RDIF a preprocessing Filter for HDF5

Raka Jovanovic
Texas AM University
At Qatar
PO Box 23874,Doha
Qatar
rakabog@yahoo.com

Rudolph A. Lorentz
Texas AM University
At Qatar
PO Box 23874, Doha
Qatar
rudolph.lorentz@qatar.tamu.edu

Abstract: The aim of this paper is to provide the users of the data format HDF5 with a preprocessor package for lossless compression for all of its predefined numerical data types. Combining this package with the built in compression filters generally compresses the data better than any of them individually, while not being essentially slower than the standard compression filter. In addition, it is as easy to use as any of the built in filters. We have tested the increase in compression ratio that occurs when this new filter (RDIF) is combined with compression filters that are a part of HDF5 like SZIP and GZIP.

Key-Words: - Data Compression, HDF5, Szip

1 Introduction

With the increased use of computers in a wide range of problems from government, engineering to science the amount of created digital data is constantly increasing. The problem of storage is of great importance. One approach is using more hardware, to store the data. Although in many cases the need for extra hardware cannot be avoided, it can be seriously decreased using data compression. One of the areas that data compression has proven very valuable is application to images[1, 2] and sound[3]. There are different approaches to data compression like lossy and lossless. Lossy data compression achieves much higher compression rations but in contrast to the lossless version, the data cannot be fully restored. In our paper we focus on lossless data compression. The compression process can in most cases be divided in two parts: the encoding stage and the preprocessing stage. The goal of the encoding is to decrease the data size, by using long bit sequences for rarely appearing values and short ones for frequent values or some similar method. Encoding methods do not take into account higher level properties like continuity and periodicity of data, which makes them more robust but in many cases significantly less efficient. Preprocessing takes advantage of these properties and transforms data so it could be more efficiently encoded. To be able to use these properties, it is necessary to keep data in a more structured form than just a sequence of bits for which only the creating program knows their meaning.

HDF5 is a data model, library and file format that is used for saving data that describes complex systems with a wide range of monitored parameters. It records the data type, data space, connections between data objects and other properties. It has become wide-spread and is used by many scientific projects [4] and is used by important research software like Wolfram Mathematica [5] and MatLab [6]. A great feature of HDF5 is the existence of data compression filters like GZIP and SZIP [7, 8]. HDF5 allows creating filter pipelines or in other words a sequence of filters that will be applied on data. This makes the creation and use of preprocessing filters simple. HDF contains as a standard filter Shuffle [9] that is used for preprocessing data. Shuffle changes the order of bytes in the data stream.

We have developed a new filter for HDF5 that uses the multidimensional correlation of neighboring data values. We use the concept of predicting the value of a point using its neighbors and storing the difference with the original. With a good prediction function these differences are easier to compress than the initial values. In our filter we use one, two and three dimensional differences using Lorenzo predictor [10] The use of differences is not directly applicable to floating point numbers, at least not when using floating point computations. For solving this problem, we adopted the method for converting floating point data to integers proposed by Martin Isenburg [11]. We show in our tests that combining our filter with other compression and preprocessing filters increases the compression ratio without a significant rise in calculation time. These tests have been conducted on different examples of scientific data.

This article is organized as follows. In Section 1 we give details about the prediction functions being used. In the following section we explain the software implementation and integration into HDF5. In the final section we show the results applying our filter to different types of data sets using different combination of filters.

2 The RDIF Filter in HDF5

HDF5 is used for managing and storing data in a structured way. One of the main features of HDF5 is the possibility of automatic compression of data when it is saved to data storage devices. HDF5 has several standard compression filters. N-bit and ScaleOffset are two basic compression filters that are effective when used on data that belongs to some sub range of the possible values standard data types [12]. HDF5 also contains more powerful general compression filters like GZIP which implements the deflate algorithm[13], and SZIP which implements an enhanced version of Rice encoding [7, 14]. Another group of filters are BZIP2[15] and LZF [16] that are used for compression in HDF5 but have been developed by third parties and are not included as standard parts of HDF5.

The HDF5 library has a possibility of applying multiple filters on data sets by creating filter pipelines. Specific preprocessing filters can be created by users that can prepare data for better compression using known properties of it. These user made filters can exploit properties of a very high level that are specific to the data (weather data, crash test data). Shuffle is a general preprocessing filter that is a part of HDF5, and can be applied to any data type. It rearranges the byte order of a data set by grouping them by their significance or in other words by their position in individual data elements. Shuffle manages to increase the compression ratio by taking into account the type of data that is being compressed. One of the drawbacks of this filter is that it has no effect on byte data and in general, it is more effective on larger data types.

Data that is stored by HDF5 is more complex than just having information about its type. We have developed a new general preprocessing filter (RDIF) for HDF5 based on the Lorenzo predictor that does not use just the type of data but also its structure. In RDIF we use the concept of predicting the value of a data element using the values of its neighbors. If the prediction function is good these differences should be small and better suited for compression. In our filter we use the information about the dimensions of the data space and depending on it, we implement one, two or multidimensional predictor. It can be applied to all standard types of data 8, 16, 32 and 64 bit signed and unsigned integers and on 32 and 64 bit floating point numbers.

The filter is written in ANSI C and it is used as a library file (lib). It can easily be added to existing software that uses HDF5 by adding RDIFLIB.lib to linking process. To use the filter we need to register it with HDF5 with the following code.

```
#include "HDF5_RDif.h"
....
RegisterRDif();
```

When packing data the following code should be add to for including RDIF to the filter pipeline.

```
H5Pset_filter(prop, FILTER_RDIF, 0, 0, NULL);
```

As previously mentioned, the prediction is done differently for one, two and multi dimensional data. In the case of one dimensional data we use the nearest neighbor as does SZIP, Equation 1

$$Pred_i = Data_i \tag{1}$$

In the two dimensional case we use one dimensional predictions on the borders $i=0$, $j=0$ and for the remaining elements the two dimensional Lorenzo predictor Equation 2.

$$Pred_{i,j} = data_{i-1,j} + data_{i,j-1} - data_{i-1,j-1} \tag{2}$$

In the case of higher dimensions, we implement the two dimensional prediction on the lowest level.

One problem with using differences is that more bits may be needed to store it than the original data. As an example: we take the case of byte data and we have a sequence of the following numbers 0xff, 0x0, 0xff. Their differences are -0xff and 0xff and it is obvious that an extra bit will be needed for storing the sign. This is a well known problem and we have solved it using interleaving

$$\delta_i = \begin{cases} 2\Delta_i & , \ 0 \le \Delta_i \le \theta_i \\ -2|\Delta_i| - 1 & , -\theta_i \le \Delta_i < 0 \\ \theta_i + |\Delta_i| & , |\Delta_i| > \theta_i \end{cases} \tag{3}$$

Where $\theta_i = \min(Data_i - \min, \max - Data_i)$, [min, max] is the possible range of data type. Δ is the difference of prediction and actual data value. In the two dimensional predictions this problem is even

more complex due to the fact that $|\Delta|$ can also have an overflow, but we do not go into the details of this problem article.

This type of prediction and differences cannot be directly applied to floating point numbers when floating point arithmetic is used. When floating point subtraction is calculated, there is possibility of underflows appearing with irreversible loss of data. There have been several methods of avoiding this problem. One is using bit XOR instead of subtraction [17]. Another approach is applying integer arithmetic on the floating point numbers in the sense that bits of float pointing numbers are read as integers [18]. In our filter we adopted the approach given in article [11] for converting floating point numbers to integers. The IEEE standard for floating point numbers [19] is given by the following Eq. 4

$$(-1)^s 2^{e-2^{n_e-1}-n_m+1}(2^{n_m}+m) \qquad (4)$$

An IEEE single (double) precision number consists of a sign bit s, an n_e = 8 (11) bit exponent e, and an n_m= 23(52) bit mantissa m that generally represent the number. The conversion we use is given in Eq. 5

$$Int(f)=\begin{cases} f\ XOR(1<<(BitLength-1)), f<=0 \\ f\ XOR((1<<BitLength)-1), f<0 \end{cases} \qquad (5)$$

In practical applications, this means that if the number is positive change its leading bit to 1, and in the case it is negative invert all of its bits. The result of this conversion of floating point number is a monotonic mapping to unsigned integers that preserves ordering and even linearity of differences for floats with the same sign and exponent.

3 Performance evaluations

In this section we present our test for evaluating the performance of RDIF. The goal of these tests is to show that adding our preprocessor to the filter pipeline increases the compression ratio with negligible extra calculation time. We also analyze the effect of combining RDIF with other filters. We have performed tests on existing scientific data saved in HDF5 format. These files are combination of several data types that are stored in arrays of varying size and dimensions. We first observe the overall compression ratio on these files. The second part of our analysis is observing the accomplished compression ratio on data generated by sampling functions of two variables.

In our tests, we use data acquired from the NASA web site, using the MIRADOR web interface for data users. The data is in HDF5 format. We collected data files of different sizes and from different measurements. We use data from the OMI (The Ozone Monitoring Instrument), MLS-Aura (The Microwave Limb Sounder aboard the EOS-Aura spacecraft) and HIRDLS-AURA (The High Resolution Dynamics Limb Sounder aboard the EOS Aura spacecraft). We have also test data acquired from the web site of the Laboratory for computational Astrophysics, University of California, San Diego. In these tests we compare the speed of data repacking using GZIP and SZIP as compression filters and their combination with Shuffle and RDIF. We have done tests with the GZIP optimization parameter having values of 5 and 9. In the case of SZIP, we used blocks of size 8. When we use SZIP separately, it has the nearest neighbor set to true. When it was used in combination with RDIF it would have just entropy encoding, due to the fact that RDIF and nearest neighbor are partially overlapping. The packing/unpacking tests have been done using the standard HDF5 tool H5repack slightly modified to be able to use RDIF filter. We have measured the time needed for H5repack to load a HDF5 file with no filters applied and save it with a new set of filters for packing and in the opposite direction for unpacking. In all of our tests, we used a Dell Optiplex 755, Intel(R) Core(TM) Duo CPU E8500 @ 3.16GHz, 3.25 GB of Ram with Windows XP Professional Service pack 3. The results of these tests are given in the Tables [1, 2, 3, 4]. In all the tables, we use SZIP to indicate that SZIP filter has been used. GZIP and GZIP9 in the case GZIP filter is used with compression parameter set to 5 or 9 respectively. If extra filters have been added to the pipeline we use "+S" for Shuffle, and "+D" for RDIFF.

Table 1. Comparison of different combinations of HDF5 filters on file test.float.hdf5 from CLA. The size of the file is 0.1 megabytes. Compression and decompression times are measured in seconds.

Filters	Com. time	Com ratio	Dec. time
SZIP	0.0030	1.525794	0.0030
SZIP+S	0.0062	1.439842	0.0030
SZIP+D	0.0030	1.397089	0.0030
GZIP	0.0030	2.098752	0.0030
GZIP9	0.0062	2.132849	0.0062
GZIP+S	0.0062	**2.271142**	0.0030
GZIP+D	0.0030	2.058578	0.0062
GZIP+D+S	0.0062	2.233088	0.0030

Table 2. Comparison of different combinations of HDF5 filters on file MLS-Aura_L2GP-GPH_v01-52-c01_2007d059.he5. The size of the file is 1.6 megabytes. Compression and decompression times are measured in seconds.

Filters	Com. time	Com ratio	Dec. time
SZIP	0.0906	1.545986	0.1156
SZIP+S	0.0936	1.482578	0.1062
SZIP+D	0.0906	1.740075	0.1030
GZIP	0.4062	1.508945	0.0906
GZIP9	0.4062	1.508945	0.1030
GZIP+S	0.4156	1.740808	0.0968
GZIP+D	0.4156	1.631318	0.0936
GZIP+D+S	0.4062	**1.804798**	0.0968

Table 3. Comparison of different combination of HDF5 filters on file HIRDLS-Aura_L2_v02-04-09-c03_2008d001.he5. The size of the file is 120.4 megabytes. Compression and decompression times are measured in seconds.

Filters	Com. time	Com ratio	Dec. time
SZIP	0.7250	10.520906	5.5530
SZIP+S	0.9312	7.654173	5.2126
SZIP+D	0.7030	11.774671	5.1750
GZIP	1.6780	11.756691	5.0624
GZIP9	1.9374	11.775440	5.2530
GZIP+S	1.6280	13.262722	6.3312
GZIP+D	1.9562	12.231457	5.9406
GZIP+D+S	1.7592	**13.589447**	5.3250

Table4. Comparison of different combination of HDF5 filters on file MLS-Aura_L1BOA_v02-21-c01_2007d044.h5. The size of the file is 319.5 megabytes. Compression and decompression times are measured in seconds.

Filters	Com. time	Com ratio	Dec. time
SZIP	16.4030	1.218118	16.3930
SZIP+S	13.9592	1.157996	14.9406
SZIP+D	12.7030	1.327255	14.8624
GZIP	31.4062	1.163908	13.5842
GZIP9	31.6624	1.163812	13.8156
GZIP+S	28.5468	1.363979	13.1718
GZIP+D	31.1374	1.411996	14.0906
GZIP+D+S	28.2906	**1.625557**	14.1203

We first notice that only in the case from Table 1 did our filter not have a good performance. It has slightly degraded the level of compression for both SZIP and GZIP. This is due to the fact that in this case the data was only one dimensional and it was separated into small chunks. The HDF5 library has been developed in a fashion that it can be further developed independent of filters. Because of this RDIF has to add extra header information that stores data type and dimensions. In all the other tested files the compression ratio has improved. The improvement varies from 10% to 30%. Adding RDIF to the filter pipeline had only minor change in calculation time. A surprising effect was that in some cases it even decreased the overall compression time. This can be explained by smaller write time to the hard drive.

Surprisingly, after the process of packing and unpacking, HDF5 files were not bit-wise identical for different combination of filters. To confirm the validity of our filter, we used the h5diff tool provided by HDF5 group. This tool is used to compare the data inside two HDF5 files. Using h5diff we have seen that the data and the structure of tested files have been exactly preserved for all combination of filters.

The second type of tests that we have conducted is on two dimensional smooth functions. We generated data that we compress and store using HDF5 and in the opposite direction read and decompressed data into memory. The function we used is $F(x,y) = (2+\sin(x)+\cos(y))m$. We tested RDIF for float 32 bit data ($m=1$) and for integer 16 bit data ($m = 16000$). The data was a 657* 660 matrix. We can see the results in Tables 5, 6.

Table 5. Comparison of different combination of HDF5 filters for generated float 32 bit

Filters	Com. mb/s	Com ratio	Dec. time
SZIP	35.21	1.675539	0.0172
SZIP+S	32.35	1.489831	0.0187
SZIP+D	11.91	9.864591	0.0125
GZIP	10.94	1.140343	0.0140
GZIP9	10.69	1.140343	0.0140
GZIP+S	11.66	1.934022	0.0109
GZIP+D	3.08	11.86597	0.0110
GZIP+D+S	2.36	**14.97928**	0.0125

Table 6. Comparison of different combination of HDF5 filters for generated float 32 bit

Filters	Com. mb/s	Com ratio	Dec. time
SZIP	20.13	1.389290	0.0015
SZIP+S	28.62	1.091209	0.0016
SZIP+D	10.33	8.647445	0.0016
GZIP	14.38	1.026243	0.0016
GZIP9	14.40	1.026243	0.0016
GZIP+S	11.27	1.686536	0.0016
GZIP+D	9.63	9.635869	0.0016
GZIP+D+S	2.25	**11.214086**	0.0016

In these tests the compression ratio has been greatly improved even up to 10 times when RDIF was added. This shows that RDIF is very suitable for preprocessing smooth data. We believe that in many of the cases of smooth functions even higher order differences would give even better results, which we plan to add to our filter in the future.

Our tests have shown that the combined use of Shuffle and SZIP gives poor results. Shuffle improves the performance of GZIP. If RDIF is added before SHUFFLE in the filter pipeline, even greater compression is achieved by GZIP. Shuffle should not be used before our filter due to the fact after changing byte order differences will not be calculated on true values of data. RDIF improves the compression ratio of GZIP, but the best results are acquired when it is combined with Shuffle. Decompression time in all the tested cases was very similar for all the tested combination of filters and data sets. In the case of compression, SZIP was 2-4 times faster than GZIP.

4 Conclusions

In this paper we have presented RDIF, a new preprocessing filter for HDF5 that is currently being developed. The main goal of RDIF is to improve the compression ratio achieved in HDF5 using existing compression filters. The use of this filter is simple and can be easily added to existing software that uses HDF5. We have tried to increase compression ration by exploiting properties like data type and data dimensions that are preserved inside HDF5. Preprocessing is done using predictions based on values of neighboring elements and storing the difference between the prediction and the actual value. RDIF can be applied to all HDF5 supported numerical data types. The use of our filter has proven to be efficient on real data acquired by NASA and gave an increase in compression ratio 10%-30% with a small effect on calculation time. We have also tested the use of the new filter on data generated by two dimensional smooth functions. For this type of data RDIF has shown to be extremely efficient and improvement the compression ratio even up to 10 times.

In the future we plan to add a possibility of selecting predictions with higher level differences, extending the prediction calculations to higher dimensions.

References:

[1] S. Amat, J. Ruiz, and J. C. Trillo, “Compression of Color Images Using Nonlinear Multiresolution Schemes,” *WSEAS Transactions on Signal Processing,* vol. 2, no. 2, pp. 302-308, 2006.

[2] B. Carpentieri, “Image compression via textual substitution,” *WSEAS Transactions Information Science. and Application,* vol. 6, no. 5, pp. 768-777, 2009.

[3] S. Saha, and V. Reddy, “Audio Compression Technology for Voice Transmission,” *WSEAS Transactions on Circuits and Systems,* vol. 9, no. 3, pp. 1858-1862, 2004.

[4] The HDF5 group. "HDF5 Users," www.bigdata.org/HDF5/users5.html

[5] Wolfram Research Inc. "HDF5, Wolfram Mathematica documentation Center," http://reference.wolfram.com/mathematica/ref/format/HDF5.html.

[6] The MathWorks Inc. "HDF5, MATLAB Function Reference," http://matlab.izmiran.ru/help/techdoc/ref/hdf5.html.

[7] Y. Pen-Shu., X.-S. Wei., M. Lowel. *et al.*, “Implementation of ccsds lossless data compression in hdf,” in Earth Science Technology Conference, Pasadena, 2002.

[8] Consultative Committee for Space Data Systems, *BLUE BOOK*, Washington, DC: CCSDS Secretariat Program Integration Division (Code MG) National Aeronautics and Space Administration, 1997.

[9] "Performance evaluation report: gzip, bzip2 compression with and without shuffling algorithm,"http://www.hdfgroup.org/HDF5/doc_resource/H5Shuffle_Perf.pdf.

[10] L. Ibarria, P. Lindstrom, J. Rossignac *et al.*, “Out-of-core compression and decompression of large n-dimensional scalar fields,” *Computer Graphics Forum,* vol. 22, pp. 343-348, 2003.

[11] M. Isenburg, “Fast and efficient compression of floating-point data,” *IEEE Transactions on Visualization and Computer Graphics,* vol. 12, no. 5, pp. 1245-1250, 2006.

[12] M. Yang. "N-bit and ScaleOffset filters," http://www.hdfgroup.org/training/hdf5-class/HDF5-nbit-scaleoffset.pdf.

[13] D. Salomon, *Data Compression: the Complete reference*, London: Springer-Verlang, 2007.

[14] P.-S. Yeh, "The CCSDS Lossless Data Compression Recommendation for Space Applications," *Lossless Compression HandBook*, pp. 311-326 London: Academic Press, 2003.

[15] S. Julian. "BZIP2 and LIBBZIP2, version 1.0.5: A program and library for data compression," November, 2009; www.bzip.org

[16] A. Collette. "LZF Compression Filter for HDF5," http://h5py.alfven.org/lzf/.
[17] P. Ratanaworabhan, J. Ke, and M. Burtscher, "Fast lossless compression of scientific floating-point data," in Data Compression Conference, 2006, pp. 133-142.
[18] V. Engelson, D. Fritzson, and P. Fritzson, "Lossless compression of high-volume numerical data from simulations," in D. Data Compression Conference, 2000, pp. 574-586.
[19] IEEE 754: , *Standard for binary floating-point arithmetic*, 1985.