PaSTRI: Error-Bounded Lossy Compression for Two-Electron Integrals in Quantum Chemistry

Ali Murat Gok,[1] Sheng Di,[2] Yuri Alexeev,[2] Dingwen Tao,[3] Vladimir Mironov,[4] Xin Liang,[5] Franck Cappello[2,6]
[1]Northwestern University, USA
[2]Argonne National Laboratory, USA
[3]The University of Alabama, USA
[4]Lomonosov Moscow State University, Russia
[5]University of California, Riverside, USA
[6]University of Illinois at Urbana-Champaign, USA
amg@u.northwestern.edu, {sdi1,yuri}@anl.gov, tao@cs.ua.edu,
vmironov@lcc.chem.msu.ru, xlian007@ucr.edu, cappello@mcs.anl.gov

***Abstract*—Computation of two-electron repulsion integrals is the critical and the most time-consuming step in a typical parallel quantum chemistry simulation. Such calculations have massive computing and storage requirements, which scale as $\mathcal{O}(N^4)$ with the size of a chemical system. Compressing the integral's data and storing it on disk can avoid costly recalculation, significantly speeding the overall quantum chemistry calculations; but it requires a fast compression algorithm. To this end, we developed PaSTRI (Pattern Scaling for Two-electron Repulsion Integrals) and implemented the algorithm in the data compression package SZ. PaSTRI leverages the latent pattern features in the integral dataset and optimizes the calculation of the appropriate number of bits required for the storage of the integral. We have evaluated PaSTRI using integral datasets generated by the quantum chemistry program GAMESS. The results show an excellent 16.8 compression ratio with low overhead, while maintaining 10^{-10} absolute precision based on user's requirement.**

I. Introduction

Today's scientific parallel high-performance computing (HPC) applications are able to work with extremely large amounts of data, typically on the order of terabytes or even petabytes. The sheer size of the data, however, causes serious bottlenecks to the calculations and data processing for extreme-scale applications. In this work, we focus on the compression of the data generated or used by the quantum chemistry programs GAMESS, NWChemEx, and QMCPACK, which are part of three projects in the U.S. Department of Energy Exascale Computing Project. These quantum chemistry programs can generate petabytes of data. The data can be stored in a parallel file system such as Lustre or recomputed on the fly. Because of scaling factors, the storage bandwidth is a serious bottleneck compared with other types of resources such as caches, memories, or high-speed processors.

An efficient data compressor therefore is essential in order to significantly reduce the data size while respecting user-required error bounds. Since lossless compression algorithms such as GZIP/DEFLATE [1] suffer from limited compression ratios [2]–[4], we develop an error-bounded lossy data compression method for quantum chemistry simulations.

In quantum chemistry, the researchers need to solve the Schrödinger differential equation to obtain the wavefunction, which contains all the information about a chemical system. The most time-consuming step typically involves computing the two-electron repulsion integrals (ERIs). The ERI calculations have enormous computing and storage requirements, which scale as $\mathcal{O}(N^4)$ with the size of the chemical system. To make matters worse, solving the Schrödinger equation requires an iterative solver with the number of iterations typically ranging from 10 to 30. Thus, ERIs need to be read from a disk or recomputed from scratch every iteration, adding a massive overhead to the quantum chemistry calculations. Compressing ERIs will enable large quantum chemistry calculations and speed up calculations by reducing the data footprint if ERIs are stored on disk. In this study, we used the popular general-purpose parallel quantum chemistry package GAMESS [5]–[7]. To be specific, all results were based on the integral datasets generated by ERI programs implemented in GAMESS.

The work presented here can benefit many quantum chemistry methods such as restricted Hartree-Fock, unrestricted Hartree-Fock, and density functional theory implemented in virtually all quantum chemistry programs. Furthermore, post-Hartree-Fock methods need to assemble molecular integrals from ERIs. Compressing and storing the latter can lead to considerable speedup of the calculations. Thus we focus on how to optimize the ERI compression quality.

We encountered three major challenges in our work. First, the ERI dataset has a complicated data structure; for example, it consists of blocks of 4D tensors. Second, exploring an effective pattern-matching approach with the best-fit scaling metric is a nontrivial task because datasets are diverse. Third, optimizing the quantization method and developing an encoding strategy is also nontrivial because it requires in-depth analysis of multiple related techniques, as well as careful calculation of the appropriate number of bits required for the storage bytes outputted by them.

To compress ERIs, we developed the algorithm PaSTRI (Pattern Scaling for Two-electron Repulsion Integrals) and implemented it in the data compression package SZ [2], [3]. Compared with the traditional compression algorithms used by SZ, PaSTRI significantly improves the data prediction, quantization method, and encoding techniques based on the specific pattern feature we explored in ERI datasets. The PaSTRI algorithm is implemented as a generic compression algorithm that can work for any dataset as long as it exhibits similar features we explored.

Our contributions are summarized below:

- We developed a sophisticated model to understand how to compress ERIs efficiently, which is a fundamental step for developing an efficient compression algorithm.
- We explored the inherent pattern features in the ERI datasets by different scaling methods.
- We proposed an effective lossy compression technique for ERI datasets by leveraging an optimized pattern matching strategy. We also calculated the proper number of bits required for storing the involved data bytes.
- We implemented a novel lossy compression algorithm and released it as an open source [8].
- We evaluated PaSTRI performance using GAMESS by comparing it with two other state-of-the-art lossy compressors, SZ and ZFP.

The rest of the paper is organized as follows. In Section II we discuss the related work. In Section III we describe the key quantum chemistry concept. Section IV covers the detailed design and implementation of our PaSTRI algorithm. The benchmarking results are demonstrated in Section V. Our conclusions are summarized in Section VI. The key notations used in this paper are summarized in the Appendix.

II. Related Work

Compression of scientific data is an important problem because of ever-increasing volume of data produced by parallel simulations. There are two types of data compression algorithms: lossy and lossless. Popular lossless compressors for scientific datasets are Gzip [1], FPC [9], Zstd [10], and Blosc [11]. Since scientific data is diverse and dynamic, lossless compressors suffer from poor compression ratios (1.1$\sim$2 in most cases), as presented in our prior work [2], [3].

Error-bound lossy compressors offer an attractive alternative because they can significantly improve the compression ratio, while maintaining the required accuracy provided by an user. Many different types of lossy compressors have been proposed, but the most popular are SZ [2], [3], ZFP [12], FPZIP [13], wavelet-based compressors [14], and ISABELA [15]. In our previous work [3], we showed SZ and ZFP provide the best compression ratios for a large variety of HPC applications. In another previous work [16], we investigated how to significantly improve the error-controlled lossy compression for N-body simulations [17], [18]. However, SZ and ZFP are general purpose compressors focusing on the traditional mesh datasets. For example, ZFP works particularly well on 3D datasets, but suffers from the low compression ratio for 1D datasets [3], [4]. This restriction presents a problem in the case of the ERIs because they utilize 1D arrays. SZ exhibits a higher compression ratio on 1D data sets than ZFP, but it does not completely utilize the data features in the chemistry datasets to improve the compression ratio. One common technique is to store data by using a customized real number format, which may lead to a compression ratio of only approximately 1.5-2.5 times, also depending on the target precision of the numerical representation of floating-point numbers [19].

To the best of our knowledge, improving lossy compression based on latent data features for quantum chemistry applications has not been studied yet and our work fills this gap.

III. Two-Electron Repulsion Integrals

A. Overview of Quantum Chemistry Simulation

The goal of quantum chemistry methods is to obtain the wavefunction of a chemical system by solving the Schrödinger equation. The wavefunction is usually represented as a combination of well-known functions, called basis set functions (BFs). Solving the Schrödinger equation in quantum chemistry involves computing different types of integrals. One of the most important types is ERI:

$$(i,j|k,l) = \int_{3D} d\mathbf{r}_1 \int_{3D} d\mathbf{r}_2 \frac{\phi_i(\mathbf{r}_1)\phi_j(\mathbf{r}_1)\phi_k(\mathbf{r}_2)\phi_l(\mathbf{r}_2)}{|\mathbf{r}_1 - \mathbf{r}_2|}, \quad (1)$$

where indices 1 and 2 denote the first and second electron; ϕ_i, ϕ_j, ϕ_k, and ϕ_l are the BFs; and $\mathbf{r}$ is the vector of atomic coordinates. The number of BFs (N) scales linearly with the size of the chemical system, while the number of ERIs scales as $\mathcal{O}(N^4)$. The ERI values are actually repulsion energies between two electron clouds ($\phi_i\phi_j$ and $\phi_k\phi_l$) in the vacuum.

All BFs in a basis set are grouped in shells. Each shell is a set of BFs that share the same center, the exponential coefficient, and the total angular momentum ($l_x + l_y + l_z$). Thus, each shell with the angular momentum l consists of $(l+1)(l+2)/2$ BFs. An example of shell structure is presented in Fig. 1. Depending on the total angular momentum value, a shell in chemistry has an unique one-letter name: $s, p, d, f, \ldots$ for angular momentum $l = 0, 1, 2, 3, \ldots,$ respectively. For example, ERI shell block $(pd|df)$ was formed by p, d, d, f shells.

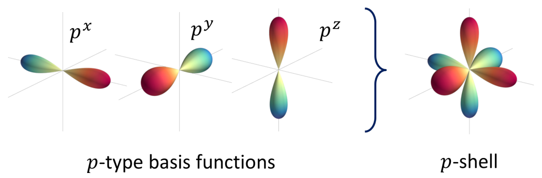


Fig. 1. Demonstration of the shell structure for a p shell. The p shell ($l = 1$) contains three Cartesian Gaussians. BFs with $(l_x, l_y, l_z) \in \{(1,0,0),(0,1,0),(0,0,1)\}$ are denoted as p^x, p^y, and p^z, respectively.

In practice, the amount of data required to store all ERIs for realistic chemical systems is too large (more than hundreds of gigabytes) to fit in the system memory and is usually recomputed from scratch whenever it is needed. As a result, ERIs are recomputed about 10–30 times and in some cases up to 100 times each. This is a very inefficient aproach in terms of computational costs, and an alternative approach would be to store ERIs on a hard disk. However, this also is usually not practical given the size and speed of hard disks. Although storage space is becoming more and more available, disk-based quantum chemistry methods are still bandwidth-limited

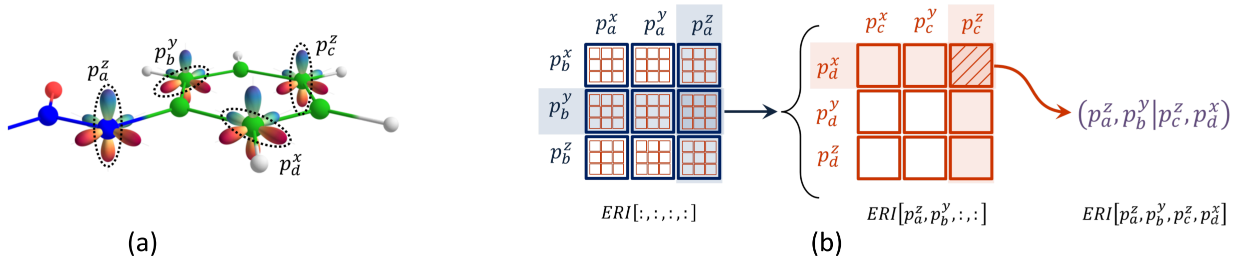


Fig. 2. (a) An illustration of ERI $(p_a^z p_b^y | p_c^z p_d^x)$ on an organic molecule. Each BF of this ERI is a member of a p-type shell, centered on different atoms. (b) Position of the ERI $(p_a^z p_b^y | p_c^z p_d^x)$ in the ERI shell block of $(pp|pp)$ type. This block (denoted as $ERI[:,:,:,:]$) is a 4D tensor, containing all ERIs for corresponding shells.

even for modern solid-state drives [20]. A possible solution for both problems is to store ERIs in compressed form. ERI compression not only can reduce the storage requirements but also can increase the hard disk bandwidth (the increased speed is proportional to the compression rate). Moreover, for relatively small chemical systems, compressed ERIs can even fit in the system memory, which can dramatically increase the speed of quantum chemistry calculations.

B. Patterns in ERI Blocks

A part of an actual integral block of $(dd|dd)$ type generated by GAMESS is shown on Fig. 3. The 4D ERI block (see Fig. 2) is mapped to a 1D array, preserving the original ERI sequence as it is used internally in GAMESS data structures. At first glance, the data shown in Fig. 3 looks random. In reality, if one divides [0:215] dataset into 6 subblocks containing 36 elements, one will find that every subblock has the same pattern. For example, if one compares the first two subblocks [0:35] and [36:71] shown in Fig. 3(a) and Fig. 3(b), one will find the same pattern but with different scaling factors as shown in Fig. 3(c). The deviation and absolute error are close to zero as shown on Fig. 3(d). To explain this result, we developed a quantum chemistry model described in the next paragraph.

According to the Coulomb's law, the farther two electron clouds are from each other (see eq. (1)), the smaller will be ERI value due to the distance $r_{12}^{-1} = |r_1 - r_2|^{-1}$ between clouds. An important consequence of r^{-1} nature of Coulomb's law is that the repulsion energy of the distant non-overlapping electronic clouds does not depend much on their actual shape; rather, it depends mostly on the distance between them. This fact is especially important for Gaussian BFs and their products. They can be safely considered as non-overlapping at some distance because of the exponential decrease of electronic density away from the Gaussian center. Thus, the shape of each ERI can be attributed to the following factors:

$$(i,j|k,l)|_{r_{12}\to\infty} \approx G_{ij}(l_i,l_j,\alpha_i,\alpha_j)G_{kl}(l_k,l_l,\alpha_k,\alpha_l)D_{ij,kl}(r_{12}^{-1}), \tag{2}$$

where G_{ij} and G_{kl} are shape factors depending only on the angular momentum and exponents of the BFs and $D_{pq,rs}$ is a distance factor depending on the distance between BF centers r_{12}^{-1}. Applying eq. (2) to the ERI shell block, we can multiply out the distance factor because it is *approximately* the same for the whole block:

$$(pq|uv)|_{r_{12}\to\infty} \approx (\mathbf{G}_{pq} \otimes \mathbf{G}_{uv})D_{pq,uv}(r_{12}^{-1}), \tag{3}$$

where $\mathbf{G}_{pq}$ is a 2D matrix of shape factor products for shells p and q. $\mathbf{G}_{uv}$ is consequently a product of shells u and v. We note that the direct product of the shape factors in eq. (3) of the whole ERI block carries the shape pattern of both $(pq|$ and $|uv)$ shell pairs. Thus, the shape of the pattern generated by the $|uv)$ shell pairs repeats itself. Consequently, we can observe the shape of $|uv)$ shell pairs by just comparing sub-blocks, where the group size is determined by the $|uv)$ shell parts. When we plot the first two sub-blocks on the same scale as shown on Fig. 3(b), we can see similarities in the patterns. But the similarity becomes more obvious when both curves are rescaled to match each other, as shown in Fig. 3(c). The actual difference is negligible between the two curves (see Fig. 3(c)). This is an important observation, which is exploited in our compression algorithm PaSTRI.

In order for PaSTRI to exploit the latent pattern in the data, the user should provide the information about which BF configuration is being used. After BF configuration is specified, we will show that it is rather trivial to calculate the period and determine the pattern. Since the BF configuration is closely related to the nature of the applications run by the user, such information would typically be available to the user even before the run-time.

IV. Error-Bounded Compression with PaSTRI

In this section, we describe our compression algorithm PaSTRI, and explain how we designed and optimized the error-bounded lossy compression for the data generated by GAMESS ERIs. Our double precision floating-point compression technique solves the following three critical problems:

- Exploration and recognition of potential repeated patterns with one scaling coefficient in each sub-block.
- Optimization of the quantization in order to minimize the number of bits representing the patterns, scaling coefficients, and error correction codes.
- Optimization of encoding technique to significantly reduce the storage size.

The pseudocode of our compression algorithm is presented in Algorithm 1. The first step is reading the input data during which each block is loaded into a full sized array and screened

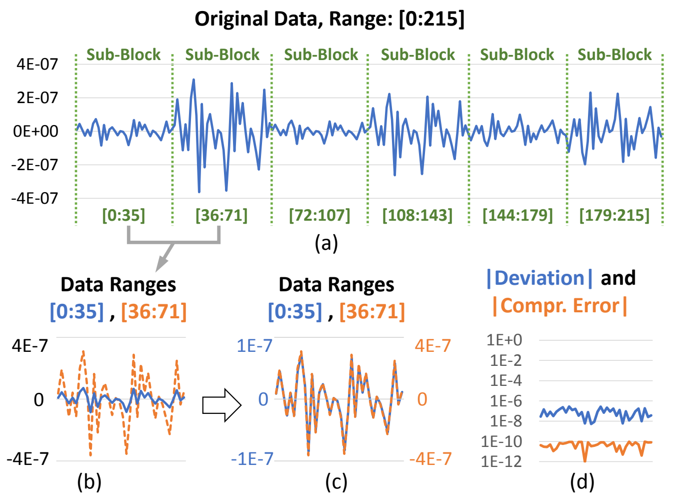


Fig. 3. ERIs can be presented as a 1D array where x-axis is the BF index and y-axis is the integral value. A part of the generated $(dd|dd)$ ERI block is shown in (a). Overlapped first two sub-blocks are shown in (b). Rescaled to match curves are shown in (c). The deviation and absolute compression error between curves are shown in (d), when the the error bound is set to be 10^{-10}.

elements are represented as zeros. The pattern formation occurs only on full-sized blocks; hence, PaSTRI always works on the full-sized blocks. We generate and use one such pattern per block.

The pattern is constructed based on the subblock with the maximum scaling metric in the whole block (see lines 5–11 in Algorithm 1). As presented in Fig. 3(c), all data points in one subblock can match the corresponding data points in another subblock closely as long as we introduce an appropriate scaling coefficient (or multiplication factor) between the two subblocks. Hence, the constructed pattern is also called a *scaled pattern (SP)* in our work.

We explore various scaling metrics to optimize the pattern-matching effect (detailed in Section IV-A). After constructing the pattern in a block, the algorithm compares each subblock against the pattern and calculates just one scaling coefficient to represent all the data values in the subblock.

We model the data points in terms of the SP. Each data point may have a certain deviation from the SP:

$$data_i = S \cdot P_i + dev_i, \quad (4)$$

where $data_i$ and P_i denote the values of the data point i in the subblock and in the pattern, respectively; dev_i refers to the deviation between the two values; and S is the scaling coefficient. In an ideal case, the absolute value of dev_i never exceeds the requested error bound (EB). In this case, using only the pattern and the scaling coefficients is enough to represent the entire block with the required EB. In more general cases, however, there could be one or more *outliers*, whose deviations are larger than EB. Their dev_i values are stored explicitly (lines 12–16 in Algorithm 1) during the *error correction (EC)* step. The saved dev_i values will be further used for reconstructing the original data during decompression.

The compressed file consists of the data points in the pattern, scaling coefficients and EC values (i.e., EC codes) as well as a small header holding the block metadata. Hence, the algorithm will quantize the pattern, scaling coefficients, and error correction codes to realize the compression effect (lines 17–22 in Algorithm 1).

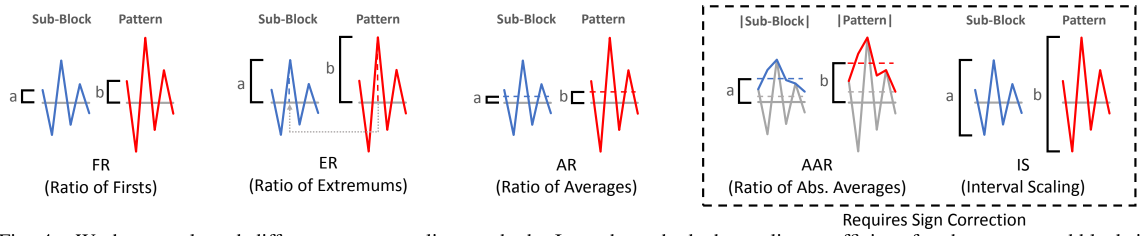
Algorithm 1 PaSTRI Compression Algorithm

```
1:  allocate($ERI(:)$)
2:  for all Blocks do
      ▷ Determine the number of sub-blocks and sub-block size:
3:    $num\_SB \leftarrow N_i^{BF} \cdot N_j^{BF}$
4:    $SB\_size \leftarrow N_k^{BF} \cdot N_l^{BF}$
5:    for all Sub-Blocks do
6:      Calculate pattern scaling metrics
7:    end for
8:    Construct pattern based on the best-fit scaling metric.
9:    for all Sub-Blocks do
10:     Calculate scale coefficients
11:   end for
12:   for all Points do
13:     Calculate error correction (EC) with binSize = $2 \cdot EB$.
14:     $ECQ \leftarrow Quantize(EC)$
15:     Update maxECQ
16:   end for
      ▷ Write the output file
17:   Quantize and write the pattern and the scaling coefficients
18:   Decide encoding method for ECQ
19:   for all Points do
20:     Encode and write ECQ
21:   end for
22: end for
```

The compression ratio could be very high because of the high effectiveness of pattern matching and optimization of quantization in our PaSTRI algorithm. As an example, for a $(fd|ff)$ GAMESS data set, each block contains $10 \times 6 \times 10 \times 10 = 6000$ data points. There are $10 \times 6 = 60$ subblocks, each subblock containing $10 \times 10 = 100$ points. Using our method, only 60 pattern points, 100 scaling coefficients and some error correction codes for the outliers should be stored in the output file. In an ideal case, this corresponds to storing just 160 numbers instead of 6,000, leading to a compression ratio of $6000/160 = 37.5$. Our algorithm reduces the number of bits to store for these values based on the given error bound, pushing the theoretical upper limit of the compression ratio for $(fd|ff)$ to be even higher than 37.5. For example, many blocks have few outliers with relatively small EC code values, leading the compression ratio to be even higher than 100 for those specific blocks.

Because of space limitations, we do not show the pseudocode of the decompression algorithm. In fact, the decompression algorithm is just an inverse procedure of the compression algorithm, including the following steps: (1) assign the number of subblocks and subblock size; (2) reconstruct the pattern (P); (3) reconstruct scaling coefficients (S); (4) calculate l, the local ID inside a subblock; and (5) calculate decompressed values based on EC codes.

In what follows, we detail our floating-point data compression technique, including exploration of the best-fit pattern-scaling metrics, optimization of the quantization method, and a tailored bit-encoding approach.



Method	Comp. Ratio
FR	N/A
ER	17.46
AR	16.92
AAR	17.44
IS	17.20

Fig. 4. We have evaluated different pattern-scaling methods. In each method, the scaling coefficient for the current subblock is calculated as a/b. Some methods may also require sign correction, which either multiplies this coefficient by -1 or not.

A. Exploration of Best-Fit Pattern-Scaling Metrics

We explore multiple pattern-scaling methods and metrics, in order to minimize the data approximation errors and the number of outliers. We note that some of these deviations are naturally occurring as the features of the original data, and they should remain as they are since they contain valuable information. The five possible scaling metrics explored in our study are as follows: *ratio of firsts (FR)*, *ratio of extremums (ER)*, *ratio of averages (AR)*, *ratio of absolute averages (AAR)*, and *interval scaling (IS)*, as illustrated in Fig. 4. Each scaling method selects a particular subblock matching the corresponding condition as the pattern. FR, for example, selects the subblock with the largest absolute value of the first data point as the pattern to be used for matching other subblocks. Similarly, ER, AR, AAR, and IS mean that the pattern will be set to the subblock with extremum point (the data with the largest absolute value in the block), with largest average value, with the largest average absolute value (i.e., the mean value of data points' absolute values), or with the largest value range, respectively. Then, the scaling coefficient of each subblock will be set to the ratio of the corresponding data value in the subblock to that in the pattern.

The reason we always adopt the *largest* value for a scaling method is that the closer the scaling metric is to zero, the more unreliable the scaling effect we will achieve. We note that the scaling coefficient of any subblock must be in the range [-1,1] and typically spans through that range with at least one value being equal to 1. We will take advantage of this property($|S| \leq 1$) during quantization to reduce the size of the output (detailed in Section IV-B). We also note that the scaling coefficients can be positive or negative, implying that we have to deal with the sign of the coefficients properly. Otherwise we may encounter over-large EC values (up to twice as large as the extremum of that block), significantly reducing the compression ratio.

We evaluated all five candidate scaling metrics and noted that the ER metric always leads to the best and most reliable matching effect, as presented in Fig. 4. The reason is that the extremum data point in the whole block represents the amplitude of the data set, which reflects the scaling correlation between different subblocks most precisely. Unlike the ER metric, FR results in much higher matching deviations in that the first data points' values can be close to zero. Moreover, ER has the lowest computation complexity. Specifically, AR and AAR both need to compute a mean value ($\mathcal{O}(N)$), and IS needs to cope with the sign in the operation, introducing extra overhead.

B. Optimization of Quantization Method

The main objective of the quantization is to convert the error correction (EC) values from floating-point data to integer numbers, such that the integer encoding techniques can be applied to reduce storage size. In our study, we first investigate the general quantization equations and then explore the best-fit quantization method based on them.

We denote the quantized versions of pattern, scaling coefficients, and EC values as PQ, SQ, and ECQ, respectively. We denote the number of bits required for them as P_b, S_b, and EC_b, respectively. For simplicity of description, we mainly use ECQ as an example to present how we perform the quantization. The derived equations will also be suitable for SQ and PQ, in addition to ECQ.

The quantized values can be calculated by the following equation with a rounding function:

$$ECQ = round(EC/ECQ_{binsize}) \tag{5}$$

The data reconstructed from the quantized values (i.e., ECQ) actually represent slightly different values than their originals (i.e., EC), where the difference is the quantization error (Equation (6)). This quantization error is represented by ΔEC and is bounded by $ECQ_{binsize}/2$.

$$EC = ECQ \times ECQ_{binsize} \pm \Delta EC, \tag{6}$$

The range of the quantized values can be calculated by using the range of EC and $ECQ_{binsize}$ as follows:

$$ECQ_{range} = round(EC_{range}/ECQ_{binsize}), \tag{7}$$

where $EC_{range} = 2 \times |EC_{ext}|$ and EC_{ext} is the extremum value of EC in the current block.

Considering a symbol-by-symbol, fixed-length encoding to be designed for the quantization, we can calculate the number of bits required per quantized value as follows:

$$EC_b = \lceil log_2(ECQ_{range}) \rceil. \tag{8}$$

Rearranging Equation (7) with the definition of EC_{range}:

$$\begin{aligned} EC_{binsize} \approx EC_{range}/ECQ_{range} &= 2 \times |EC_{ext}|/2^{EC_b} \\ &= |EC_{ext}| \times 2^{-EC_b+1}. \end{aligned} \tag{9}$$

After quantization, Equation (4) becomes

$$\begin{aligned} DecompVal = SQ \times SQ_{binsize} \times PQ \times PQ_{binsize} \\ +ECQ \times ECQ_{binsize}. \end{aligned} \tag{10}$$

In order to guarantee the decompressed values to be within EB distance to the original value, $ECQ_{binsize}$ is chosen to be $2 \times EB$, where EB is the given error bound. Consequently, the EC term should accommodate for both the deviation term in Equation (4) and the quantization errors from S and P. Plugging in the P and S versions of eq. (6), we get

$$\begin{aligned} DecompVal &= (S \pm \Delta S) \times (P \pm \Delta P) \\ &\quad + ECQ \times ECQ_{binsize} \\ &= S \times P \pm S \times \Delta P \pm P \times \Delta S \pm \Delta S \times \Delta P \\ &\quad + ECQ \times ECQ_{binsize}. \end{aligned} \tag{11}$$

Our ultimate goal is to have the compression error, that is the difference between the original value and the decompressed value, to be less than EB for every point in the block:

$$EB \geq \max(\ |CompErr|\ ) = \max(\ |Val - DecompVal|), \tag{12}$$

Plugging in Equations (4) and (11) cancels $S \times P$:

$$\begin{aligned} EB \geq \max(\ |Dev &\pm S \times \Delta P \pm P \times \Delta S \pm \Delta S \times \Delta P \\ &- ECQ \times ECQ_{binsize}|\ ). \end{aligned} \tag{13}$$

We can rewrite this equation without the absolute values and the max operator by considering the worst case, in which all variables get their extremum values and $\pm$'s are selected to make ECQ have its maximum range :

$$\begin{aligned} ECQ_{ext} \times ECQ_{binsize} + EB &\geq |Dev_{ext}| + |S_{ext} \times (\Delta P)_{ext}| \\ &+ |P_{ext} \times (\Delta S)_{ext}| + |(\Delta S)_{ext} \times (\Delta P)_{ext}|. \end{aligned} \tag{14}$$

Some of the variables in this equation are constant within a block. Since PaSTRI works at the block level, we can consider them as constants for the purpose of our calculations. EB and Dev_{ext} are such constants in the equation above. We also know that $ECQ_{binsize} = 2 \times EB$ and $S_{ext} = \pm 1$. Keeping in mind that $(\Delta S)_{ext} = S_{binsize}/2$ and $(\Delta P)_{ext} = P_{binsize}/2$, we can use S and P versions of Equation (9) to represent ΔS and ΔP in terms of S_b and P_b as follows:

$$(\Delta S)_{ext} = S_{binsize}/2 \approx |S_{ext}| \times 2^{-S_b - 2} = 2^{-S_b - 2}, \tag{15a}$$

$$(\Delta P)_{ext} = P_{binsize}/2 \approx |P_{ext}| \times 2^{-P_b - 2}, \tag{15b}$$

where $|P_{ext}|$ is constant within a block. ECQ_{ext} stands for the maximum integer representable by the EC, which is approximately $2^{EC_b - 1}$. Plugging in all the constants and Equations (15) and (9) into Equation (14), we have

$$\begin{aligned} 2^{EC_b - 1} \times 2 \times EB + EB &\geq |Dev_{ext}| + |P_{ext}| \times 2^{-P_b - 2} \\ &+ |P_{ext}| \times 2^{-S_b - 2} + |P_{ext}| \times 2^{-S_b - P_b - 2}, \end{aligned} \tag{16}$$

When S_b and P_b are not close to zero, the last term becomes much smaller in comparison with the others, so it can be dropped. We can also divide both sides by EB, and we end up with the following relationship between S_b, P_b and EC_b:

$$\begin{aligned} 2^{EC_b} \geq (|Dev_{ext}|/EB) - 1 &+ |P_{ext}| \times 2^{-P_b - 2}/EB \\ &+ |P_{ext}| \times 2^{-S_b - 2}/EB. \end{aligned} \tag{17}$$

For simplicity, we rewrite this equation by merging the constants to have one coefficient per term as follows:

$$2^{EC_b} \geq C_1 + C_2 \times 2^{-P_b} + C_3 \times 2^{-S_b}, \tag{18}$$

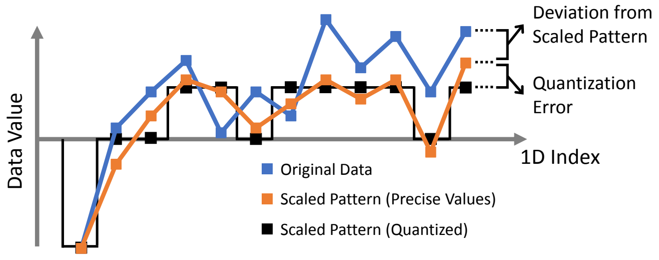


Fig. 5. Demonstration of the quantization affects on the scaled pattern.

where $C_1 = (|Dev_{ext}|/EB) - 1$ and $C_2 = C_3 = |P_{ext}|/4 \times EB)$. Analyzing this equation, we can deduce that EC_b should be large enough to cover the deviations from the SP (the term C_1), and the quantization errors of the pattern and the scaling coefficients (the terms with C_2 and C_3). If S_b and P_b are too large, meaning that the pattern and scaling coefficients have very high resolution, then the term C_1 dominates, and the error correction should still be able to cover the first term. So, the lower bound of EC_b could be calculated as follows:

$$lower\_bound(EC_b) = \lceil log_2(C_1) \rceil. \tag{19}$$

We can see this in Fig. 5, when the quantized SP converges to its precise values, the range required to be covered by ECQ converges to the deviation. On the other hand, if the pattern and scaling coefficients have very low resolution, this means they are represented by fewer bits, the terms with C_2 and C_3 get more important, possibly causing EC_b to increase.

It is not an easy task to find an optimal solution for S_b, P_b and EC_b just from this relationship. Even if we assume the encoding algorithm to be fixed size and symbol by symbol, calculating the optimal number of bits is still too complex. It is a nonlinear integer optimization problem as follows: According to the criteria in Equation (17), we can minimize the number of total output bits:

$$\begin{aligned} TotalBits = SB\_size \times P_b &+ num\_SB \times S_b \\ &+ NOL \times (1DIndexBits + EC_b) \end{aligned} \tag{20}$$

where NOL stands for Number of Outliers. Although one can solve this problem, the computational cost will be undesirably high, and the assumptions for the encoding, specifically the fixed-length part, may not be necessarily true.

The main problem here is to optimally distribute the range covered by EC to the maximum deviation along with the quantization errors of S and P, as indicated in Equation (17).

One can limit the quantization errors on both PQ and SQ by using $P_{binsize} = S_{binsize} = 2 \times EB$, which would result in good resolution on both PQ and SQ, but unnecessarily large S_b. As an example, assuming $EB = 10^{-10}$, some typical ranges for P and EC are $[-10^{-7}, 10^{-7}]$ and $[-10^{-8}, 10^{-8}]$ respectively. S values are always spanning the range of $[-1, 1]$. $EC_{binsize}$ is always equal to $2 \times EB$. One can calculate S_b, P_b and EC_b as 33, 10, and 7, respectively. Consequently, for some blocks the size of the scale coefficients at the compressed output can be significant.

We use a much more practical approach in PaSTRI as follows. We limit the quantization error on only PQ by using

$P_{binsize} = 2 \times EB$. Using $|P_{ext}|$, we can calculate P_b by using the P version of Equation (8). Then we use $S_b = P_b$ during the compression. Consequently, EC_b will be calculated as it is supposed to be by using Equation (8).

With this method, $(\Delta P)_{ext} = EB$ since we limited the quantization error on PQ. Additionally, the largest representable PQ value is $\approx 2^{P_b - 1}$. Consequently $|P_{ext}| = 2^{P_b-1} \times 2 \times EB$. Using these in addition to $S_b = P_b$ and plugging them into Equation (14), we end up with the following:

$$\begin{aligned} ECQ_{ext} \times ECQ_{binsize} + EB \geq |Dev_{ext}| + |S_{ext} \times EB| \\ + |2^{P_b-1} \times 2 \times EB \times (\Delta S)_{ext}| + |(\Delta S)_{ext} \times (\Delta P)_{ext}|, \end{aligned} \tag{21}$$

Following similar steps to our deductions as before, while keeping in mind that $S_b = P_b$, we end up with:

$$\begin{aligned} 2^{EC_b-1} \times 2 \times EB + EB \geq |Dev_{ext}| + |1 \times EB| \\ + |2^{P_b-1} \times 2 \times EB \times 2^{-P_b-1}|, \end{aligned} \tag{22}$$

Rearranging and then dividing both sides by EB, we get:

$$2^{EC_b} \geq (|Dev_{ext}| - 1) + 3/2. \tag{23}$$

EC should accommodate for only 2 more bins compared to the best case, where it needs to accommodate for C_1 bins (Equation (19)). In most cases requiring this will not cause an increase in EC_b, letting it stay at its optimal value.

We have shown that by enforcing the maximum EB quantization error on PQ and then selecting $S_b = P_b$, we have decreased S_b to a significantly lower value while having almost no adverse effects on EC_b. Consequently, our practical method boosts the compression ratio significantly while requiring no computationally expensive steps.

C. Optimization of Encoding

Although precisely estimating the proportions of SQ, PQ, and ECQ in the compressed output file without the encoding phase is hard, ECQ should occupy the largest size in the output file according to our results, thanks to our optimizations during quantization. Additionally, SQ and PQ do not have suitable distributions to exploit during the Encoding stage. Consequently, we adopt an optimized symbol-by-symbol fixed-length encoding method for SQ and PQ, and a variable-length one for ECQ.

We start by analyzing the ECQ value distribution of a real compression execution where thousands of blocks have been compressed in Fig. 6. We group the ECQ values into different bins according to the minimum number of bits required to represent the range $[-|ECQval|, +|ECQval|]$. For example, the value 0 requires at least 1 bit, -1 and 1 require at least 2 bits, $\pm[2, 3]$ requires 3 bits, $\pm[4, 6]$ requires 4 bits, and so on. In general, i bits are required to represent the value range of $\pm[2^{i-2}, 2^{i-1}]$. All values in that range reside in the ith bin. The y-axis represents the total frequency of the values that reside in each bin. We will use this representation to analyze the ECQ distribution to find a good variable-size encoding method for our purposes. Additionally, we also calculate the maximum bin number required per block (denoted as $EC_{b,max}$) and store it in the compressed output file.

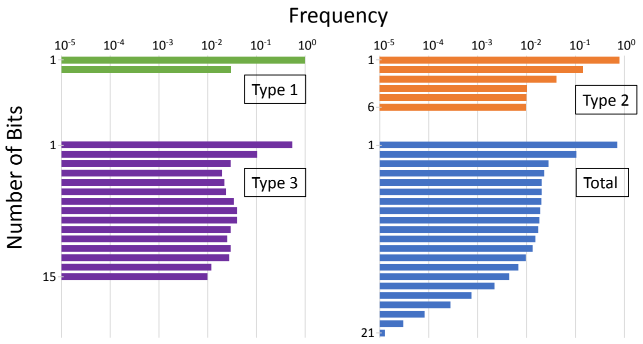


Fig. 6. ECQ value distribution of different block types and of the total data set from a real workload.

According to their ECQ ranges, we observe four types of data blocks (Fig. 6):

- *Type 0*: All ECQ values are 0, so there is no need to keep them in the output file. Even though $EC_{b,max} = 1$, regardless of the encoding tree used, this will be handled as a special case, and no bits will be spent to encode ECQ.
- *Type 1*: Most ECQ values are 0, but there are also 1's or -1's. Consequently, 1 or 2 bits are enough to represent any ECQ value within this block: $EC_{b,max} = 2$
- *Type 2*: The bits required to represent ECQ values is limited to a few (typically ≤ 6), and the vast majority of them are concentrated in lower number of bits.
- *Type 3*: Many bits are required to represent ECQ values (> 6), and there is a significant presence of larger bits. $EC_{b,max}$ typically does not exceed 22 for $EB = 10^{-10}$.

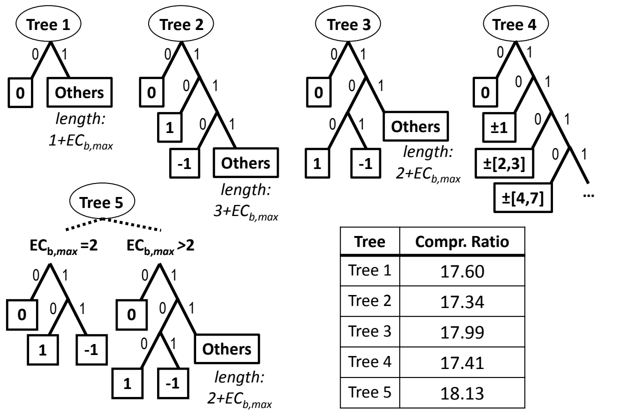
Please note that the type of the block can be determined from the value of $EC_{b,max}$.

The vast majority of the blocks (70-80%) can be categorized as Type 0 or Type 1. Consequently, fixed-length encoding works well enough for these types of blocks. On the other hand, Type 2 or 3 blocks have fewer occurrences, but larger $EC_{b,max}$. If a fixed-length encoding is used, a large ECQ value may cause a large $EC_{b,max}$ the encoded length of all values to increase, which would be wasteful because there are more small values than large ones. In order to solve this problem, we have evaluated multiple different variable-length encoding methods.

We represent each encoding algorithm we have tried as encoding trees in Fig. 7. The leaf nodes in an encoding tree indicate the value being encoded. Starting from the top, the values on the branches to reach such leaves indicate the bit stream used to encode that value. For example, in Tree 2, (Fig. 7) 0 is encoded as 0, 1 as 10, and -1 as 110.

The encoding methods we have evaluated are as follows:

- *Tree 1*: The value 0 is encoded by using only 1 bit (which is bit 0), and the rest of the values are encoded as the bit 1 followed by just the ECQ value encoded in $EC_{b,max}$ bits. With this tree, we aim to focus on encoding 0's as short as possible, since they are the most frequent values. This method results in reasonably good compression ratios.



Tree	Compr. Ratio
Tree 1	17.60
Tree 2	17.34
Tree 3	17.99
Tree 4	17.41
Tree 5	18.13

Fig. 7. Different symbol-by-symbol and variable-length encoding algorithms which we have evaluated.

- *Tree 2*: We put +1 and -1 at higher locations in the tree, leaving the rest requiring 3+EC_b bits to represent. Greedily putting +1 and -1 as high as possible pushes the rest of the values further down in the tree, requiring 2 more bits to encode them compared with Tree 1. Consequently, the compression ratio is degraded because the occurrences of ±1 are not frequent enough to justify such rearrangement.
- *Tree 3*: We rearrange the tree to push "Others" higher, and hence they require 1 less bit. Consequently, the compression ratio is improved.
- *Tree 4*: The depth of a value in the tree depends on its bin in Fig. 6. Each leaf contains power-of-2, namely 2^i, elements in it. In order to obtain the encoded bit patterns, the branches are traversed regularly; and when a leaf is reached, i extra bits are used to represent the value. For example, 0 is encoded as 0, ±1 is encoded by 10 followed by 0 for 1 and 1 for -1, and $\pm[2,3]$ is encoded by 110 followed by 2 bits that uniquely distinguished the value among -3,-2,2,3. We note that this tree ends up being even worse than Tree 1.
- *Tree 5*: This is a smart tree that has two different behaviors depending on the $EC_{b,max}$ value. If it is 2, then the ECQ range is just [-1,1], and we need not consider any other value. Consequently 0 is encoded by 0, 1 by 10 and -1 by 11. For larger $EC_{b,max}$ values, it works the same as Tree 3. This tree results in the best compression ratio due to its adaptive behavior tailored for different block types.

Since the five trees exhibit similar processing speed as we characterized, we select Tree 5 for the implementation of PaSTRI.

One can suggest employing Huffman encoding, which generates the optimal encoding tree by counting the frequencies of values occurrences. This approach has three shortcomings compared with our algorithm:

- It requires storing a dictionary: Since our encoding trees are fixed, they are a part of the implementation and do not need to be included in the compressed data.
- The ECQ range can be huge, with many single-occurrence values. Consequently, huge number of bins would be required for constructing the Huffman tree. Additionally, such single-value occurrences degrade the compression ratio of Huffman encoding, because Huffman encoding requires high-frequency of repeated values to reduce the size effectively.
- It requires generating the dictionary, and the compression cannot continue until this process is completed. In order to amortize for the size of the dictionary, thousands of blocks should be used to create it first, thus serializing the workload and slowing the whole process. On the other hand, PaSTRI is highly parallelizable thanks to its block-level scope.

On the other hand, our Tree 5 uses the optimal tree when $EC_{b,max}=2$, and captures the most important aspects of all the optimal trees for other $EC_{b,max}$ values, where the most frequently occurring values are encoded with fewer bits.

During the encoding stage, PaSTRI decides whether to use sparse representation or non-sparse representation (without indices) to represent the ECQ data. This adaptive behavior also helps boosting compression ratios.

PaSTRI is highly parallelizable, since it has almost no non-parallelizable parts, and it has very small granularity of parallel workloads. Patterns can be considered as symbols in a dictionary, but due to differences in between blocks, each block requires its own pattern. Consequently, there is no need to bundle blocks together to generate a dictionary. Instead, PaSTRI stores one pattern per block, allowing each block to be compressed and decompressed completely independent from each other. Each block, typically, can contain a maximum of 10000 elements with the total data size of 80 kB (for the largest configuration that is commonly used, $(ff|ff)$), which makes each parallelizable workload to be very small for both compression and decompression, resulting in very homogeneous workloads for parallel systems.

V. Performance Evaluation

In this section, we present the evaluation results for double precision floating point data compression with PaSTRI, compared to the state-of-the-art lossy compressors SZ and ZFP.

A. Experimental Setting

We evaluate our results using data sets that correspond to tri-Alanine, Benzene and Glutamine molecules (Fig.8) with $(dd|dd)$ and $(ff|ff)$ BF configurations, which are representative in the quantum chemistry calculations. We have chosen d and f BFs because they are commonly used, their total data size is large and their ERI calculations are costly. In our experiments, we have also used d and f hybrid BF configurations ($(df|fd)$, etc.) but we have reported only the pure configurations for simplicity while presenting the general trends in our results. Metrics for hybrid configurations follow very similar trends of the metrics of pure configurations.

Since full sizes of such quantum chemistry data sets are impractically large, we have sampled them down to at least 2

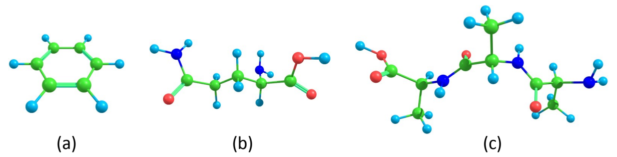


Fig. 8. The molecules used in the benchmarks: (a) Benzene, (b) Glutamine, and (c) tri-Alanine.

GB in size for each BF configuration. We have used Z-Checker [21] infrastructure to evaluate our results, using different error bound (EB) settings of 10^{-9}, 10^{-10} and 10^{-11} (Fig. 9). Typical GAMESS applications require EB to be around 10^{-10}.

In addition, we run a parallel experiment with up to 2,048 cores, in order to assess the I/O performance gain when facilitated with the lossy compression techniques. We conduct our experiments on the Bebop supercomputer [22] using up to 2,048 cores (i.e., 64 nodes, each with two Intel Xeon E5-2695 v4 processors and 128 GB of memory, and each processor with 16 cores). The storage system uses General Parallel File Systems (GPFS). These file systems are located on a raid array and served by multiple file servers. The I/O and storage systems are typical high-end supercomputer facilities. We use the file-per-process mode with POSIX I/O [23] on each process for reading/writing data in parallel. [1]

B. Evaluation Results

As shown in Fig. 9, our compressor PaSTRI exhibits much higher compression ratios than both SZ and ZFP with different error bound settings. SZ and ZFP reach 7.24x and 5.92x double precision floating point data compression ratios, respectively, when the error bound is set to 10^{-10}, while PaSTRI gets the compression ratio up to 16.8X. The key reason is that PaSTRI exploits latent patterns in the data, and employs an optimized quantization and encoding methods additionally.

We break up the structure of storage by PaSTRI as follows: PQ and SQ constitute around 20-30% of PaSTRI's output data size, whereas ECQ constitutes around 70-80%. A tiny portion of the output data, typically less than 0.5%, consists of other bookkeeping bits, for example P_b and EC_b.

PaSTRI runs much faster than SZ and ZFP on both compression and decompression, due to its optimized design and avoiding unnecessary computations with its simple yet effective encoding method as explained in Section IV-C. Specifically, the compression rate of PaSTRI is greater than 660 MB/s on average, whereas it is 308.5 MB/s for ZFP and 104.1 MB/s for SZ (see Fig. 9(c)). PaSTRI achieves over 1110 MB/s decompression rate because of its few decompression operations, whereas ZFP and SZ suffer from 260.5 MB/s and 148.6 MB/s, respectively (see Fig. 9(d)).

Rate-distortion is a commonly used metric to evaluate the compression ratio against the overall distortion of the data. Specifically, the term *rate* here refers to bit rate, which denotes the number of bits used per input data value. This rate can be calculated as 64/compression_ratio. The distortion of data is generally assessed by using the peak signal-to-noise ratio (PSNR), defined as $20\log_{10}(\frac{value\_range}{\sqrt{MSE}})$, where MSE refers to the mean squared error. Having a rate-distortion curve to be close to the upper-left corner means a lower bit rate (or higher compression ratio) and a lower distortion of data. We present the rate distortion of double precision floating-point data compression for Alanine $(dd|dd)$ in Fig. 9(b). We can see that PaSTRI is far better than the other two solutions with respect to the rate-distortion metric. Specifically, with the same PSNR (data distortion), the size of the compressed data generated by PaSTRI is half less than the size of the data generated by SZ or ZFP. All other data sets follow the same trend as Alanine $(dd|dd)$ in terms of PSNR vs. Bitrate, in which PaSTRI is far superior.

We evaluate the I/O performance by employing the compression techniques in the data dumping and data loading as shown in Fig. 10. We do not show the I/O time of writing/reading the original data without compressors because they take extremely long time (more than thousands of seconds). As shown in the two figures, PaSTRI leads to much higher performance (2X or higher) than the other two compressors mainly because of significantly reduced size of data to write/read.

On single dataset, PaSTRI also achieves much better results (e.g., higher compression ratios and higher execution performance) than the other two compressors. On average, PaSTRI achieves an approximately 2.5x better compression ratio while maintaining very low overhead because of the low time complexity ($\mathcal{O}(N)$) of our efficient pattern-matching method and built-in variable-length encoding design. It also achieves a significantly better PSNR vs. Bitrate curve, along with superior parallel execution performance.

We observed that compression ratios do not have large variations for different molecules and BF configurations for each compression algorithm. On the other hand, compression and decompression rates have larger variations. There does not seem to be a clear trend for different BF configurations or molecules in terms of compression ratios, rates and decompression rates. PSNR vs. Bitrate curves have similar trends for all our data sets, with some minor shifts, but PaSTRI always results in a much more favorable curve. MPI execution times are also always in favor of PaSTRI, while being dominated by the disk access times for reading and writing. Since the execution does not take much time, the advantage of PaSTRI depends on its superior compression ratios.

Finally, we have performed a comparison between the original GAMESS infrastructure and PaSTRI infrastructure in terms of total computation time required. In fig. 11, *Original* represents generating the data every time it is needed using original GAMESS functions, whereas *PaSTRI infrastructure* represents generating the data once, then compressing it once by using PaSTRI, and then decompressing it whenever it is needed again. About 87% of GAMESS Hartree-Fock computation time is spent on computing the integrals, which is significantly slower ($(dd|dd)$: 322.82 MB/s, $(ff|ff)$: 622.81 MB/s) than PaSTRI decompression (around 1GB/s for EB = 10^{-10}).

[1]POSIX I/O performance is close to other parallel I/O performance such as MPI-IO [24] when thousands of files are written/read simultaneously on GPFS, as indicated by a recent study [25].

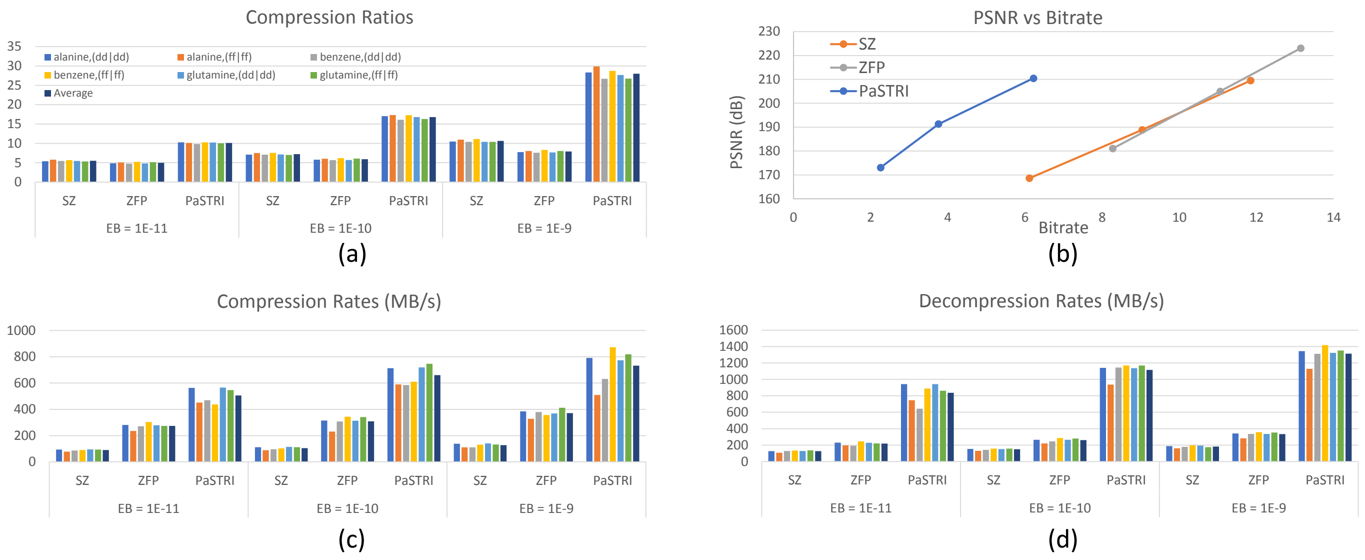


Fig. 9. Comparison of PaSTRI versus SZ and ZFP. (a,c,d): Metrics are calculated for different compression algorithms (SZ, ZFP, PaSTRI), using different data sets (Alanine, Benzene, Glutamine) with different EBs (10^{-11}, 10^{-10}, 10^{-9}). (b): PSNR vs. Bitrate graph for Alanine $(dd|dd)$

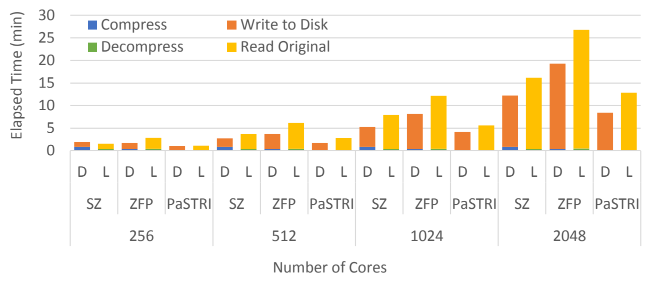


Fig. 10. Parallel Performance of Dumping (D) and Loading (L) Alanine data (dd|dd) to PFS with 256, 512, 1024 and 2048 cores

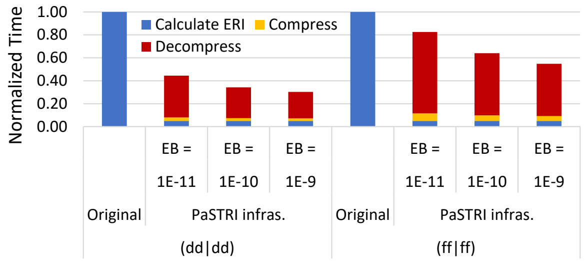


Fig. 11. Total computation time needed to obtain integral data. Disk access times are not included, as in both GAMESS and PaSTRI infrastructures the data is assumed to fit into the memory.

Consequently, it is possible to achieve significant speed-ups by using PaSTRI infrastructure instead of the original GAMESS even for relatively small data reuse values. In fig. 11 we have assumed that the same integral data is used for a total of 20 times, which is a conservatively acceptable value for ERIs.

VI. Conclusions

We proposed a fast compression algorithm for two-electron repulsion integrals used in quantum chemistry. We first performed an in-depth analysis of the latent pattern features of the integral data. We then designed a novel error-bounded lossy compression algorithm called PaSTRI, which utilizes the mined patterns. In PaSTRI, we optimized the number of bits required for storing integral data, patterns, scaling efficients, and error correction values. All the steps have $\mathcal{O}(N)$ complexity, where N is the number of the points per block. We implemented PaSTRI in the SZ package designed for compressing data generated by scientific applications. We then evaluated PaSTRI by using integrals data generated by the quantum chemistry package GAMESS. Our benchmarks show that PaSTRI achieves an excellent 16.8X compression ratio with low overhead, while maintaining 10^{-10} absolute precision based on the requirement of the corresponding researchers. This work not only directly benefits the exascale computing chemistry project GAMESS, but it can be used for compressing any data with pattern features. In the future, we plan to extend it to suit more chemistry applications.

Acknowledgments

This research was supported by the Exascale Computing Project (ECP), Project Number: 17-SC-20-SC, a collaborative effort of two DOE organizations – the Office of Science and the National Nuclear Security Administration, responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering and early testbed platforms, to support the nation's exascale computing imperative. This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357. This research is also supported by the National Science Foundation under Grant No. 1619253. This work is supported by Intel Parallel Compute Center program.

Appendix

TABLE I
List of Abbreviations Used in the Paper

Notation	Description	Notation	Description
BF	basis set function	$N^{BF}_{i/j/k/l}$	size on four dimensions of BF
EB	error bound	ERI	electron repulsion integrals
SB	sub block	SP	scaled pattern
EC	error correction	ECQ	quantized EC values
PQ	quantized pattern values	SQ	quantized scale values
$P_b/S_b/EC_b$	# bits required for pattern, scale, and error correction		

References

[1] P. Deutsch, “DEFLATE Compressed Data Format Specification Version 1.3,” RFC 1951, United States, 1996.

[2] S. Di and F. Cappello, “Fast error-bounded lossy HPC data compression with SZ, year=2016, pages=730-739, issn=1530-2075, month=May,,” in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.

[3] D. Tao, S. Di, Z. Chen, and F. Cappello, “Significantly improving lossy compression for scientific data sets based on multidimensional prediction and error-controlled quantization,” in *2017 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2017, Orlando, FL, USA, May 29 - June 2, 2017*. IEEE Computer Society, 2017, pp. 1129–1139.

[4] S. Di and F. Cappello, “Optimization of error-bounded lossy compression for hard-to-compress HPC data,” *Transactions on Parallel and Distributed Systems (TPDS)*, 2017.

[5] M. W. Schmidt, K. K. Baldridge, J. A. Boatz, S. T. Elbert, M. S. Gordon, J. H. Jensen, S. Koseki, N. Matsunaga, K. A. Nguyen, S. Su, T. L. Windus, M. Dupuis, and J. A. Montgomery, “General atomic and molecular electronic structure system,” *Journal of Computational Chemistry*, vol. 14, no. 11, pp. 1347–1363, 1993.

[6] V. Mironov, A. Moskovsky, M. D‘Mello, and Y. Alexeev, “An efficient MPI/OpenMP parallelization of the Hartree-Fock-Roothaan method for the first generation of Intel Xeon Phi processor architecture,” *The International Journal of High Performance Computing Applications*, 2017.

[7] V. Mironov, Y. Alexeev, K. Keipert, M. D‘mello, A. Moskovsky, and M. S. Gordon, “An efficient MPI/OpenMP parallelization of the Hartree-Fock method for the second generation of Intel Xeon Phi processor,” *arXiv preprint arXiv:1708.00033*, 2017.

[8] “SZ: Fast Error-Bounded Floating-point Data Compressor for Scientific Applications.” [Online]. Available: https://collab.cels.anl.gov/display/ESR/SZ

[9] M. Burtscher and P. Ratanaworabhan, “Fpc: A high-speed compressor for double-precision floating-point data,” *IEEE Transactions on Computers*, vol. 58, no. 1, pp. 18–31, Jan 2009.

[10] Zstandard, http://www.zstd.net, 2017, online.

[11] Blosc compressor, http://blosc.org/, 2017, online.

[12] P. Lindstrom, “Fixed-rate compressed floating-point arrays,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 20, no. 12, pp. 2674–2683, 2014.

[13] P. Lindstrom and M. Isenburg, “Fast and efficient compression of floating-point data,” *TVCG*, vol. 12, no. 5, pp. 1245–1250, 2006.

[14] N. Sasaki, K. Sato, T. Endo, and S. Matsuoka, “Exploration of lossy compression for application-level checkpoint/restart,” in *IPDPS 2015*, 2015, pp. 914–922.

[15] S. Lakshminarasimhan, N. Shah, S. Ethier, S. Ku, C. Chang, S. Klasky, R. Latham, R. B. Ross, and N. F. Samatova, “ISABELA for effective in situ compression of scientific data,” *Concurrency and Computation: Practice and Experience*, vol. 25, no. 4, pp. 524–540, 2013.

[16] D. Tao, S. Di, Z. Chen, and F. Cappello, “In-depth exploration of single-snapshot lossy compression techniques for N-body simulations,” 2018.

[17] S. Habib, A. Pope, H. Finkel, N. Frontiere, K. Heitmann, D. Daniel, P. Fasel, V. Morozov, G. Zagaris, T. Peterka *et al.*, “Hacc: Simulating sky surveys on state-of-the-art supercomputing architectures,” *New Astronomy*, vol. 42, pp. 49–65, 2016.

[18] A. Omeltchenko, T. J. Campbell, R. K. Kalia, X. Liu, A. Nakano, and P. Vashishta, “Scalable I/O of large-scale molecular dynamics simulations: A data-compression algorithm,” *Computer Physics Communications*, vol. 131, no. 1, pp. 78 – 85, 2000.

[19] M. P. Fülscher and P. O. Widmark, “An electron repulsion integral compression algorithm,” *Journal of Computational Chemistry*, vol. 14, no. 1, pp. 8–12, 1993.

[20] V. Mironov, A. Moskovsky, and Y. Alexeev, “Power measurements of Hartree-Fock algorithms using different storage devices,” in *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2017, pp. 1004–1011.

[21] D. Tao, S. Di, H. Guo, Z. Chen, and F. Cappello, “Z-checker: A framework for assessing lossy compression of scientific data,” *The International Journal of High Performance Computing Applications*, 2017. [Online]. Available: https://doi.org/10.1177/1094342017737147

[22] “Bebop supercomputer.” [Online]. Available: https://www.lcrc.anl.gov/systems/resources/bebop

[23] B. Welch, “Posix io extensions for hpc,” in *4th USENIX Conference on File and Storage Technologies (FAST05)*, 2005.

[24] R. Thakur, W. Gropp, and E. Lusk, “On implementing mpi-io portably and with high performance,” in *sixth workshop on I/O in parallel and distributed systems*, 1999, pp. 23–32.

[25] A. Turner, “Parallel i/o performance.” [Online]. Available: https://www.archer.ac.uk/training/virtual/2017-02-08-Parallel-IO/2017_02_ParallelIO_ARCHERWebinar.pdf