Technical Memo

ECMWF
European Centre for Medium-Range Weather Forecasts

900

Impact of GRIB compression on weather forecast data and data-handling applications.

Eugen Betke, Tiago Quintino, Simon Smart, Tomas Wilhelmsson

August 2022

Series: ECMWF Technical Memoranda

A full list of ECMWF Publications can be found on our web site under:
http://www.ecmwf.int/en/publications/

Contact: library@ecmwf.int

European Centre for Medium Range Weather Forecasts, Shinfield Park, Reading, RG2 9AX, UK

Contents

Abstract

As part of its scientific ambitions, ECMWF plans to increase the resolution of its ensemble forecasts from 18 to 9 kilometres for IFS cycle Cy48r1. This single upgrade will be responsible for a four times increase in the volume of data produced (assuming all other factors constant), from roughly 30 TiB per cycle to 120 TiB. In aggregate, ECMWF's forecasts will produce 480 TiB per day.

Data reduction strategies provide one approach to mitigating the impact of this expected growth. The bulk of ECMWF's forecast output data is generated in GRIB format. As such, this report re-examines the GRIB packing methods available and investigates the use of compression algorithms to reduce the volume of data produced.

This report focuses on the `grid_ccsds`[1] packing type supported in the WMO GRIB standard and `grid_second_order` developed by ECMWF, both implemented by the ecCodes library. We present estimates for the impact of these compression methods on current and future data volumes. We also look at the first-order impact on other components of the operational system, including the IFS, MARS and the FDB.

We demonstrate that the packing type `grid_ccsds` is particularly suitable for forecast data. Adopting this packing type for data that is already encoded in GRIB2 format would result in a 32% reduction in the volume of data produced. Further, a reduction of 42% is possible given the completion of the progressive migration from GRIB1 to GRIB2.

Although it appears as a one-shot solution, we also conclude that `grid_ccsds` would be a good investment, as the method also scales well and further improves its compression efficiency with future resolution increases.

Plain language summary

ECMWF plans to double the resolution of the weather forecasts in the near future to be able to make them more accurate. However, after the change, the forecast data will increase four times in size. Since this data is to be kept on a long-term storage for decades, preferably forever, it would of course be good to reduce the amount of data as much as possible.

The GRIB data format, in which the data is stored, provides options for data reduction. It exists in two versions, GRIB1 and GRIB2. Both offer a number of packing types that have the ability to compress data. Currently, both GRIB versions are in use at ECMWF, but in a parallel project ECMWF is working on the migration to the new data format.

In this report we show that not every packing type is equally suitable for our data. We've picked out the best packing type candidates (`grid_ccsds` and `grid_second_order`) and examined them closely for how effectively they compress data, how fast they encode and decode data, and what impact compression has on our applications. We have shown that `grid_ccsds` is particularly suitable for our data.

Unfortunately `grid_ccsds` in only supported by the new data format. If we apply the compression now, before the migration is finished, we will reduce the generated forecast data by 32%. In the future, when the migration is complete then it will be possible to decrease the generated data volume by 42% .

We also demonstrate that using the `grid_ccsds` packing type is scalable — that is that the computational cost increases roughly linearly with the data volume and that the compression actually becomes more effective as the resolution increases.

[1] Consultative Committee for Space Data Systems Adaptive Entropy Coding (AEC) based on Golomb-Rice algorithm

1 Introduction

ECMWF produces two-week weather forecasts four times daily, complemented with additional forecasts for monthly and seasonal timescales. Operational forecast data is currently produced at the rate of 120 TiB per day, in short, time-critical bursts of one hour, supplemented by data produced in research.

ECMWF's data archive (MARS[2]) contains more than 30 years of historical forecast data and has grown to over 330 PiB. It currently grows at an average rate of about 250 TiB per day. The growth rate itself grows exponentially as a result of increases in resolution of the forecast, the addition of physical parameters and the addition of vertical levels. Over the last 30 years, the MARS archive has experienced an average compound growth rate of approximately 40% on year. With the upcoming ensemble horizontal resolution change from 18 to 9 kilometres, we expect to produce four times (an increase of 300%) the forecast data volume.

The IFS model is the largest data producer in ECMWF's production workflow (Figure 1). It produces raw model output in the form of global fields (forecasting data) and stores them into the indexed object store (the FDB) on the parallel file system (Lustre). Within minutes, and whilst the model is still running, 70% of the data are read by product generation (PGen) to produce user-specific forecast products, currently outputting 45 TiB per day. The data remain in the FDB for only a few days due to limited storage capacity and are migrated to the tape archive, where they are kept in long-term storage. The output of PGen is stored temporarily on the parallel file system for a short time until it is disseminated to the end customer by ECPDS[3].

Reducing the data size has obvious positive impacts on storage requirements (and thus cost). Alternatively, it allows us to be more scientifically ambitious with the same storage resources. Further, the technology to do this is already supported in the WMO GRIB format and is already implemented within ecCodes as alternative GRIB data packing types.

Using this approach implies technical compromises. Compression comes with additional computational demands, shifting the resource burden somewhat from storage to compute. As the computational landscape is moving towards lower cost-per-FLOP, by using multi-core high compute density CPUs, modern HPC facilities are likely to have spare compute cycles but restricted I/O throughput. This makes this compute vs storage trade-off attractive.

At ECMWF, field data handling uses the ecCodes library to encode and decode the GRIB message format. GRIB supports several packing types. Some are tailored to specific data (e.g., `spectral_complex`), while others focus on gridded data (e.g., `grid_simple`). A previous study [Qui12] concluded that spectral complex packing is already sufficiently compressed such that further compression of this data is out of the scope of this study. Although spectral complex packing type is a good compressor for data with certain characteristics, compression and decompression requires a high compute intensity that scales quadratically with resolution, making it not affordable for fields that were not originally in spectral form inside the IFS model.

[2] the Meteorological Archival and Retrieval System
[3] ECMWF Production Data Store

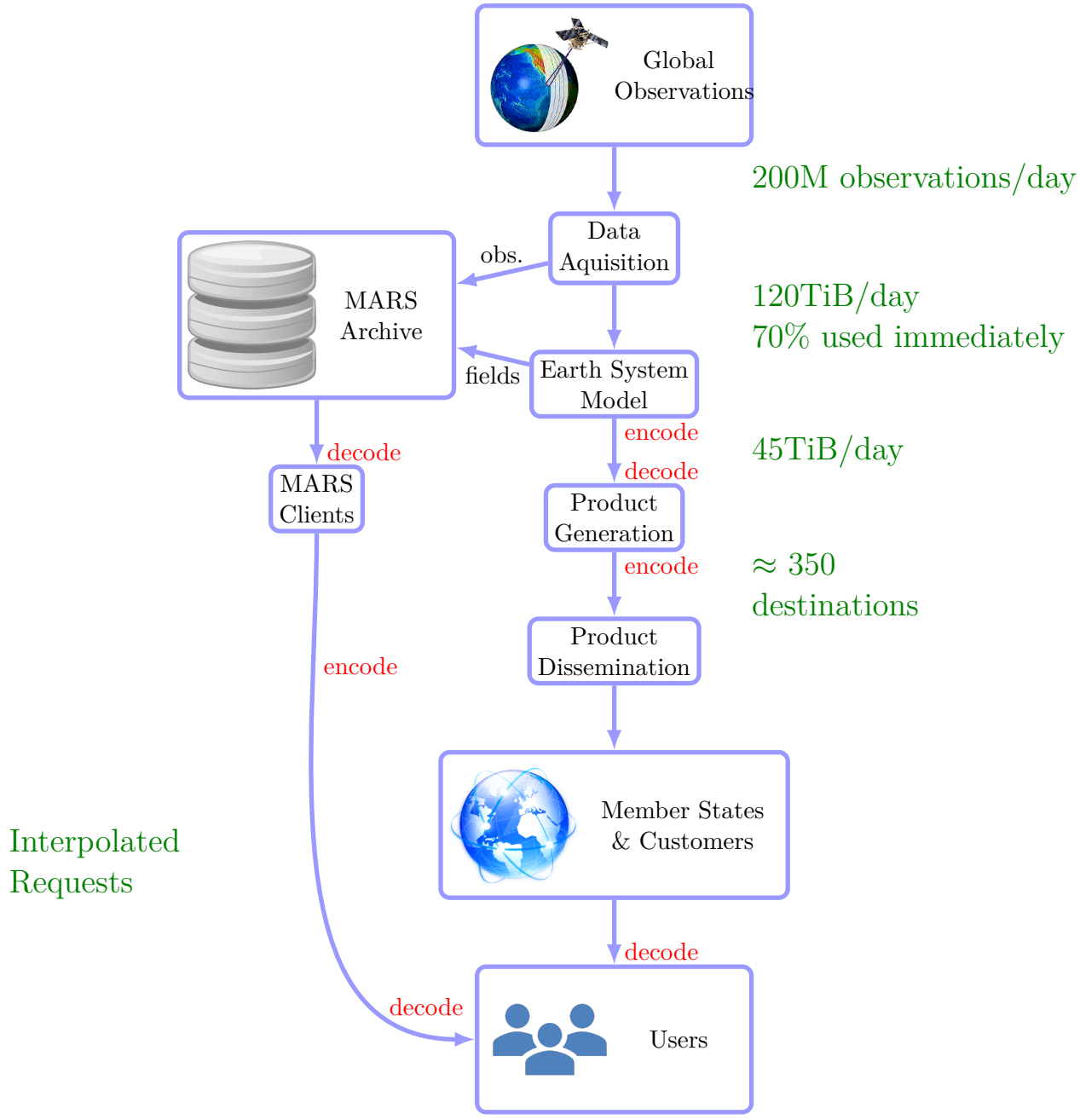


Figure 1: ECMWF's production workflow. Encoding and decoding occurs all the time.

1.1 Requirements

Data packing algorithms must meet strict requirements for use in ECMWF's production workflows. These requirements derive from multiple aspects of our complex workflows and the use of our data by our Users and the Member States. The most important requirements to take into account when comparing different data packing methods are:

- **WMO GRIB Standard:** data packing methods must be recognised in the WMO GRIB standard, as this is the data interchange format recognised and maintained by the community. This is paramount because data distribution is part of our core mission.
- **Reliability:** The algorithm must be stable, and must consistently and efficiently handle *all* of the data that ECMWF produces. We must also ensure that the algorithm's implementation will be maintained into the future, to prevent historical data from becoming orphaned and inaccessible.
- **Speed and efficiency:** Compression and decompression of the data must be computationally efficient and operable in our workflows in a timely manner relative to the current data flows. Decompression speed is slightly more important than compression, as most users will decompress data in serial on smaller systems, whereas our workflows run on massively parallel computational infrastructures.
- **Scalability:** The compression algorithm has to scale well to higher resolutions, as ECMWF's scientific ambition will drive the production of ever-higher resolutions and overall data volumes.
- **Granularity:** Data must be compressed field-by-field rather than as a whole. This allows direct access to individual fields without requiring the whole data set to be decoded. This requirement is based on the historical usage patterns of MARS and the FDB; most users select a minority of fields particular to their needs.

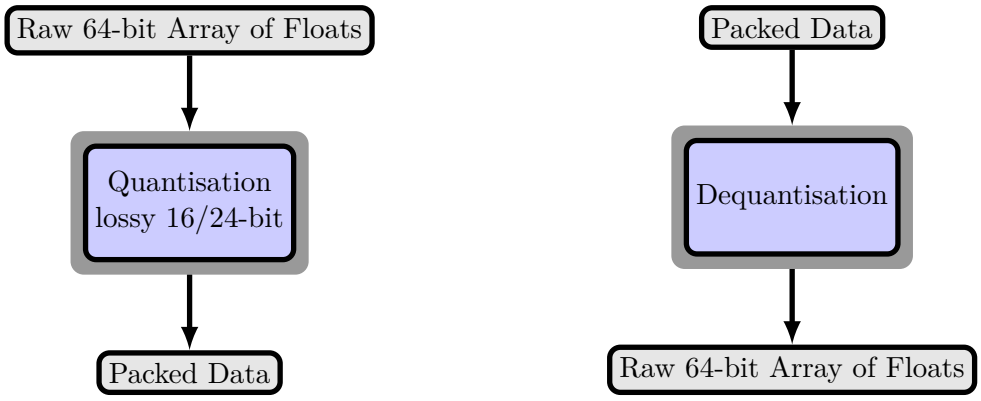
1.2 Encoding and decoding

Data encoding in GRIB implies packaging floating-point arrays into a specific data format, whilst decoding is the reverse process. The processes behind each packing method are transparent to the users, implying that users rely on the packing method specifications to understand its effects on the data, such as precision loss.

In Figures 2 and 3, we visualize the processes of packing types considered in this work. In the simple case (Figure 2), data is compressed by reducing the precision of the values. In the more complex case (Figure 3), compression is done by lossy compression in the first stage and lossless compression in the second stage. The additional lossless compression has no scientific impact because the decompressed data are bit-identical to the original data.

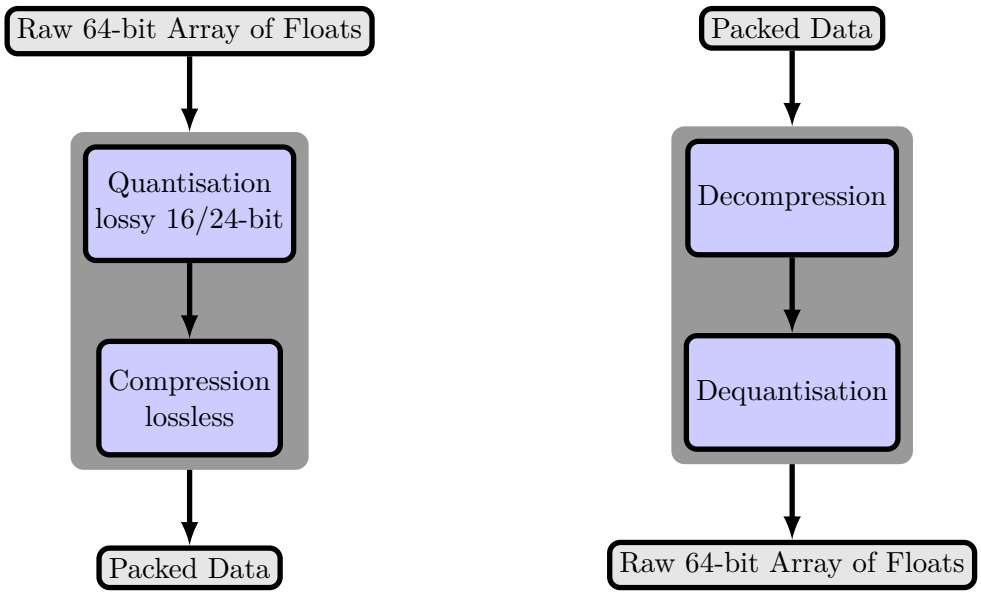
1.3 GRIB simple packing

In GRIB simple packing, data is coded using the minimum number of bits necessary to ensure a predefined accuracy, usually agreed with the users receiving the data. This required accuracy/precision is achieved by scaling the data by multiplication by an appropriate power of 10



(a) Encode *(b) Decode*

Figure 2: Internal structure of packing types that reduce data volume by lossy quantisation (e.g., `grid_simple`). Quantisation projects data from 64-bit floating points to 16-bit or 24-bit integers.



(a) Encode *(b) Decode*

Figure 3: Internal structure of packing types that reduce data volume by lossy quantisation and lossless compression (e.g., `grid_ccsds`, `grid_second_order`). Lossless compression has no scientific impact, because it implies that decompressed data are bit-identical to the original data.

Y	$A = Y \cdot 10^D$	$B = A - R$		$X = B/2^E$	$B' = X \cdot 2^E$	$A' = B' + R$	$Y' = A'/10^D$
		Decimal	Binary				
1.004	10040	0	0000.000 0.000	0.0000	0000	0	10040	1.004
1.038	10380	340	0000.000 1.010	1.0100	1010	320	10360	1.036
1.053	10530	490	0000.000 1.111	0.1010	1111	480	10520	1.052

Table 1: Example: Compression of an array containing three elements $Y = [1.004, 1038, 1053]$ using the parameters $D = 4, E = 5, R = 10040$. For simplicity we assume that the operations are executed on a 16-bit machine (i.e., integers are 16-bit long). D was chosen so that it is maximally large and A doesn't not exceed 16 bits. $R = min(A)$, it follows that B contains non-negative values only. X includes the four highest non-zero positions (marked with gray color). It is done by two operations: by shifting bits using an appropriate value for E and then by removing the first 12 bits.

(which may be 0) before forming the non-negative differences and then using the binary scaling to select the precision of the transmitted value.

Data is coded as non-negative scaled increments from a reference value. The reference value is normally the minimum value of the represented data set. The actual value Y is linked to the coded value X, the reference value R, the binary scale factor E and the decimal scale factor D using the following formula:

$$Y \cdot 10^D = R + X \cdot 2^E \tag{1}$$

In GRIB, this packing type is called `grid_simple`. Depending on user settings, it converts the 64-bit floating-point data into 16-bit or 24-bit data.

The packing type is applied to most of the ECMWF forecast data. One of its features is that it can limit the values to a certain number of bits and thus reduce the amount of data at the expense of precision, but it does not apply compression to the data afterwards. For example, this lossy packing type can convert double values (64-bit) to short integers (16-bit), resulting in a data size that is one-fourth of the original data. The advantage of such a simple compression technique is a high compression speed and, given fixed-length data encoding, the ability to access small data regions without unpacking data by just using an offset. The disadvantage is its relatively low compression ratio. This can be improved by further compressing the data with a compression algorithm. This is what packing types `grid_ccsds` and `grid_second_order` do. Below we include a short description of what these packing types implement.

1.4 GRIB second-order packing

The lossless compression algorithm does not belong to any official standard. It was developed and has been maintained since 1991 by ECMWF.

This algorithm works with integer arrays, therefore the incoming 64-bit floating-point values must first be converted into 16-bit or 24-bit integer values. In GRIB, this happens by a similar procedure used for grid simple packing in Section 1.3. Once that first quantization takes place, the second-order packing splits the integer array into several blocks and calculates a reference

value (the first-order value) for each block. The second-order values are positive derivatives of the reference value. For typical forecast data, they are usually small and can be packed with only a few bits per value, with the algorithm computing the required number of bits to pack the data efficiently.

This packing type is available in the GRIB standard for gridded data in GRIB1 as in GRIB2 under the name `grid_second_order`.

1.5 GRIB CCSDS Adaptive Entropy Encoding

WMO GRIB supports a packing type that implements the extended Golomb-Rice coding as defined by the CCSDS recommended standard 121.0-B-3 [Spa20]. This packing type is limited to gridded data in GRIB2 messages. This is an encoding used extensively by DKRZ, who is also the maintainer of the widely available open-sourced library libAEC[4] that implements the algorithm.

This is a ubiquitous algorithm for encoding scientific data, already used within the HDF5 format for multiple years and therefore available for the NetCDF-4 format.

In GRIB, this is identified as `grid_ccsds` which involves a first pass on the data, using the quantisation method of the grid simple packing in Section 1.3, encoding data from floating-point to integer space followed by a compression step using the extended Golomb-Rice coding. As with `grid_simple`, the transformation to integer space (16-bit or 24-bit) is lossy, but the subsequent compression step is lossless and fully reversible.

1.6 Overview of report

In this report, we aim to demonstrate the impact of each packing type on storage systems and users' experience with a series of experiments. In Section 2, we perform a first round of analysis of the multiple packing types supported by ecCodes. The most promising packing types, `grid_ccsds` and `grid_second_order`, are further explored in Sections 3 and 4. After such experiments, it will be shown that the `grid_ccsds` is an outstanding candidate to compress ECMWF's forecasting data further. In Section 5, we further test this packing type for scalability, and finally, we conduct an impact study on application run-times in Section 6.

[4] gitlab.dkrz.de/k202009/libaec

```
retrieve,
  time=0000,
  class=od,
  date=20210520,
  levtype=pl,
  stream=oper,
  domain=g,
  expver=0001,
  param=60/133/133/246/247/248,
  step=240,
  type=fc,
  levelist=1000,
  target="data/data.grib"
```

Listing 1: oper stream

```
retrieve,
  time=0000,
  class=od,
  number=10,
  date=20210520,
  levtype=pl,
  stream=enfo,
  domain=g,
  expver=0001,
  param=60/133/203,
  step=306,
  type=pf,
  levelist=500,
  target="data/data.grib"
```

Listing 2: enfo stream

Figure 4: Mars requests used to download the test data.

2 Initial study of available packing types

The WMO GRIB standard supports many different packing types. In addition, ecCodes brings other algorithms with it. Many of these options can be eliminated by a first analysis of the data and their behaviour under compression.

For this comparison we are interested in:

1. Can our data be successfully packed with a packing type?
2. How efficiently does the packing type compress the data in comparison to `grid_simple`?
3. How fast can the data be encoded or decoded?

Our tests took sample GRIB data, output from the operational forecast, and re-packed this data into a range of packing types supported by ecCodes. The resulting data sizes and timing of the encoding phase were recorded and followed by a test of the performance of decoding the resultant data. [5] [6] We then determined the compression ratio and performance across different physical parameters and compared varios packing types supported by ecCodes.

2.1 Data

The data for the following analysis was retrieved by the MARS requests shown in Listings 1 and 2. It contains a number of parameters that often appear in the production data. Additionally, these parameters are packed with `grid_simple` and are organised in octahedral reduced gaussian grids. These are the characteristics of most of our data.

A characteristic property of octahedral reduced gaussian grids is that latitude lines close to the poles are shorter than close to the equator, growing linearly from the poles to the equator. If we visualize that strictly from the perspective of the grid structure, we get a representation that resembles a diamond (Figures 5a and 6a). Image compression algorithms like PNG and JPEG are

[5] The compression benchmark is available at `ssh://git@git.ecmwf.int/~maee/compression-benchmark.git`

[6] These tests were conducted on a workstation equipped with an Intel®Core™i7-9700 CPU running at 3.00 GHz and 16 GiB DDR4 RAM. Tests used ecCodes built with standard (release) compiler optimisation options for the Linux platform

ParameterID	ShortName	Name	Shannon Entropy		Ideal Compressibility	
			enfo	oper	enfo	oper
60	pv	Potential vorticity	6.16	4.80	2.60	3.33
133	q	Specific humidity	7.34	7.82	2.18	2.05
203	o3	Ozone mass mixing ratio	7.76	7.83	2.06	2.04
246	clwc	Specific cloud liquid water content	-	0.54	-	29.65
247	ciwc	Specific cloud ice water content	-	0.27	-	58.89
248	cc	Fraction of cloud cover	-	0.18	-	90.61

Table 2: The entropy of the 16-bit integer arrays of data for several parameters. The entropy shows the number of bits of information carried per value in the array, and can be used to estimate the compressibility (for 16-bit integers, compressibility = 16 / Entropy).

not adapted to the diamond shape and therefore can not benefit from their 2d compression capabilities. Instead, as with all other algorithms, the data must be processed as a one-dimensional data series, and these algorithms will not be able to compress the data as efficiently as perhaps expected.

Shannon entropy[7] is a quantitative metric that tells us how much information content is in the data and thus indirectly indicates compressibility. Encoding high entropy data requires more bits per value than encoding low entropy data, so the compressibility is low. The reverse is also true. In our case, it makes the most sense to calculate the entropy of the quantised, 16-bit data, which will be passed directly to the compression algorithm (see Section 1).

According to the results in Table 2, the parameters can be divided into two groups. In the high entropy group are pv, q and o3 and in the low entropy group are cc, ciwc and clwc. From the entropy, we can also calculate the compression ratio that could be achieved if data was encoded with the theoretical minimum number of bits. The maximum compression ratio for the high entropy group would be between 2.04 and 3.33. In the low entropy group, it would be much higher, ranging from 29.65 to 90.61.

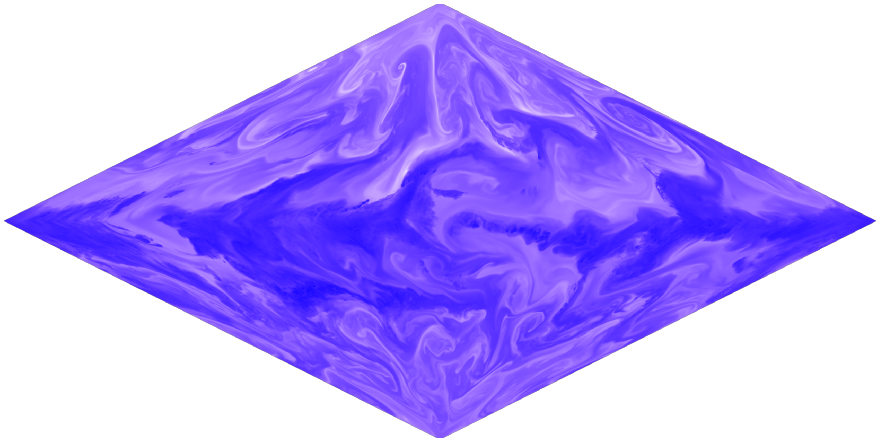
Visualising the density and variability of the data values for each parameter can help give an insight into the data. Figures 5 and 6 show a visualisation of the distribution of the values within two representative fields and how they are distributed in the first 5000 values in the file. In the high-entropy group, we can see that the values for parameter q (specific humidity) are distributed fairly uniformly and randomly across the data indexes. This makes the work harder for the compression algorithm, as it tries to find patterns and sequences in the data. By contrast, in the low-entropy group, parameter cc (cloud cover) has a smaller range of possible values, and these are sparsely distributed. The large regions of null values can be very easily compressed.

2.2 Compression performance

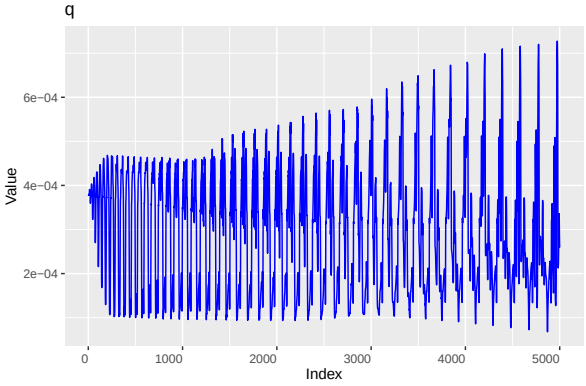
Figure 7 gives an overview of the behaviour of different packing types for data coming from the high-resolution (oper) or ensemble (enfo) datasets. Packing types `bifourier_complex` and `grid_png` were not able to encode and decode all the data without failing with run-time errors, and have been excluded.

The `grid_complex_spatial_differencing`, `grid_simple_log_processing` and `grid_complex`

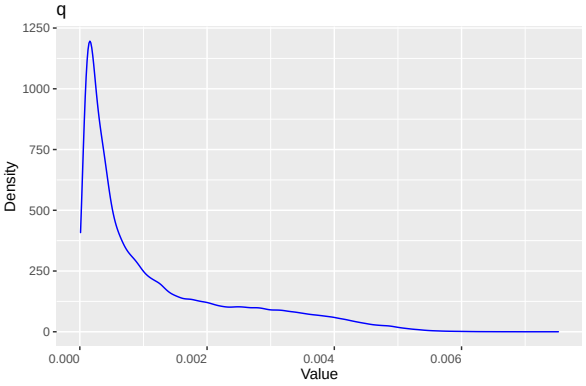
[7] $H(X) = -\sum_{x \in X} p(x) \log(p(x)) = \mathbb{E}[-\log(p(X))]$



(a) Data is organized in lines in octahedral grids. The lines are shorter and contain fewer points on poles (on top and on bottom) than on the equator (centre). Most area in this data is covered by non-zero values (non-white areas).

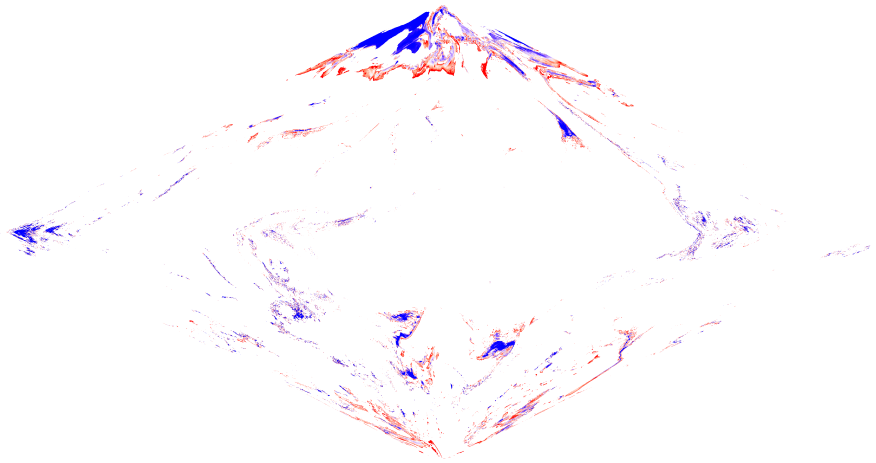


(b) The first 5000 values as ordered in memory. Each oscillation represents a latitude line in an octahedral grid. The random-like data may be difficult to compress.

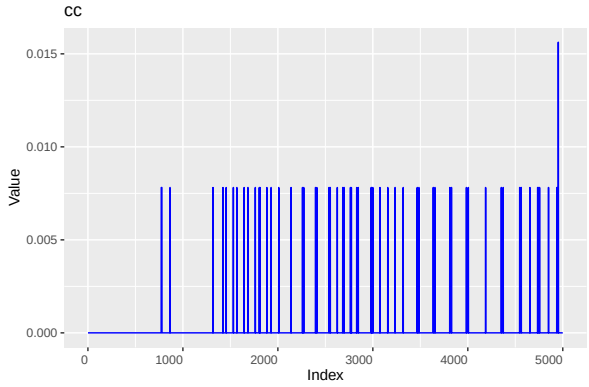


(c) Density. The values are distributed over a large range of values. This can be an additional hurdle for compression.

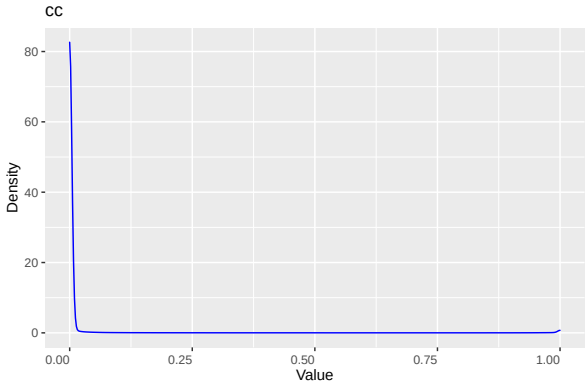
Figure 5: Example of dense octahedral data using the specific humidity (q) parameter. These data usually do not reach a very high compression ratio. Similar parameters are pv/o3.



(a) Data is organized in lines in octahedral grids. The lines are shorter and contain fewer points on poles (on top and on bottom) than on the equator (centre). The white area is 47.7% and is represented by the value 0.



(b) The first 5000 values as ordered in memory. There are many zeros between values. It can have a positive effect on the compression ratio. Many algorithms can use this to achieve better compression rates.



(c) Density. The values are distributed over a small range of values so that fewer different values have to be compressed.

Figure 6: Example of sparse octahedral data using the fraction of cloud cover (cc) parameter. Such data usually has a very high compression ratio. Similar parameters are clwc/ciwc.

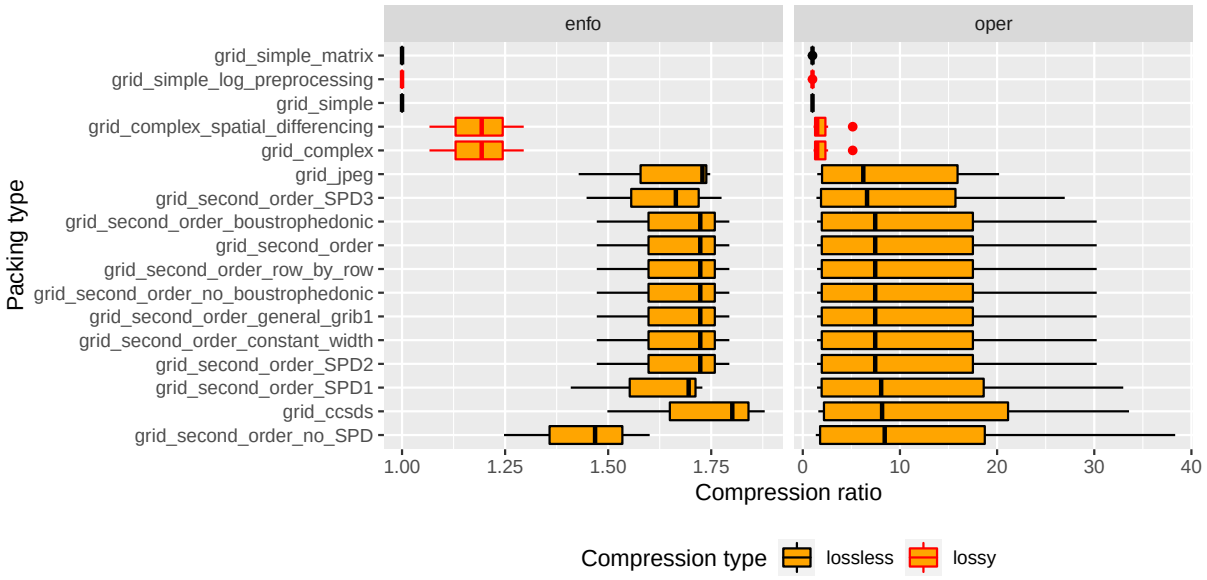


Figure 7: Compression ratios. ***grid_ccsds*** *and* ***grid_second_order*** *packing type family show the best compression ratios.*

packing types demonstrate lossy compression and cannot encode and decode the sample data without further loss of precision beyond the quantisation common to all these packing types. As such, they will not be considered further in this report.

There are several natural groups of packing types with similar performance and functionality. We will only consider one packing type from each group as representative of the whole. `grid_second_order` will represent all `grid_second_order*` and `grid_simple` all `grid_simple*` packing types. `grid_ccsds` and `grid_jpeg` are considered individually.

Figure 8 clearly shows the difference in compressibility of various parameters with different entropy. The compression ratios reported in this figure are relative to `grid_simple` and, as such, show the improvement that can be obtained with a change of packing type. The different algorithms achieve different compression ratios relative to the source data in `grid_simple` and are quite far from the theoretical compression ratios in Table 2. This is expected as real compression algorithms use heuristics and algorithms to compress data rather than a theoretical investigation of the information content.

The `grid_jpeg` packing type does not compress sparse data (cc, ciwv and clwc) as well as its competitors. This is likely due to the algorithm being optimised for two-dimensional data but not being able to use the octahedral grid's two-dimensional nature.

The `grid_ccsds` and `grid_second_order` packing types have very similar compression performance, with `grid_ccsds` having only a slight edge. As such, we will consider these two packing types going forward.

Table 3 shows the compression and decompression speeds that can be achieved with the two packing types. The first thing that stands out is that the `grid_jpeg` packing type is extraordinarily slow on compression and decompression. As this compression type also achieves a worse compression ratio, it will no longer be considered.

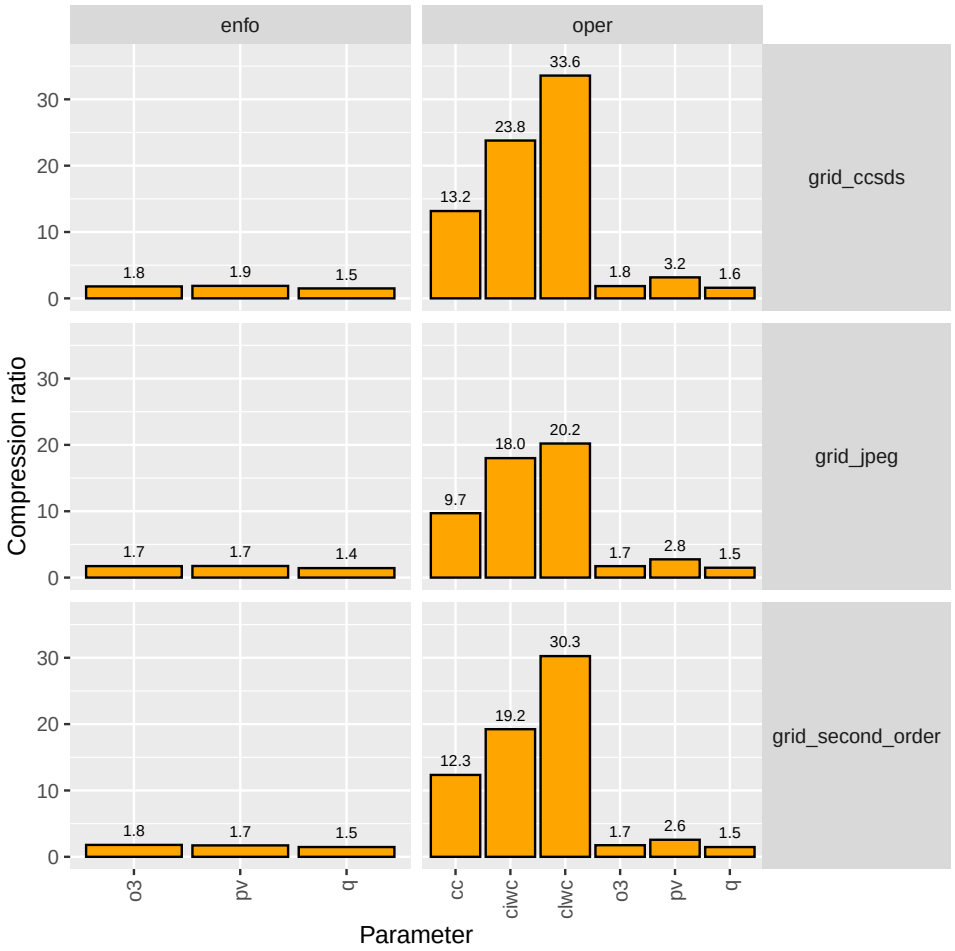


Figure 8: Compression ratio of individual parameters relative to `grid_simple`.

Packing type	Compression MiB/s	Decompression MiB/s
`grid_simple`	1683.06	1791.22
`grid_ccsds`	339.12	452.12
`grid_second_order`	81.97	622.63
`grid_jpeg`	14.18	42.99

Table 3: Compression and decompression speeds for different packing types refer to the unpacked data, i.e., the calculation is performed with the formula: speed $= \frac{uncompressed\ size}{duration}$.

compr_packing_type	mean_compr_perf_mib	mean_decompr_perf_mib

Table 4: Mean de-/compression speeds.

Both of the two remaining compression algorithms are significantly slower than `grid_simple`. This is no surprise, as they are doing substantially more work! Later in the report, we will demonstrate that the computational cost incurred is manageable for either of those algorithms. `grid_ccsds` has well-balanced compression and decompression speeds, with compression only a little slower than decompression. `grid_second_order` is heavily biased in favour of decompression speed, which is 7-8 times faster.

2.3 Conclusion

Many packing types can be immediately excluded as they are either lossy or cannot handle all of the data without errors. Of the remaining packing types, the `grid_second_order` family and `grid_ccsds` remain as good candidates to replace the `grid_simple` algorithm. These packing types will be considered in the remainder of the report.

3 Savings in data volumes

An evaluation of the data savings possible with different packing types needs to answer three central questions:

1. Which packing type is most effective at compressing the data?
2. What is the maximum data volume saving available under ideal conditions?
3. What data volume saving can be expected under real conditions?

As different parameters compress to different degrees, it is necessary to consider a larger data set of all appropriate parameters to estimate the overall impact of compression algorithms on the forecast process. Here we consider a data set containing all data produced over a full day's NWP operations[8], as stored in the FDB[9].

A look inside the data (Figure 9) shows that five of the streams are pure GRIB1 streams, and five are mixed. Combining this view with Figure 9a makes it clear that the vast bulk of the data belongs to these mixed streams (specifically enfo, elda, oper and scda in that order). The plot highlights that future work would do well to look at the waef stream. Note that data encoded with the `spectral_complex` is not considered, as it is already well compressed.

Two approaches are considered. In Experiment A, all data which is encoded with `grid_simple` in GRIB2 format is considered. This data is taken from the FDB and re-encoded using either the `grid_ccsds` or `grid_second_order` packing types. The resulting data set and its data volumes are then compared to original forecast data volume. This experiment gives an insight into the benefit that can be obtained simply by turning compression on in the operational pipeline.

In the second case, Experiment B, data which is originally stored in GRIB1 format is also considered. This data cannot be natively encoded using both the packing types being considered, as the GRIB1 standard does not support them, but most of this data can be first converted to GRIB2 and then encoded. This experiment gives an insight into the benefit that can be achieved in the medium term, as ECMWF completes its migration from GRIB1 to GRIB2. This data set is not perfect as some data cannot (yet) be converted to GRIB1, so it provides a worst-case estimate of what can be achieved.

3.1 Evaluation

To consider which packing type is most effective, we compress the data set using both `grid_ccsds` and `grid_second_order`. The results can be seen in Figure 10a. There is very little overall difference between the two packing types, with `grid_ccsds` having 0.13% advantage. Compressing only the GRIB2 data (50% of the source data) results in approximately a 32% saving in total data volume.

To calculate the potentially available volume saving, we convert all possible GRIB1 data into GRIB2 format before encoding. The interpolation shows that we can achieve a 41% overall data saving. However, compression fails for 15% of the overall data, as some parameters are not yet supported. Assuming that these issues will be fixed as part of data migration to GRIB2, we extrapolate that a volume saving of 51% could be achieved.

[8] 30 November 2021

[9] Operational data files located in the FDB, according to the regular expression `".*od:.*:.*:$date.*.data"`

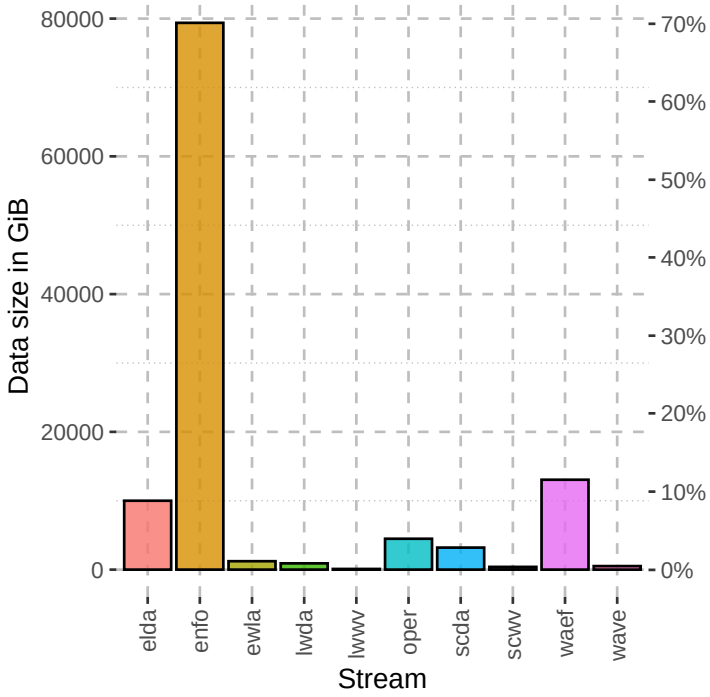
Looking forward, we wish to understand how the forecast will behave with the next resolution of data. Figure 10a presents data from the O640 grid. We can estimate the impact of compression on data from an O1280 grid by interpolating the data onto the higher resolution grid and then making the same experimental comparison. With the caveat that this is not true O1280 data, Figure 10b shows that the savings available will increase to 35% and 46% respectively.

Figure 11 shows a breakdown of the data savings by level type (levtype). It is notable that we can compress model-level (ml) data significantly better than other level types. The ml data is also all already in GRIB2 form, so it does not need to be converted from GRIB1 to demonstrate what can be achieved. It is also notable that all of the GRIB1 data that could not be safely converted to GRIB2 and compressed were surface fields (levtype sfc).

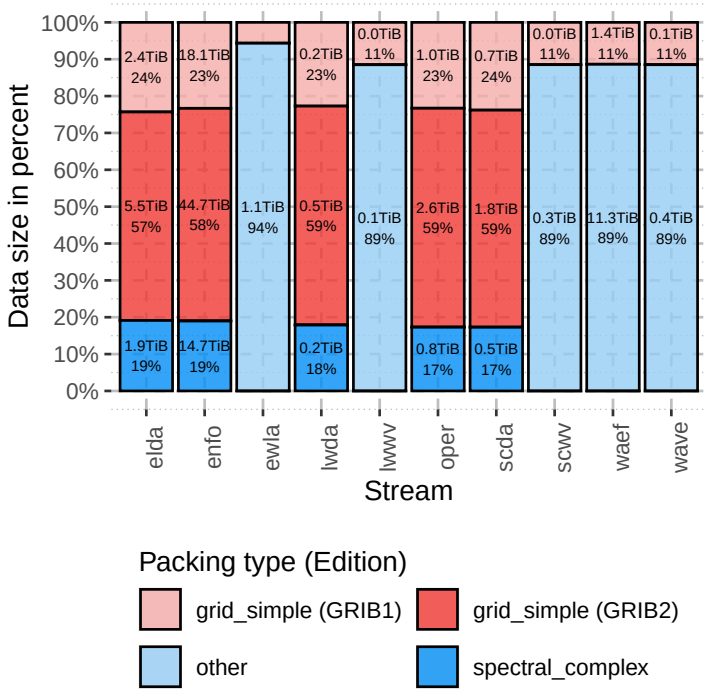
3.2 Conclusion

The two packing types, `grid_ccsds` and `grid_second_order`, are almost indistinguishable in compression effectiveness for ECMWF's operational data set. Turning on compression could offer a 32% immediate benefit, with at least 42% available as the data migration to GRIB2 is completed.

The data savings are scalable as the resolution increases, with compression performance increasing to 35% and 46%, respectively, at the O1280 resolution.

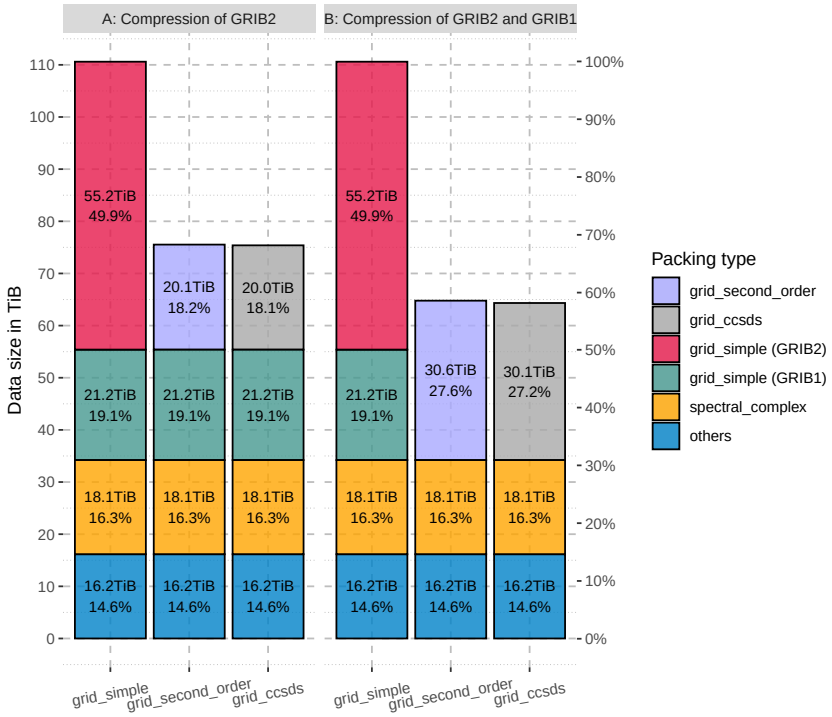


(a) Total operational data volumes per stream. The total data set amounts to 111 TiB, comprised of 190,603,787 messages.

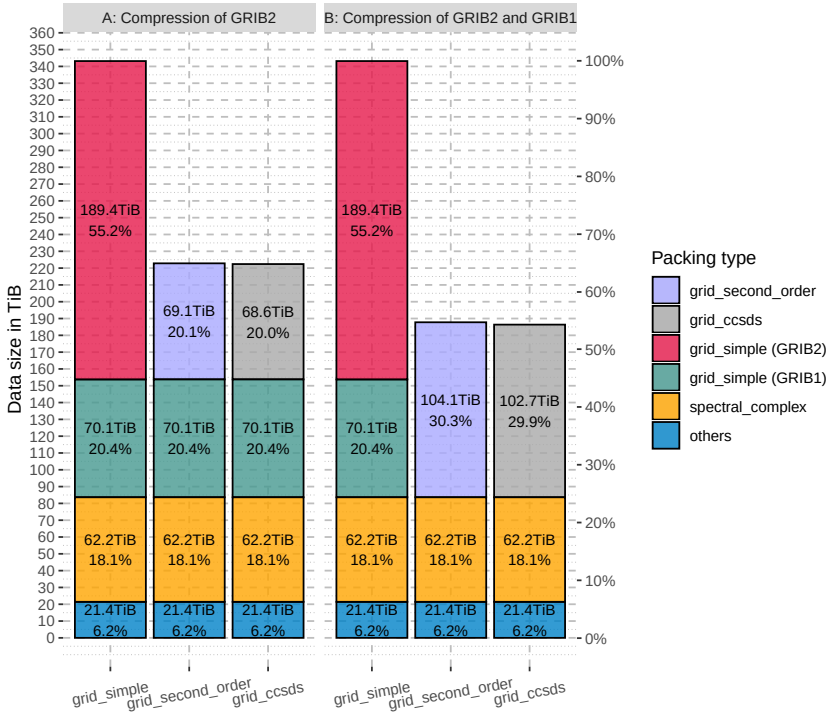


(b) The distribution of the operational data set by packing type, in each stream. Overall `spectral_complex` *accounts for 16% and* `grid_simple` *(GRIB1+2) for 69%.*

Figure 9: Composition of operationally produced data from November 30, 2021.



(a) Data volume savings achieved with O640 resolution data. Overall savings of 32% and 42% in experiment A and B, respectively.



(b) Data volume savings achieved with data interpolated to O1280 resolution data. Overall savings of 35% and 46% in experiment A and B, respectively.

Figure 10: Data volume savings achieved when compressing `grid_simple` messages.

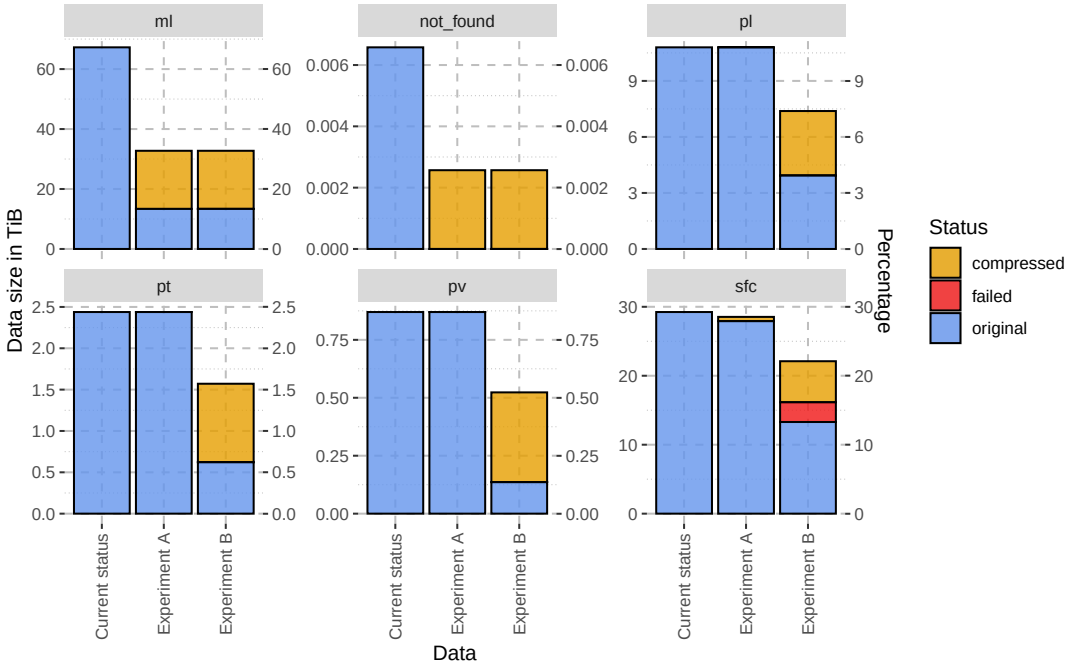


Figure 11: Data volume savings broken down by level type. For some data the levtype is not specified, and these have been labelled "not_found" and visualised separately.

4 Compression performance

Compression algorithms have two relevant performance axes to consider when being introduced into the forecast pipeline. Firstly, the extent to which each packing type compresses the supplied data, as it varies for a range of different parameters. Secondly, introducing compression into the forecast pipeline introduces a computational cost in exchange for the savings in storage space. Given that the `grid_ccsds` and `grid_second_order` packing types achieve very similar compression ratios, the difference in their performance is likely to be the deciding factor.

This section presents the relative performance of the `grid_ccsds` and `grid_second_order` packing types for a range of different parameters, considered at a range of different resolutions. This will allow us to consider the impact of these algorithms in a more fine-grained way and also consider the impact of future resolution increases.

The data used for these tests originate from a high-resolution experiment conducted on the OLCF's Summit supercomputer. It has a much higher resolution (1km) than current production data (18km). Each of the GRIB messages uses the O8000 grid, is packed with `grid_simple`, and occupies 489 MiB[10]. To test behaviour at a range of resolutions, these fields were then downscaled to O320, O400, O640, O800, O1024, O1280, O1600, O2000 and O4000 grids. One side effect of downscaling to lower resolutions is that the images become sharper, placing a higher demand on the compression algorithms than would otherwise be typical.

A selection of parameters that are typically output by the IFS in `grid_simple` form are recompressed using the `grid_ccsds` and `grid_second_order` packing types. The resulting data, and timing information from the conversion process are then considered[11].

The studied parameters can be divided into two groups, the air and surface fields. The difference between the groups are the transitions, e.g., the air fields (ciwc, clwc, crwc, cc, cswc, pv, q) have smooth transitions throughout the whole area, while the surface fields (asn, blh, ci, dsrp, istl1, lsp, lspf, pev, ro, rsn, sf, smlt, sro, ssro, sst, swvl1, tciw, tclw, tcwv, u10n, uvb, v10n) can have sharp transitions between land and sea. Since this can have an effect on compression, we have shown the fields separately in Figures 12 and 13.

4.1 Evaluation

Figures 12 and 13 give us an insight into the behaviour of the two packing types under consideration, both relative to each other and the `grid_simple` reference.

Looking first at Figure 12, we can see the compression ratios of the `grid_second_order` and `grid_ccsds` relative to `grid_simple`. In all cases, the compressed data is smaller than that encoded using `grid_second_order`, but there are several interesting patterns. Although the compression performance is very similar for the two packing types, it is highly parameter-dependent, varying by (in the most extreme case) up to 70x. This likely varies according to the entropy content and the sparsity of the different parameters.

Secondly, the achieved compression ratios are resolution-dependent - especially for the surface fields. At higher resolutions, the data compresses more efficiently. This is a very useful property,

[10] The cc parameter has fields of 245 MiB

[11] The experiments are conducted on a workstation equipped with 32GB DDR4-RAM and an Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz.

as it implies that the impact of using compression will only grow as our data usage increases.

Figure 13 gives some insight into the relative behaviour of the two packing types. Essentially the red regions (ratio > 1) of the plot show `grid_ccsds` outperforming `grid_second_order`, whereas the grey regions demonstrate the reverse. It can be clearly seen that although the margin is slight at the current operational resolution, `grid_ccsds` outperforms `grid_second_order` the majority of the time. Further (and notably), as the resolution is increased further, its compression ratio lead increases — including some parameters which previously favoured `grid_second_order` switching optimal packing algorithm. For this reason, it would appear that using `grid_ccsds` would be preferable from a data volume perspective.

It is worth noting that there are some outlier parameters in both of these plots. A few parameters (e.g. q & u10n) do not appreciably compress at any resolution — likely as a result of having high internal entropy. Further, some fields (e.g. asn) significantly favour the `grid_second_order` packing type. But these parameters are small relative to the overall bulk of the data.

Moving onto computational performance, Figure 14 shows the compression and speeds achieved using all three packing types across various parameters and resolutions. The darker solid lines show the average performance at the specified resolution, whereas the coloured stripe indicates the range of performances observed between the most and least performant parameters.

It is not surprising, that the reference `grid_simple` packing type outperforms the others across all parameters and resolutions, as it does not need to perform any compression or decompression. It appears that the decompression routines are less optimal than compression for `grid_simple`, with the performance dropping off rapidly with the resolution, possibly indicating when the data no longer fits in cache — and this can likely be fixed with further optimisation.

It is very clear that `grid_ccsds` compresses data much more rapidly than `grid_second_order`, consistently being roughly 2.5-3x faster. The compression speeds are remarkably consistent across a range of different resolutions.

For decompression, the picture is entirely different. Across the bulk of resolutions, and especially at higher resolution (from O1024 and above), the performance of the two packing types is very similar, with `grid_ccsds` being roughly 11-15% slower.

The sensitivity of ecCodes and the various compression algorithms to the nature of the input data can be clearly seen in Figure 15. Here we can see the ratio between the compression speed and ratio, and similarly for decompression. There is a strongly visible trend that the more that data can be compressed, the higher the computational cost. By contrast, decompression speed is not significantly dependent on the parameter type - and is rather largely dominated by the resolution. At higher resolution, data decompresses faster.

4.2 Conclusion

The `grid_ccsds` packing type is favoured over `grid_second_order` from the perspective of compressed data volume. Although notably slower than `grid_simple`, as expected, `grid_ccsds` is also significantly favoured in terms of compression speed and is only marginally less performant for decompression.

It is notable that the overall compression ratios achieved using the `grid_ccsds` packing type and its compression ratios relative to `grid_second_order` improve with resolution. Further, the

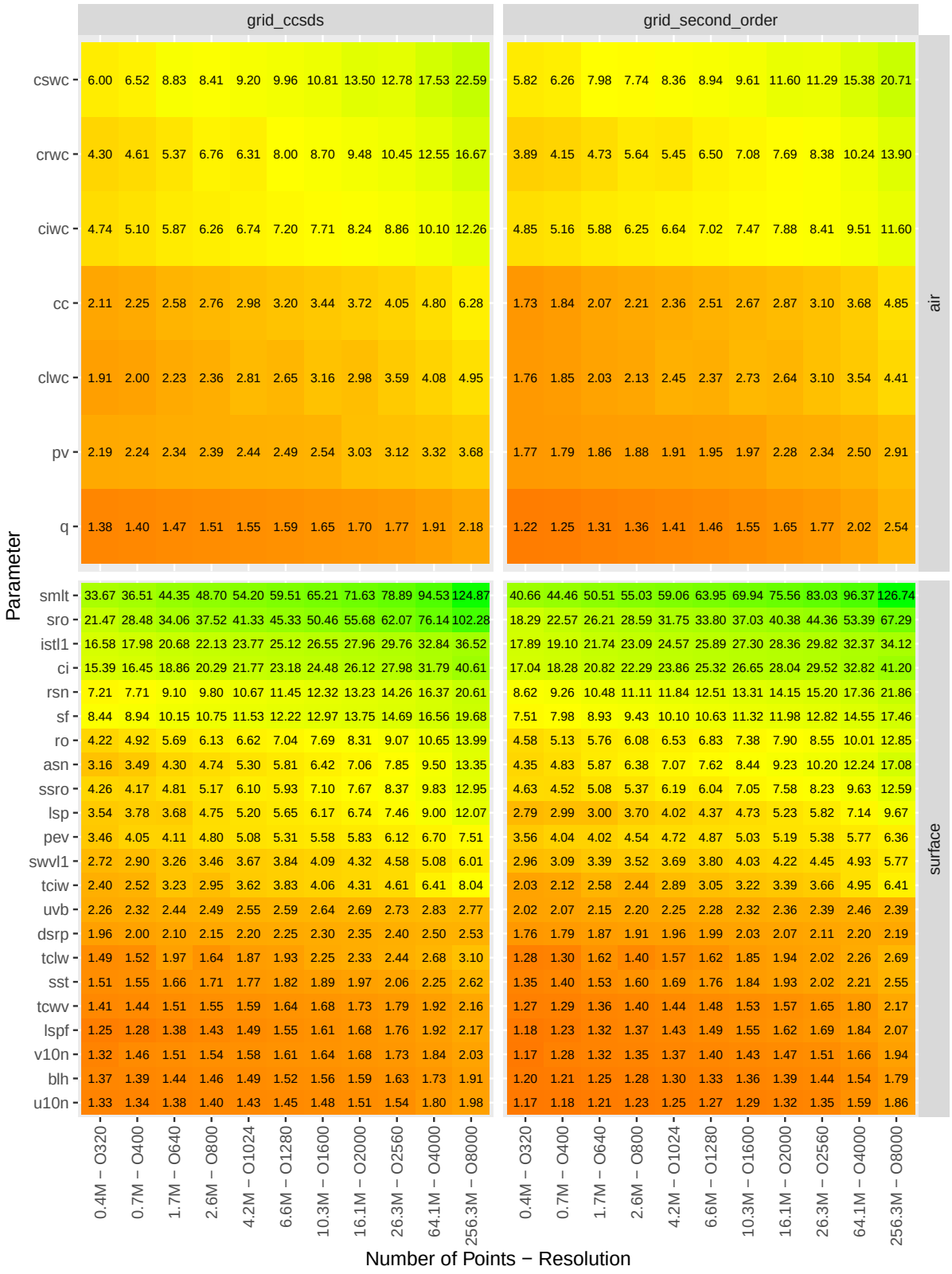


Figure 12: Compression ratio relative to the `grid_simple` *packing type (e.g., a compression ratio* $= 2$ *means that the resultant data is half the size of that encoded with* `grid_simple`*).*

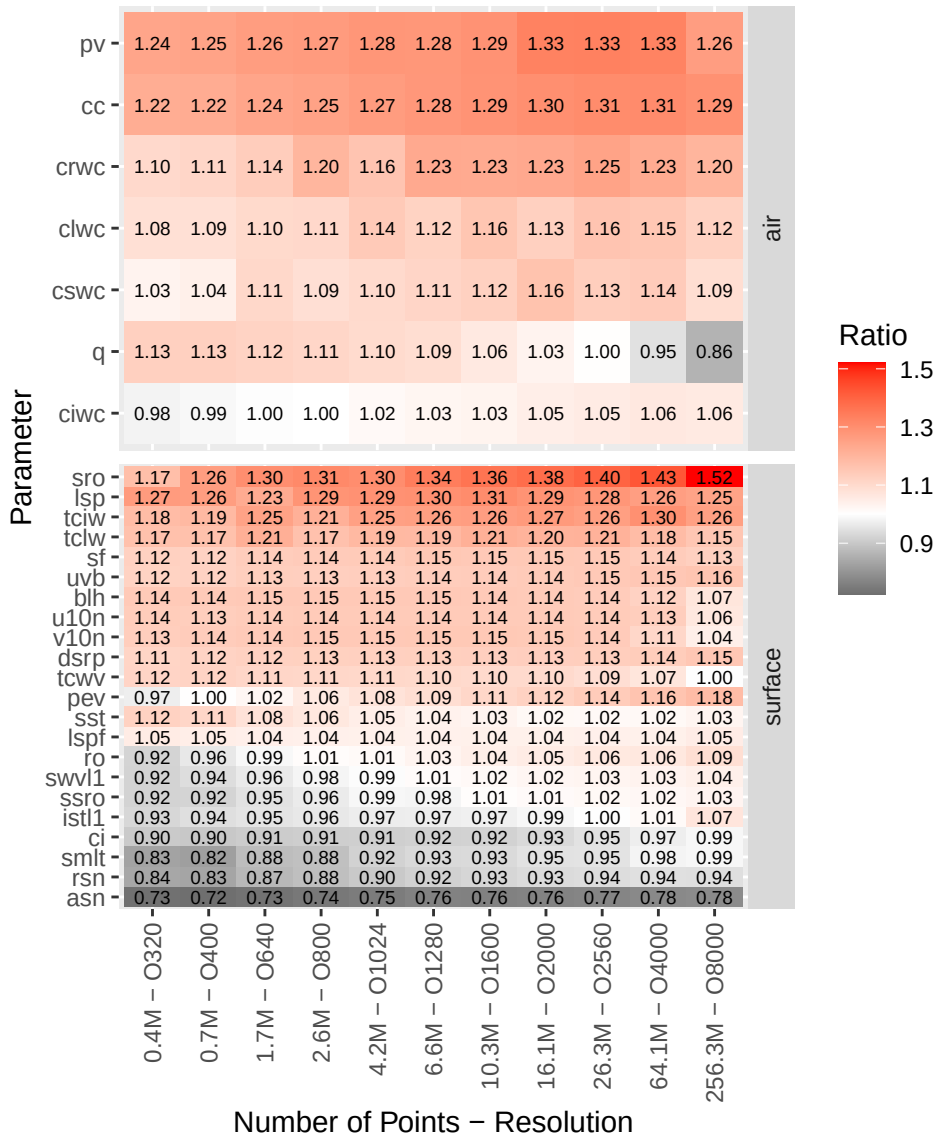


Figure 13: Comparison of grid_second_order and `grid_ccsds`*. The ratio* $= \frac{compression\ ratio_{grid\_ccsds}}{compression\ ratio_{grid\_second\_order}}$ *means, that if ratio* < 1 *(coloured grey), then* `grid_second_order` *compresses better, else if ratio* > 1 *(coloured red), then* `grid_ccsds` *compresses better.*

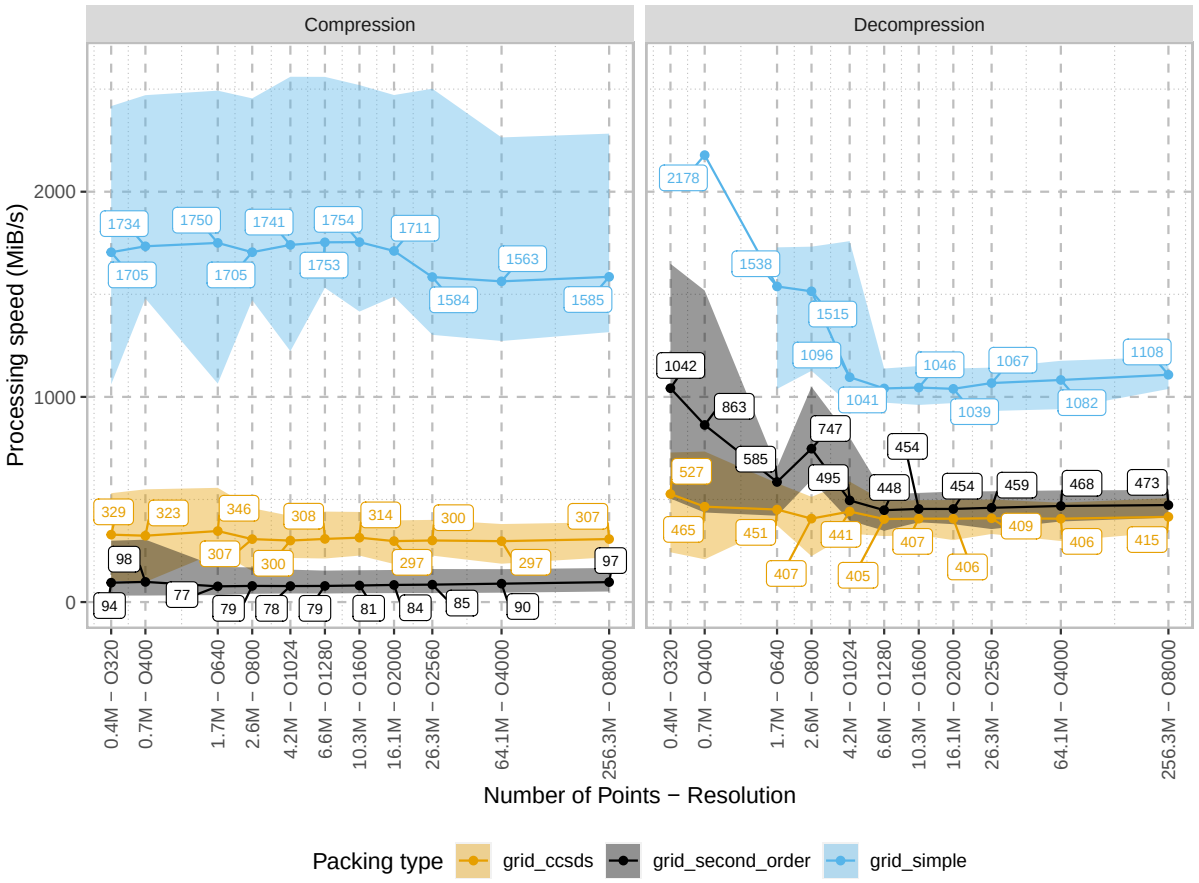
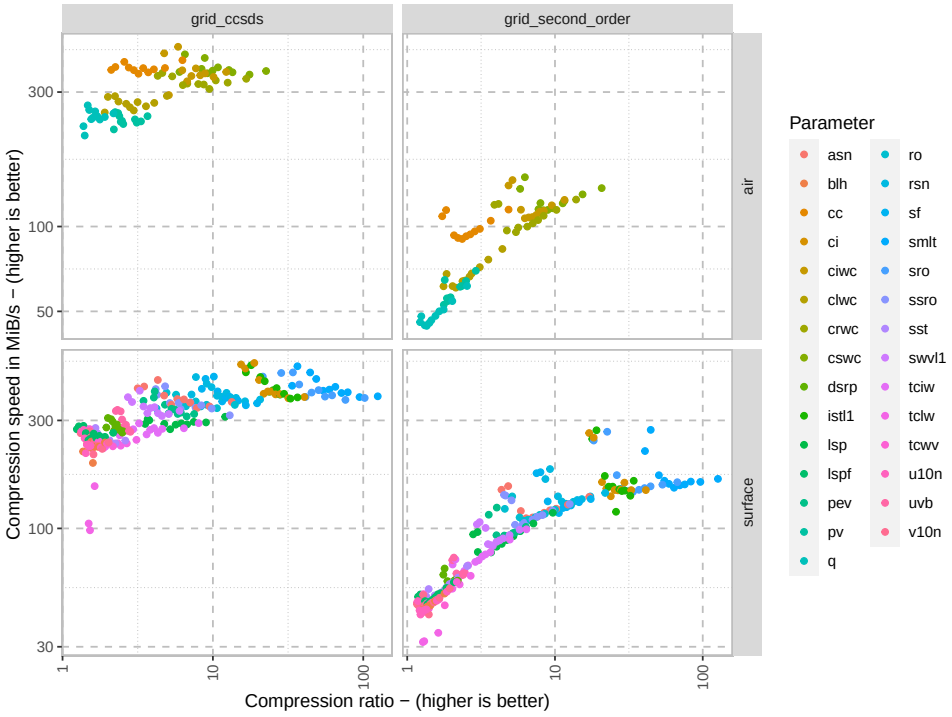
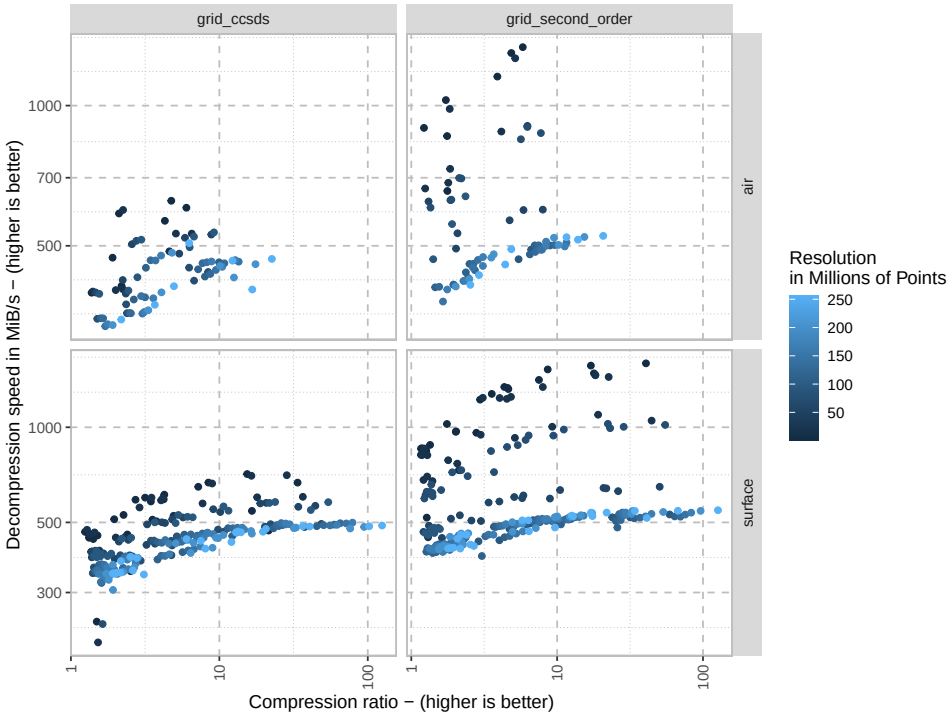


Figure 14: Data compression and decompression speeds. The area around the lines indicate the range between the minimum and maximum recorded speeds for different parameters.



(a) Compression speed vs compression ratio. The data content has an effect on the compression speed and compression ratio. Sparse data seems to be compressed better and faster.



(b) Decompression speed vs compression ratio. Decompression is faster for low resolutions and high compression ratios.

Figure 15: Relationship between speed and ratio.

compression and decompression performance of these algorithms are stable as the resolution is increased.

As such, `grid_ccsds` should be recommended as the most suitable packing type to consider through the remainder of this work.

5 Performance profile and scalability

The ecCodes library is used at ECMWF for encoding and decoding GRIB data. It is sufficiently performant for current use, but its performance profile will change after the upcoming resolution increase. As the library processes much larger data volumes, the its overall encoding run-time will extend significantly. It is important to understand how this additional effort will be expended.

There are two overall sets of actions involved in encoding a GRIB message. Firstly, there are per-field operations, such as encoding the field metadata, which should be identical irrespective of the size of the overall data. When considering the rate of encoding the data (per MiB, or per data point) this effort should provide a fixed overhead per field. Secondly, there is the encoding of the data itself, comprised of the quantisation and compression routines. Ideally these will scale linearly with the volume of data being compressed, but this is by no means guaranteed as larger data volumes can interact poorly with cache infrastructure, and compression algorithms may not be comprised of a simple pass through the data.

ecCodes was instrumented on a workstation[12] with the use of a profiling tool, and the run-times of the constituent functions were recorded. Data in a range of different resolutions, from O320 to O8000, was repacked from `grid_ccsds` to `grid_ccsds`. Before the experiments, we emptied the cache so as not to influence the reading speed through the cache.

5.1 Evaluation

An example of the data obtained can be seen in Figure 16, which shows functions with a run-time greater than one percent for data at an O8000 resolution. Note that the run-times do not include run-times of nested functions of the ecCodes library, but run-times of other called libraries (e.g. of the C standard library) are included.

It can be seen on run-time costs of the three most significant functions in Figure 16b that these three functions are slowly consuming the majority of the run-time as the resolution increases, with the fixed overhead falling. We can see from Figure 16a that the cost per data point for decoding the source data and encoding the output is very constant through the runs. This component of the effort scales linearly. Reading data becomes cheaper the bigger it gets. But there are three observations to couple with this.

First, at lower resolutions the fixed overheads per field are large. The majority of our effort is not being spent processing the data itself, but is expended on I/O, memory management and metadata handling. It is a little disappointing that the data coding routines do not dominate from lower resolutions.

Second, a look at the `grid_ccsds` source code reveals that the `grib_decode_unsigned_long()` function is called for each point. At high resolutions that are several millions function calls that slow down the decoding process considerably. There is considerable potential here to increase decoding speed.

Finally, the runtime of the `stdio_read()` function is considerable, but fortunately, I/O becomes cheaper with the higher resolution compared to the total runtime of the application. Interestingly, we see a growth stop after the O1280 resolution. This may be related to the hardware

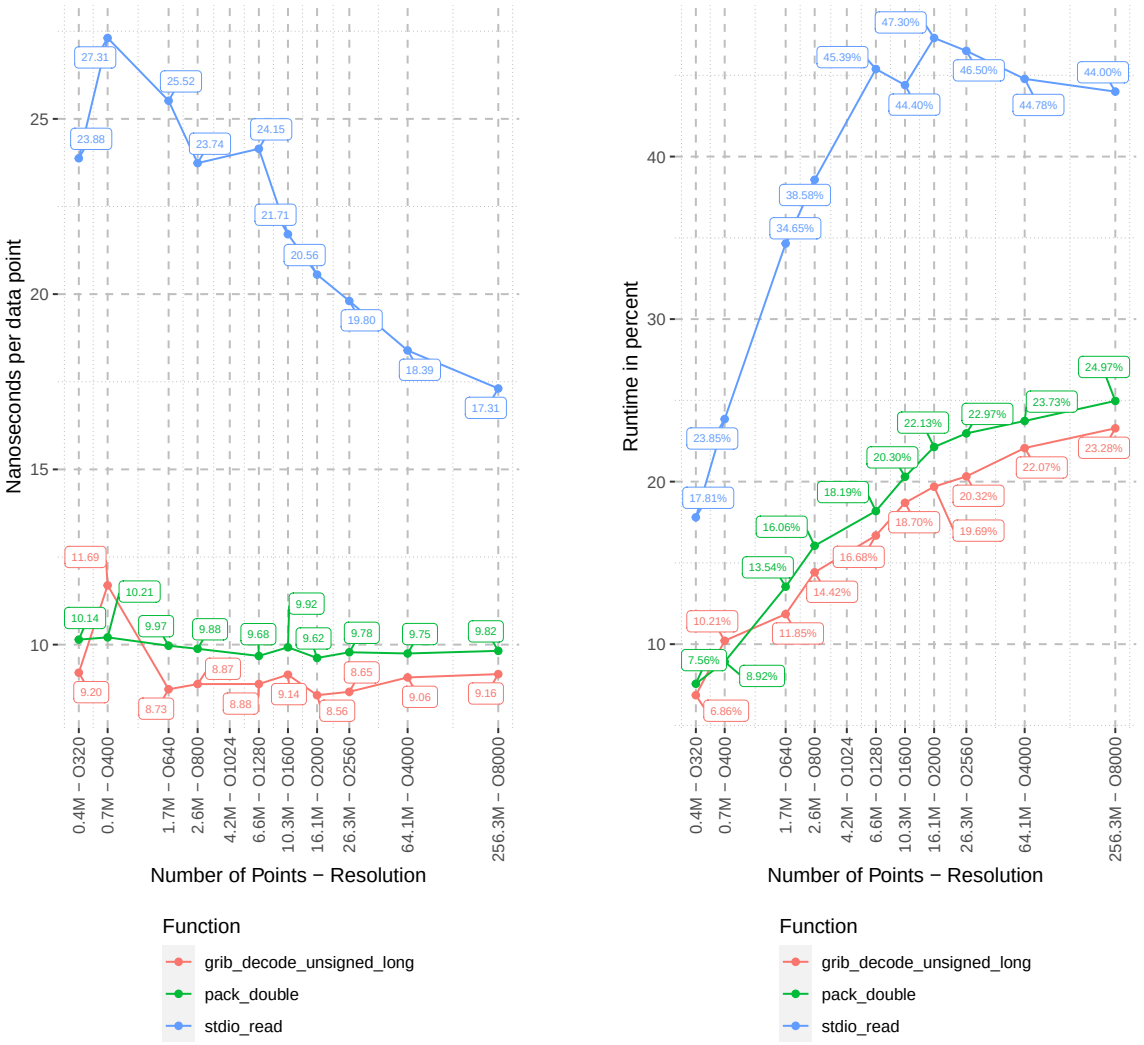
[12] Test environment: CPU: Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz; RAM: DDR3 Synchronous 1600 MHz; HDD: Western Digital RE4-GP WDC WD2002FYPS-02W3B0

used for the the experiment, but we will not go into this in detail because I/O is beyond the scope of this paper.

5.2 Conclusion

As the resolution, and hence volume, of the data increases the distribution of computational effort shifts. There is a certain per-field computational overhead which remains roughly constant, and an ever larger proportion of the run-time is spent processing the data points. The packing routines have a constant cost per data point, and as such the computational cost tends towards linear with the size of the data being processed.

The many calls to the `grib_decode_unsigned_long()` function slow down the unpacking process. There is a lot of potential for improvement here.



(a) Processing time of a data point. The costs for processing a data point are nearly constant. From this we can infer the linear data processing speed, which is actually an expected and good result. The one exception is the `stdio_read()` function. The reading of the data becomes cheaper with increasing data size.

(b) The run-time related to the total run-time of the application. If we disregard the `stdio_read()` function, the remaining functions that are presented are increasing in their run-time percentage. From this we can conclude that the ratio between fixed overhead and data processing improves with increasing resolution.

Figure 16: The most compute-intensive functions when compressing data with the O8000 resolution. (The measurements may be inaccurate due to some profiling overhead and serve only as a rough overview.)
Description:
`pack_double()` *is the* `grid_ccsds` *encoding function. It encodes 64-bit floating point arrays to 16-bit integer arrays and compress data with the AEC compression algorithm.* `grib_decode_unsigned_long()` *decodes a value consisting of n-bits from an octet-bitstream to long-representation.* `stdio_read()` *is a wrapper for* `fread()` *and is used to read sections and metadata from GRIB files.*

6 Impact on run-time of IFS and MARS

As has been seen earlier in this report, changing the GRIB packing type from `grid_simple` to `grid_ccsds` will result in GRIB encoding and decoding taking longer and using more computational resources. This may impact overall application run-times.

But the time of encoding and decoding is only a small part of our larger applications, and the impact of slowing down these processes does not straightforwardly translate into application run-times. An application may hide the cost of encoding by running encoding tasks in concurrent processes while computational tasks continue to run uninterrupted, or the changes in data volumes may cause other parts of a workflow to become faster!

6.1 IFS experimental suites

An IFS experiment is a collection of tasks with a fixed execution order. We can think of it as a directed graph, where each node represents a job with a specific task (such as setup, initialisation, simulation or archival). Once an experiment is started, a scheduler executes the jobs according to a specified dependency structure and logs the execution times.

We examine the run-times of the ensemble of tasks comprising IFS experimental suites running at a range of resolutions, using both the `grid_simple` and `grid_ccsds` packing types. The impact on the change of packing type between otherwise identical experiments.

Once the run-times are collected, the relative run-time can be calculated as

$$\text{Speed-up in } \% = \frac{t_{\texttt{grid\_ccsds}}}{t_{\texttt{grid\_simple}}} \cdot 100 \tag{2}$$

As these tasks are run on a shared supercomputing resource and use shared resources in the Data Handling System, there is a considerable amount of noise in the numbers obtained. As the overall trends can be clearly seen, and the experiments are resource- and time-intensive, these runs have not been repeated to obtain convergence, and the precise values obtained must be considered in that light.

Most of the tasks involved in the workflow, as seen in Figure 17 do not involve the encoding, decoding or transfer of significant volumes of GRIB data. Unsurprisingly, they are unaffected by the change in packing type. The primary tasks that need further analysis can be seen to be main/fcgroup/model.1 (which will be considered here) and lag/archive/* (which are considered in the next section).

The notable conclusion drawn from the model speed-up in Figure 18 is that there are no statistically significant changes in the run-time performance of the model when compression is enabled. This can be explained by the data output pathway in the model. Simulation data is produced on many parallel compute nodes in a disaggregated fashion, such that different nodes threat different geographic regions. The output data is then sent to a smaller number of I/O server nodes, where the pieces are aggregated into global fields and encoded into GRIB messages before being output to the storage system. The I/O server is designed to decouple the I/O processes from the compute processes, such that the I/O servers take ownership of the data and compute nodes, then continue with further compute in parallel to the I/O processes.

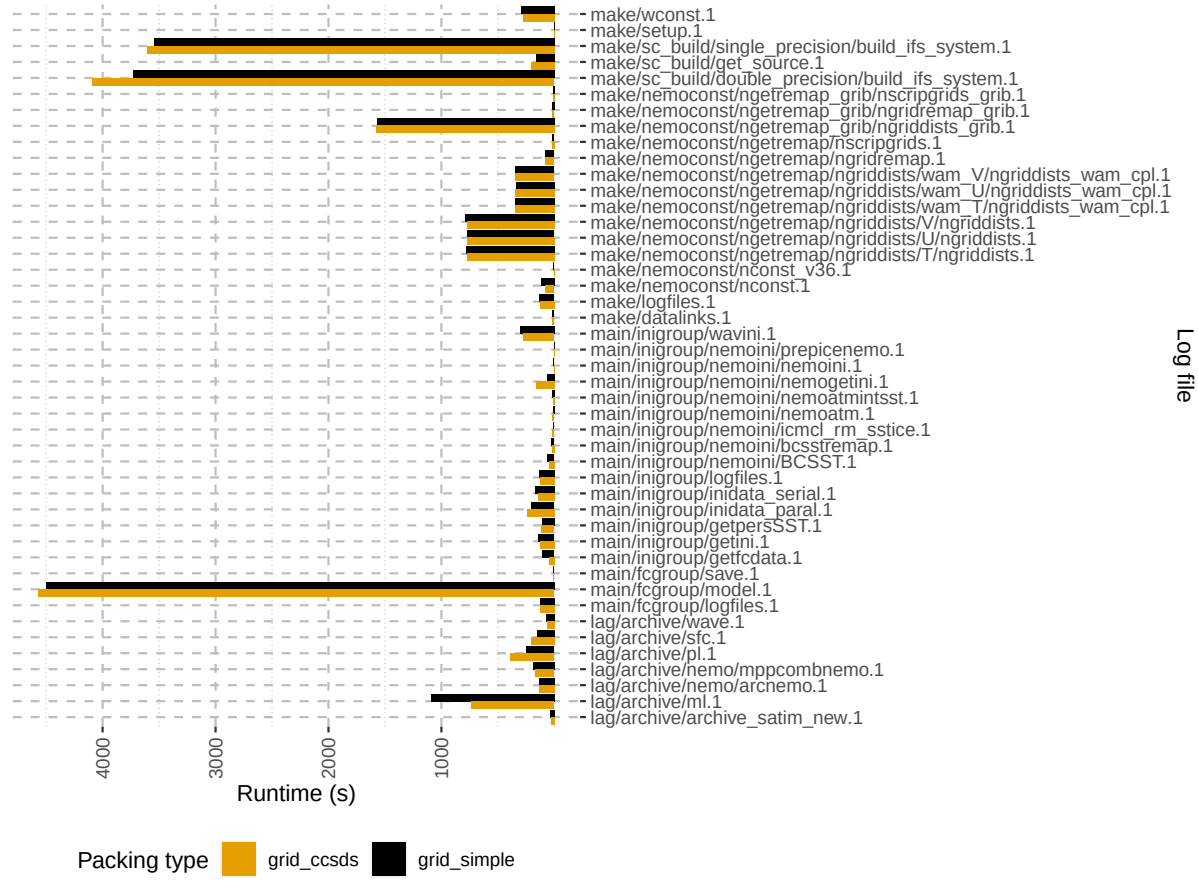


Figure 17: Job run-times of the TCo639 experiment. The job log files contain job run-times (horizontal bars). Some tasks, e.g., archiving, can be splitted over several jobs.

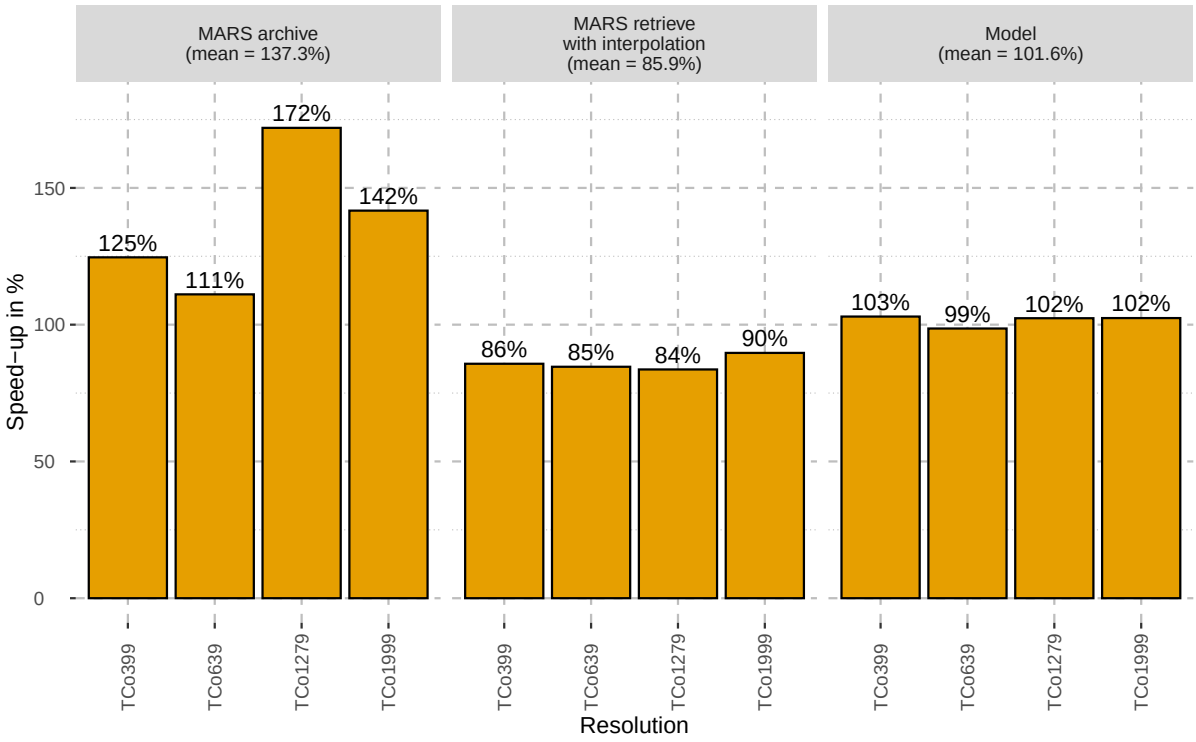


Figure 18: Relative run-time of various IFS workflow processes when using `grid_ccsds` *compared to* `grid_simple`*.*

The additional computational work required to compress the data does not exceed the resources available to the I/O server nodes, and as such the extra effort is entirely hidden from the overall run-time of the job. As the resolution is increased further, it is possible that the I/O server will not be able to keep up - but adding further I/O server resources should be able to maintain the decoupling from compute.

6.2 MARS performance

There are three types of MARS tasks to be considered. Simple archival and retrieval tasks do not directly encode or decode data, but as data transfer and storage/retrieval tasks they are heavily impacted by the reduction in data volume. Finally, MARS retrievals involving compute have two competing effects. Reductions in data volume will reduce the run-time, but the computation or interpolation places decoding and re-encoding data into the execution pathway.

Archival performance figures are extracted from the experimental suite shown in Figure 17. Retrieval is not separately considered, as it is a symmetric process. A separate interpolation task was then used to examine the impact of encoding and decoding, using the request in Listing 3. The parameter `grid:1/1` enables interpolation of the data to a different grid, and thus also decompression of the retrieved data and re-compression of the computed result on the MARS client.

Figure 18 shows that the archival tasks run notably faster with `grid_ccsds` encoded data that the reference `grid_simple` encoded data, across all of the resolutions. The bulk of the time

```
retrieve,
  domain   : g,
  levtype  : ml,
  date     : 20200721,
  time     : 0000,
  param    : q/o3,
  class    : rd,
  type     : fc,
  levelist : 1,
  stream   : lwda,
  expver   : %s,
  grid     : 1/1,
  anoffset : 9
```

Listing 3: Mars request that was used for testing data retrieval. The request initiates data compression on the server by activating interpolation with the `grid:1/1`.

taken by the MARS archival task is spent on transferring data across the network, and the reduction in data volume reduces this straightforwardly. This effect grows as the resolution increases, which is consistent with the increasing impact of compression at higher resolutions.

By contrast, MARS retrieval in Figure 18 shows that the interpolation case is approximately 14% slower. This clearly shows that the run-time impact of decoding and encoding is larger than the benefit obtained by the reduced data volume transferred over the network.

7 Summary

An analysis of several GRIB packing types showed the most promising ones to be `grid_ccsds` and `grid_second_order` packing. The `grid_ccsds` has a slightly higher compression ratio and has a balanced de-/compression performance, whereas the `grid_second_order` compression is somewhat slower. The experiments showed that `grid_ccsds` has a mean performance of 339 MiB/s and 452 MiB/s, and `grid_second_order` 82 MiB/s and 663 MiB/s for compression and decompression, respectively. The data compression speed is strongly dependent on the physical parameter type. Parameters with high variability compress less and take longer to process.

A detailed investigation of `grid_ccsds` and `grid_second_order` showed that we could reduce current data volumes per cycle in the FDB by 32% by adopting compression. This can be further increased to 42% when more fields are converted from GRIB1 to GRIB2, enabling `grid_ccsds` compression for more of the data.

Experiments with varying resolutions from O320 to O8000 have shown that the savings in data volume scale favourably with resolution increase, showing that the benefits of using compression increase as the resolution is increased. For example, with the implementation of Cy48R1 and the increase of ensemble resolution from 18 to 9 kilometres, the savings increase to 35% and 46%, respectively.

The run-time impacts on the IFS and on MARS are rather small but often positive. On experiments up to a resolution of TCo1279, the asynchronous I/O servers were able to accommodate for the increased computational effort of data compression, thus hiding the slower packing process (compared to `grid_simple`) and resulting in IFS model runs that are neutral or even slightly faster than before ($\approx 2\%$) due to the reduced I/O load.

MARS tasks are impacted in different ways, depending on if decompression is involved or not. Experiments showed that MARS archiving research data is 37% faster due to smaller data volume. MARS data retrieval when incurring interpolation by user request (e.g. grid=1/1) was 14% slower due to the need to decompress the data before interpolation and compress the results. However, this impact depends strongly on the details of specific interpolation requests.

The performance analysis also showed that the ecCodes' design and implementation currently scale well with data size and allows for efficient data processing. It was also shown that data packing is the most compute-intensive part of ecCodes in these use cases. As the resolution increases, the proportion of time spent in this part also increases, and the overall computational cost tends towards a linear dependence on the input data size.

8 Recommendations

Based on the contents of this report, we make the following recommendations

- The `grid_ccsds` packing type should be adopted for all fields that are already encoded in GRIB2.
- Further consideration should be given to which fields are truncated to 16-bit vs 24-bit precision before compression.
- The adoption of compression should commence with Cy48R1 to compensate for the increase in ensemble resolution from 18 to 9 km, thus alleviating its impact on the data volumes.
- The `grid_ccsds` packing type should also be adopted for research experiments, even at lower resolutions. The reduction in data volumes results in MARS archiving tasks running more quickly, which is highly desirable as these often act as bottlenecks for the progression of experiments.
- The migration from GRIB1 to GRIB2 should be accelerated to maximise the proportion of the data that can be encoded using the `grid_ccsds` packing type. This will give us the maximum benefit, especially in light of future resolution increases, and the Destination Earth programme.
- The application of the `grid_ccsds` packing type to Copernicus CAMS data should be explored. More specifically, the entropy content of the chemical species data seems suitable for compression.
- The use of multi-threaded encoding tasks within the new MultIO server should be explored to enable overlapping of the computational work of compression with data movements in the output pipeline and remove the potential run-time impact of this from the time-critical parts of the forecast model.
- ECMWF should undertake a performance study of the libAEC library, which implements the compression algorithm underlying `grid_ccsds`. ECMWF should work collaboratively with its original authors (DKRZ) to ensure optimal performance.
- The possibility of replacing the output of data in `spectral_complex` with gridded data stored using `grid_ccsds` packing should be investigated. For the relevant parameters, this will avoid a final transformation from grid-point space into spherical harmonic space inside the IFS FullPos routines.

References

[Qui12] Tiago Quintino. *Consequences of changing IFS output to Grid Point.* Tech. rep. ECMWF, July 2012.

[Spa20] The Consultative Commitee for Space Data Systems. *Lossless Data Compression.* https://public.ccsds.org/Pubs/121x0b3.pdf. Washington, DC, USA: CCSDS Secretariat, National Aeronautics and Space Administration, 2020.