

(b) Matrix A partitioned into submatrices of column vectors

Figure 4.3: Each block of matrix A partitioned into n column vectors

elements contiguous, each column vector gets transposed and temporarily stored in CPU registers until it is written out to a full cache line. The general notation for each transposed column vector, referred to as a *row vector*, is denoted: $V_{k,j}^R = [e_{(k-1)m_b+1,j}, e_{(k-1)m_b+2,j}, \cdots, e_{km_b,j}]$.

For clarity, consider a specific example. Suppose block B_1 is currently being partitioned into n column vectors, namely $V_{1,j}^c$ for $j = 1$ to n. The elements of a column vector $V_{1,j}^c$ consist of the elements $e_{1,j}, e_{2,j}, ..., e_{m_b,j}$ from A as seen in Figure 4.3a. Starting with the first column vector ($j = 1$), the elements must be loaded into a buffer of registers and in the next step written into the extra memory space that was created for the transposition matrix. Using CPU registers as a buffer to store these elements constitutes a transposition of the column vector as the elements will now be contiguous instead of strided.

The motivation for using the registers as a temporary buffer is that each column vector must be transposed into some storage location in order to achieve full cache line writes, which is the strength of the strided transposition method. The registers provide the most efficient location to store the row vectors due to their minimal CPU cycles per operation [31]. In addition, using a buffer of registers in this manner is a viable option since typically a CPU provides enough hardware registers where the buffer size is at least equal to the cache line size.

4.3.3 Optimizing for Full Cache Line Writes

Once the elements of a row vector are loaded into the register buffer, our method then writes out this data into the memory space that was created for the deinterleaved output. During the write process, our method utilizes the full cache line due to the row vector containing m_b elements, where m_b was chosen to fill the cache line. By utilizing full cache line writes, our method emulates the strength of the strided transposition method [20], while avoiding the inefficient write process of the standard transposition method.

During the write process, our method must leave enough room for m elements of A (an entire column) between the start of each column vector, meaning there will be a stride of m between the memory storage offset of each column vector. So, for a given row vector $V^R_{k,j}$, the elements get mapped consecutively into the new memory storage location offset starting at $(k-1)m_b + (j-1)m$.

After this process is completed and all the row vectors have been written, the process is repeated. The next block, which should already reside in cache, is partitioned into column vectors that are consecutively loaded into the register buffer and written out. The entire process is completed for each block B_k for $k = 1$ to M. Once every block has gone through this process, the output location will contain the transpose of matrix A. The entire deinterleaving process is illustrated by the example given in the following section.

4.3.4 A Simple Example of Our Deinterleaving Method

For clarity, consider a simple example of 24 data elements consisting of three different variables interleaved in memory. Figure 4.4a shows the matrix representation of these interleaved variables, with each column of the matrix storing data corresponding to a particular variable. For the sake of this example, suppose the elements are 8-byte doubles (common in simulation data) and the cache line size of the system is 32 bytes. The elements of the matrix are initially stored in row major format, meaning the elements are ordered as $e_1, e_2, e_3, e_4, ..., e_{24}$ in memory. The goal of our deinterleaving method is to obtain the transpose of the matrix, illustrated in Figure 4.4e, so that the elements of each column will be contiguous in memory and thus deinterleaved.

The initial step of our deinterleaving method is to create a new output memory space to hold the transposed matrix. Next, the matrix is partitioned into two 4×3 block matrices, B_1 and B_2 consisting of elements e_1 through e_{12} and e_{13} through e_{24}, respectively. The number of rows in each block was chosen as $m_b = 4$ so that each column within a block will entirely fill the cache line, as four 8-byte doubles is exactly the cache line size of the system.

With the matrix partitioned into two blocks, the next step is to load B_1 into the cache. The block itself is then partitioned into the three column vectors V^C_1, V^C_2, and V^C_3, as depicted in Figure 4.4b. After this partition, the first column vector of B_1, meaning the elements e_1, e_4, e_7,

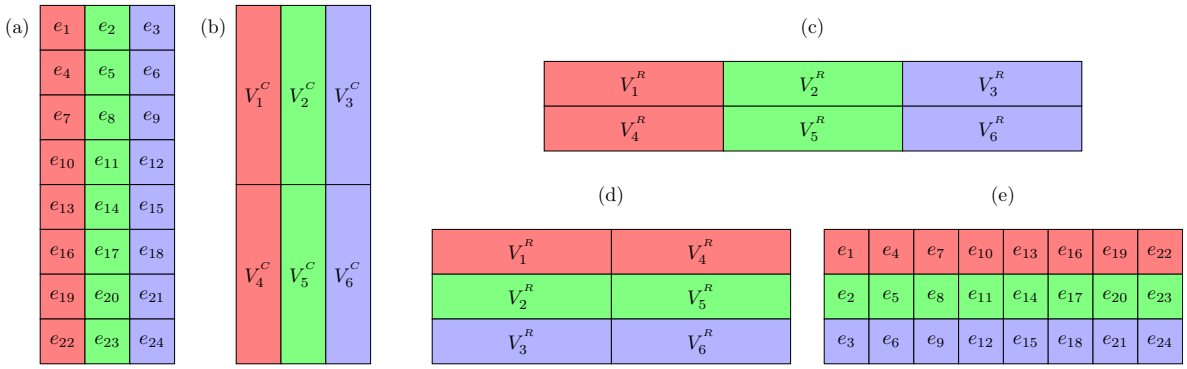


Figure 4.4: The partition and transposition steps of our deinterleaving method performed on a simple 8×3 matrix of 8-byte elements optimized for cache line writes of 32 bytes

and e_{10}, is transposed into a row vector and temporarily stored in the register buffer, see Figure 4.4c. The full cache line is then utilized to write out the elements of the row vector into the output memory space that was created for the transposed matrix, see Figure 4.4d. This process is repeated on the remaining column vectors of B_1 until all of them have been written into the output memory space.

After B_1 has finished transposing and writing each of its column vectors, the same process is repeated on the second block, B_2. This block would have been prefetched into cache during the time B_1 was being processed, thus saving the time of retrieving B_2 from memory. After B_2 is processed, the matrix will be transposed and the variables deinterleaved, as illustrated in Figure 4.4e.

4.4 Performance Evaluation

In this section, we present the empirical evaluations of our deinterleaving method via a set of micro-benchmarks to evaluate throughput and energy performance. We compare the results of our deinterleaving method against those of the standard and strided transposition methods. For brevity, we will refer to our Out-of-Place Deinterleaving method as *OPD method* in the remainder of the paper.

4.4.1 Experimental Setup

Performance measurements were collected on the Lens Linux cluster at Oak Ridge National Laboratory and on a dedicated Intel server. The Lens cluster is primarily used for data analysis and high-end visualization. Each cluster node consists of four quad-core 2.3 GHz AMD Opteron processors and 128GB of memory. Each processor has three cache levels: L1 cache is 64KB, L2

cache is 512KB, and the shared last level cache (LLC) is 5118KB. The Intel server consists of a quad-core i7 2.93 GHz processor and 16GB of memory running CentOS-6.3. The Intel processor has three cache levels: L1 is 32KB, L2 is 256KB, and LLC is 8MB. All multi-core evaluations for both the throughput and energy experiments were done utilizing all available processors and computational cores.

For collecting performance metrics, we added micro-benchmarks of all deinterleaving methods into the STREAM [59] framework, compiled with GNU Compiler Collection (GCC) version 4.7.1. STREAM is useful for evaluating memory throughput performance of single- and multi-core I/O-intensive functions that are sensitive to system architecture characteristics [36]. We compared the throughput performance metrics collected from 105 STREAM micro-benchmarks tested across the AMD and Intel systems. The test cases spanned a diverse set of data including multiple data types, column sizes, and input buffer sizes. Specifically, the data types evaluated were bytes, single-precision floating-points, and double-precision floating-points. For each data type, the variables interleaved (columns) were $2, 4, 8$, and 16. The input buffer sizes ranged from $64, 128, \cdots, 4096$ kilobytes per core. To obtain the performance measurements seen in Figure 4.5 and Figure 4.6, each micro-benchmark was run 100 times for each deinterleaving method. The highest throughput of the 100 runs was recorded.

For our set of micro-benchmarks, we restricted our input buffer size between 64KB and 4096KB. The reason this lower bound was chosen is due to the precision of the timer used in the STREAM benchmark, which states at what point the clock measurement becomes unreliable. For input sizes less than 64KB, our deinterleaving technique ran too fast for a reliable throughput measurement. However, at sizes of 64KB and higher, the throughput could be measured accurately. The upper bound of 4096KB was chosen to represent an input size that was beyond the size of the LLC for multi-core evaluations.

4.4.2 Deinterleaving Throughput Performance

In all multi-core evaluations, our deinterleaving method performed better than the standard and strided transposition methods, see Figure 4.5 and Figure 4.6. In the best case, our deinterleaving method performed at a 26.2x faster throughput, when compared to the next best method. In addition, our method consistently reported gains of over 40GB/s on smaller input sizes (corresponding to lower cache levels). The performance gains of our deinterleaving method were more pronounced on smaller input buffer sizes because memory latency starts to become a significant factor on larger buffer sizes.

Another characteristic seen in our results is that neither the standard transposition nor the strided transposition was consistently better than the other. In some cases, the standard transposition would significantly outperform the strided transposition and vice versa, irrespective

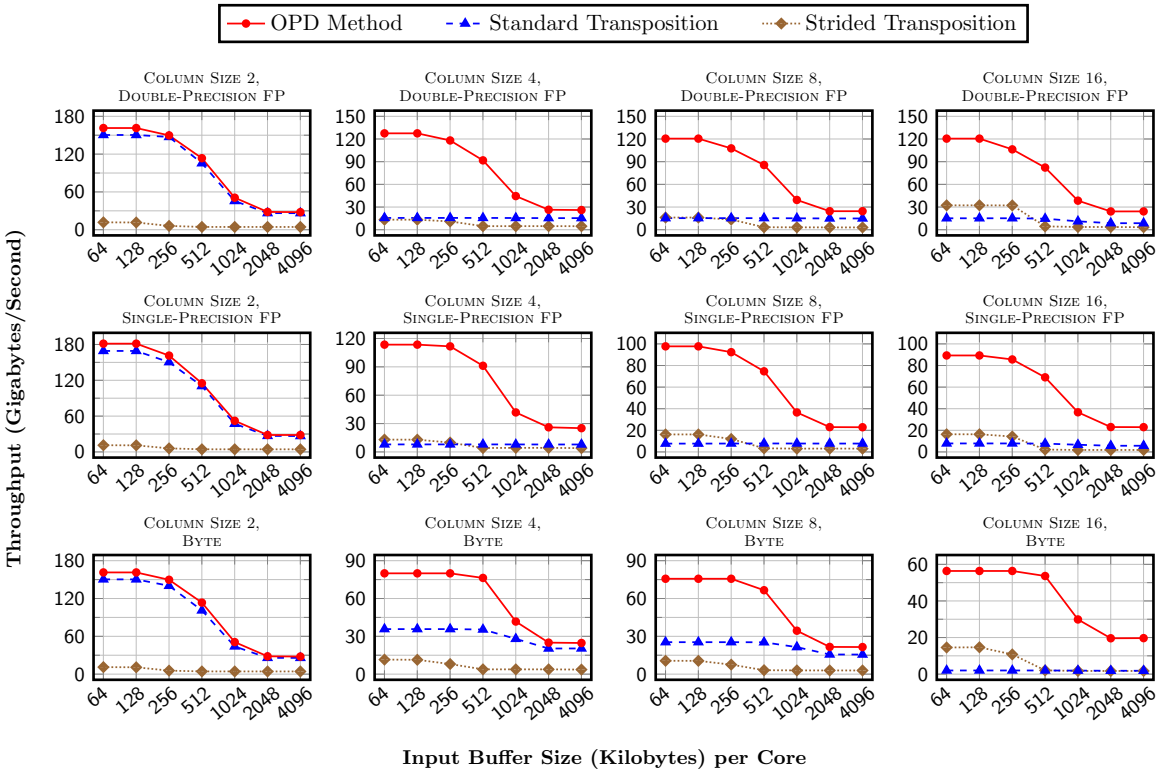


Figure 4.5: Throughput performance applying STREAM micro-benchmarks when deinterleaving single-precision, double-precision floating-point (FP), and byte variables on the AMD Opteron system utilizing all 16 cores

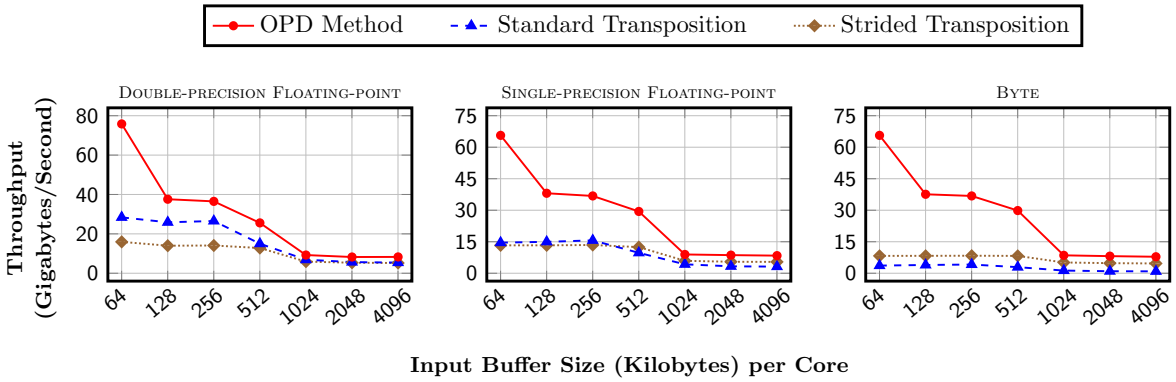


Figure 4.6: Throughput performance applying STREAM micro-benchmarks when deinterleaving single-precision, double-precision floating-point, and byte variables with 16-variable interleaved data on the Intel i7 system utilizing all cores

Table 4.1: Instruction Set Architecture for Deinterleaving Methods

Data Type	**Method**	**Column Size**			
		2	4	8	16
Double	Standard	SSE2	x86_64	x86_64	x86_64
	Strided	x86_64	x86_64	x86_64	x86_64
Float	Standard	SSE	SSE	SSE	SSE
	Strided	SSE	x86_64	x86_64	x86_64
Byte	Standard	SSE2	SSE2	SSE2	SSE2
	Strided	x86_64	x86_64	x86_64	x86_64

of the instruction set architecture being used, see Table 4.1. The performance inconsistency of these two techniques is another strength of our deinterleaving method, as ours consistently outperformed the other two methods.

Although not depicted in throughput performance figures, our method was also compared against the other methods when all were utilizing only a single core of the system. In this case, our method reported similar, but scaled down trends to those seen in multi-core evaluations. Even in this case, our method always had better throughput performance than the other two methods.

4.4.3 Deinterleaving Energy Performance

The energy performance measurements were performed on a dedicated Intel server connected to a Watts Up Pro meter, which provides a recording of power measurements (watts) per second during the collection of throughput metrics. The power was measured for each deinterleaving method on 21 micro-benchmarks of 16-variable interleaved data of varying input sizes and data types. Energy performance normalization was done for the deinterleaving methods by calculating gigabytes per joule (throughput/watt) for each test case.

In all cases tested, our deinterleaving had better energy utilization than the other methods, with throughput/watt improvements up to 7.8x, when compared to the next best method. The results of our energy experiments can be seen in Figure 4.7. The improved energy performance of our method is attributed to the increased throughput (Figure 4.6), the effective cache utilization similar to the standard transposition method, and the optimized cache line writes like the strided transposition method.

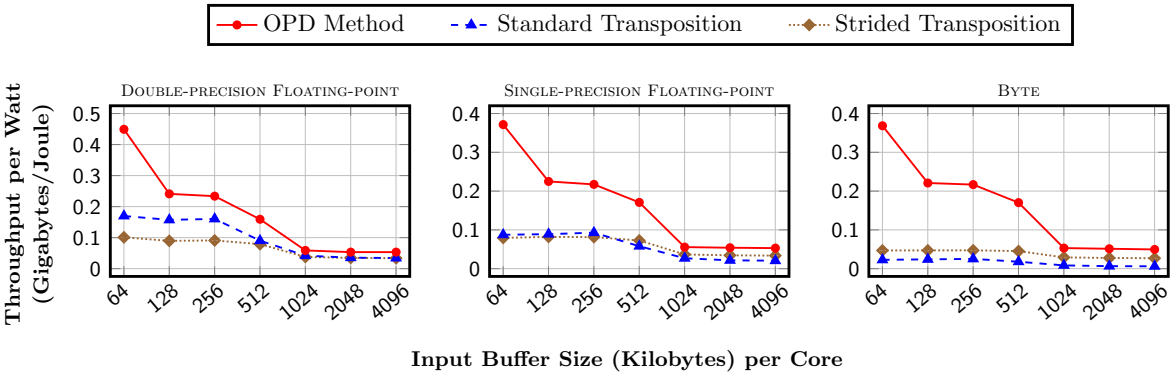


Figure 4.7: Normalized energy performance measurements (throughput/watt) collected with power meter during STREAM throughput benchmarks on Intel system

4.5 Related Work

Out-of-place matrix transpositions have been studied extensively in the past. Majority of these transposition algorithms, initially proposed decades ago, focus on methodologies for optimizing use of secondary storage (tapes, disks, etc.). Although these algorithms are not well suited for modern computer systems due to processor cache inefficiency, we still use these techniques for references since secondary storage of the past is analogous to RAM in modern systems. A fast matrix transposing method was given in [22] where the algorithm was specifically designed for $2^n \times 2^n$ square matrices and it is compared with many other matrix transposition algorithms. Another algorithm called single radix algorithm was proposed in [44], and shows better performance in disk seeks and accesses. For transposing a large arbitrary matrix, PRIM was introduced in [34].

In-place matrix transpositions can be used as an alternative to out-of-place methods; however, in-place methods are often complex and can be performance inefficient for simulations requiring interleaved variables to continue with the calculation phase. Furthermore, in-place methods commonly have constraints on row and column sizes making them unusable as a generic method for deinterleaving scientific data. Six algorithms are investigated in [15] for transposing a large square matrix in-place. They use 32-bit single-precision floating-point numbers and have the length of both the row and column equal to 2^n. In their experiments, the non-linear array layout algorithm outperforms other algorithms as it uses "Morton ordering" [61]. This algorithm also uses recursion to divide the problem into smaller subproblems, as in [27], but terminates at an architecture-specific tile size. Even by using a "blocking" and "tiling" technique, a higher cache efficiency might not be achieved as claimed in [31]; instead, they proposed a buffer must be used in order to be cache efficient.

Although much attention has been paid to matrix transposition, very few of the studies focus on the utilization of cache in a specific domain requiring deinterleaving of variables. Our method applies to any data type and utilizes full cache line writes to be throughput and energy efficient when deinterleaving data. Blocking, shuffling, and compression library, Blosc, was introduced in [4], which uses a high-performance byte deinterleaving technique to reduce activity on the memory bus. Our approach differs from this technique in that we support not just byte-level but float- and double-level as well. Moreover, Blosc currently utilizes 16-byte SSE2 register writes instead of full cache line writes compared to our deinterleaving method.

4.6 Conclusion

We proposed a deinterleaving method that is high performance, energy efficient, and generic to any data type. Our method has increased throughput and energy performance by utilizing the system architecture in three ways: 1) improving data cache prefetching, 2) reducing memory accesses, and 3) optimizing the use of full cache line writes.

Our method results in better throughput and energy performance when compared against two common deinterleaving methods during 105 STREAM standard micro-benchmarks evaluations, which includes 84 throughput and 21 energy performance test cases. When compared to the next best case, our method improved throughput up to 26.2x and throughput/watt up to 7.8x.

Chapter 5

Conclusion

Our thesis introduced realizable preconditioner-base methods for optimizing end-to-end throughput by applying lossless compression and layout reorganization towards modern-day HPC and next generation systems for storage, retrieval, and analysis. Our future work focuses on advancing this research of utilizing preconditioners to further minimize the disparity between the ever-increasing amounts of data being generated by these systems and their predicted data handling limitations. In particular, application of preconditioner-base compression and deinterleaving can provide the foundation for supporting in-place techniques to reduce system memory utilization and optimize cache awareness of the architecture.

5.1 Future Work

As HPC systems scale towards exascale computing, data storage, retrieval, and analytic frameworks suffer more from limited in-core memory performance and capacity available. In particular, HPC systems will offer less Random Access Memory (RAM) available per computational core, RAM I/O bandwidth does not scale proportionally to computational performance, and energy cost is increasing due to poor utilization of data caching mechanisms. Furthermore, the level of network and disk I/O performance offered by these systems has not kept up with computational needs, leading to a serious bottleneck when reading and writing data out-of-core. These issues become apparent during data movement operations that are commonly associated with in-transit and in situ processing. Using in-place preconditioning methods to reduce the amount of RAM, network, and disk I/O activity is a promising approach to further address these challenges and will work alongside of the preconditioner techniques already presented in our thesis.

Our thesis has presented methods that are shown to improve overall end-to-end throughput performance but currently require extra memory especially when handling lossless compression. Unfortunately, a downside to all known lossless compression methods is that they require

extraneous use of RAM during the compression process. Lossless compression is a necessary technique for addressing the challenges of improving the performance of data movement without sacrificing data fidelity. On the other hand, overutilization of RAM during compression can negatively affect application viability and performance.

Movement of compressed data, especially between in situ processing stages, needs to be innocuous when it comes to memory utilization. Effective lossless compression without the need for extra memory, called *in-place lossless compression*, is a difficult problem to address due to reasons discussed in the next subsections. Furthermore, traditional compression methodologies, unlike in-place compression, have a deleterious effect on hardware caches available within HPC processors. In-place compression reduces the possibility of cache thrashing for applications that are already cache aware but require the use of compression. This improves overall computational performance and energy consumption.

To address the HPC challenges of limited in-core memory bottlenecks, I propose an effective in-place and cache-aware lossless compression methodology stemming from the previously presented ISOBAR and deinterleaving work. The primary approach for supporting in-place lossless compression lies within supporting *in-place deinterleaving* of a matrix of bytes that can represent an array of data types. The data types can be of any given size including complex data types such as those containing multiple instances of doubles.

5.1.1 In-place Deinterleaving Future Work

Utilizing in-place deinterleaving for a set of bytes is an unexplored area of compression with several complexities. The most difficult aspects of an effective in-place deinterleaving method for an array of data elements are transposing a non-square matrix and accessing memory at a byte-level granularity [69]. The matrix is non-square when dimensions are not the same. There has been recent research towards in-place transposition but they do not provide effective throughput for transposing at the byte level.

On the other hand, out-of-place transposition of bytes has been used as part of a byte transposition process, also called byte *shuffling*, to improve compression efficiency. Byte shuffling is typically a specialization of the deinterleaving preconditioner presented in this thesis. The reason that an out-of-place shuffling process is commonly ignored during a compression workflow is due to requirement of a temporary RAM buffer that is the same size of the input dataset being compressed.

A future in-place transposition process can build on the prior out-of-place deinterleaving work to support disparate cache-aware hardware architectures, such as Cray and Blue Gene systems. In addition, the process can begin immediately within 64 or 128-byte cache-line chunks as data is being generated. Furthermore, it should not cause cache thrashing which is a major

concern for the future of HPC energy consumption and RDMA utilization.

5.1.2 In-place Lossless Compression Future Work

An in-place compression methodology requires approaches beyond efficient in-place byte shuffling. In our proposed future work, the in-place shuffling process combined with a novel improvement on the latest ISOBAR hybrid compression-I/O interleaving optimization is expected to result in a first to exist in-place compression technique.

In-place compression is possible due to the transposition process allowing a single pass of ISOBAR analysis and ISOBAR partitioning without the need for extraneous memory. Additionally, contiguous memory regions become immediately available for next stage of the compression process once the transposition has been applied. The immediate availability of contiguous memory regions allows pipelining of compression without the need of extra memory allocation and without introducing extra timing overhead due to lags in synchronization and multiple data passes.

The following is a consolidated list of several major areas of expected impact for in-place lossless compression:

- To the best of my knowledge and research, this will be the first in-place lossless compression methodology to be utilized for HPC I/O performance gains.
- This will be the second known in-place decompression technique I could find, after LZO decompression. However, a negative against LZO is it requires special non-contiguous segmentation of the compressed input buffer. This is the responsibility of the invoking application and not the decompressor itself.
- The compression/decompression methodology will not adversely affect data movement of buffers that are already optimized for cache utilization. A common example is when applications generate data to fit specifically within L2 cache for optimizing Remote Direct Memory Access (RDMA) operations.
- The in-place transposition algorithm should be effective at the byte level or any other larger granularity. Current published work focuses on only in-place matrix transposition techniques assuming the matrix contains elements of byte size 4 or 8.
- A highly beneficial byproduct of an in-place cache aware lossless compression methodology is that it will be energy efficient by reducing operations to RAM. Accessing RAM instead of cache is very expensive in terms of energy cost.

REFERENCES

[1] H. Abbasi, G. Eisenhauer, M. Wolf, K. Schwan, and S. Klasky. Just in time: Adding value to the IO pipelines of high performance applications with JITStaging. In *Proceedings of the 20th International Symposium on High Performance Distributed Computing*, HPDC '11, pages 27–36. ACM, 2011.

[2] H. Abbasi, M. Wolf, G. Eisenhauer, S. Klasky, K. Schwan, and F. Zheng. DataStager: Scalable data staging services for petascale applications. In *Proceedings of the 18th International Symposium on High Performance Distributed Computing*, HPDC '09, pages 39–48. ACM, 2009.

[3] N. Ali, P. Carns, K. Iskra, D. Kimpe, S. Lang, R. Latham, R. Ross, L. Ward, and P. Sadayappan. Scalable I/O forwarding framework for high-performance computing systems. In *International Conference on Cluster Computing and Workshops*, CLUSTER '09, pages 1–10. IEEE, 2009.

[4] F. Alted. Why modern CPUs are starving and what can be done about it. *Computing in Science and Engineering*, 12(2):68–71, 2010.

[5] P. Beckman, K. Iskra, K. Yoshii, and H. Naik. The ZeptoOS project. http://www.zeptoos.org/.

[6] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate. PLFS: A checkpoint filesystem for parallel applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 21:1–21:12. ACM, 2009.

[7] M. Benzi. Preconditioning techniques for large linear systems: A survey. *Journal of Computational Physics*, 182:418–477, 2002.

[8] D. A. Boyuka, S. Lakshminarasimham, X. Zou, Z. Gong, J. Jenkins, E. R. Schendel, N. Podhorszki, Q. Liu, S. Klasky, and N. F. Samatova. Transparent in situ data transformations in ADIOS. In *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*, pages 256–266. IEEE, 2014.

[9] G. Bronevetsky and A. Moody. Scalable I/O systems via node-local storage: Approaching 1 TB/sec file I/O. Technical report, Lawrence Livermore National Laboratory, 2009.

[10] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. *HP Labs Technical Reports*, 1994.

[11] M. Burtscher and P. Ratanaworabhan. High throughput compression of double-precision floating-point data. In *IEEE Data Compression Conference*, pages 293–302, 2007.

[12] M. Burtscher and P. Ratanaworabhan. FPC: A high-speed compressor for double-precision floating-point data. *IEEE Transactions on Computers*, 58:18–31, 2009.

[13] M. Burtscher and I. Szczyrba. Numerical modeling of brain dynamics in traumatic situations - Impulsive Translations. In *Conference on Mathematics and Engineering Techniques in Medicine and Biological Sciences*, pages 205–211, 2005.

[14] C. Chang and B. Girod. Direction-adaptive discrete wavelet transform for image compression. *IEEE Transactions on Image Processing*, 16(5):1289 –1302, May 2007.

[15] S. Chatterjee and S. Sen. Cache-efficient matrix transposition. In *Sixth International Symposium on High-Performance Computer Architecture*, pages 195–205. IEEE, 2000.

[16] J. H. Chen, A. Choudhary, B. de Supinski, M. DeVries, E. R. Hawkes, S. Klasky, W. K. Liao, K. L. Ma, J. Mellor-Crummey, N. Podhorszki, R. Sankaran, S. Shende, and C. S. Yoo. Terascale direct numerical simulations of turbulent combustion using S3D. *Computational Science and Discovery*, 2(1):015001, 2009.

[17] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes. Cache hierarchy and memory subsystem of the AMD Opteron processor. *IEEE Micro*, 30(2):16–29, 2010.

[18] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51:107–113, Jan. 2008.

[19] M. Dow. Transposing a matrix on a vector computer. *Parallel Computing*, 21(12):1997–2005, 1995.

[20] U. Drepper. What every programmer should know about memory. *Tech. Rep., Red Hat, Inc*, 2007.

[21] C. Dubnicki, L. Gryz, L. Heldt, M. Kaczmarczyk, W. Kilian, P. Strzelczak, J. Szczepkowski, C. Ungureanu, and M. Welnicki. HYDRAstor: A scalable secondary storage. In *Proccedings of the 7th Conference on File and Storage Technologies*, pages 197–210. USENIX Association, 2009.

[22] J.O. Eklundh. Efficient matrix transposition. *Two-Dimensional Digital Signal Prcessing II*, pages 9–35, 1981.

[23] R. D. Falgout. An introduction to Algebraic Multigrid. *Computing in Science and Engineering*, 8:24–33, Nov. 2006.

[24] P. F. Fischer, J. W. Lottes, and S. G. Kerkemeier, 2008. http://nek5000.mcs.anl.gov/.

[25] N. Fout and K.-L. Ma. An adaptive prediction-based approach to lossless compression of floating-point volume data. *IEEE Transactions on Visualization and Computer Graphics*, 18:2295–2304, 2012.

[26] R. W. Freund and N. M. Nachtigal. QMR: A quasi-minimal residual method for non-Hermitian linear systems. *Numerische Mathematik*, 60:315–339, 1991.

[27] M. Frigo, C.E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Symposium on Foundations of Computer Science*, pages 285–297. IEEE, 1999.

[28] B. Fryxell, K. Olson, P. Ricker, F. X. Timmes, M. Zingale, D. Q. Lamb, P. MacNeice, R. Rosner, J. W. Truran, and H. Tufo. FLASH: An adaptive mesh hydrodynamics code for modeling astrophysical thermonuclear flashes. *The Astrophysical Journal Supplement Series*, 131:273–334, November 2000.

[29] J. Gailly and M. Adler. Zlib general purpose compression library. http://zlib.net/, Jan. 2012.

[30] O. Gamoudi, N. Drach, and K. Heydemann. Using runtime activity to dynamically filter out inefficient data prefetches. *Euro-Par Parallel Processing*, pages 338–350, 2011.

[31] K.S. Gatlin and L. Carter. Memory hierarchy considerations for fast transpose and bit-reversals. In *Proceedings of Fifth International Symposium High-Performance on Computer Architecture*, pages 33–42. IEEE, 1999.

[32] R. Ge, X. Feng, and X.H. Sun. SERA-IO: Integrating energy consciousness into parallel I/O middleware. In *Proceedings of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 204–211, 2012.

[33] B. Goeman, H. Vandierendonck, and K. D. Bosschere. Differential FCM: Increasing value prediction accuracy by improving table usage efficiency. In *Seventh International Symposium on High Performance Computer Architecture*, pages 207–216, 2001.

[34] G.C. Goldbogen. PRIM: A fast matrix transpose method. *IEEE Transactions on Software Engineering*, (2):255–257, 1981.

[35] Z. Gong, S. Lakshminarasimhan, J. Jenkins, H. Kolla, S. Ethier, J. Chen, R. Ross, S. Klasky, and N. F. Samatova. Multi-level layout optimization for efficient spatio-temporal queries on ISABELA-compressed data. In *Proceedings of the 26th IEEE International Parallel and Distributed Processing Symposium*, IPDPS '12, 2012.

[36] P. Gschwandtner, T. Fahringer, and R. Prodan. Performance analysis and benchmarking of the Intel SCC. In *Proceedings of International Conference on Cluster Computing*, pages 139–149. IEEE, 2011.

[37] F. Guo and P. Efstathopoulos. Building a high-performance deduplication system. In *Proceedings of the 2011 USENIX Annual Technical Conference*, 2011.

[38] Y. He, R. Lee, Y. Huai, Z. Shao, N. Jain, X. Zhang, and Z. Xu. RCFile: a fast and space-efficient data placement structure in MapReduce-based warehouse systems. In *Proceedings of the 27th IEEE International Conference on Data Engineering*, ICDE '11, pages 1199–1208, 2011.

[39] L. Ibarria, P. Lindstrom, J. Rossignac, and A. Szymczak. Out-of-core compression and decompression of large n-dimensional scalar fields. *Computer Graphics Forum*, 22:343–348, 2003.

[40] K. Iskra, J. M. Romein, K. Yoshii, and P. Beckman. ZOID: I/O-forwarding infrastructure for petascale architectures. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 153–162, 2008.

[41] J. Jenkins, I. Arkatkar, S. Lakshminarasimhan, D. A. Boyuka II, E. R. Schendel, N. Shah, S. Ethier, C.-S. Chang, J. Chen, H. Kolla, S. Klasky, R. Ross, and N. F. Samatova. ALACRITY: Analytics-driven lossless data compression for rapid in-situ indexing, storing, and querying. *Transactions on Large-Scale Data-and Knowledge-Centered Systems X*, 8220:95–114, 2013.

[42] J. Jenkins, E. R. Schendel, S. Lakshminarasimhan, D. A. Boyuka II, T. Rogers, S. Ethier, R. Ross, S. Klasky, and N. F. Samatova. Byte-precision level of detail processing for variable precision analysis. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis*, page 48, 2012.

[43] Y. Jin, S. Lakshminarasimhan, N. Shah, Z. Gong, C.S. Chang, J. Chen, S. Ethier, H. Kolla, S-H. Ku, S. Klasky, R. Latham, R. Ross, K. Schuchardt, and N. F. Samatova. S-preconditioner for multi-fold data reduction with guaranteed user-controlled accuracy. In *Proceedings of the IEEE 11th International Conference on Data Mining*, ICDM '11, pages 290–299, 2011.

[44] S.D. Kaushik, C.H. Huang, J.R. Johnson, R.W. Johnson, and P. Sadayappan. Efficient transposition algorithms for large matrices. In *Proceedings of Supercomputing'93*, pages 656–665. IEEE, 1993.

[45] K. Konstantinides and K. B. Natarajan. An architecture for non-linear noise filtering via piecewise linear compression. *HP Labs Technical Reports*, 1994.

[46] S. Ku, C.S. Chang, and P.H. Diamond. Full-f gyrokinetic particle simulation of centrally heated global ITG turbulence from magnetic axis to edge pedestal top in a realistic Tokamak geometry. *Nuclear Fusion*, 49(11):115021, 2009.

[47] S. Lakshminarasimhan, J. Jenkins, I. Arkatkar, Z. Gong, H. Kolla, S-H Ku, S. Ethier, J. Chen, C.S. Chang, S. Klasky, R. Latham, R. Ross, and N. F. Samatova. ISABELA-QA: Query-driven data analytics over ISABELA-compressed scientific data. In *Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 31:1–31:11. ACM, 2011.

[48] S. Lakshminarasimhan, N. Shah, S. Ethier, S. Klasky, R. Latham, R. Ross, and N. Samatova. Compressing the incompressible with ISABELA: In-situ reduction of spatio-temporal data. In *Proceedings of the 17th International European Conference on Parallel and Distributed Computing*, Euro-Par '11, pages 366–379, 2011.

[49] R. Latham, C. Daley, W. Liao, K. Gao, R. Ross, A. Dubey, and A. Choudhary. A case study for scientific I/O: Improving the FLASH astrophysics code. *Computational Science & Discovery*, 5(1):015001, 2012.

[50] M. Laurenzano, M. Meswani, L. Carrington, A. Snavely, M. Tikir, and S. Poole. Reducing energy usage with memory and computation-aware dynamic frequency scaling. *Euro-Par Parallel Processing*, pages 79–90, 2011.

[51] J. K. Lawder and P. J. H. King. Querying multi-dimensional data indexed using the Hilbert Space-Filling Curve. *SIGMOD Record*, 30:2001, 2001.

[52] J. Li, W-K. Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale. Parallel netCDF: A high-performance scientific I/O interface. In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, SC '03, page 39. ACM, 2003.

[53] P. Lindstrom and M. Isenburg. Fast and efficient compression of floating-point data. *IEEE Transactions on Visualization and Computer Graphics*, 12:1245–1250, 2006.

[54] J. Liu, J. Wu, and D. Panda. High performance RDMA-based MPI implementation over InfiniBand. *International Journal of Parallel Programming*, 32:167–198, 2004.

[55] J. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin. 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, pages 15–24. ACM, 2008.

[56] J. Lofstead, F. Zheng, S. Klasky, and K. Schwan. Adaptable, metadata rich IO methods for portable high performance IO. In *Proceedings of the 2009 IEEE International Symposium on Parallel and Distributed Processing*, IPDPS '09, pages 1–10, 2009.

[57] J. Lofstead, F. Zheng, Q. Liu, S. Klasky, R. Oldfield, T. Kordenbrock, K. Schwan, and M. Wolf. Managing variability in the IO performance of petascale storage systems. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–12, 2010.

[58] K.L. Ma. In situ visualization at extreme scale: Challenges and opportunities. *Computer Graphics and Applications, IEEE*, 29(6):14–19, 2009.

[59] J. D. McCalpin. STREAM: Sustainable memory bandwidth in high performance computers, 2000. http://www.cs.virginia.edu/stream/.

[60] D. T. Meyer and W. J. Bolosky. A study of practical deduplication. *ACM Transactions on Storage*, 7(4):1–20, Feb. 2012.

[61] G. M. Morton. *A Computer Oriented Geodetic Database and a New Technique in File Sequencing.* IBM, Ltd, 1966.

[62] B. K. Natarajan. Filtering random noise via data compression. *Data Compression Conference*, pages 60–69, 1993.

[63] B. K. Natarajan. Occam's razor for functions. In *Proceedings of the Sixth Annual Conference on Computational Learning Theory*, COLT '93, pages 370–376. ACM, 1993.

[64] A. Nisar, W-K. Liao, and A. Choudhary. Scaling parallel I/O performance through I/O delegate and caching system. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '08, pages 1–12, 2008.

[65] W. D. Pence, R. Seaman, and R. L. White. Lossless astronomical image compression and the effects of noise. *Publications of the Astronomical Society of the Pacific*, 121:414–427, Apr. 2009.

[66] M. Polte, J. Lofstead, J. Bent, G. Gibson, S. Klasky, Q. Liu, M. Parashar, N. Podhorszki, K. Schwan, M. Wingate, and M. Wolf. ...and eat it too: high read performance in write-optimized HPC I/O middleware file formats. In *Proceedings of the 4th Annual Workshop on Petascale Data Storage*, PDSW '09, pages 21–25. ACM, 2009.

[67] J. M. Prusa, P. K. Smolarkiewicz, and A. A. Wyszogrodzki. Simulations of gravity wave induced turbulence using 512 PE CRAY T3E. *International Journal of Applied Mathematics and Computational Science*, 11(4):883–898, 2001.

[68] R. Rew and G. Davis. NetCDF: an interface for scientific data access. *IEEE Computer Graphics and Applications*, 10(4):76–82, Jul. 1990.

[69] E. R. Schendel, S. Harenberg, H. Tang, V. Vishwanath, M. E. Papka, and N. F. Samatova. A generic high-performance method for deinterleaving scientific data. In *Euro-Par 2013 Parallel Processing*, pages 571–582. Springer, 2013.

[70] E. R. Schendel, Y. Jin, N. Shah, J. Chen, C.S. Chang, S-H. Ku, S. Ethier, S. Klasky, R. Latham, R. Ross, and N. F. Samatova. ISOBAR preconditioner for effective and high-throughput lossless data compression. In *Proceedings of the 28th International Conference on Data Engineering*, ICDE '12, pages 138–149. IEEE, 2012.

[71] E. R. Schendel, S. V. Pendse, J. Jenkins, D. A. Boyuka II, Z. Gong, S. Lakshminarasimhan, Q. Liu, H. Kolla, J. Chen, S. Klasky, R. Ross, and N. F. Samatova. ISOBAR hybrid compression-I/O interleaving for large-scale parallel I/O optimization. In *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*, pages 61–72. ACM, 2012.

[72] P. Schwan. Lustre: Building a file system for 1000-node clusters. In *Proceedings of the 2003 Linux Symposium*, pages 400–407, Jul. 2003.

[73] Y. Sehoon and W. A. Pearlman. Critical encoding rate in combined denoising and compression. In *IEEE International Conference on Image Processing*, volume 3, pages III – 341–4, Sep. 2005.

[74] R. Sevens, A. White, S. Dosanjh, A. Geist, B. Gorda, K. Yelick, J. Morrison, H. Simon, J. Shalf, J. Nichols, and M. Seager. Scientific grand challenges: Architectures and technologies for extreme scale computing. *Tech. Rep., DOE*, 2009.

[75] J. Seward. The bzip2 and libbzip2 official home page. http://bzip.org, 2000.

[76] C. E. Shannon. Prediction and entropy of printed English. *Bell Systems Technical Journal*, 30:50–64, 1951.

[77] V. Vishwanath, R. Burns, J. Leigh, and M. Seablom. Accelerating tropical cyclone analysis using LambdaRAM, a distributed data cache over wide-area ultra-fast networks. *Future Generation Computer Systems*, 25(2):184–191, 2009.

[78] W. X. Wang, Z. Lin, W. M. Tang, W. W. Lee, S. Ethier, J. L. V. Lewandowski, G. Rewoldt, T. S. Hahm, and J. Manickam. Gyro-kinetic simulation of global turbulent transport properties in Tokamak experiments. *Physics of Plasmas*, 13(9):092505, 2006.

[79] A. B. Watson. Image compression using the discrete cosine transform. *Mathematica Journal*, 4:81–88, 1994.

[80] J. Wei, H. Jiang, K. Zhou, and D. Feng. MAD2: A scalable high-throughput exact deduplication approach for network backup services. In *Proceedings of the IEEE 26th Symposium on Mass Storage Systems and Technologies*, MSST '10, pages 1–14, 2010.

[81] T. A. Welch. A technique for high-performance data compression. *Computer*, 17(6):8–19, June 1984.

[82] B. Welton, D. Kimpe, J. Cope, C.M. Patrick, K. Iskra, and R. Ross. Improving I/O forwarding throughput with data compression. In *International Conference on Cluster Computing*, CLUSTER '11, pages 438–445. IEEE, 2011.

[83] M. Yang, R. E. McGrath, and M. Folk. HDF5 - a high performance data format for earth science. In *21st International Conference on Interactive Information Processing Systems (IIPS) for Meteorology, Oceanography and Hydrology*, 2005.

[84] S. Yiannakis and J. E. Smith. The predictability of data values. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 30, pages 248–258, 1997.

[85] M. Zukowski, S. Heman, N. Nes, and P. A. Boncz. Super-scalar RAM-CPU cache compression. In *Proceedings of the International Conference of Data Engineering*, page 59. IEEE, 2006.

APPENDIX

Appendix A

Dataset Descriptions

Datasets with the prefix “gts" and “xgc" in name are generated from the scientific applications: Gyrokinetic Tokamak Simulation (GTS) [78] and full-function X-point included Gyrokinetic Code (XGC) [46].

1. *gts_phi_l*: linear potential fluctuation variable values of particle-based simulations of fusion plasmas to study plasma micro-turbulence in reactor core and edge.
2. *gts_phi_l*: nonlinear potential fluctuation variable values of the same simulations of fusion plasmas.
3. *gts_chkp_zeon*: values for zeon variable's checkpoint restart data for each 10th time-step of GTS simulation.
4. *gts_chkp_zion*: values for zion variable's checkpoint restart data for each 10th time-step of GTS simulation.
5. *xgc_igid*: ID number of each particle on the fusion plasma edge during XGC simulation.
6. *xgc_iphase*: indicates 8 phase variables of each ion during XGC simulation.

In the application of velocity in the field of astrophysics, there are three datasets for 3 variables respectively generated by code development at the Flash Center: *flash_velx, flash_vely* and *flash_gamc* [28]. Here is the brief illustration of the three datasets.

1. *flash_velx*: fluid velocity x variable values for FLASH.
2. *flash_vely*: fluid velocity y variable values for FLASH.
3. *flash_gamc*: fluid velocity *gamc* variable values for FLASH.

Two single floating-point datasets *s3d_temp* and *s3d_vmag* are generated by the three-dimensional solver application (S3D) [16] used for direct numerical simulations of turbulent combustion.

1. *s3d_temp*: temperature values of S3D simulation.
2. *s3d_vmag*: magnitude of vectors sensed by the toroidal devices.

These dataset names starting with "obs" [12] and "num" [11] comprise measurements from scientific observational instruments and numeric simulations:

1. *obs_error*: data values specifying brightness temperature errors of a weather satellite.
2. *obs_info*: latitude and longitude information of the observation points of a weather satellite.
3. *obs_spitzer*: data from the Spitzer Space Telescope showing a slight darkening as an extra-solar planet disappears behinds its star.
4. *obs_temp*: data from a weather satellite denoting how much the observed temperature differs from the actual contiguous analysis temperature field.
5. *num_brain*: simulation of the velocity field of a human brain during a head impact.
6. *num_comet*: simulation of the comet Shoemaker-Levy 9 entering Jupiter atmosphere.
7. *num_control*: control vector output between two minimization steps in weather-satellite data assimilation.
8. *num_plasma*: simulated plasma temperature evolution of a wire array z-pinch.

Parallel messages datasets have the prefix "msg" [12]. These 5 datasets contain the numeric messages sent by a node in a parallel system running NAS Parallel Benchmark (NPB) and ASCI Purple applications:

1. *msg_bt*: NPB computational fluid dynamics pseudo-application `bt`.
2. *msg_lu*: NPB computational fluid dynamics pseudo-application `lu`.
3. *msg_sp*: NPB computational fluid dynamics pseudo-application `sp`.
4. *msg_sppm*: ASCI Purple solver `sppm`.
5. *msg_sweep3d*: ASCI Purple solver `sweep3d`.