Noname manuscript No.
(will be inserted by the editor)

Reducing the HPC-Datastorage Footprint with MAFISC - Multidimensional Adaptive Filtering Improved Scientific data Compression

Nathanael Hübbe · Julian Kunkel

Received: date / Accepted: date

Abstract Large HPC installations today also include large data storage installations. Data compression can significantly reduce the amount of data, and it was one of our goals to find out, how much compression can do for climate data. The price of compression is, of course, the need for additional computational resources, so our second goal was to relate the savings of compression to the costs it necessitates.

In this paper we present the results of our analysis of typical climate data. A lossless algorithm based on these insights is developed and its compression ratio is compared to that of standard compression tools. As it turns out, this algorithm is general enough to be usefull for a large class of scientific data, which is the reason we speak of MAFISC as a method for scientific data compression. A numeric problem for lossless compression of scientific data is identified and a possible solution is given. Finally, we discuss the economics of data compression in HPC environments using the example of the German Climate Computing Center.

Keywords Data Compression · NetCDF · HDF5

1 Introduction

During the last decades, chip designers have worked hard to uphold Moore's Law of exponential transistor count growth, giving the supercomputers of today tremendous amounts of computational power. Data storage and data transfer, however, have not been able to keep up [14]. Consequently, the costs for data storage are taking up an ever increasing share of HPC investments when data intensive sciences are concerned. This is especially true for climate research, where supercomputers take the role of data production tools. The produced data needs to be held available for further research, analysis and documentation of scientific results. For the DKRZ (German Climate Computing Center) this means annual costs of over 100,000 € just for tapes.

Data compression methods have been available for approximately half a century already. Some lossy compression methods, like `jpeg`[7] and `mp3`[2] have since become state of the art in consumer products. In the HPC domain, however, huge amounts of data are still stored without any compression at all. For tape archives, the state of the art is to use an `sldc`[5] based compression, an algorithm that cannot reach the level of compression achieved by other standard algorithms.

Scientists shy away from all lossy compression methods since it is usually not clear in what ways the compression degrades the scientific value of the data. This does not stop people from evaluating lossy methods for scientific data compression [16], though. For this reason, we restricted ourselves to systematically investigating what lossless data compression can do for climate data.

1.1 Paper outline

First the current state of the art in data compression is outlined and the current handling of scientific data is presented in Section 2. After that, Section 3 takes a closer look at the nature of the data that is to be com-

Nathanael Hübbe
University of Hamburg
E-mail: nathanael.huebbe@informatik.uni-hamburg.de

Julian Kunkel
University of Hamburg
E-mail: kunkel@informatik.uni-hamburg.de

pressed. A few issues concerning the representation of floating point data are discussed in Section 4.

In Section 5 we start to detail the actual filters we used to reduce the size of the data; the final algorithm of MAFISC is outlined in Section 6.

Finally, the performance of MAFISC is evaluated and compared to a number of standard compression algorithms in Section 7, and the impact of data compression on HPC economics is analysed in Section 8. Section 9 wraps up the discussion and provides an outlook on our future work.

2 State of the art

2.1 Data compression

Even though compression schemes are diverse, all algorithms tend to use the same three step structure: The first step applies some kind of filter to the data, the second step eliminates repeating patterns before the data is fed to an entropy coder, which constitutes the final step. Many compression algorithms omit one of these three steps; however, the steps that are present are always applied in this order.

Filter (step 1): This first step could also be called a redundancy transformation step. For example, an array of numbers counting up does not contain much information, however long the sequence is. Yet no number ever appears a second time, so many compression algorithms will see no repetitions and fail to compress it well. Filtering the data first by replacing each number with its difference to the previous one will result in an array of ones. And this sequence can be compressed by simple run length encoding. The redundancy present in the original data has been converted into an exploitable form by changing the representation of the data.

The goal of this filtering step is not to reduce the amount of data itself, it may even require additional data to be stored to be able to undo the filtering step. Examples for such filters are the discrete cosine transform used by jpeg compression [7] and the wavelet transform [8] implemented in jpeg2000[3]. For lossy compression algorithms, this filtering step also introduces changes that cannot be undone, like quantizing the data or deleting small entries.

A rather radical filter has been proposed by Lakshminarasimhan et al. [10], sorting the data blockwise to generate a smooth curve that can easily be compressed using curve fitting. Of course, for invertability, this filter requires the indices of all the sorted data points to be stored in the compressed format. Finding new filters for specific kinds of data is a topic of ongoing research.

General compression algorithms, like `zip`, `lzma` and `sldc`[5] do not include a filtering step since they do not know what kind of data is fed to them.

Repetition elimination (step 2): Almost all data files, even binary ones, contain sequences of bytes that appear quite frequently. Consequently, most standard compression algorithms include a step where they replace byte sequences by small references to a previous occurence. This is usually an offset, but other dictionary based aproaches exist. There are huge differences in the quality of this step. `sldc`, for instance, can only eleminate repetitions that occur within one KiB of data, while `lzma` can find and use matches several mebibytes back.

`bzip2` is special at this point: Like all other good general compressors, it takes advantage of recurring patterns, but it does not use a dictionary for this purpose. It relies on the Burrows-Wheeler transformation to translate recurring patterns into repeating bytes, and this is, technically, a filter[6].

Entropy coding (step 3): The last step, common to all good compression schemes, is the entropy coder. The data is compressed by using fewer bits for the more frequent symbols than for the less frequent ones. This is a closed subject in research. It can be proven that no scheme will be better than the Shanon limit, and arithmetic coding already works at that limit[12].

Even though the use of an entropy coder as the last step can be regarded as state of the art, there are notable compression schemes that omit this step; `sldc` [5] is such an example. These methods rely entirely on the elimination of recurring patterns and compress with a severely suboptimal ratio. Nevertheless, they are used in applications where simplicity is valued higher than a good compression ratio; in online tape compression for example.

Since standard compression tools are already very good at eliminating recurring patterns and perfect at entropy coding, we developed filters adapted to climate data, relying on the standard tools for the other two steps.

2.2 Data handling in data intensive sciences

Today, we see the need to handle large datasets in a variety of sciences. While specialized file formats exist in a number of these fields, many scientists use NetCDF [11] or HDF5 [9] files to store their data. The advantage of these two formats is that they allow any kind of multidimensional data to be stored in a general file format, using a unified library interface. These files can even be fairly self-descriptive through the use of attributes that store meta information along with the data itself.

The HDF5 library contains a filter interface that can be used to compress the data before it is written to disc and to decompress it transparently when it is read again. The library itself includes support for GZIP, SZIP and NBIT compression currently (as of Version 1.8). More filters can be added by programs using the library. Using a nonstandard filter, however, limits the usability of the data to the programs that know this filter.

Originally, the NetCDF file format did not allow for compression. With Version 4 however, the library has replaced its own file format with the HDF5 file format. Consequently, NetCDF-4 allows to use some of the standard filters provided by HDF5, including SZIP and GZIP compression. Transparent compression as it is offered by these libraries has the advantage that the data will never be stored uncompressed. The downside, of course, is the neccessity to (de)compress the data whenever it is accessed.

For long term storage, the current state of the art is to use tape archives. At this point, compression is currently state of the art. The tape drives can compress the data on the fly as it is written, actually increasing the throughput by fitting more data on a fixed length of tape. On reading, it is again the tape drive that transparently decompresses the data to deliver the original data written to the archive.

Another way to reduce archive sizes is the use of deduplication methods like the one presented in [4]. The basic idea here is to divide the data into blocks and to employ bloom filters to determine if a specific data block has already been seen. If this is applied on a file system level, it can automatically eliminate repeated copies of a single file. Deduplication can be seen as a coarse form of compression.

While the tape drive compression automatically saves a lot of storage space, it was likely not to be the best practice possible. In this paper we compare the compressions achieved by different methods, including our newly developed algorithm, and analyze their economical impact. Doing so, it becomes evident that the tape drive compression is indeed not the best choice, even though it is the simplest option.

3 Properties of scientific data

Many scientific datasets, consist of a number of different physical variables sampled on a multidimensional grid. Such data can be stored in two fundamentally different ways: A record containing the values for all the different variables can be stored for each grid point, or a multidimensional array of values can be stored for each variable. Scientists tend to use the later variant, with the consequence that *similar data is stored together*. This similarity encompases the legal value range, the value distribution (possibly as a function of the location within the multidimensional array) and the smoothness properties of the data.

Since the data is *multidimensional*, each value has more than just two neighbours. These neighbourhood relations may be exploited for compression. However, they are invisible to the standard compression algorithms.

Many scientific datasets describe real valued variables which are stored as *floating point numbers*, usually with single precision. Consequently, each value occupies four or eight bytes that are in a close relationship to each other.

Climate data tends to be *smooth* in one direction at least. However, at the same time, climate data also tends to include sharp jumps, many of which are triggered by areas for which the variable is undefined. Another problem are variables with saturating values: The cloud area fraction, for example, frequently reaches zero or one, but cannot exceed these values. The saturation leads to sharp peaks in the second derivation. Finally, smoothness is not a property of all variables. In short: Smoothness is abundant and should be exploited, yet it cannot be relied upon.

4 Developing invertable filters for floating point data

For the purpose of lossless compression, invertible operations are mandatory. Floating point arithmetic, however, is not even invertible for such simple operations as additions and subtractions; and some of the filters we developed rely on differences to produce small values. So a way had to be found to make the necessary calculations invertible without producing unnecessarily large values.

Most of the time, floating point data is represented either in the single or double precision format standardized by the IEEE. These floating point formats have an interesting and, as detailed below, helpful property: The order of the positive floating point numbers is the same as the order of their binary representations interpreted as integers. In other words: If all positive floating point numbers were sorted in an array, the index for each number (an integer) would have the same bit pattern as the floating point number itself. This property is due to the relative position of the exponent bits to the mantissa, and the fact that the exponent is stored as an unsigned biased exponent.

Since two floating point numbers with similar values will also be close to one another in the array of all positive floating point values, their bit representations (interpreted as integers) will also be close to one another. So, calculating their difference in fixed point arithmetic will yield small values as well. An example of this is given in Figure 1: The values 2.125 and 1.875 use different exponents (the tinted bits). The difference computed with fixed point arithmetic, however, has no bits set in the exponent area.

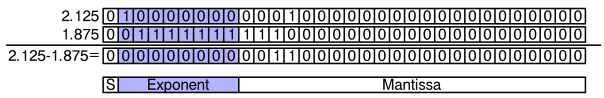

Fig. 1 Fixed point arithmetic difference of two positive single precision IEEE floating point numbers that are close to one another. Even though the exponent is different, the fixed point difference has no bits set in the exponent part.

Unfortunately, this is only true for positive floating point values. Negative values show the inverse sorting as compared to negative integer values in 2-complement representation: Negative zero is represented by 0x80000000 (single precision), which is the smallest possible signed integer in two complement representation. All the other negative values count up from this value. This fact leads to the situation shown in Figure 2: The first line shows the smallest representable positive number 2^{-149}, below is the negative of it. Subtracting the two numbers with fixed point arithmetic yields the most negative signed integer even though only the two zero values lie in between.

Fig. 2 Fixed point arithmetic difference of the smallest positive and the smallest negative single precision number. Even though only the two zeros lie in between, the computed difference has the most significant bit set.

All that needs to be done to solve this problem, is to invert all bits except the sign bit for negative values. This inverts the sorting of the negative values. In Figure 3 this has been done to 2^{-149} and -2^{-149}, yielding a fixed point arithmetic difference of three, which is the number of steps to go from -2^{-149} over -0 and $+0$ to 2^{-149}. The result is a representation of floating point values that looks like the two complement representation of signed integers.

What this transformation cannot alleviate is the fact that one half of the valid floating point values lie in the interval between -2 and 2, even the interval from

Fig. 3 Fixed point arithmetic differences of the smallest positive and the smallest negative single precision number after transformation. Now the difference is as small as it should be.

-10^{-19} to 10^{-19} encompasses a quarter of the valid values. The consequence is, that any variable with a value range including zero will produce very large fixed point arithmetic differences, and, even worse, the differences will be on different scales depending on the scale of the values compared. Unfortunately, there is no way to alleviate this problem without losing precision, with the consequence that the compression would be lossy. So, for now, we decided to ignore the problem for our compression algorithm.

5 Developed filters

This section describes some of the different filters we developed and tested. Not all of them are actually used by our final algorithm; yet, they all have been implemented and subjected to preliminary tests to evaluate their potential. It is important to understand that all the filters presented in this section can be applied one after the other, theoretically in any order and any number of times. Nevertheless, some combinations do not make sense at all and would never be able to enhance compression. Also note, that these filters use only fixed point arithmetic to be invertible; floating point data is converted as described in section 4 in an implicit first filtering step so that it can be handled using fixed point arithmetic as well.

5.1 Simple difference filters

Each value is replaced by its difference to a neighbouring value to exploit the smoothness of the data. The smoothness leads to small differences being stored, and small values imply low entropy. Since a notable amount of scientific datasets exhibit smoothness in their derivations as well, it can be profitable to apply several difference filters in sequence.

5.2 Linear spline filter

The goal of this filter was to better exploit global properties of the data. This filter is based on the interpolation between two values to provide a prediction for a value in between. Again, the difference between the

value and the prediction is stored. This filter was implemented hierarchically, so that the value in the middle is used as a predicting value for the next step of finer resolution.

One problem with this filter is, that it cannot be applied multiple times sensibly. However, our tests have also shown that the compression achieved by using the linear spline filter is less than that produced by a simple difference filter run. Together with the added implementation complexity, linear spline filtering appears to be inferior to difference filtering in all aspects.

5.3 Bitsorting filters

With this class of filters it has been investigated whether it is possible to enhance compression by changing the order of bits in the file. Several distinct approaches have been tested; sorting all the most significant bits into one block for example, followed by a block with the 30th bits of each value and so on. Since there are almost always bits in scientific data that are constant in all values, this transformation creates long runs of zeros or 255s that can be compressed very easily. In the example shown in Table 1, this would lead to a quarter of the file containing only six such runs.

byte 3		byte 2		byte 1		byte 0	
bit	freq.	bit	freq.	bit	freq.	bit	freq.
31	0%	23	97.6%	15	49.8%	7	49.9%
30	100%	22	2.3%	14	50.1%	6	49.9%
29	0%	21	56.9%	13	50.0%	5	50.0%
28	0%	20	50.0%	12	50.0%	4	49.9%
27	0%	19	47.3%	11	49.9%	3	50.0%
26	100%	18	46.0%	10	50.0%	2	49.9%
25	0%	17	50.5%	9	50.0%	1	50.0%
24	100%	16	50.0%	8	50.0%	0	50.0%

Table 1 An example for the frequencies at which the bits inside the single precision float values are set. The variable used gives the air pressure at the ozone maximum.

byte 3		byte 2		byte 1		byte 0	
bit	freq.	bit	freq.	bit	freq.	bit	freq.
31	0%	30	100%	29	0%	28	0%
27	0%	26	100%	25	0%	24	100%
23	97.6%	22	2.3%	21	56.9%	20	50.0%
19	47.3%	18	46.0%	17	50.5%	16	50.0%
15	49.8%	14	50.1%	13	50.0%	12	50.0%
11	49.9%	10	50.0%	9	50.0%	8	50.0%
7	49.9%	6	49.9%	5	50.0%	4	49.9%
3	50.0%	2	49.9%	1	50.0%	0	50.0%

Table 2 The bit frequencies of Table 1 after value internal reordering.

Another approach was to reorder the bits of each value internally. Using this method, the four/eight most significant bits of a value are distributed to the most significant bits of its four/eight bytes, and so on. This distributes the entropy evenly across the bytes, which is better for entropy coding. Reordering the bits of the example shown in Table 1 in this fashion produces the bit frequencies shown in Table 2.

While we saw in our preliminary tests that these filters do help `gzip`, `zip` and `sldc` to shrink the data, they tend to be counterproductive for `bzip2`. For some files, the value internal reordering was actually quite successful when applied after the adaptive filter and the prefix transformation described below.

5.4 Prefix transformation

For reasons presented in Section 4, all these filters see only integer values in two complement representation. Negative values in this representation have their most significant bits set while positive values are zero prefixed. When small values are stored in multibyte integers, the bytes of higher significance are either zero or 255, depending on the sign of their value. The prefix transformation inverts all bits of negative values, excluding the sign bit itself, and rotates the value left one bit so that all small values use a zero prefix, whether they are positive or not. It is mandatory to apply this transformation before bitsorting methods are used, since it changes the frequencies at which many bits are set from 50% to zero.

Note that a part of applying this filter actually reverses the effect of the sign transformation described in Section 4, which is always applied to floating point data before any other filter sees the data. These two transformations serve two different purposes: The sign transformation of Section 4 is applied to allow the filters to use fixed point arithmetic, the prefix transformation described here produces constant prefixes for better compression and should be used after all arithmetic based filters.

5.5 Adaptive filter

Plain difference filters are simple and powerful, yet they have a disadvantage: They cannot adapt to the given data. Scientific data, however, consists of many different variables with very diverse properties, often changing their characteristics within a record. The adaptive filter exploits this fact. Like the difference filters, the adaptive filter replaces each stored value by its difference to some other value which is used as a prediction.

However, instead of using a fixed neighbour for prediction, the adaptive filter chooses between a number of applicable predictors. The predictors we use are as follows:

Zero predictor
: The simplest possible predictor, in effect it does not change the saved value at all.

Constant predictors
: One predictor for each dimension. The predicted value is equal to the value of the next neighbour in the given direction. Using only a single constant predictor is the same as the corresponding difference method.

Linear predictors
: One predictor for each dimension, including the time. Each predictor works by extrapolating the last two values in a direction linearly. Using only a single linear predictor is the same as applying the corresponding difference method twice.

Quadratic predictors
: One predictor for each dimension, including the time. Like linear predictors, but extrapolating three values quadratically. This corresponds to triple difference filtering.

Our preliminary tests showed that it is best to always use the predictor that made the best prediction for the last value seen. Note that there is no need to store the result of the predictor decision since it depends on the last value seen and the predictions that were made for it. All of this information is available to the decoder, so that it can make the same decision.

6 Towards a better compression algorithm

As noted above, we did not implement an algorithm to eliminate repetitions nor did we implement an entropy coder; we simply relied on standard compression tools for these steps. So the structure of MAFISC is to apply a sequence of filters to the data and compress it with a standard compression algorithm afterwards. Since our tests have shown that `lzma` is superior to all the other standard compression tools available, we currently use `lzma`.

All of the filters described above have their advantages and disadvantages. Most of them can be combined with good results for specific variables. Yet, there is no combination that performs best on all our test cases, not to mention the many different variables scientists store in NetCDF and HDF5 files that we know not of. This diversity calls for more flexibility than a single filter combination can provide; for best results, the method combination must be chosen to fit the data at hand.

Unfortunately, testing all sensible filter combinations in a brute force manner is far too inefficient due to the large number of possibilities. Even though this would produce the best compression, other more time efficient methods have to be used. We used two different approaches: The first is to test all the method combinations on a small subset of the data. This yields good results iff the subset is representative. This is true for many subsets, but there are exceptions: A few time slices of climate data will usually be representative for example. A few latitude slices of the same data will not.

The second approach to select a good filter combination without resorting to brute force is to work on small chunks of data at a time. Each chunk is compressed using two different combinations; the one with the better compression ratio is selected (or the uncompressed data in the case that both combinations inflate the data), and that decision is remembered as one of the methods that will be tested for the next chunk.

Obviously, this approach limits the overhead to a factor of two. Since there is usually a large number of different filter combinations that are only slightly suboptimal, one of them is quickly selected and the amount of disk space lost, compared to brute force testing, is small. This is the method we prefer. Technically, we implemented this algorithm as an HDF5 filter.

Of course, the filter chain that was used for compression needs to be stored together with the compressed data, otherwise the data would not be readable afterwards. To this end, we defined a concise string representation of the filterchains using one or two characters to describe each filter in the chain. These strings are prepended to each compressed chunk.

7 Evaluation

We used two real datasets for the final evaluation. The first contains data taken from the CMIP5 project [15] which covers the entire globe using the standard latitude/longitude coordinate system. The resolution is roughly two degrees in both directions and one day in the time dimension, and the file format is the classical `NetCDF` format. The second dataset, generated by the COSMO-CLM model code [13], contains data of only a part of the earth surface at roughly half degree resolution and a time resolution of down to three hours. Also, the grid used for the COSMO-CLM data is based on a rotated pole coordinate system, and the file format employed here is the `NetCDF-4` file format, which is actually the `HDF5` file format.

Both the fact that the COSMO-CLM data includes the day/night cycle, and the fact that it uses a rotated coordinate system imply a reduced compressability of the data. The consequence of both facts is that more change is to be expected in a direction, which is generally bad for compression.

The results for these two test cases are shown in Figure 4 and 5, respectively. The measurement used is simply the total size of the data. As expected, the CMIP5 dataset shows generally more compressability than the COSMO-CLM dataset.

The compression methods that have been tested are as follows: `sldc` is the algorithm employed by the tape drives. `zip`, `gzip`, `bzip2` and `lzma` are the standart compression tools applied to the entire file. `szip` is a compression that can be compiled into `HDF5` by non-commercial scientific users, so that it can be used transparently. In both test cases we used the szip parameter set that yielded the best compression (blocksize = 8, coding method = NN). It should be noted, that the EC coding method of szip delivered marginal compression of our data at best. `HDF5` also support transparent `gzip` compression, yet since compression is generally better if it is applied to the entire file, we used the later method for our tests. `mafisc` is our new algorithm, applied either via a custom `HDF5` filter (for the CCLM dataset) or by applying the filters to the data stored in `NetCDF-2` files and compressing it with lzma afterwards (in the case of cmip5 data).

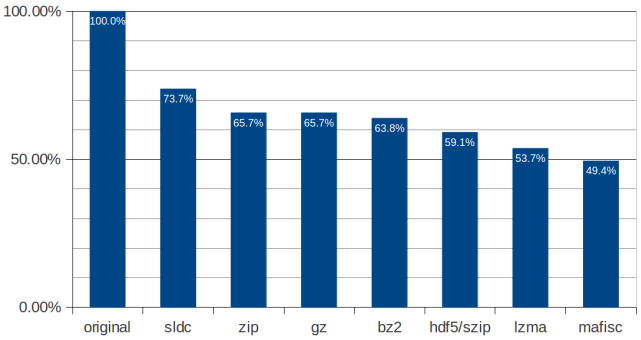

Fig. 4 Compressed sizes of the COSMO-CLM dataset using different algorithms.

Comparing the results, it is clear that the general picture is the same. `sldc`, the compression used in tape drives, performs worst in both test cases due to its lack of an entropy coder. `zip`, `gzip` and `bzip2` all perform roughly on the same level, with `bzip2` performing only 1.5% to 1.8% better than the other two. `lzma` significantly outperforms all the other standard algorithms by at least 10% due to its probabilistic model and highly effective match finders. Since MAFISC employs `lzma` as

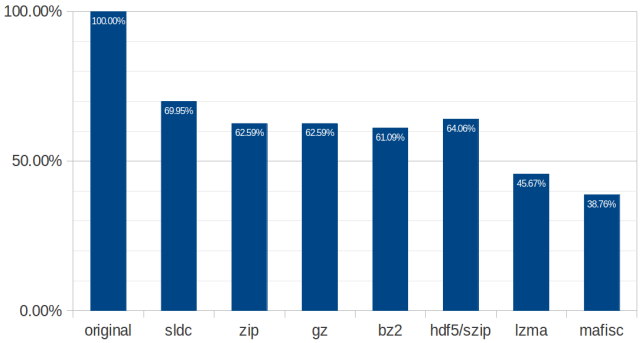

Fig. 5 Compressed sizes of the CMIP5 dataset using different algorithms.

the last step, to which it can fall back, it is expected to be at least as good as plain `lzma`. The tests, however, show that it saves another 4.2% to 6.9% of the original data size compared to plain `lzma`.

The only algorithm, that falls out of this general picture is `szip` which performs better on the CCLM dataset than on the cmip5 dataset, the reason for this behavior remains unclear. There could be a connection to the fact that we had to convert this dataset to the `NetCDF-4` format first before `szip` could be applied. However, it is unlikely that this is the only reason, since the bulk of the data is still stored as multidimensional arrays of single precision float values. Anyhow, the main advantage of using szip compression is its good compromize between compression ratio and compression speed.

If the best evaluated standard algorithm is taken as the basis for the calculation (`lzma`), MAFISC saves 7.9% to 15.1% of the compressed file size.

8 Impact on HPC installations

With the knowledge of the space savings of the different compression algorithms on climate data, we can now extrapolate to estimate their impact on the hardware costs for real HPC installation. We used the raw tape costs as a metric since it is easily calculated and a sufficiently small unit. Consequently, the figures presented here are a lower bound of the real costs. Of course, using compression will also reduce the number of tape drives that are needed, the number of tape libraries that have to be installed, the amount of space that the system needs, the costs for service, and so on. These costs, however, are heavily quantized (like the costs of tape archives) and possibly site dependent (the amount of available space for example) and therefore left out of the equation.

In addition to this lower bound of the costs and savings, we provide some throughput test results to estimate the costs of the necessary hardware to do this

compression. We will base this calculation on the example of the DKRZ, which takes part in the climate calculations conducted by the Intergovernmental Panel on Climate Change (IPCC) [1].

Last year, the DKRZ has archived approximately five petabyte of data on tape per year. Taking the COSMO-CLM test case (the one with the smaller savings) to be representative for the entire five petabytes, we arrive at the savings given in Table 3, using $^{24\,€}/_{800\,\mathrm{GB}}$ as the current tape costs (the base line are costs of 150000 € per year for uncompressed data); Table 4 shows the results of the same calculation based on the CMIP5 test case. These figures represent what can be saved every year. This shows that the current way of handling compression, which is to use the tape drive compression, a variant of the `sldc` algorithm, is severely suboptimal. Using MAFISC, the DKRZ could save between 36490 € and 46789 € a year in comparison.

	savings			
method	space	tapes	money	rel. to sldc
sldc	1313 TB	1641	39386 €	0 €
zip	1715 TB	2144	51465 €	12079 €
gzip	1716 TB	2144	51465 €	12079 €
bzip2	1808 TB	2261	54253 €	14867 €
hdf5/szip	2046 TB	2557	61371 €	21985 €
lzma	2317 TB	2896	69501 €	30115 €
mafisc	2529 TB	3161	75876 €	36490 €

Table 3 Data compression savings of 5PB data comparable to the COSMO-CLM dataset.

	savings			
method	space	tapes	money	rel. to sldc
sldc	1502 TB	1878	45068 €	0 €
zip	1871 TB	2338	56115 €	11047 €
gzip	1871 TB	2338	56115 €	11047 €
bzip2	1945 TB	2432	58363 €	13295 €
hdf5/szip	1797 TB	2246	53903 €	8835 €
lzma	2716 TB	3396	81494 €	36426 €
mafisc	3062 TB	3827	91857 €	46789 €

Table 4 Data compression savings of 5PB data comparable to the CMIP5 dataset.

The question that remains to be answered is: How much would it cost to achieve the compression? To provide an estimate of the costs of compression, we ran a few throughput tests on a server that was purchased by the University of Hamburg in the summer of 2011. The machine has four AMD Opteron CPUs with 12 cores each, running at 1.9 GHz. It also has a rather large installation of 128 GiB RAM and was purchased for 5627.51 €.

The throughput tests were done using 48 concurrent processes (one process per core), each working on the compression of a different file. In this setup, $26.4\,^{\mathrm{MB}}/_{s}$ of compressed data were produced by MAFISC. The throughput measurement was done on the compressed side, since this gives much more stable results than on the uncompressed side. This is due to the fact that `lzma` spends most of its time searching for matches for the data it is currently compressing, and the longer the matches it finds, the fewer matches it needs to find to compress the same amount of data.

Extrapolating again, our server can produce $833\,^{\mathrm{TB}}/_{a}$ of compressed data, which amounts to 1.68 to 2.15 petabyte of uncompressed data per year. Accordingly, three such machines would be able to compress all of the data archived by the DKRZ, an investment that would pay off within a few months.

Yet it is possible to do even better: The options passed to the `lzma` library can be easily adjusted to increase the throughput on our test system up to $86.1\,^{\mathrm{MB}}/_{s} = 2.72\,^{\mathrm{PB}}/_{a}$, with the tradeoff of losing 1% of compression rate, equivalent to 1500 € per year for the DKRZ. In this case, our test server can compress all the data archived by the DKRZ single-handedly.

In short: An investment of 5600 € is enough to save an HPC installation like the DKRZ ca. 40000 € a year compared to the current way of storing the data. It would pay for itself within two months. It should also be noted that this entire calculation is based on the tape costs only, extending compression to data stored on disks would reduce the costs even further. It seems benefical that vendors of tape archives allow to plugin additional compressors into their infrastructure. Probably the servers of the tape library could provide a solid basis of compute power that could be used to run more advanced algorithms, compressing the data as transparently as the current tape drive compression does, yielding much higher savings.

9 Summary and future work

We have analysed examples of climate research data in order to develop an enhanced compression algorithm for this specific field. This new algorithm could save an additional 8% to 15% relative to the best reference algorithm. Climate data, however, is diverse, with the consequence that the algorithm we developed does not only fit climate model data, but will be able to handle any multidimensional, somewhat smooth scientific data well. We also saw big differences between the evaluated compression algorithms, the worst of which are the current state of the art in HPC installations. We analysed

the economics of employing better compression algorithms and came to the conclusion that the necessary investment is actually an order of magnitude lower than the annual savings.

A patch that transparently integrates MAFISC into the `HDF5` library is already available, but more work is required to enhance the interchangeability of the compressed data, since the current implementation requires the program reading the data to provide the decoder. Future work will also address the problem of data stored on irregular grids, like the data produced by climate models working on icosahedral grids.

Acknowledgements We want to thank Wolfgang Stahl for his measurements of the tape drive compression ratio at the DKRZ and the fruitful discussion about compression impact on HPC installations.

References

1. Alfsen, K., Skodvin, T.: The intergovernmental panel on climate change (ipcc) and scientific consensus (2009)
2. Bosi, M.: MPEG audio compression basics. In: L. Chiariglione (ed.) The MPEG Representation of Digital Media, pp. 97–123. Springer New York (2012). URL `http://dx.doi.org/10.1007/978-1-4419-6184-6_6`
3. Christopoulos, C., Skodras, A., Ebrahimi, T.: The JPEG2000 still image coding system: an overview. Consumer Electronics, IEEE Transactions on **46**(4), 1103–1127 (2000)
4. Dutta, S., Bhattacherjee, S., Narang, A.: Towards "Intelligent Compression" in Streams: A Biased Reservoir Sampling based Bloom Filter Approach. ArXiv e-prints (2011)
5. ECMA: Streaming lossless data compression algorithm - (sldc). `http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-321.pdf` (2001). ECMA Standart 321
6. Fenwick, P.: The burrows–wheeler transform for block sorting text compression: principles and improvements. The Computer Journal **39**(9), 731 (1996)
7. Furht, B.: A survey of multimedia compression techniques and standards. Part I: JPEG standard. Real-Time Imaging **1**(1), 49–67 (1995)
8. Jain, C., Chaudhary, V., Jain, K., Karsoliya, S.: Performance analysis of integer wavelet transform for image compression. In: Electronics Computer Technology (ICECT), 2011 3rd International Conference on, vol. 3, pp. 244 –246 (2011). DOI 10.1109/ICECTECH.2011.5941746
9. Koranne, S., Koranne, S.: Hierarchical data format 5: HDF5. In: Handbook of Open Source Tools, pp. 191–200. Springer US (2011). URL `http://dx.doi.org/10.1007/978-1-4419-7719-9_10`
10. Lakshminarasimhan, S., Shah, N., Ethier, S., Klasky, S., Latham, R., Ross, R., Samatova, N.: Compressing the incompressible with ISABELA: in-situ reduction of spatio-temporal data. Euro-Par 2011 Parallel Processing pp. 366–379 (2011)
11. Latham, R.: The parallel-netCDF I/O library. `http://www.mcs.anl.gov/uploads/cels/papers/P1713A.pdf` (2010)
12. Nagaraj, N., Vaidya, P., Bhat, K.: Arithmetic coding as a non-linear dynamical system. Communications in Nonlinear Science and Numerical Simulation **14**(4), 1013–1020 (2009)
13. Rockel, B., Will, A., Hense, A.: The regional climate model COSMO-CLM (CCLM). Meteorologische zeitschrift **17**(4), 347–348 (2008)
14. Szalay, A.: Extreme data-intensive scientific computing. Computing in Science Engineering **13**(6), 34 –41 (2011). DOI 10.1109/MCSE.2011.74
15. Taylor, K.E., Stouffer, R.J., Meehl, G.A.: A summary of the CMIP5 experiment design. World **4**(January 2011), 1–33 (2007)
16. Woodring, J., Mniszewski, S., Brislawn, C., DeMarle, D., Ahrens, J.: Revisiting wavelet compression for large-scale climate data using JPEG2000 and ensuring data precision. In: Large Data Analysis and Visualization (LDAV), 2011 IEEE Symposium on, pp. 31 –38 (2011). DOI 10.1109/LDAV.2011.6092314