Optimizing NetCDF usage

Valentín KIVACHUK BURDÁ

INTRODUCTION

- Original data sources comes in different formats (CSV, GRIB, numpy arrays, …)
- Must **convert** to NetCDF format.
- Efficient most of the times. But **NOT** always.

3 Vars (***Floats***) *with:*

- 1461 **time** slots
- 384 points of **latitude**
- 288 points of **longitude**

A - %	| Nebulosity (**0 – 100**)
B - ºC	| Temperature (**-10 – 40**)
C - m/s	| Wind Speed (**0.xx – 400.xx**)

Algorithm example

```
img_len = 64
list_rand = 6000 random values of time, lat and lon
for time, lat, lon in list_rand:
  for var in [ 'A', 'B', 'C' ]:
    data = NetCDF[var][time, lat + img_len, lon + img_len ]
    acumulate_mean(var, data)
```

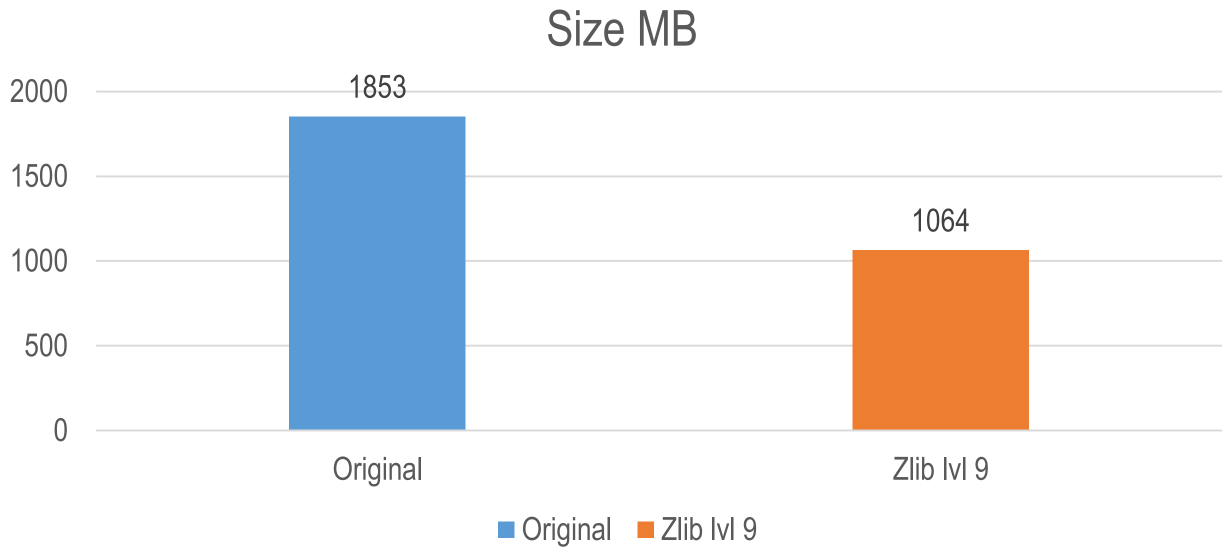
We collect 6,000 samples for each variable, randomly in all dimensions, and compute the mean thereof.

WHAT IS NetCDF?

NetCDF is a set of software and data formats that manage scientific data.

- Self-describing format
- Multidimensional variables
- **Native compression (zlib)**

Apply NetCDF compression (**zlib**) with maximum level (**9**).



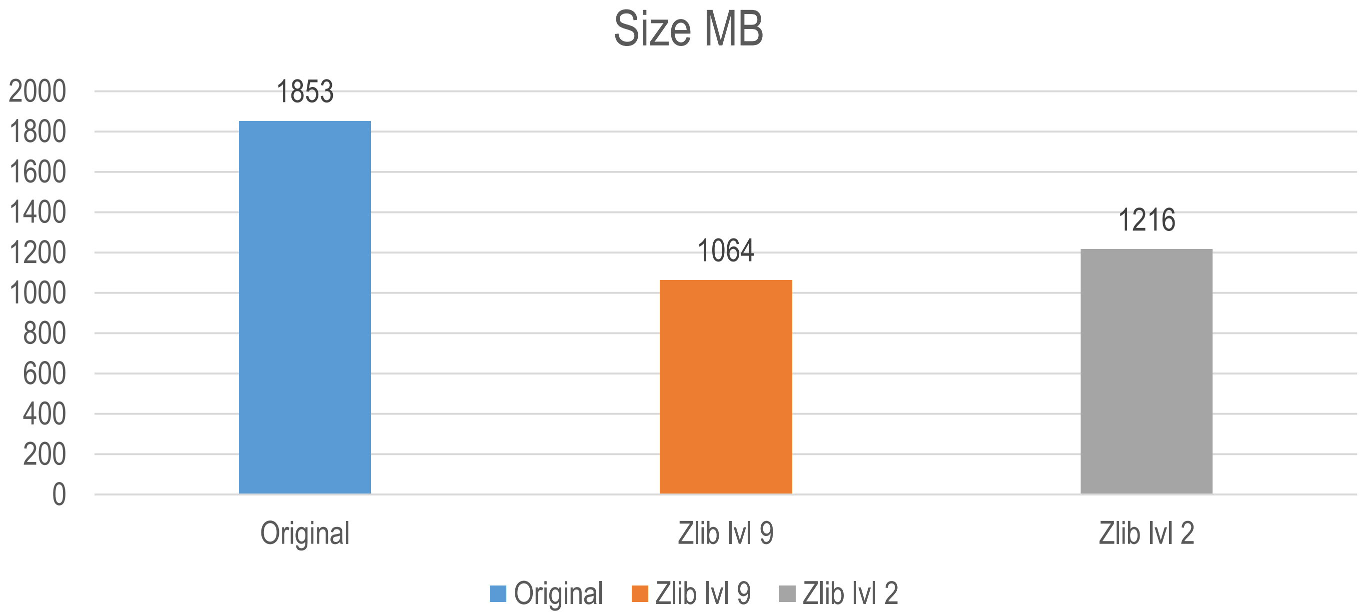
EXTERNAL COMPRESSION

The performance is **REALLY BAD**.

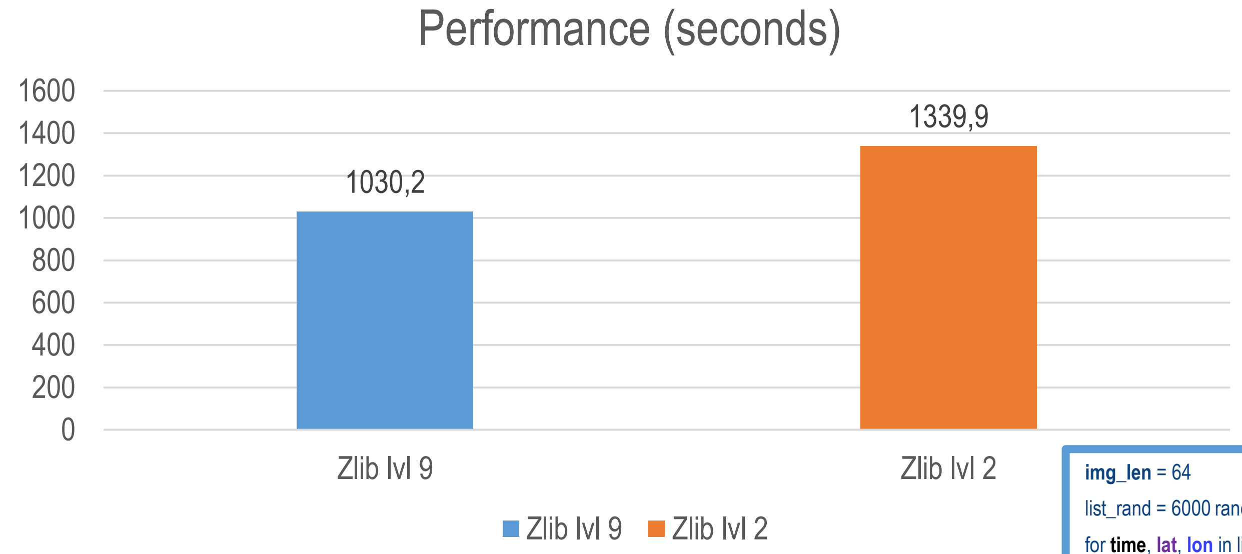
1030s (17min 10s)

What can I do?

Higer compression level needs more CPU time.



BUT, bigger size -> more data to read -> **slower**




```
img_len = 64
list_rand = 6000 random values of time, lat and lon
for time, lat, lon in list_rand:
  for var in [ 'A', 'B', 'C' ]:
    data = NetCDF[var][time, lat + img_len, lon + img_len ]
    acumulate_mean(var, data)
```

IRT SAINT EXUPERY

TYPE OF DATA

- A value can be stored in different formats (**int**, **float**, ..)
- All values within a variable have the same format
- A format have a size (per value) and can represent a delimited range of values
- Choosing a format with **smallest size** that can represent our <u>range of values</u>

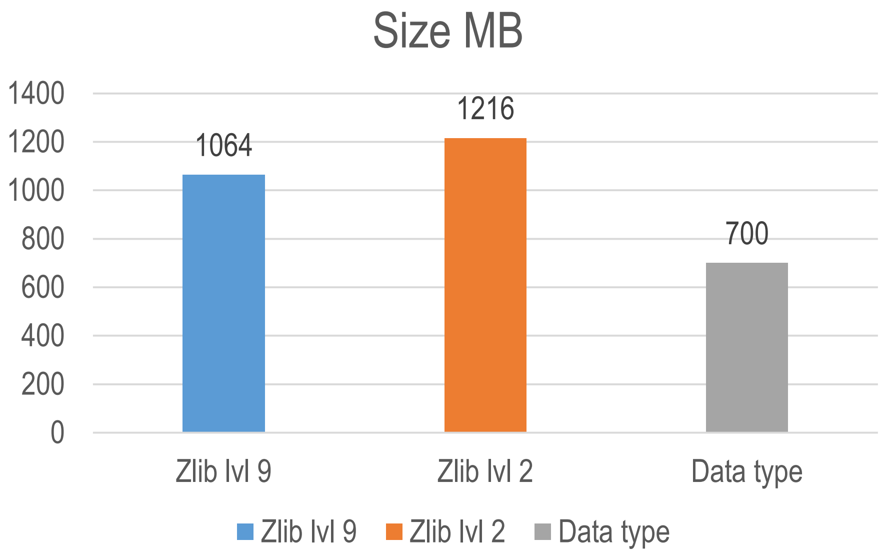
Reduce size per value

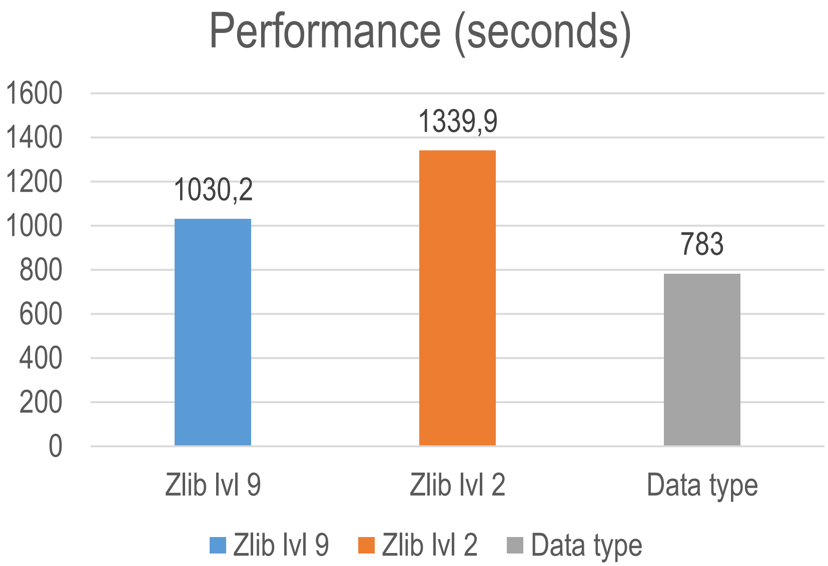
Name	Size (bytes)	Range
BYTE	1	-127 ... 128
UNSIGNED BYTE	1	0 … 255
SHORT	2	-32,768 … 32,767
UNSIGNED SHORT	2	0 … 65,535
INT	4	-2,147,483,648 … 2,147,483,647
UNSIGNED INT	4	0 … 4,294,967,295
INT64	8	$-2^{63} \ldots 2^{63} - 1$
UNSIGNED INT64	8	$0 \ldots 2^{64} - 1$
FLOAT	4	$3.4 \pm E38$ *
DOUBLE	8	$1.7 \pm E308$ *

A - % | Nebulosity (0 – 100)
B - ºC | Temperature (-10 – 40)
C - m/s | Wind Speed (0.xx – 400.xx)

* Can represent decimals

TYPE OF DATA






```
img_len = 64
list_rand = 6000 random values of time, lat and lon
for time, lat, lon in list_rand:
  for var in [ 'A', 'B', 'C' ]:
    data = NetCDF[var][time, lat + img_len, lon + img_len ]
    acumulate_mean(var, data)
```

TRANSFORMATION

Linear Packing

- Pack *floats*, *doubles* (**4, 8 Bytes**) inside smaller formats (**4, 2, 1 Bytes**)
- Loss precission (depends of data range and output type)

PPC

- Set 0s some mantissa positions (IEEE-754)
- Loss precission (can be controlled)
- Impact on external compressors (zlib)

- **C** range (*floats*) can be packed inside *shorts*

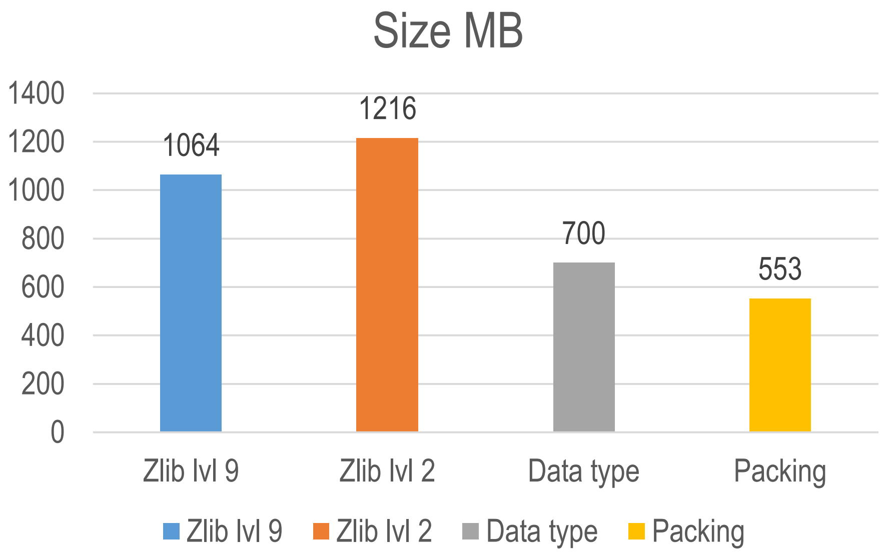
Name	Size (bytes)	Range
BYTE	1	-127 ... 128
UNSIGNED BYTE	1	0 ... 255
SHORT	2	-32,768 ... 32,767
UNSIGNED SHORT	2	0 ... 65,535
INT	4	-2,147,483,648 ... 2,147,483,647
UNSIGNED INT	4	0 ... 4,294,967,295
INT64	8	$-\frac{2^{64}}{2} \dots \frac{2^{64}}{2} - 1$
UNSIGNED INT64	8	$0 \dots 2^{64} - 1$
FLOAT	4	$3.4 \pm E38$
DOUBLE	8	$1.7 \pm E308$

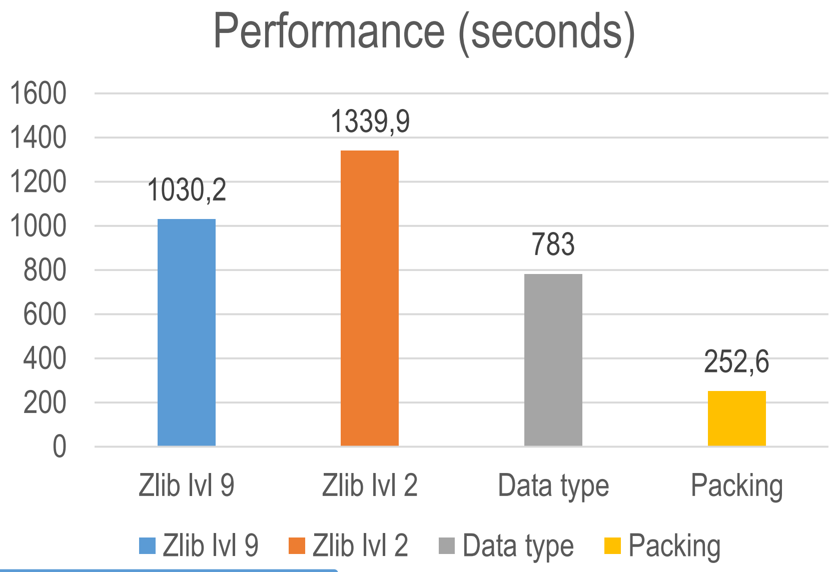
A - % | Nebulosity (0 – 100)

B - ºC | Temperature (-10 – 40)

C - m^2/s | Wind Speed (0.xx – 400.xx)

TRANSFORMATION



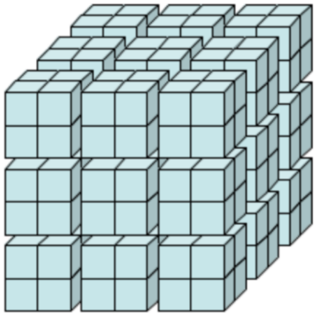



```
img_len = 64
list_rand = 6000 random values of time, lat and lon
for time, lat, lon in list_rand:
  for var in [ ‘A’, ‘B’, ‘C’ ]:
    data = NetCDF[var][time, lat + img_len, lon + img_len ]
    acumulate_mean(var, data)
```

ACCESS PATTERN

ACCESS PATTERN

- Multidimensional data can be stored in chunks.
- Each chunk is processed internally as atomic set of data.
- The shape of the chunk is closely related with the performance of final application.



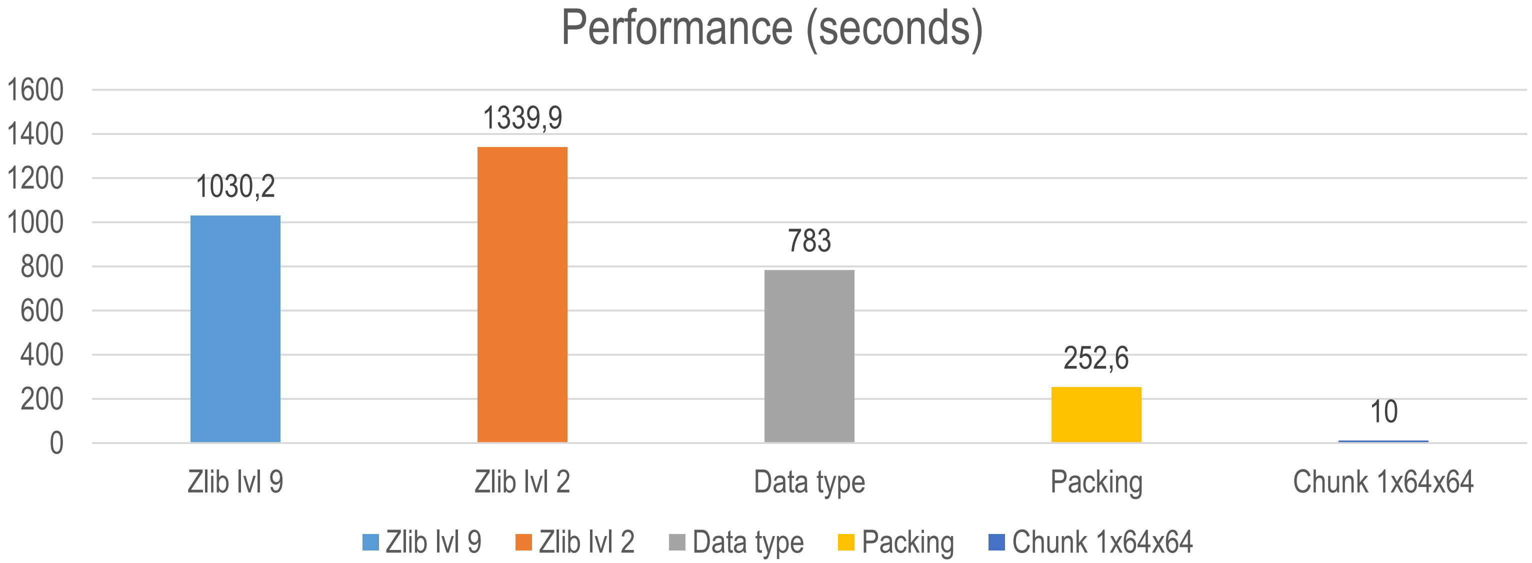
chunked

ACCESS PATTERN

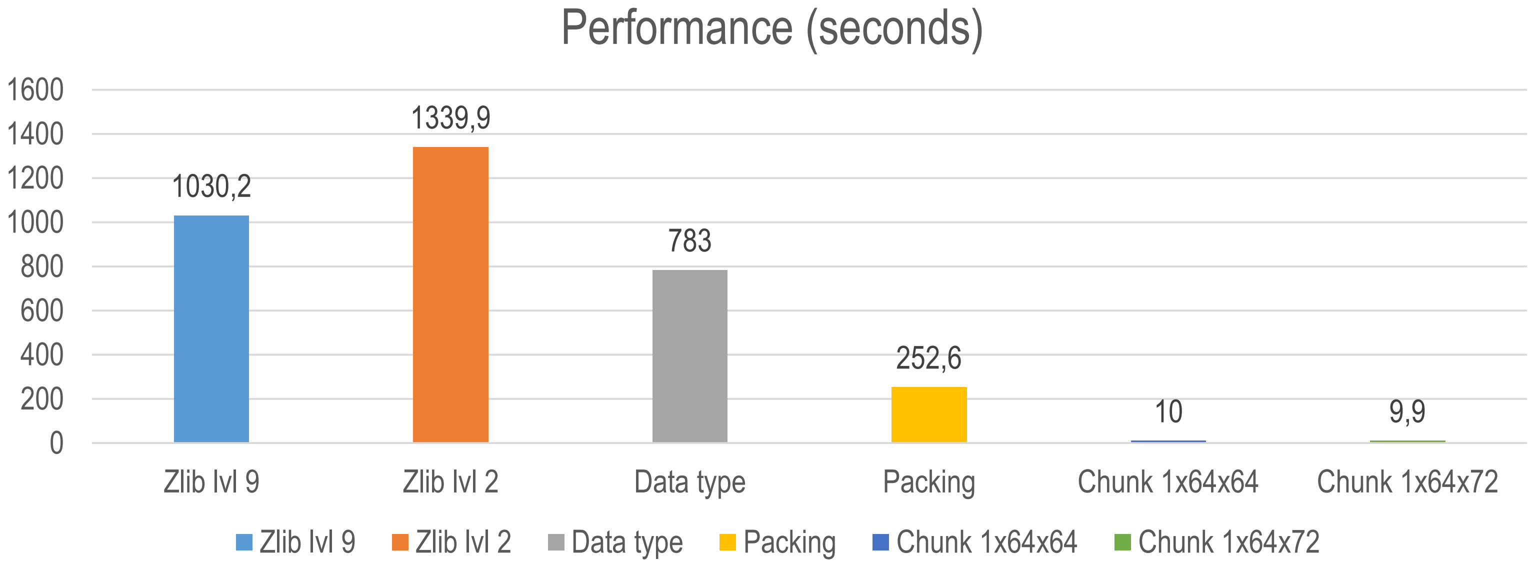
- How we access the data?
- Each time: 1 **time**, 64 **lat** and 64 **lon**
- Different aproach for **Read**, **Write** or **Both**
- A **REALLY GOOD** chunking for one access pattern can be **REALLY BAD** for other.

```
img_len = 64
list_rand = 6000 random values of time, lat and lon
for time, lat, lon in list_rand:
  for var in [ 'A', 'B', 'C' ]:
    data = NetCDF[var][time, lat + img_len, lon + img_len ]
    acumulate_mean(var, data)
```

Each chunk with 1 **time**, 64 **lat** and 64 **lon**.



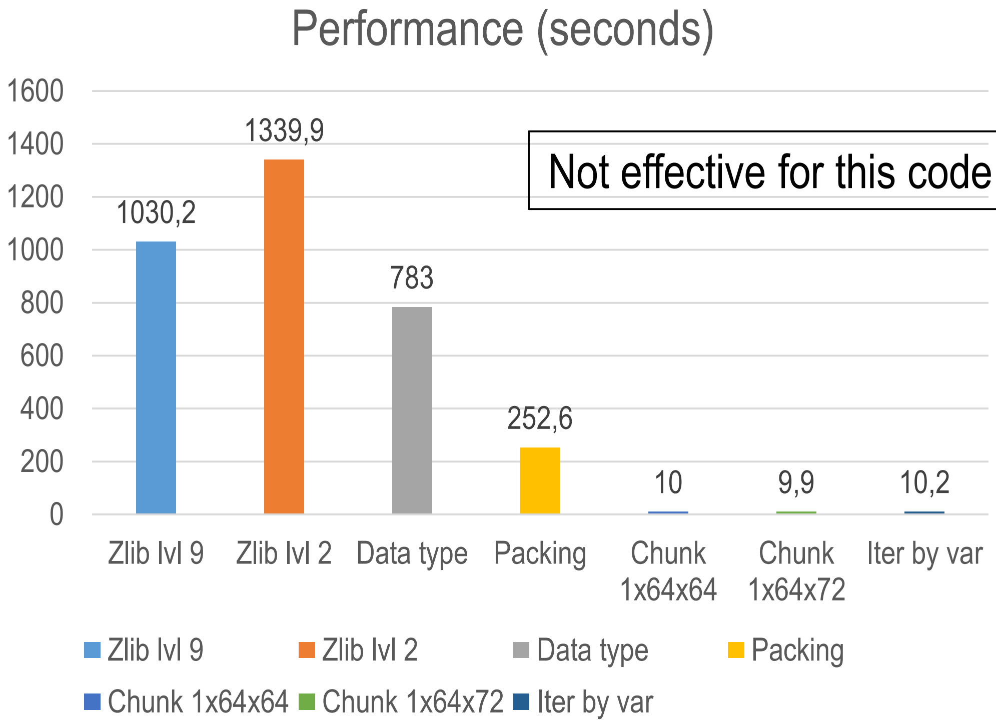
Find the best chunksize: A complex problem (1x64x72)



ITERATE BY VARIABLE

- NetCDF read data from disk to RAM. A expensive operation.
- Stores data in cache (RAM) to quickly retrieve previously read data.
- Cache is per variable (not reused between variables)

ITERATE BY VARIABLE



Not effective for this code

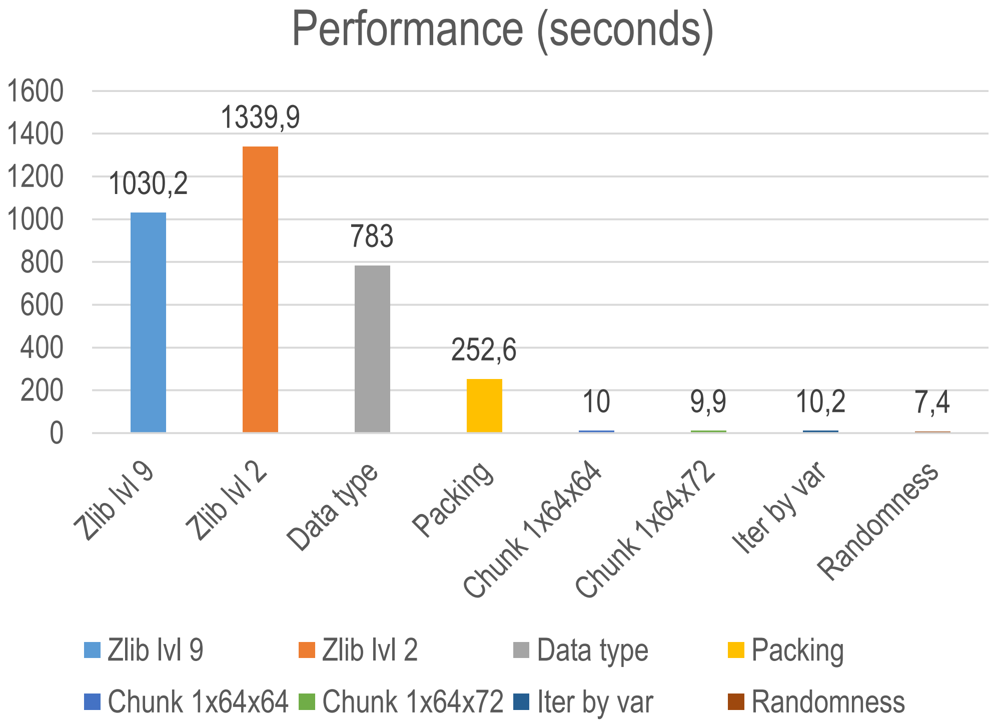
```
img_len = 64
list_rand = 6000 random values of time, lat and lon
for time, lat, lon in list_rand:
 for var in [ 'A', 'B', 'C' ]:
   data = NetCDF[var][time, lat + img_len, lon + img_len ]
   acumulate_mean(var, data)
```

```
img_len = 64
list_rand = 6000 random values of time, lat and lon
for var in [ 'A', 'B', 'C' ]:
 for time, lat, lon in list_rand:
   data = NetCDF[var][time, lat + img_len, lon + img_len ]
   acumulate_mean(var, data)
```

RANDOMNESS

- Arbitrary access to data is very slow compared to sequential.
- Some applications requires random order of values.
- Order of data access **is independent** from application data consumption.
- Better usage of **cache** mechanisms (spatial locality)

RANDOMNESS




```
img_len = 64
list_rand = 6000 random values of time, lat and lon
for var in [ 'A', 'B', 'C' ]:
  for time, lat, lon in list_rand:
    data = NetCDF[var][time, lat + img_len, lon + img_len ]
    acumulate_mean(var, data)
```

```
img_len = 64
list_rand = 6000 random values of time, lat and lon
list_seq, permutation = order_list(list_rand)
for var in [ 'A', 'B', 'C' ]:
  for time, lat, lon in list_seq :
    data = NetCDF[var][time, lat + img_len, lon + img_len ]
    acumulate_mean(var, data)
```

FINAL RESULTS

- Storage
  - Original 1853 MB (**100,00 %**)
  - Optimized **553** MB (**29,84 %**)
- Performance
  - Original 1030,2s
  - Optimized **7,4s** (**~x139 faster**)

CONCLUSIONS

- Find the best **type of data** for the values you will store.
- A **bad chunking** have a big impact in the performance.
- Choosing the best **chunking** is a **complex problem.**
- Always try to access the data as much **sequential** as possible.

- The « ***NetCDF: Performance and Storage Optimization of Meteorological Data*** » contain more details and reproducible example.

Merci de votre attention