PaSTRI: A Novel Data Compression Algorithm for Two-Electron Integrals in Quantum Chemistry

Ali Murat Gok[1,6], Dingwen Tao[2], Sheng Di[6], Vladimir Mironov[3], Yuri Alexeev[5], Franck Cappello[4,6]
[1]Northwestern University (USA); [2]University of California, Riverside (USA); [3]Lomonosov Moscow State University (Russia); [4]University of Illinois Urbana-Champaign (USA); [5]Leadership Computing Facility, Argonne National Laboratory (USA); [6]Argonne National Laboratory (USA);

Abstract

Integral computations for two-electron repulsion energies are very frequently used applications in quantum chemistry. Computational complexity, energy consumption and the size of the output data generated by these computations scales with O(N⁴), where N is the number of atoms simulated in the system. In many applications, the same integrals are required to be calculated multiple times. Storing these values and reusing them requires impractical amounts of storage space; whereas recalculating them requires a lot of computations. On the other hand, generated data typically requires much less precision than the built-in floating point data types. We propose PaSTRI (**Pa**ttern **S**caling for **T**wo-electron **R**epulsion **I**ntegrals), a fast novel compression algorithm which makes it possible to calculate these integrals only once, store them, and reuse them at much smaller computational cost then recalculation. PaSTRI is “lossy” compared to floating point numbers, but still maintains the precision level required by the integral computations. PaSTRI is a part of ECP-EZ project, implemented as one of the compression algorithms in the SZ[1] compressor. We have evaluated our compressor using GAMESS[2] dataset, and achieved 17.5:1 compression ratio whereas compression ratios for original SZ was 8.0:1 and ZFP[3] was 7.1:1.

Exascale Computing Project for Data Compression (ECP-EZ)

The ECP-EZ project aims at significantly improving the compression factors and the usability of SZ. Our goal is to produce a production-quality lossy compressor responding to the needs of exascale application users. PaSTRI is a part of ECP-EZ project, implemented as one of the compression algorithms in the SZ compressor.

Introduction

All properties of the quantum mechanical system are determined by the wave functions which are obtained by solving the Schrödinger Equation. These wave functions define the molecular orbitals, and they can be represented as finite sets of Linear Combination of Atomic Orbitals (LCAO approximation), as shown in Eq.1:

$$\varphi_i = \sum_{\mu=1}^{n} c_{\mu i}\phi_\mu \qquad \text{(Equation 1)}$$

where φ_i is the i-th molecular orbital, $c_{\mu i}$ are the coefficients of linear combination, ϕ_μ is the μ–th atomic orbital, and n is the number of atomic orbitals. Solving the Schrödinger Equation requires finding $c_{\mu i}$'s, which is done by constructing and solving Fock matrix, where the most time-consuming step is computation of two-electron repulsion integrals. Number of integrals scale with N⁴, where N is the number of atomic orbitals(basis functions). Typically, the integrals cannot fit into memory and iterative solution of Schrödinger equation requires these integrals to be recalculated for every iteration, which is typically 20-30 times. A much better strategy is to compute election repulsion integrals(ERIs) once, compress and store them in memory, and read ERIs from memory every time we need them. Consequently, we can potentially speed up calculations by 20-30 times.

GAMESS

(General Atomic and Molecular Electronic Structure System)

- *Ab initio* quantum chemistry package.
- Maintained by the research group of Prof. Mark Gordon at Iowa State University (http://www.msg.ameslab.gov/gamess).
- Enables most major quantum mechanical methods (Hartree-Fock, Møller-Plesset perturbation theory, coupled cluster, multiconfiguration self consistent field, configuration interaction, density functional theory).
- Ported to most major computer architectures.
- Free and widely used on everything from laptops to supercomputers.
- About a million lines of code, with an associated parallelization library comprising 15,000 lines.
- Highly scalable, including many distributed data algorithms.

Two-Electron Integrals

Two-electron repulsion integral calculation involve two couples of atoms, where each couple is sharing one electron. Each one of the four atoms are represented by an index, leading each output data point to include four 16-bit integers as indexes, and a 64-bit double precision floating point number for the resulting integral value. There are 4 types of orbitals commonly used in quantum chemistry, each having a different number of shells: s(1 shell), p(3 shells), d(6 shells), f(10 shells). The data points are generated in blocks, where typically four nested for loop are used to traverse over different types of orbital shells of each atom. Each block is completed when all shell combinations are traversed. Although electrons are shared by two atoms, they are represented by a single Gaussian function during calculations. This results in periodic behavior in the output data, where the periodicity is defined by the shell types in atom couples. There are different block types ((sp|df), (df|pd), (dd|ff), (ff|ff), etc.) which are represented by orbital shell types of each atom.
It should also be noted that integral values typically require less precision than the double precision floats, leading the original data to be unnecessarily large. Maximum tolerable error is called Error Bound(EB), which is 10^{-10} by default in GAMESS. PaSTRI always satisfy the specified error bound in our decompressed values. Data points may also have a sparse representation, which may decrease the original data size when most of the values inside a block is smaller than EB. In sparse representation, each data point whose integral value is greater than EB is represented with their indexes and integral value. On the contrary in non-sparse representation only the integral values for all possible index combinations are represented, in a lexicographical order. If needed, corresponding indexes can be calculated with the help of lexicographical ordering. All blocks in GAMESS application by default have sparse representation.

Periodicity and Pattern Determination

In order to observe the periodicity, we have a non-sparse representation of the integral values in a single block (Fig 1a). All possible index combinations are included in the lexicographical order, and if the original block has sparse representation the skipped values can simply be replaced by zeros.
In this representation, we observe a periodic behavior within each block. Although there are some relatively small deviations, the general behavior is observed as there is a “Pattern”(Fig 1b) that repeats itself in different scales.

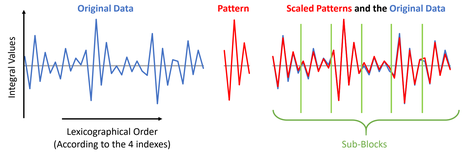

Figure 1 – Original data fits onto a scheme that a pattern repeats itself at different scales, with small deviations

We name each one of these repetitions as a Sub-Block(SB) (Fig 1c).
SB length is fixed, and determined by the multiplication of the number of the last 2 orbital's shells.
Number of SBs is determined by the multiplication of the number of the first 2 orbital's shells.
Example: (sd|pf) : SB length = Pattern length = 3*10 =30, Number of SBs = 1*6 = 6
We need to choose one of the SBs to be the Pattern(P). In order to eliminate the potential effects of the deviations, and also to ensure that Scale values will be in the interval of [-1:1] the Pattern is selected to be the SB with the largest metric, whereas this metric is determined by the scale calculation method.

Calculating the Scales

One scale value is calculated per SB. We evaluated several methods to calculate the scales (Fig 2):
1) FR (Ratio of Firsts): Scale = SB[1] / P[1] (Surprisingly good results, but unreliable)
2) ER (Ratio of Extremums): Find the index of extremum in P, namely e. Scale = SB[e] / P[e]
3) AR (Ratio of Averages): Same as RatioScale = (∑ SB[i]) / (∑ P[i])
4) AAR (Ratio of Absolute Averages): Scale = (∑ |SB[i]|) / (∑ |P[i]|) * SignCorrection
5) IS (Interval Scaling): Scale = Range(SB) / Range(P) * SignCorrection

Where Sign Correction is 1 or -1. AAR and IS methods require Sign Correction because they can only produce positive values without it, whereas the scale values can be positive or negative. Reliable sign correction methods increase computational costs.

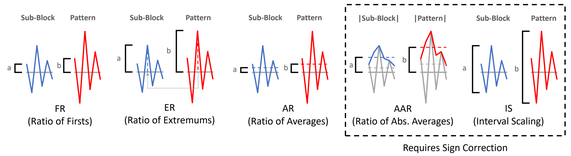

Figure 2 – Different scale calculation methods. Scale is calculated as a/b (Sign correction may also be applied)

Handling the Outliers

A data point whose values is not within EB distance to the scaled pattern is called an Outlier.
We only store the difference between the value of each outlier point and the scaled pattern, and name it Error Correction(EC). This difference is typically much smaller than the outlier values, consequently contributing to a higher compression ratio. EC representation also employs quantization, always using bin size of 2*EB, which guarantees all decompressed values to be within EB distance to the original data values. Outliers can be encoded in a sparse or a non-sparse manner. Sparse representations of Outliers would require encoding indexes along with EC values. We observe that index values within a block may be large, but their ranges are low.
Example: Index ranges for (df|df): [150:155] [300:309] [450:455] [600:609]
In order to represent the indexes, index offsets are separated and then four indexes are merged into one (Fig 3).

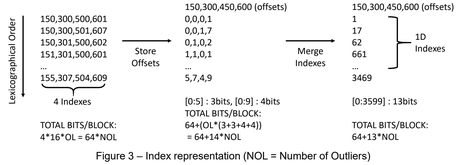

Figure 3 – Index representation (NOL = Number of Outliers)

The method mentioned above was more successful than GZIP (LZ77+Huffman) due to the fact that indexes have a very uniform distribution inside a block.

Quantization

Decompressed Value = Pattern * Scale + Error Correction

How to represent Pattern(P), Scale(S) and Error Correction(EC) in terms of quantization bins efficiently?
EC should always have quantization bin size of 2*EB to guarantee errors to be bounded by EB.
S is always in [-1,1], and example ranges for others: P:[-10⁻⁶:-10⁸], EC:[-10⁻⁶:10⁸].

We tried different approaches for quantization and calculating the bin sizes of S, P and EC:
1) No quantization: Represent them with doubles/floats. Too small compression!
2) Enforce EB on all representations: 2*EB will be used as bin size for all. Better than (1), but S_bits is too large!
3) Use optimal number of bits: Optimal number of bits for a symbol-by-symbol, fixed length compression can be obtained by solving a nonlinear integer optimization problem. A (dd|ff) example is shown below:

$$\text{Constraint: } C_1 + C_2{*}2^{\text{P\_bits}} + C_3{*}2^{\text{S\_bits}} \le 2^{\text{EC\_bits}}$$

$$\text{Minimize: } \{100{*}\text{P\_bits} + 36{*}\text{S\_bits} + \text{NOL}{*}(13 + \text{EC\_bits})\} = \text{Total compressed bits}$$

Where C_1, C_2 and C_3 are some constants. Solving this problem introduces too much computational cost. We use a more practical approach as follows:
4) Enforce EB on P. Calculate P_bits. Use: S_bits = P_bits. It can be mathematically proven that EC will require only one extra bin(compared to (2) above), but most probably not an extra bit. Consequently, we have reduced S_bits at almost no cost, while still keeping total computational cost low. We achieve acceptable compression ratios with this approach.

EC has a highly concentrated distribution around values 0, 1 and -1. Utilizing this fact, we have tried numerous variable length encoding methods, and decided on the following technique:
If EC range is [-1:1], then: 0→“0”, 1 →“10”, -1 →“11”
Otherwise, 0 →“0”, 1 →“100”, -1 →“101”, Others →{ “11” followed by EC value (EC_bits+2 bits per symbol) }
We have also tried GZIP to encode quantized EC values, but the technique mentioned above achieves slightly better compression ratio, at much smaller computational costs.

Encoding

During encoding phase, firstly the quantized pattern and scale values are written to the output file. Then our algorithm chooses the output format with the smallest size of the following:
1) Uncompressed, Sparse: Compression may not be useful when there were very few values that were above EB in the current block(highly-sparse block)
2) Compressed, Non-Sparse: Only EC values are encoded (no indexes)
3) Compressed, Sparse: Outliers are represented with their quantized EC value and 1D index

Evaluation

Due to their computational complexity and output data size, we have focused on d and f orbitals during our evaluations. We present average compression ratios for (dd|dd), (dd|ff), (ff|dd), (ff|ff) orbitals using different scale calculation methods in Fig 4.

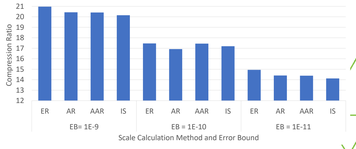

Figure 4 – Compression ratios for different scale calculation methods

We choose ER(Ratios of Extremums) in the implementation of PaSTRI because ER has slightly better compression ratios, and also requiring the smallest amount of computations. We then compare PaSTRI with vanilla SZ and ZFP compressors in Fig 5, where integer indexes are compressed using GZIP[4] in the latter two.

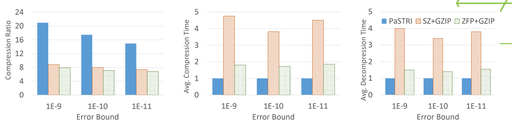

Figure 5 – PaSTRI in comparison with other compressors

PaSTRI achieves ~2x better compression ratio while maintaining the lowest execution time.

Conclusion

We propose PaSTRI, a fast, novel compression algorithm for two-electron repulsion integrals which is a frequent application in quantum chemistry. Our algorithm consist of determining the pattern, calculating the scales, calculating error corrections and then encoding the whole block. All the steps have O(N) complexity, where N is the points per block. PaSTRI achieves 2.5 times better compression ratios at lowest computational costs. We have evaluated our results on GAMESS dataset, but also expect to have similar results for other ERI applications because the way integrals are calculated would be similar.

References

[1] 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”, *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Orlando, FL, USA, 2017, pp. 1129-1139.
[2] M. W. Schmidt, K. K. Baldridge, J. A. Boatz, S. T. Elbert, M. S. Gordon, J. H. Jensen, S. K. N. Matsunaga, K. A. Nguyen, S. Su, T. L. Windus, M. Dupuis, J. A. Montgomery Jr., S. Koseki, N. Matsunaga, K. A. Nguyen, S. Su, and others, “General atomic and molecular electronic structure system”, J. Comput. Chem., vol. 14, no. 11, pp. 1347–1363, 1993.
[3] P. Lindstrom, “Fixed-Rate Compressed Floating-Point Arrays”, in IEEE Transactions on Visualization and Computer Graphics, 20(12):2674–2683, 2014.
[4] Gzip compression. Available at http://www.gzip.org.

Acknowledgments
This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. This work was supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research Program, under Contract DE-AC02-06CH11357.
The submitted manuscript has been created by UChicago Argonne, LLC, Operator of Argonne National Laboratory (“Argonne”). Argonne, a U.S. Department of Energy Office of Science laboratory, is operated under Contract No. DE-AC02-06CH11357. The U.S. Government retains for itself, and others acting on its behalf, a paid-up nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan. http://energy.gov/downloads/doe-public-access-plan.