Understanding Data Chunking

Chunking demo for Edu Pilot

Lina M. Estupinan-Suarez Felix Cremer
Fabian Gans

2023-04-28

Table of contents

Introduction **1**

Data exploration **2**
Visual inspection 3
Chunks overview 5
Data access performance 7

Statistical computation **8**
Mean 9
Mean by pixel 9
Mean by time step 10
Median 11
Median by pixel 11

Final remarks **14**

Introduction

In this notebook you will learn how the chunking of large gridded data sets affects the reading and processing speed depending on data chunking and on the access patterns you need for your analysis.

To start, chunking is important when working with datasets that do not fit into memory which is often the case for climate and remote sensing data. This can severely limit the computation performance of data analysis due to the time to access, load and process the data. To learn more about hardware and the speed implications see: https://biojulia.github.io/post/hardware/

Task: Use your favorite NetCDF package and method to compute the a) mean and b) median per spatial pixel for the *air_temperature_2m* variable without loading the whole data set into memory (7 GB uncompressed file).

For this tutorial, we will use two different chunk sizes of the same *air_temperature_2m* data set which has three dimensions (i.e., longitude, latitude, time). The files and chunks are:

- `t2_map.nc`: This chunked file setting aims at fast access to spatial layers i.e., grids in latitude and longitude
- `t2_blocks.nc`: This chunked file setting aims at an intermediate access to both the spatial and temporal dimensions (box chunking)

Data exploration

Let's launch Julia and explore the data

```
# load environment
using Pkg
Pkg.activate(".")
Pkg.instantiate()

# load libraries
using Pkg.Artifacts
using NetCDF # for loading of NetCDF data
using DiskArrays: eachchunk # for exploring the chunking of the data
using DelimitedFiles # for handling delimited files like cs

# import code for plotting maps and figures
include("plots4chunking.jl")
```

```
# to access data in the cloud
filebase = artifact"example_nc"

# load files metadata
xmap0 = NetCDF.open(joinpath(filebase,"t2m_map.nc"))
xbox0 = NetCDF.open(joinpath(filebase,"t2m_blocks.nc"));

# load data indices
xmap = xmap0["air_temperature_2m"];
xbox = xbox0["air_temperature_2m"];
```

```
  Activating project at `~/nfdi4earth/chunking_tutorial/v002/scripts`
```

```
WebIO._IJuliaInit()
```

Visual inspection

To get more familiar with the data let's do a few plots. First, let's observe air temperature at 2 m for one day at the global scale.

```
# map of air temperature at 2 m for the first time step
# calling a function from the code imported previously
geoplotsfx(xmap[:,:,1])
```

```
  Warning: Could not find font regular, using TeX Gyre Heros Makie
  @ Makie ~/.julia/packages/Makie/Iqcri/src/conversions.jl:983
```

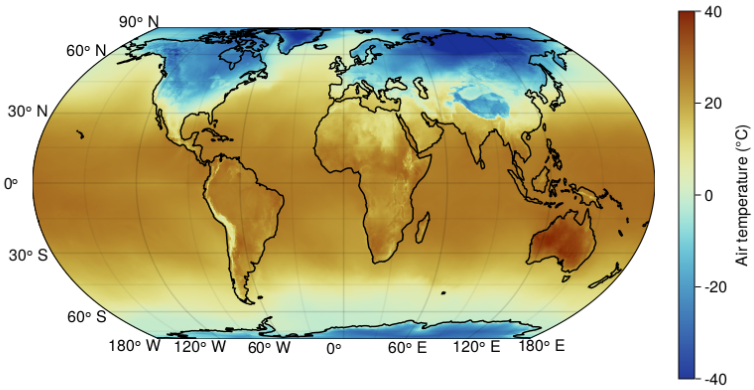


Figure 1: Global map of air temperature at 2 m on 05-Apr-1979.

Now, let's observe a time series of air temperature from 2012 to 2021 at the crossing between the equator and the prime meridian.

```
# libraries for plotting
using CFTime, Dates

# load time axis
xmap0["time"].atts
timevalues = timedecode(xmap0["time"][:],xmap0["time"].atts["units"]);

# subset dates for plotting and assing the dates format
dict1 = Dict(Dates.value(x) => Dates.format(x, "u-yy") for x in timevalues)
tsub = timevalues[1519:end] # subset for the selected years
tax = Dates.value.(tsub)
tax2 = map(x->dict1[x],tax);
```

```
# time series of air temperature at 2 m for
# a pixel at 0° N 0° E
```

```
# define input data and y-axis limits
timeseriesfx(xmap[720,360,1519:end], 20, 30)
```

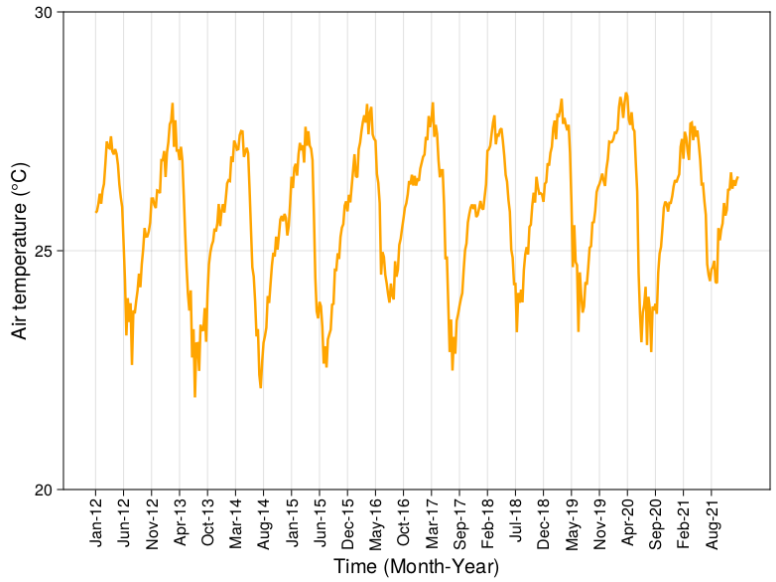


Figure 2: Time series of air temperature at 2 m at at 0° N 0° E from Jan-2012 to Dec-2021.

Chunks overview

Now, we will access two files for the same *air_temperature_2m* data set, but with different chunk settings to explore and compare their properties.

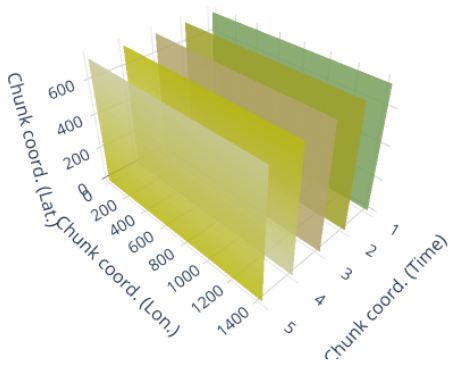
```
# with the 'eachchunk' command we visualize
# the index range of each chunk
chunk1 = eachchunk(xmap) # spatial chunk
size1 = first(chunk1)
chunk2 = eachchunk(xbox); # box chunk
size2 = first(chunk2)
print("The indices for the first spatial chunk are", size1)
println("\nThe ndices for the first box chunk are", size2)
```

```
The indices for the first spatial chunk are(1:1440, 1:720, 1:1)
The ndices for the first box chunk are(1:90, 1:90, 1:256)
```

As we notice, the spatial chunk size is 1440x720x1 which corresponds to the dimensions of one map layer. On the other hand, the box chunk is stored in small blocks with the following dimensions 90x90x256[1] As we will discover later, these storage settings have different implications for the computation speed.

A graphical representation of the spatial chunk is below.

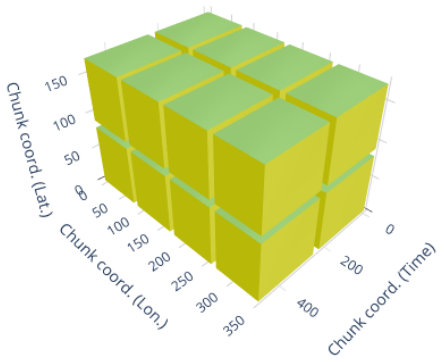
```
# call function for plotting box chunks
spatialchunkfx(chunk1)
```



Similarly, we can also represent a few blocks of the box chunk file in the next plot.

```
# import function for plotting box chunks
boxchunkfx(chunk2)
```

[1] The spatial chunk's size is based on the spatial grid's size. In this example the grid is 720 x 1440 pixels (a map layer). Conversely, The box chunk's size is more flexible and determined by considering the target analyses.



Data access performance

Now, let's estimate the time requiered to access data along different axes for each chunk storage.

```
# spatial access (one map layer)
@time xmap[:,:,1];

# temporal access (time series)
@time xmap[1,1,:];
```

```
  0.019205 seconds (43 allocations: 3.957 MiB)
```

```
 38.290645 seconds (210.81 k allocations: 11.063 MiB, 0.50% compilation time)
```

As expected, for the spatial chunk (xmap), access along spatial strides is much faster than access to time series because of the internal storage in the NetCDF file. In this particular case we can access a map layer a few hundred times faster than a time series even though the map stores much more data than the single time series.

```julia
# spatial access (one map layer)
@time xbox[:,:,1];

# temporal access (time series)
@time xbox[1,1,:];
```

```
4.760620 seconds (43 allocations: 3.957 MiB)
```

```
0.339872 seconds (41 allocations: 9.453 KiB)
```

For the intermediate chunk (xbox), there is a good compromise between accessing the spatial and temporal axes. In this case, access to the temporal axis is faster than access to the spatial axis. These intermediate chunks are prefered when performing analyses in all axes.

> Take home message (1)
>
> In summary, the time required to access geospatial data and time series varies depending on the characteristics of the chunks in the dataset. In this example, we found that:
>
> - Spatial chunking can access spatial layers about a hundred times faster than time series.
> - The box chunk provides a good trade-off when analyses are required across all axes.

Statistical computation

Now, we want to compute the mean and median values for both chunks and across differente axes. Keep in mind that the computational resources needed for the mean and median are different. Specifically, the mean is a cumulative computation that does not require to load the entire data into memory. Conversely, the median needs to load and sort all data to be computed. We will use the same input variable

air_temperature_2m variable with the two different chunk settings.

```
# input chunks
xmap # spatial chunking
xbox; # box chunking
```

Mean

Our input data is stored on disk and accessed as DiskArrays. The `DiskArrays.jl` package uses the internal chunking structure and provides special implementations for the `mapreduce` function used in the implementation of the mean for `AbstractArray`.

Mean by pixel

```
using Statistics
@time xmean1 = mean(xmap, dims=3);

@time xmean2 = mean(xbox,dims=3);
```

```
42.760523 seconds (4.50 M allocations: 7.869 GiB, 0.64% gc time, 4.37% compilation time)
```

```
44.082409 seconds (50.26 k allocations: 7.646 GiB, 0.35% gc time)
```

Note that the computational time of the mean across all dimensions is similar regardless of the chunking.

Next, we plot the output of our computation.

```
geoplotsfx(xmean1[:,:])
```

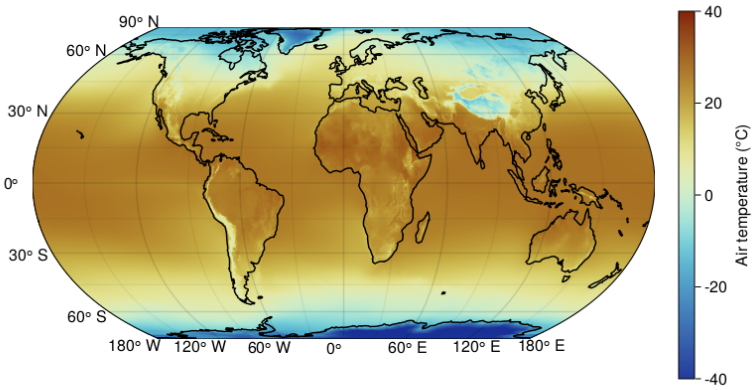


Figure 3: Multianual mean of air temperature at 2 m from 1979 to 2021 pixel-wise.

Mean by time step

Our next step is to compute the global mean per time step.

```
@time tmean1 = mean(xbox,dims=(1,2));

@time tmean2 = mean(xmap, dims=(1,2));
```

```
40.884688 seconds (670.62 k allocations: 7.672 GiB, 0.77% gc time, 0.84% compilation time)

38.877180 seconds (100.96 k allocations: 7.644 GiB, 0.36% gc time)
```

```
# time series of air temperature at 2 m for all pixels
timeseriesfx(tmean1[1519:end], 0, 10) # define input data and y-axis limits
```

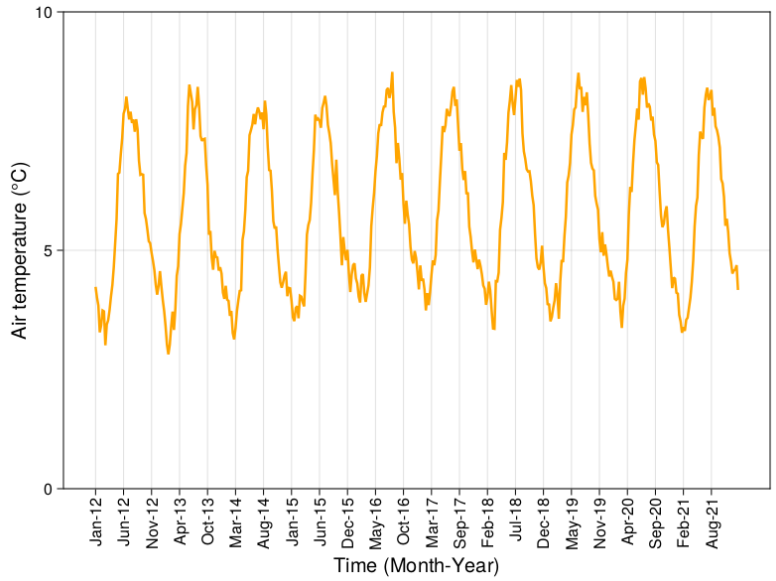


Figure 4: Global mean of time series of air temperature at 2 m from Jan-2012 to Dec-2021.

Take home message (2)

- In the case of the mean, the computation time is similar regardless of the chunking properties and used axes. This is due to the fact that the mean is a cumulative operation and does not need to load the data set along the entire reduction dimension at the same time.
- The mean computation is properly handled by DiskArrays.jl.

Median

Median by pixel

The computation of a reduction gets more difficult for the median, because here we need the full time series in memory. This

makes it impossible to compute the median in a single pass. Let's try this on a small subset.

```
# subset spatial chunking
sub1 = view(xmap,1:2, 1:2,:)
out1 = zeros(size(sub1,1),size(sub1,2));

# Note: this way of reading the data is used for demonstrative purposes,
# but keep in mind, it is a very unefficient looping through the dataset.
# More efficient approaches are mentioned at the end.
@time for ilat in axes(sub1,2), ilon in axes(sub1,1)
    out1[ilon,ilat] = median(sub1[ilon,ilat,:])
end
```

```
137.690002 seconds (2.16 M allocations: 119.666 MiB, 0.55% compilation time)
```

This already takes ages with 4 grid cells when working with spatial chunking. For this calculation it would be better to read e.g. approx. 1 GB of data each time and perform the calculations one after the other as we show later.

```
# subset box chunking
sub2 = view(xbox,1:2, 1:2,:)
out2 = zeros(size(sub2,1),size(sub2,2))

@time for ilat in axes(sub2,2), ilon in axes(sub2,1)
    out2[ilon,ilat] = median(sub2[ilon,ilat,:])
end
```

```
  1.202766 seconds (386 allocations: 77.078 KiB)
```

Regarding the subset with the box chunking, the computation runs much faster, because this chunking is more suitable to access time series. Here we compare and see that both results are exactly the same:

```
out1
```

```
2×2 Matrix{Float64}:
 -48.8034  -49.0315
 -48.8035  -49.0294
```

```
out2
```

```
2×2 Matrix{Float64}:
 -48.8034  -49.0315
 -48.8035  -49.0294
```

One way to deal with these inefficient calculations is to read the data in blocks. This means that a block is read and the calculation immediately follows. In this way, one block after the other is read and calculated until all the data has been read and calculated. In this way, the calculation becomes more efficient.

```
# here we fix latitude ranges that will be used to read the data in blocks
out3 = zeros(size(xmap,1),size(xmap,2))
latsteps = 90
latranges = [(i*90-latsteps+1):(i*90) for i in 1:(720÷latsteps)];
```

```
@time for ilat in latranges
    out3[:,ilat] = median(xmap[:,ilat,:],dims=3)
end
```

```
343.106342 seconds (16.23 M allocations: 8.078 GiB, 0.59% gc time, 0.45% compilation time)
```

```
out4 = zeros(size(xbox,1),size(xbox,2));

@time for ilat in latranges
    out4[:,ilat] = median(xbox[:,ilat,:],dims=3)
end
```

```
 92.808612 seconds (14.08 M allocations: 7.962 GiB, 0.86% gc time)
```

In general, results are obtained from the entire dataset in a reasonable time for both chunking settings. Nevertheless, we find that chunking by boxes is again more efficient than chunking by maps. Alternatively, we can use YAXArrays.jl, which performs exactly this workflow for a given cache size (see last section).

```
geoplotsfx(out3[:,:])
```

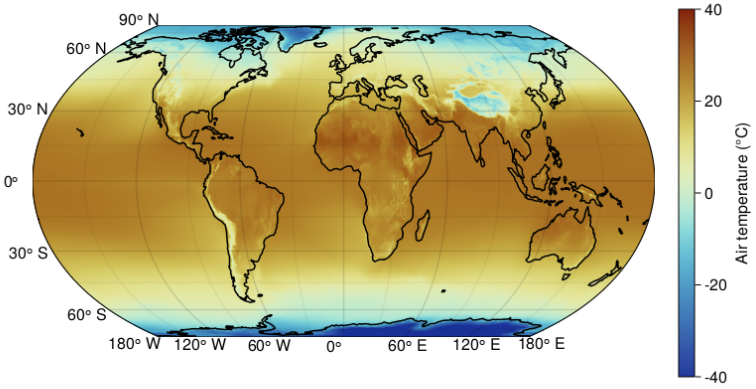


Figure 5: Multiannual median of air temperature at 2 m from 1979 to 2021 pixel-wise.

Final remarks

Take home message (3)

Our last remarks are:

- Chunking is critical for efficient data access when the entire data set cannot be loaded into memory.
- Calculations such as the mean are not affected by

chunking if an appropriate library and code are used.
- To ensure optimal performance for all operations an appropriate chunking size should be chosen considering the type of analyses and required dimensions.
- For a given dataset rechunking is only feasible if the rechunked data needs to be accessed multiple times.

As a final **note**, there are already libraries that efficiently deal with data partitioning and processing, one example is YAXArrays (https://github.com/JuliaDataCubes/YAXArrays.jl). These libraries contribute significantly to improve processing performance but full efficieny is only achieved when considering chunking.

A short syntax for the median example using YAXArrays is:

```
ds = open_dataset(joinpath(filebase,"t2_map.nc"))
ds.air_temperature_2m
medtair = mapslices(median, ds.air_temperature_2m, dims="Time",
 max_cache=1e9)
```