esm-analysis

Release 0.0.2

Martin Bergemann

Jun 15, 2020

CONTENTS

esm-analysis is a Python 3 library for accessingand working with output data from various *Earth System Models*. It has been developed to process output data from global storm resolving simulations at the Max-Planck-Institute for Meteorology.

Note: The source code is available via GitHub.

CHAPTER ONE

INSTALLATION

The crrent master branch can be installed from the GitHub repository:

```
pip install git+https://github.com/antarcticrainforest/esm_analysis.git
```

CHAPTER

TWO

CONTENTS

2.1 Reading data files

You will normally access data from a model experiement, which is stored in a directory containing netcdf files. You can load the **meta-data** which is associated with such an experiement by calling `RunDirectory()`. Applying `RunDirectory()` only loads meta data, that is which model, who many data files are present and any other meta-data that is important for the experiment.

class esm_analysis.**RunDirectory**(*run_dir*, ***, *prefix=None*, *model_type=None*, *overwrite=False*, *f90name_list=None*, *filetype='nc'*, *client=None*)

Open data in experiment folder.

__init__(*run_dir*, ***, *prefix=None*, *model_type=None*, *overwrite=False*, *f90name_list=None*, *filetype='nc'*, *client=None*)

Create an RunDirecotry object from a given input directory.

```
run = RunDirectory('/work/mh0066/precip-project/3-hourly/CMORPH')
```

The RunDirectory object gathers all nesseccary information on the data that is stored in the run directory. Once loaded the most important meta data will be stored in the run directory for faster access the second time.

Parameters

- **run_dir** (*str*) – Name of the directory where the data that should be read is stored.
- **prefix** (*str, optional (default: None)*) – filname prefix
- **model_type** (*str, optional (default: None)*) – model name/ observation porduct that created the data. This will be used to generate a variable lookup table. This can be useful for loading various model datasets and comparing them while only accessing the data with one set of variable names. By default no lookupt table will be generated.
- **overwrite** (*bool, optional (default : False)*) – If true the meta data will be generated again even if it has been stored to disk already.
- **f90name_list** (*str, optional (default: None)*) – Filename to an optional f90 namelist with additional information about the data
- **filetype** (*str, optional (default: nc)*) – Input data file format
- **client** (*dask.distributed cleint, optional (default: None)*) – Configuration that is used the create a dask client which recieves tasks for multiprocessing. By default (None) a local client will be started.

run_dir

The name of the directory that has been loaded

files
Apply a given function to the dataset via the dask scheduling client

close_client()
Close the opened dask client.

restart_client()
Restart the opened dask client.

status
Returns the status of the associated dask worker client

remap(*grid_description*, *inp=None*, *out_dir=None*, ***, *method='weighted'*, *weightfile=None*, *options='-f nc4'*, *grid_file=None*)
Regrid to a different input grid.

```
run.remap('echam_griddes.txt', method='remapbil')
```

Parameters

- **grid_description** (*str*) – Path to file containing the output grid description
- **inp** (*(collection of) str, xarray.Dataset, xarray.DataArray*) – Filenames that are to be remapped.
- **out_dir** (*str (default: None)*) – Directory name for the output
- **weight_file** (*str (default: None)*) – Path to file containing grid weights
- **method** (*str (default: weighted)*) – Remap method that is applyied to the data, can be either weighted (default), bil, con, laf, nn. If weighted is chosen this class should have been instanciated either with a given weightfile or using the gen_weights methods.
- **weightfile** (*str (default: None)*) – File containing the weights for the distance weighted remapping.
- **grid_file** (*str (default: None)*) – file containing the source grid describtion
- **options** (*str (default: -f nc4)*) – additional file options that are passed to cdo

Returns Collection of output

Return type (str, xarray.DataArray, xarray.Dataset)

static apply_function(*mappable*, *collection*, ***, *args=None*, *client=None*, ***kwargs*)
Apply function to given collection.

```
result = run.apply_function(lambda d, v: d[v].sum(dim='time'),
                            run.dataset, args=('temp',))
```

Parameters

- **mappable** (*method*) – method that is applied
- **collection** (*collection*) – collection that is distributed in a thread pool
- **args** – additional arguments passed into the method
- **client** (*dask distributed client (default: None)*) – worker scheduler client that submits the jobs. If None is given a new client is started
- **progress** (*bool (default: True)*) – display tqdm progress bar

- ****kwargs** (*optional*) – additional keyword arguments controlling the progress bar parameter

Returns **combined output of the thread-pool processes**

Return type collection

close_client()

Close the opened dask client.

property files

Return all files that have been opened.

classmethod gen_weights(*griddes*, *run_dir*, ***, *prefix=None*, *model_type='ECHAM'*, *infile=None*, *overwrite=False*, *client=None*)

Create grid weigths from grid description and instanciate class.

```
run = RunDirectory.gen_weights('echam_grid.txt',
                '/work/mh0066/precip-project/3-hourly/CMORPH/',
                infile='griddes.nc')
```

Parameters

- **griddess** (*str*) – filename containing the desired output grid information
- **run_dir** (*str*) – path to the experiment directory
- **prefix** (*str*) – filename prefix
- **model_type** (*str*) – Model/Product name of the dataset to be read
- **infile** (*str*) – Path to input file. By default the method looks for appropriate inputfiles
- **overwrite** (*bool, optional (default: False)*) – should an existing weight file be overwritten

Returns **RunDirectory**

Return type RunDirectory object

load_data(*filenames=None*, ***kwargs*)

Open a multifile dataset using xrarray open_mfdataset.

```
dset = run.load_data('*2008*.nc')
```

Parameters

- **filenames** (*collection/str*) – collection of filenames, filename or glob pattern for filenames that should be read. Default behavior is reading all dataset files
- ****kwargs** (*optional*) – Additional keyword arguments passed to xarray's open_mfdataset

Returns **Xarray (multi-file) dataset**

Return type xarray.Dataset

remap(*grid_description*, *inp=None*, *out_dir=None*, ***, *method='weighted'*, *weightfile=None*, *options='-f nc4'*, *grid_file=None*)

Regrid to a different input grid.

```
run.remap('echam_griddes.txt', method='remapbil')
```

Parameters

- **grid_description** (*str*) – Path to file containing the output grid description
- **inp** (*(collection of) str, xarray.Dataset, xarray.DataArray*) – Filenames that are to be remapped.
- **out_dir** (*str (default: None)*) – Directory name for the output
- **weight_file** (*str (default: None)*) – Path to file containing grid weights
- **method** (*str (default: weighted)*) – Remap method that is applyied to the data, can be either weighted (default), bil, con, laf, nn. If weighted is chosen this class should have been instanciated either with a given weightfile or using the gen_weights methods.
- **weightfile** (*str (default: None)*) – File containing the weights for the distance weighted remapping.
- **grid_file** (*str (default: None)*) – file containing the source grid describtion
- **options** (*str (default: -f nc4)*) – additional file options that are passed to cdo

Returns **Collection of output**

Return type (str, xarray.DataArray, xarray.Dataset)

restart_client()
Restart the opened dask client.

property run_dir
Get the name of the experiment path.

property status
Query the status of the dask client.

2.1.1 Loading the Data

Creating an instance of the `RunDirecotry()` object won't load any data. To get access to the netcdf data the `load_data()` method has to be apply

class esm_analysis.**RunDirectory**

load_data(*filenames=None*, ***kwargs*)
Open a multifile dataset using xrarray open_mfdataset.

```
dset = run.load_data('*2008*.nc')
```

Parameters

- **filenames** (*collection/str*) – collection of filenames, filename or glob pattern for filenames that should be read. Default behavior is reading all dataset files
- ****kwargs** (*optional*) – Additional keyword arguments passed to xarray's open_mfdataset

Returns **Xarray (multi-file) dataset**

Return type xarray.Dataset

dataset
xarray dataset that contains the model data

remap (*grid_description*, *inp=None*, *out_dir=None*, ***, *method='weighted'*, *weightfile=None*, *options='-f nc4'*, *grid_file=None*)
Regrid to a different input grid.

```
run.remap('echam_griddes.txt', method='remapbil')
```

Parameters

- **grid_description** (*str*) – Path to file containing the output grid description
- **inp** (*(collection of) str, xarray.Dataset, xarray.DataArray*) – Filenames that are to be remapped.
- **out_dir** (*str (default: None)*) – Directory name for the output
- **weight_file** (*str (default: None)*) – Path to file containing grid weights
- **method** (*str (default: weighted)*) – Remap method that is applyied to the data, can be either weighted (default), bil, con, laf, nn. If weighted is chosen this class should have been instanciated either with a given weightfile or using the gen_weights methods.
- **weightfile** (*str (default: None)*) – File containing the weights for the distance weighted remapping.
- **grid_file** (*str (default: None)*) – file containing the source grid describtion
- **options** (*str (default: -f nc4)*) – additional file options that are passed to cdo

Returns **Collection of output**

Return type (str, xarray.DataArray, xarray.Dataset)

static apply_function (*mappable*, *collection*, ***, *args=None*, *client=None*, ***kwargs*)
Apply function to given collection.

```
result = run.apply_function(lambda d, v: d[v].sum(dim='time'),
                            run.dataset, args=('temp',))
```

Parameters

- **mappable** (*method*) – method that is applied
- **collection** (*collection*) – collection that is distributed in a thread pool
- **args** – additional arguments passed into the method
- **client** (*dask distributed client (default: None)*) – worker scheduler client that submits the jobs. If None is given a new client is started
- **progress** (*bool (default: True)*) – display tqdm progress bar
- ****kwargs** (*optional*) – additional keyword arguments controlling the progress bar parameter

Returns **combined output of the thread-pool processes**

Return type collection

classmethod gen_weights(*griddes, run_dir, *, prefix=None, model_type='ECHAM', infile=None, overwrite=False, client=None*)

Create grid weigths from grid description and instanciate class.

```
run = RunDirectory.gen_weights('echam_grid.txt',
                '/work/mh0066/precip-project/3-hourly/CMORPH/',
                infile='griddes.nc')
```

Parameters

- **griddess** (*str*) – filename containing the desired output grid information
- **run_dir** (*str*) – path to the experiment directory
- **prefix** (*str*) – filename prefix
- **model_type** (*str*) – Model/Product name of the dataset to be read
- **infile** (*str*) – Path to input file. By default the method looks for appropriate inputfiles
- **overwrite** (*bool, optional (default: False)*) – should an existing weight file be overwritten

Returns **RunDirectory**

Return type RunDirectory object

2.2 Useful methods

The `progress_bar` method gives you the ability to get some feedback while processing data. It brings together the functionality of `tqdm` and `dask-distributed`.

`esm_analysis.`**progress_bar**(**futures, **kwargs*)

Connect dask futures to tqdm progressbar.

The probress_bar method gives you the ability to get some feedback while processing data.

```
from dask.distributed import Client
dask_client = Client()
futures = dask_client.map(lambda x: x*2, [0, 2, 4, 6])
progress_bar(futures)
Progress: 100%|| 4.00/4.00 [00:00<00:00, 487it/s]
results = dask_client.gather(results)
```

Parameters

- **futures** (*collection*) – collections of (dask, concurrent) futures
- **notebook** (*bool, optional (default: False)*) – whether or not to display a progress bar optimized for jupyter notebooks
- **label** (*str, optional (default: Progress)*) – Title of the progress bar
- **kwargs** – Additional keyword arguments passed to the tqdm object

Returns **collection**

Return type futures

`esm_analysis.`**`icon2datetime`**(*icon_dates*)

Convert datetime objects in icon format to python datetime objects.

```
time = icon2datetime([20011201.5])
```

Parameters **`icon_dates`** (*`collection`*) – Collection of icon date dests

Returns **dates**

Return type pd.DatetimeIndex

2.3 Variable Calculation

The `Calculator` sub module offers some methods to calculated common variables like relative humidity.

`class` `esm_analysis.`**`Calculator`**

`calc_rh`(*temp*, *pres*, *temp_unit='K'*, *pres_unit='hPa'*)

Calculate Realtive Humidity.

Parameters

- **`q`** (*`float, nd-array`*) – Specific humidity that is taken to calculate the relative humidity
- **`temp`** (*`float, nd-array`*) – Temperature that is taken to calculate the relative humidity
- **`pres`** (*`float, nd-array`*) – Pressure that is taken to calculate the relative humidity
- **`temp_unit`** (*`str, optional (default: K)`*) – Temperature unit (C: Celsius, K: Kelvin)
- **`pres_unit`** (*`str, optional (default: hPa)`*) – Pressure unit (hPa: ha Pascal, Pa: Pascal)

Returns **Relative Humidity in percent**

Return type float/nd-array

`calc_sathum`(*pres*, *temp_unit='K'*, *pres_unit='hPa'*)

Calculate Saturation Humidity.

Parameters

- **`temp`** (*`float, nd-array`*) – Temperature that is taken to calculate the sat. humidity
- **`pres`** (*`float, nd-array`*) – Pressure that is taken to calculate the sat. humidity
- **`temp_unit`** (*`str, optional (default: K)`*) – Temperature unit (C: Celsius, K: Kelvin)
- **`pres_unit`** (*`str, optional (default: hPa)`*) – Pressure unit (hPa: ha Pascal, Pa: Pascal)

Returns **Saturation Humidity**

Return type float/nd-array

`calc_satpres`(*unit='K'*)

Calculate saturation presure.

Parameters

- **temp** (*float, nd-array*) – Temperature that is taken to calculate the saturation pressure
- **unit** (*str, optional (default: K)*) – Temperature unit (C: Celsius, K: Kelvin)

Returns **Saturation Pressure in hPa**

Return type float/nd-array

2.4 Creating a cluster for distributed processing

esm_analysis supports creating HPC style clusters for distributed data processing using dask-mpi. At the moment only clusters created by the slurm workload manager are supported.

class esm_analysis.**MPICluster**(*script*, *workdir*, *submit_time=None*, *batch_system=None*, *job_id=None*)

Create Cluster of distrbuted workers.

classmethod load(*workdir*)

Load the information of a running cluster.

This method can be used to connect to an already running cluster.

```
from esm_analysis import MPICluster
cluster = MPICluster.load('/tmp/old_cluster')
```

Parameters **workdir** (*str*) – Directory name where information of the previously created cluster is stored. The information on the work directory can be retrieved by calling the workdir property

Returns **Instance of the MPICluster object**

Return type *esm_analysis.MPICluster*

classmethod slurm(*account*, *queue*, ***, *slurm_extra=['']*, *memory='140G'*, *workdir=None*, *walltime='01:00:00'*, *cpus_per_task=48*, *name='dask_job'*, *nworkers=1*, *job_extra=None*)

Create an MPI cluster using slurm.

This method sets up a cluster with help of the workload manager slurm.

```
from esm_analysis import MPICluster
cluster = MPICluster.slurm('account', 'express', nworkers=10)
```

The jobs will immediately be submitted to the workload manager upon creation of the instance.

Parameters

- **account** (*str*) – Account name
- **queue** (*str*) – partition job should be submitted to
- **walltime** (*str, optional (default: '01:00:00')*) – lenth of the job
- **name** (*str, optional (default: dask_job)*) – name of the job
- **workdir** (*str, optional (default: None)*) – name of the workdirectory, if None is given, a temporary directory is used.

- **cpus_per_task** (*int, optional (default: 48)*) – number of cpus per node
- **memory** (*str, optional (default: 140G)*) – allocated memory per node
- **nworkers** (*int, optional (default: 1)*) – number of nodes used in the job
- **job_extra** (*str, optional (default: None)*) – additional commands that should be executed in the run sript
- **slurm_extra** (*list, optional (default: None)*) – additional slurm directives

Returns Instance of the MPICluster object

Return type *esm_analysis.MPICluster*

job_script
A representation of the job script that was submitted

submit_time
datetime.datetime ojbect representing the time the job script was submitted

workdir
The working directory that was used to submit the job to the cluster

job_id
The Id of the submitted job script

2.5 Processing big model output data

This notebook serves as a stack of examples of how to make you of various `python` based data processing libraries to process *large* data from high resolution model output.

In this notebook we are going to process data from the *Dyamond Winter* project as an example. The data is available on mistral hence the notebook should be applied in a *mistral* computing environment to access the data.

2.5.1 Installing Libraries:

To be able to import all libraries that a necessary to run this notebook install the `esm_analysis` repository:

```
python3 -m pip install git+https://github.com/antarcticrainforest/esm_analysis.git
```

2.5.2 Import Libraries:

[1]:
```
from getpass import getuser # Libaray to copy things
from pathlib import Path # Object oriented libary to deal with paths
from tempfile import NamedTemporaryFile, TemporaryDirectory # Creating temporary␣
↪Files/Dirs
from subprocess import run, PIPE

from cartopy import crs as ccrs # Cartography library
import dask # Distributed data libary
from dask_jobqueue import SLURMCluster # Setting up distributed memories via slurm
from distributed import Client, progress, wait # Libaray to orchestrate distributed␣
↪resources
from hurry.filesize import size as filesize # Get human readable file sizes
```

(continues on next page)

(continued from previous page)

```
from matplotlib import pyplot as plt # Standard Plotting library
from metpy import calc as metcalc # Calculate atmospheric variables
from metpy.units import units as metunits # Easy to use meteorological units
import numpy as np # Standard array library
import pandas as pd # Libary to work with labeled data frames and time series
import seaborn as sns # Makes plots more beautiful
import xarray as xr # Libary to work with labeled n-dimensional data and dask
```

2.5.3 2.0 Setup a distributed computing cluster where we can process the data:

The data we are going to process is in the order of TB. On a single machine using a single core this can not only be slow but also the fields we are trying to process won't fit into a single computers memory. Therefore we're setting up a distributed cluster using the `dask_jobqueue` library. Specifically we will involve the *Slurm* workload manager to to that. More information on the `dask_jobqueue` library can be found here: https://jobqueue.dask.org/en/latest/ .

To create the slum cluster we need some information, like the account that is going to be charged and the partition that is going to be used. In this example we are going to use the GPU partition but any other partition can be involved:

```
[2]: # Set some user specific variables
account_name = 'mh0731' # Account that is going to be 'charged' fore the computation
partition = 'prepost' # Name of the partition we want to use
job_name = 'dyamondProc' # Job name that is submitted via sbatch
memory = "200GiB" # Max memory per node that is going to be used - this depends on␣
↪the partition
cores = 42 # Max number of cores per that are reserved - also partition dependend
walltime = '12:00:00' # Walltime - also partition dependen
```

```
[3]: scratch_dir = Path('/scratch') / getuser()[0] / getuser() # Define the users scratch␣
↪dir
# Create a temp directory where the output of distributed cluster will be written to,␣
↪after this notebook
# is closed the temp directory will be closed
dask_scratch_dir = TemporaryDirectory(dir=scratch_dir, prefix='DyamondProc')
cluster = SLURMCluster(memory=memory,
                       cores=cores,
                       project=account_name,
                       walltime=walltime,
                       queue=partition,
                       local_directory=dask_scratch_dir.name,
                       job_extra=[f'-J {job_name}',
                                  f'-D {dask_scratch_dir.name}',
                                  f'--begin=now',
                                  f'--output={dask_scratch_dir.name}/LOG_cluster.%j.o
↪',
                                  f'--output={dask_scratch_dir.name}/LOG_cluster.%j.o'
                                 ],
                       interface='ib0')
```

So far nothing has happened, lets order 10 nodes which will give us 420 cores and 2 TB distributed memory to work on:

```
[4]: cluster.scale(10)
cluster
```

```
VBox(children=(HTML(value='<h2>SLURMCluster</h2>'), HBox(children=(HTML(value='\n<div>
↪\n  <style scoped>\n    ...
```

Now we have submitted two jobs that establish the distributed computing resources. After some queuing time, depending on how busy the computer is. The resources will be available. This can be seen when the above interfaces changes from

Workers 0/10

to

Workers 10

We can also check the status by calling the `squeue` command from bash:

[5]:
```
! squeue -u $USER
```

```
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
          21778712   prepost dyamondP  m300765  R       2:29      1 m11543
          21778713   prepost dyamondP  m300765  R       2:29      1 m11555
          21778714   prepost dyamondP  m300765  R       2:29      1 m11556
          21778715   prepost dyamondP  m300765  R       2:29      1 m11557
          21778716   prepost dyamondP  m300765  R       2:29      1 m11558
          21778717   prepost dyamondP  m300765  R       2:29      1 m11559
          21778718   prepost dyamondP  m300765  R       2:29      1 m11513
          21778719   prepost dyamondP  m300765  R       2:29      1 m11514
          21778711   prepost dyamondP  m300765  R       2:57      1 m11549
          21778710   prepost dyamondP  m300765  R       2:59      1 m11548
```

Now that the computing resources are made available we have to connect a client to it. This client servers as an instance between the commands we are going to use and the cluster. Let's create the client. This can be done by calling the `Client` instance with the cluster we have just created. This will tell dask the distributed library do to all calculations on the cluster.

[6]:
```
dask_client = Client(cluster)
```

If not workers are available yet not computation will be done. Once the workers become available computation can be executed on the available worker only. So in general it is a good idea to wait until as much as possible workers are available and yet more computing resources is can be utilized.

[7]:
```
# Blocking call to wait for at least 10 workers before continuing;
# This might take a while so grab a coffee or tea
dask_client.wait_for_workers(10)
dask_client
```

[7]:
```
<Client: 'tcp://10.50.32.30:41088' processes=10 threads=420, memory=2.15 TB>
```

2.5.4 3.0 Read 2D input data

Let's define the paths and the variables we are going to read. In this example we want to compare two datasets that is:

- dpp0015 : Coupled dyamond run with standard parameter configuration
- dpp0017 : Coupled dyamond run with increased ocean albedo and increased inversion parameter C

In this example we will make use of six hourly data which is store in `<exp>_2d_atm_ml_<timestep>.nc` file patterns.

```
[8]: # Define paths here
paths = {'dpp0015' : Path('/work/mh0287/k203123/GIT/icon-aes-dyw/experiments') /
↪'dpp0015',
         'dpp0018' : Path('/work/mh0287/k203123/GIT/icon-aes-dyw_albW/experiments') /
↪'dpp0018'}
glob_pattern_2d = 'atm_2d_ml'
dpp_runs = list(paths.keys())
```

The runs can be read with the `open_mfdataset` from `xarray`. This methods takes quite a number of arguments. We will try opening all the `<exp>_2d_atm_ml_<timestep>.nc` files and merging them into one big dataset. It should be noted thet `open_mfdataset` doesn't read any data until told. What it does, is just fetching meta data and creating a view to the data.

```
[9]: datasets = {}
for exp in dpp_runs:
    print(f'Reading data from {exp}', end='\r')
    datasets[exp] = xr.open_mfdataset(
        str(paths[exp] / f'{exp}*{glob_pattern_2d}*.nc'),
        combine='by_coords',
        parallel=True,
        chunks={'time': 1}) # The chunking will get important when we read 3d data.
print(f'Done {50*" "}')
```

```
Done
```

```
[10]: # dpp0017 has only data until 2st of February so we take a subset of both datasets:
for exp in dpp_runs:
    datasets[exp] = datasets[exp].sel({'time': slice('2020-01-21T00:00:00', '2020-02-
↪01T18:00:00')})
```

```
[11]: datasets['dpp0015'].time
```

```
[11]: <xarray.DataArray 'time' (time: 48)>
array(['2020-01-21T00:00:00.000000000', '2020-01-21T06:00:00.000000000',
       '2020-01-21T12:00:00.000000000', '2020-01-21T18:00:00.000000000',
       '2020-01-22T00:00:00.000000000', '2020-01-22T06:00:00.000000000',
       '2020-01-22T12:00:00.000000000', '2020-01-22T18:00:00.000000000',
       '2020-01-23T00:00:00.000000000', '2020-01-23T06:00:00.000000000',
       '2020-01-23T12:00:00.000000000', '2020-01-23T18:00:00.000000000',
       '2020-01-24T00:00:00.000000000', '2020-01-24T06:00:00.000000000',
       '2020-01-24T12:00:00.000000000', '2020-01-24T18:00:00.000000000',
       '2020-01-25T00:00:00.000000000', '2020-01-25T06:00:00.000000000',
       '2020-01-25T12:00:00.000000000', '2020-01-25T18:00:00.000000000',
       '2020-01-26T00:00:00.000000000', '2020-01-26T06:00:00.000000000',
       '2020-01-26T12:00:00.000000000', '2020-01-26T18:00:00.000000000',
       '2020-01-27T00:00:00.000000000', '2020-01-27T06:00:00.000000000',
       '2020-01-27T12:00:00.000000000', '2020-01-27T18:00:00.000000000',
       '2020-01-28T00:00:00.000000000', '2020-01-28T06:00:00.000000000',
       '2020-01-28T12:00:00.000000000', '2020-01-28T18:00:00.000000000',
       '2020-01-29T00:00:00.000000000', '2020-01-29T06:00:00.000000000',
       '2020-01-29T12:00:00.000000000', '2020-01-29T18:00:00.000000000',
       '2020-01-30T00:00:00.000000000', '2020-01-30T06:00:00.000000000',
       '2020-01-30T12:00:00.000000000', '2020-01-30T18:00:00.000000000',
       '2020-01-31T00:00:00.000000000', '2020-01-31T06:00:00.000000000',
       '2020-01-31T12:00:00.000000000', '2020-01-31T18:00:00.000000000',
       '2020-02-01T00:00:00.000000000', '2020-02-01T06:00:00.000000000',
       '2020-02-01T12:00:00.000000000', '2020-02-01T18:00:00.000000000'],
```

(continues on next page)

(continued from previous page)

```
      dtype='datetime64[ns]')
Coordinates:
  * time     (time) datetime64[ns] 2020-01-21 ... 2020-02-01T18:00:00
Attributes:
    standard_name:  time
    axis:           T
```

[12]:
```
datasets['dpp0018'].time # This data is daily data only
```

[12]:
```
<xarray.DataArray 'time' (time: 12)>
array(['2020-01-21T00:00:00.000000000', '2020-01-22T00:00:00.000000000',
       '2020-01-23T00:00:00.000000000', '2020-01-24T00:00:00.000000000',
       '2020-01-25T00:00:00.000000000', '2020-01-26T00:00:00.000000000',
       '2020-01-27T00:00:00.000000000', '2020-01-28T00:00:00.000000000',
       '2020-01-29T00:00:00.000000000', '2020-01-30T00:00:00.000000000',
       '2020-01-31T00:00:00.000000000', '2020-02-01T00:00:00.000000000'],
      dtype='datetime64[ns]')
Coordinates:
  * time     (time) datetime64[ns] 2020-01-21 2020-01-22 ... 2020-02-01
Attributes:
    standard_name:  time
    axis:           T
```

[13]:
```
# Merge both datasets in one dataet together to make it easier to work with them:
datasets = xr.concat(list(datasets.values()), dim='exp').assign_coords({'exp':␣
↪list(datasets.keys())})
datasets
```

[13]:
```
<xarray.Dataset>
Dimensions:  (exp: 2, ncells: 20971520, time: 48)
Coordinates:
  * time     (time) datetime64[ns] 2020-01-21 ... 2020-02-01T18:00:00
  * exp      (exp) <U7 'dpp0015' 'dpp0018'
Dimensions without coordinates: ncells
Data variables:
    ps       (exp, time, ncells) float32 dask.array<chunksize=(1, 1, 20971520),␣
↪meta=np.ndarray>
    psl      (exp, time, ncells) float32 dask.array<chunksize=(1, 1, 20971520),␣
↪meta=np.ndarray>
    rsdt     (exp, time, ncells) float32 dask.array<chunksize=(1, 1, 20971520),␣
↪meta=np.ndarray>
    rsut     (exp, time, ncells) float32 dask.array<chunksize=(1, 1, 20971520),␣
↪meta=np.ndarray>
    rsutcs   (exp, time, ncells) float32 dask.array<chunksize=(1, 1, 20971520),␣
↪meta=np.ndarray>
    rlut     (exp, time, ncells) float32 dask.array<chunksize=(1, 1, 20971520),␣
↪meta=np.ndarray>
    rlutcs   (exp, time, ncells) float32 dask.array<chunksize=(1, 1, 20971520),␣
↪meta=np.ndarray>
    rsds     (exp, time, ncells) float32 dask.array<chunksize=(1, 1, 20971520),␣
↪meta=np.ndarray>
    rsdscs   (exp, time, ncells) float32 dask.array<chunksize=(1, 1, 20971520),␣
↪meta=np.ndarray>
    rlds     (exp, time, ncells) float32 dask.array<chunksize=(1, 1, 20971520),␣
↪meta=np.ndarray>
    rldscs   (exp, time, ncells) float32 dask.array<chunksize=(1, 1, 20971520),␣
↪meta=np.ndarray>
```

(continues on next page)

(continued from previous page)

```
    rsus      (exp, time, ncells) float32 dask.array<chunksize=(1, 1, 20971520),␣
↪meta=np.ndarray>
    rsuscs    (exp, time, ncells) float32 dask.array<chunksize=(1, 1, 20971520),␣
↪meta=np.ndarray>
    rlus      (exp, time, ncells) float32 dask.array<chunksize=(1, 1, 20971520),␣
↪meta=np.ndarray>
    ts        (exp, time, ncells) float32 dask.array<chunksize=(1, 1, 20971520),␣
↪meta=np.ndarray>
    sic       (exp, time, ncells) float32 dask.array<chunksize=(1, 1, 20971520),␣
↪meta=np.ndarray>
    sit       (exp, time, ncells) float32 dask.array<chunksize=(1, 1, 20971520),␣
↪meta=np.ndarray>
    clt       (exp, time, ncells) float32 dask.array<chunksize=(1, 1, 20971520),␣
↪meta=np.ndarray>
    prlr      (exp, time, ncells) float32 dask.array<chunksize=(1, 1, 20971520),␣
↪meta=np.ndarray>
    prls      (exp, time, ncells) float32 dask.array<chunksize=(1, 1, 20971520),␣
↪meta=np.ndarray>
    pr        (exp, time, ncells) float32 dask.array<chunksize=(1, 1, 20971520),␣
↪meta=np.ndarray>
    prw       (exp, time, ncells) float32 dask.array<chunksize=(1, 1, 20971520),␣
↪meta=np.ndarray>
    cllvi     (exp, time, ncells) float32 dask.array<chunksize=(1, 1, 20971520),␣
↪meta=np.ndarray>
    clivi     (exp, time, ncells) float32 dask.array<chunksize=(1, 1, 20971520),␣
↪meta=np.ndarray>
    qgvi      (exp, time, ncells) float32 dask.array<chunksize=(1, 1, 20971520),␣
↪meta=np.ndarray>
    qrvi      (exp, time, ncells) float32 dask.array<chunksize=(1, 1, 20971520),␣
↪meta=np.ndarray>
    qsvi      (exp, time, ncells) float32 dask.array<chunksize=(1, 1, 20971520),␣
↪meta=np.ndarray>
    hfls      (exp, time, ncells) float32 dask.array<chunksize=(1, 1, 20971520),␣
↪meta=np.ndarray>
    hfss      (exp, time, ncells) float32 dask.array<chunksize=(1, 1, 20971520),␣
↪meta=np.ndarray>
    evspsbl   (exp, time, ncells) float32 dask.array<chunksize=(1, 1, 20971520),␣
↪meta=np.ndarray>
    tauu      (exp, time, ncells) float32 dask.array<chunksize=(1, 1, 20971520),␣
↪meta=np.ndarray>
    tauv      (exp, time, ncells) float32 dask.array<chunksize=(1, 1, 20971520),␣
↪meta=np.ndarray>
    sfcwind   (exp, time, ncells) float32 dask.array<chunksize=(1, 1, 20971520),␣
↪meta=np.ndarray>
    uas       (exp, time, ncells) float32 dask.array<chunksize=(1, 1, 20971520),␣
↪meta=np.ndarray>
    vas       (exp, time, ncells) float32 dask.array<chunksize=(1, 1, 20971520),␣
↪meta=np.ndarray>
    tas       (exp, time, ncells) float32 dask.array<chunksize=(1, 1, 20971520),␣
↪meta=np.ndarray>
Attributes:
    CDI:                   Climate Data Interface version 1.8.3rc (http://mpim...
    Conventions:           CF-1.6
    number_of_grid_used:   15
    grid_file_uri:         http://icon-downloads.mpimet.mpg.de/grids/public/mp...
    uuidOfHGrid:           0f1e7d66-637e-11e8-913b-51232bb4d8f9
    title:                 ICON simulation
```

(continues on next page)

(continued from previous page)

```
    institution:          Max Planck Institute for Meteorology/Deutscher Wett...
    source:               git@gitlab.dkrz.de:icon/icon-aes.git@6b5726d38970a4...
    history:              /work/mh0287/k203123/GIT/icon-aes-dyw/bin/icon at 2...
    references:           see MPIM/DWD publications
    comment:              Sapphire Dyamond (k203123) on m11338 (Linux 2.6.32-...
```

```
[14]: global_attrs = datasets.attrs
      var_attrs = {varn: datasets[varn].attrs for varn in datasets.data_vars}
```

```
[15]: def set_attrs(dataset):
          '''Set attributes to a dataset that might have lost its attributes.'''
          dataset.attrs = global_attrs
          first_attr = list(var_attrs.values())[0]
          for varn in dataset.data_vars:
              try:
                  dataset[varn].attrs = var_attrs[varn]
              except KeyError:
                  dataset[varn].attrs = first_attr
          return dataset
```

At this stage no data - except for meta data - has been read. Since meta data is present we can inspect the dataset a little further:

```
[16]: datasets.sel({'exp': 'dpp0015'})['ts']
```

```
[16]: <xarray.DataArray 'ts' (time: 48, ncells: 20971520)>
      dask.array<getitem, shape=(48, 20971520), dtype=float32, chunksize=(1, 20971520),␣
      ↪chunktype=numpy.ndarray>
      Coordinates:
        * time     (time) datetime64[ns] 2020-01-21 ... 2020-02-01T18:00:00
          exp      <U7 'dpp0015'
      Dimensions without coordinates: ncells
      Attributes:
          standard_name:              surface_temperature
          long_name:                  surface temperature
          units:                      K
          param:                      0.0.0
          CDI_grid_type:              unstructured
          number_of_grid_in_reference:  1
```

```
[17]: datasets.sel({'exp': 'dpp0015'})['ts'].data
```

```
[17]: dask.array<getitem, shape=(48, 20971520), dtype=float32, chunksize=(1, 20971520),␣
      ↪chunktype=numpy.ndarray>
```

The data attribute returns either data array or if the data hasn't been written yet a representation of what the *will* look like. It is also called a *future*, a very important concept for distributed computing. In our case the representation of the data is a *dask* array. *Dask* is a library that can split up the data into chunks and evenly spreads the data chunks across different cpu's and computers. In our example we can see that the *surface* temperature dataset is split up into 44 chunks. Reading the array would take 935 tasks the total dataset would take 3.69 GB of memory. We can also ask xarray how much memory the whole dataset would consume:

```
[18]: filesize(datasets.nbytes) # Use the filesize module to make the output more readable
```

```
[18]: '270G'
```

This means that the total dataset (both experiments) would need 270 GB of memory. Way to much for a local computer but we do have 1.5 TB of distributed memory. Although this data would comfortably fit there we can reduce its size a little further by throwing out variables we don't need and creating daily averages (the data is 6 hourly).

```
[19]: # Reduce the data a little further, get only interesting variables
      data_vars = ['clivi', 'cllvi', 'clt', 'hfls', 'hfss', 'pr', 'prw', 'ps', 'qgvi', 'qrvi
      ↪',
                   'qsvi', 'rlds', 'rlus', 'rlut', 'rsds', 'rsdscs', 'rsdt', 'rsus', 'rsuscs
      ↪',
                   'rsut', 'rsutcs', 'tas', 'ts', 'uas', 'vas']
      datasets = datasets[data_vars]
      # Create daily average
      datasets = datasets.resample({'time': '1D'}).mean()
      # We lost attributes set the saved once now
      datasets.attrs = global_attrs
      for varn in datasets.data_vars:
          datasets[varn].attrs = var_attrs[varn]
      datasets
```

```
[19]: <xarray.Dataset>
      Dimensions:  (exp: 2, ncells: 20971520, time: 12)
      Coordinates:
        * time     (time) datetime64[ns] 2020-01-21 2020-01-22 ... 2020-02-01
        * exp      (exp) <U7 'dpp0015' 'dpp0018'
      Dimensions without coordinates: ncells
      Data variables:
          clivi    (time, exp, ncells) float32 dask.array<chunksize=(1, 1, 20971520),␣
      ↪meta=np.ndarray>
          cllvi    (time, exp, ncells) float32 dask.array<chunksize=(1, 1, 20971520),␣
      ↪meta=np.ndarray>
          clt      (time, exp, ncells) float32 dask.array<chunksize=(1, 1, 20971520),␣
      ↪meta=np.ndarray>
          hfls     (time, exp, ncells) float32 dask.array<chunksize=(1, 1, 20971520),␣
      ↪meta=np.ndarray>
          hfss     (time, exp, ncells) float32 dask.array<chunksize=(1, 1, 20971520),␣
      ↪meta=np.ndarray>
          pr       (time, exp, ncells) float32 dask.array<chunksize=(1, 1, 20971520),␣
      ↪meta=np.ndarray>
          prw      (time, exp, ncells) float32 dask.array<chunksize=(1, 1, 20971520),␣
      ↪meta=np.ndarray>
          ps       (time, exp, ncells) float32 dask.array<chunksize=(1, 1, 20971520),␣
      ↪meta=np.ndarray>
          qgvi     (time, exp, ncells) float32 dask.array<chunksize=(1, 1, 20971520),␣
      ↪meta=np.ndarray>
          qrvi     (time, exp, ncells) float32 dask.array<chunksize=(1, 1, 20971520),␣
      ↪meta=np.ndarray>
          qsvi     (time, exp, ncells) float32 dask.array<chunksize=(1, 1, 20971520),␣
      ↪meta=np.ndarray>
          rlds     (time, exp, ncells) float32 dask.array<chunksize=(1, 1, 20971520),␣
      ↪meta=np.ndarray>
          rlus     (time, exp, ncells) float32 dask.array<chunksize=(1, 1, 20971520),␣
      ↪meta=np.ndarray>
          rlut     (time, exp, ncells) float32 dask.array<chunksize=(1, 1, 20971520),␣
      ↪meta=np.ndarray>
          rsds     (time, exp, ncells) float32 dask.array<chunksize=(1, 1, 20971520),␣
      ↪meta=np.ndarray>
          rsdscs   (time, exp, ncells) float32 dask.array<chunksize=(1, 1, 20971520),␣
      ↪meta=np.ndarray>
          rsdt     (time, exp, ncells) float32 dask.array<chunksize=(1, 1, 20971520),␣
      ↪meta=np.ndarray>
```

(continues on next page)

(continued from previous page)

```
    rsus      (time, exp, ncells) float32 dask.array<chunksize=(1, 1, 20971520),␣
↪meta=np.ndarray>
    rsuscs    (time, exp, ncells) float32 dask.array<chunksize=(1, 1, 20971520),␣
↪meta=np.ndarray>
    rsut      (time, exp, ncells) float32 dask.array<chunksize=(1, 1, 20971520),␣
↪meta=np.ndarray>
    rsutcs    (time, exp, ncells) float32 dask.array<chunksize=(1, 1, 20971520),␣
↪meta=np.ndarray>
    tas       (time, exp, ncells) float32 dask.array<chunksize=(1, 1, 20971520),␣
↪meta=np.ndarray>
    ts        (time, exp, ncells) float32 dask.array<chunksize=(1, 1, 20971520),␣
↪meta=np.ndarray>
    uas       (time, exp, ncells) float32 dask.array<chunksize=(1, 1, 20971520),␣
↪meta=np.ndarray>
    vas       (time, exp, ncells) float32 dask.array<chunksize=(1, 1, 20971520),␣
↪meta=np.ndarray>
Attributes:
    CDI:                    Climate Data Interface version 1.8.3rc (http://mpim...
    Conventions:            CF-1.6
    number_of_grid_used:    15
    grid_file_uri:          http://icon-downloads.mpimet.mpg.de/grids/public/mp...
    uuidOfHGrid:            0f1e7d66-637e-11e8-913b-51232bb4d8f9
    title:                  ICON simulation
    institution:            Max Planck Institute for Meteorology/Deutscher Wett...
    source:                 git@gitlab.dkrz.de:icon/icon-aes.git@6b5726d38970a4...
    history:                /work/mh0287/k203123/GIT/icon-aes-dyw/bin/icon at 2...
    references:             see MPIM/DWD publications
    comment:                Sapphire Dyamond (k203123) on m11338 (Linux 2.6.32-...
```

[20]:
```
datasets['tas']
```

[20]:
```
<xarray.DataArray 'tas' (time: 12, exp: 2, ncells: 20971520)>
dask.array<concatenate, shape=(12, 2, 20971520), dtype=float32, chunksize=(1, 1,␣
↪20971520), chunktype=numpy.ndarray>
Coordinates:
  * time      (time) datetime64[ns] 2020-01-21 2020-01-22 ... 2020-02-01
  * exp       (exp) <U7 'dpp0015' 'dpp0018'
Dimensions without coordinates: ncells
Attributes:
    standard_name:                tas
    long_name:                    temperature in 2m
    units:                        K
    param:                        0.0.0
    CDI_grid_type:                unstructured
    number_of_grid_in_reference:  1
```

[21]:
```
filesize(datasets.nbytes)
```

[21]:
```
'46G'
```

This means we have significantly reduced the size that will by held in memory. But beware, this is the size that will be occupied in memory once the data has been read. The reading, chunking and averaging process will take much more memory. Although dask tries to optimize the memory consumption as much as possible.

So far no data has been read. We can trigger reading the data by using the `persist` method. Persist will start pushing the data to the distributed memory. There the `netcdf-files` will be read the the data will be copied into memory and the averaging will be done. If no distributed memory is available persist uses the local memory. To use the local memory in the first place one can use the `compute` method. Please also refer to https://distributed.readthedocs.io/en/latest/manage-computation.html#dask-collections-to-futures for more information.

But before we trigger computations on the cluster, we can create some additional datasets. That is net top of the atmosphere radiation budget and net surface energy:

[22]:
```
datasets['net_sw_toa'] = datasets['rsdt'] - datasets['rsut']
datasets['net_lw_toa'] = -datasets['rlut']
datasets['net_sw_surf'] = datasets['rsds'] - datasets['rsus']
datasets['net_lw_surf'] = datasets['rlds'] - datasets['rlus']
datasets['net_toa'] = datasets['net_sw_toa'] + datasets['net_lw_toa']
datasets['net_surf'] = datasets['net_sw_surf'] + datasets['net_lw_surf']
datasets['net_surf_energy'] = datasets['net_surf'] + datasets['hfss'] + datasets['hfls
↪']
```

[23]:
```
# Push the data to the cluster and trigger all computations there
datasets = datasets.persist()
```

This will trigger the computation in the background, we don't have to wait but can keep on analyzing the data. The `progress` class from dask provides the opportunity to get a progress bar that runs in the background to inform the user about the status.

[24]:
```
# Let's inspect the progress
progress(datasets, notebook=True)
```

```
VBox()
```

We can see the tasks that are worked on in the background on the cluster. We can continue working with the data. The paradigm is that all calculations are collected and *not* executed until we explicitly instruct xarray / dask to trigger computations.

2.5.5 3.1 Sub setting the data

Suppose we wanted to work on say tropical areas only we would have to find a way of accessing the right grid triangles and omitting those once outside the tropics. Sub setting the data would be easy if the would be on a lat-lon grid. Then could simply do something like this:

```
data_tropics = datasets.sel({'lat': slice(-30, 30)})
```

Since we are working with unstructured grids things are a little more complex but can be done. Essentially we have two options:

- Creating an index and getting the data via the index.
- Masking the data that we don't need.

Both options involve more information on the grid. Let's read a land-see mask file:

[25]:
```
grid_path = Path('/work/mh0731/m300765/graupel_tuning')
lsm_file = grid_path / 'r02b09_land_frac.nc'
lsm = xr.open_dataset(lsm_file, chunks={'cell': -1}).load() # Load the file into␣
↪local memory first
lons, lats = np.rad2deg(lsm.clon), np.rad2deg(lsm.clat) #Read lon, lat vector
```

The `.load` method loads the data into *local* memory. We can see the difference of the data attribute once the data is in the local memory instead of distributed memory or not loaded at all yet:

```
[26]: lsm['sea'].data
```

```
[26]: array([-0., -0., -0., ...,  1.,  1.,  1.])
```

```
[27]: # Create sea and land masks and push the data to the distributed memory
      s_mask = np.full(lsm['sea'].shape, np.nan) # Creat nan arrays
      l_mask = np.full(lsm['sea'].shape, np.nan)
      l_mask[lsm['land'].data == 1] = 1
      s_mask[lsm['sea'].data == 1] = 1
      s_mask
```

```
[27]: array([nan, nan, nan, ...,  1.,  1.,  1.])
```

```
[28]: # Create a dask array from the sea/land_mask and push it to the cluster
      s_mask = dask.persist(dask.array.from_array(s_mask, chunks=-1))[0] # Push the data to␣
      ↪the cluster
      l_mask = dask.persist(dask.array.from_array(l_mask, chunks=-1))[0]
```

```
[29]: s_mask
```

```
[29]: dask.array<array, shape=(20971520,), dtype=float64, chunksize=(20971520,),␣
      ↪chunktype=numpy.ndarray>
```

```
[30]: l_mask
```

```
[30]: dask.array<array, shape=(20971520,), dtype=float64, chunksize=(20971520,),␣
      ↪chunktype=numpy.ndarray>
```

The most efficient way for unstructured grids is applying masks.The idea here is to mask everything outside the tropics. Let's try indexing the data for tropical values and creating an index using `np.where`

```
[31]: mask_tr = np.full(lats.shape, np.nan) # Create a mask and set the indices where lat␣
      ↪is in [-30, 30] = 1
      idx = np.where((lats >= -30) & (lats <= 30))[0]
      mask_tr[idx] = 1
      mask_tr = dask.persist(dask.array.from_array(mask_tr, chunks=-1))[0]
```

```
[32]: mask_tr
```

```
[32]: dask.array<array, shape=(20971520,), dtype=float64, chunksize=(20971520,),␣
      ↪chunktype=numpy.ndarray>
```

```
[33]: datasets = (datasets * mask_tr).persist()
```

```
[34]: progress(datasets, notebook=True)
```

```
VBox()
```

2.5.6 3.2 Plotting some maps

Lets to some plotting, first we plot maps, hence we create an average along the time axis:

```
[35]: timmean = set_attrs(datasets.mean(dim='time').persist()) # Create mean and trigger␣
      ↪computation
      progress(timmean, notebook=True)
```

```
VBox()
```

This computation is quite fast because we have already loaded the data into distributed memory. Alltogether this only took a little more than 1 minute. But the problem is that we cannot simply plot the data. To do so we are going to remap the data with `cdo`. For the remapping we need a grid file and a target grid description:

```
[36]: # Define the grid describtion
      griddes = '''#
      # gridID 1
      #
      gridtype  = lonlat
      gridsize  = 6480000
      xsize     = 3600
      ysize     = 1800
      xname     = lon
      xlongname = "longitude"
      xunits    = "degrees_east"
      yname     = lat
      ylongname = "latitude"
      yunits    = "degrees_north"
      xfirst    = -179.95
      xinc      = 0.1
      yfirst    = -89.95
      yinc      = 0.1
      '''
      # Write the grid description to a temporary file
      griddes_file = NamedTemporaryFile(dir=scratch_dir, prefix='griddes_', suffix='.txt').
      ↪name
      with Path(griddes_file).open('w') as f: f.write(griddes)
```

```
[37]: #Define the path to the grid-file
      grid_file = grid_path / 'icon_grid_0015_R02B09_G.nc'
```

The best way to remap unstructured data on the resolution is a weighted remap. We do not have a weight file let's create one first. For this we define a function that will call the `cdo gendis` command. To execute the function on the cluster we use the `dask.delayed` decorator, to tell the code it should be executed remotely.

```
[38]: @dask.delayed
      def gen_dis(dataset, grid_file, griddes, scratch_dir, global_attrs={}):
          '''Create a distance weights using cdo.'''
          if isinstance(dataset, xr.DataArray):
              # If a dataArray is given create a dataset
              dataset = xr.Dataset(data_vars={dataset.name: dataset})
              dataset.attrs = global_attrs
          enc, dataset = get_enc(dataset)
          weightfile = NamedTemporaryFile(dir=scratch_dir, prefix='weight_file', suffix='.nc
      ↪').name
          with NamedTemporaryFile(dir=scratch_dir, prefix='input_', suffix='.nc') as tf:
              dataset.to_netcdf(tf.name,  encoding=enc, mode='w')
```

(continues on next page)

(continued from previous page)

```
        cmd = ('cdo', '-O', f'gendis,{griddes}', f'-setgrid,{grid_file}', tf.name,␣
↪weightfile)
        run_cmd(cmd)
        return weightfile

def run_cmd(cmd):
    '''Run a bash command.'''
    status = run(cmd, check=False, stderr=PIPE, stdout=PIPE)
    if status.returncode != 0:
        error = f'''{' '.join(cmd)}: {status.stderr.decode('utf-8')}'''
        raise RuntimeError(f'{error}')
    return status.stdout.decode('utf-8')

def get_enc(dataset, missing_val=-99.99e36):
    """Get encoding to save datasets."""
    enc = {}
    for varn in dataset.data_vars:
        enc[varn] = {'_FillValue': missing_val}
        dataset[varn].attrs = {**dataset[varn].attrs, **{'missing_value': missing_val}
↪}
    return enc, dataset
```

[39]:
```
weightfile = gen_dis(timmean['tas'].isel({'exp':0}),
                     grid_file,
                     griddes_file,
                     scratch_dir,
                     global_attrs).compute()
```

We will remap the data on the the distributed cluster. For this we have to define a function that will remap the data is is executed on the cluster:

[40]:
```
def remap(dataset, grid_file, griddes, weightfile, tmpdir, attrs={}):
    """Perform a weighted remapping.

    Parameters
    ==========

    dataset : xarray.dataset
        The dataset the will be regriddes
    grid_file : Path, str
        Path to the grid file
    griddes : Path, str
        Path to the grid describtion file
    tmpdir : Path, str
        Directory for where temporary fils are stored

    Returns
    =======
    xarray.dataset : Remapped dataset
    """
    if isinstance(dataset, xr.DataArray):
        # If a dataArray is given create a dataset
        dataset = xr.Dataset(data_vars={dataset.name: dataset})
        dataset.attrs = attrs
    enc, dataset = get_enc(dataset)
    try:
```

(continues on next page)

(continued from previous page)

```
        # cdo doesn't like strings as data; temporarly assign the exp coordinates to␣
↪numbers (if exp present)
        exps = dataset.coords['exp'].values
        dataset = dataset.assign_coords({'exp': np.arange(len(exps))})
    except KeyError:
        exps = []
    with TemporaryDirectory(dir=tmpdir, prefix='cdo_') as indir:
            infile = Path(indir) / 'infile.nc'
            outfile = Path(indir) / 'outfile.nc'
            dataset.to_netcdf(infile, encoding=enc, mode='w')
            # Create the command to run
            cmd = ('cdo', '-O', f'remap,{griddes},{weightfile}',
                   f'-setgrid,{grid_file}',
                   str(infile), str(outfile))
            # Get the return value of the command
            run_cmd(cmd)
            try:
                # If there was a exp dimension put its values back into place
                return xr.open_dataset(outfile).load().assign_coords({'exp': exps})
            except KeyError:
                return xr.open_dataset(outfile).load()
```

[41]:
```
# Submit the remap function call to the cluster, lets do every variable in parallel
remap_futures = []
for var_name in datasets.data_vars:
    remap_futures.append(dask_client.submit(remap,
                                            timmean[var_name],
                                            grid_file,
                                            griddes_file,
                                            weightfile,
                                            scratch_dir,
                                            attrs=timmean.attrs))
progress(remap_futures, notebook=True)
```

```
VBox()
```

[42]:
```
# Merge the results from the parllel computing back together into one dataset
dset_remap = xr.merge(dask_client.gather(remap_futures))
dset_remap
```

[42]:
```
<xarray.Dataset>
Dimensions:          (exp: 2, lat: 1800, lon: 3600)
Coordinates:
  * lon              (lon) float64 -179.9 -179.8 -179.8 ... 179.8 179.9 180.0
  * lat              (lat) float64 -89.95 -89.85 -89.75 ... 89.75 89.85 89.95
  * exp              (exp) <U7 'dpp0015' 'dpp0018'
Data variables:
    clivi            (exp, lat, lon) float64 nan nan nan nan ... nan nan nan nan
    cllvi            (exp, lat, lon) float64 nan nan nan nan ... nan nan nan nan
    clt              (exp, lat, lon) float64 nan nan nan nan ... nan nan nan nan
    hfls             (exp, lat, lon) float64 nan nan nan nan ... nan nan nan nan
    hfss             (exp, lat, lon) float64 nan nan nan nan ... nan nan nan nan
    pr               (exp, lat, lon) float64 nan nan nan nan ... nan nan nan nan
    prw              (exp, lat, lon) float64 nan nan nan nan ... nan nan nan nan
    ps               (exp, lat, lon) float64 nan nan nan nan ... nan nan nan nan
    qgvi             (exp, lat, lon) float64 nan nan nan nan ... nan nan nan nan
    qrvi             (exp, lat, lon) float64 nan nan nan nan ... nan nan nan nan
```

(continues on next page)

(continued from previous page)

```
    qsvi              (exp, lat, lon) float64 nan nan nan nan ... nan nan nan nan
    rlds              (exp, lat, lon) float64 nan nan nan nan ... nan nan nan nan
    rlus              (exp, lat, lon) float64 nan nan nan nan ... nan nan nan nan
    rlut              (exp, lat, lon) float64 nan nan nan nan ... nan nan nan nan
    rsds              (exp, lat, lon) float64 nan nan nan nan ... nan nan nan nan
    rsdscs            (exp, lat, lon) float64 nan nan nan nan ... nan nan nan nan
    rsdt              (exp, lat, lon) float64 nan nan nan nan ... nan nan nan nan
    rsus              (exp, lat, lon) float64 nan nan nan nan ... nan nan nan nan
    rsuscs            (exp, lat, lon) float64 nan nan nan nan ... nan nan nan nan
    rsut              (exp, lat, lon) float64 nan nan nan nan ... nan nan nan nan
    rsutcs            (exp, lat, lon) float64 nan nan nan nan ... nan nan nan nan
    tas               (exp, lat, lon) float64 nan nan nan nan ... nan nan nan nan
    ts                (exp, lat, lon) float64 nan nan nan nan ... nan nan nan nan
    uas               (exp, lat, lon) float64 nan nan nan nan ... nan nan nan nan
    vas               (exp, lat, lon) float64 nan nan nan nan ... nan nan nan nan
    net_sw_toa        (exp, lat, lon) float64 nan nan nan nan ... nan nan nan nan
    net_lw_toa        (exp, lat, lon) float64 nan nan nan nan ... nan nan nan nan
    net_sw_surf       (exp, lat, lon) float64 nan nan nan nan ... nan nan nan nan
    net_lw_surf       (exp, lat, lon) float64 nan nan nan nan ... nan nan nan nan
    net_toa           (exp, lat, lon) float64 nan nan nan nan ... nan nan nan nan
    net_surf          (exp, lat, lon) float64 nan nan nan nan ... nan nan nan nan
    net_surf_energy   (exp, lat, lon) float64 nan nan nan nan ... nan nan nan nan
```

xarray has a powerful plot functionality that can be used to quickly plot and inspect the data:

[43]:
```
%matplotlib notebook
# Use the notebook plotting backend from matplotlib
(-dset_remap['rlut']).sel({'exp': 'dpp0015'}).plot.imshow(cmap='RdYlBu_r')
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

[43]:
```
<matplotlib.image.AxesImage at 0x2b5afb8510d0>
```

These plots are not very nice but we can make them much nicer by using `cartopy` to draw maps. Xarray supports subplots and fiddeling with the colorbar. Let's create a nicer plot of net surface energy.

[44]:
```
plot_data = dset_remap['net_surf_energy'].sel({'lat': slice(-30, 30)})
# Create a diff (exp1 - exp2)
plot_exps = list(plot_data.coords['exp'].values)
diff_data = plot_data.diff(dim='exp').assign_coords({'exp': [f'{plot_exps[1]} - {plot_
↪exps[0]}']})

#Let's rename the dimension in diff_data
diff_data = diff_data.rename({'exp': 'diff'})
```

[45]:
```
proj = ccrs.PlateCarree(central_longitude=50.) # Create ylindrical projections
plot = plot_data.plot(transform=ccrs.PlateCarree(),
                      row='exp',
                      figsize=(9,5),
                      cmap='RdYlBu_r',
                      vmin=-170,
                      vmax=170,
                      subplot_kws={'projection': proj},
                      cbar_kwargs={'label': 'Net Surface Energy [W/m$^2$]',
                                   'extend': 'both',
```

(continues on next page)

(continued from previous page)

```
                                             'anchor': (0.5, -0.15),
                                             'fraction': 0.8,
                                             'aspect': 60,
                                             'orientation': 'horizontal'},

                              )
_ = [ax.coastlines() for ax in plot.axes.flat]
plot.fig.subplots_adjust(left=0.01, right=0.99, hspace=0, wspace=0, top=1, bottom=0.2)
plot
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

[45]:
```
<xarray.plot.facetgrid.FacetGrid at 0x2b5afee79890>
```

We could also plot the difference in a similar fashion:

[51]:
```
proj = ccrs.PlateCarree(central_longitude=50.) # Create ylindrical projections
fig = plt.figure(figsize=(9,4.5))
ax = fig.add_subplot(111, projection=proj)
plot = diff_data.plot(ax=ax,
                 transform=ccrs.PlateCarree(), # This is important
                 cmap='RdBu_r',
                 vmin=-80,
                 vmax=80,
                 cbar_kwargs={'label': 'Net Surface Energy [W/m$^2$]',
                              'extend': 'both',
                              'aspect': 60,
                              'orientation': 'horizontal'},

                      )
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

You probably have noticed the `%matplotlib notebook` statements at the beginning of each cell where we plot. This is a so called *cell magic*, specifically it tells the jupyter that we are going to plot with a certain graphical user interface backend. Here we use an interface that allows us interactively change the plot. You can use the controls at the bottom to zoom and pad. You can also change the plot itself:

[52]:
```
# Draw a thin high resolution coastline to the plot above
_ = ax.coastlines(resolution='10m', lw=0.5)
fig.subplots_adjust(left=0.01, right=0.99, hspace=0, wspace=0, top=1, bottom=0.3)
```

2.5.7 3.2 Plotting some time serie

Say we wanted to plot the time series for tropical *ocean* values only. This is easy to achieve we only need to multiply the `dataset` object with the sea mask and calculate a mean over the `ncells` dimension:

[53]:
```
datasets['net_surf_energy']
```

[53]:
```
<xarray.DataArray 'net_surf_energy' (time: 12, exp: 2, ncells: 20971520)>
dask.array<mul, shape=(12, 2, 20971520), dtype=float64, chunksize=(1, 1, 20971520),␣
↪chunktype=numpy.ndarray>
Coordinates:
```

(continues on next page)

(continued from previous page)

```
  * time      (time) datetime64[ns] 2020-01-21 2020-01-22 ... 2020-02-01
  * exp       (exp) <U7 'dpp0015' 'dpp0018'
Dimensions without coordinates: ncells
```

[54]:
```
# Apply the sea mask, and create averge
fldmean = (datasets * s_mask).mean(dim='ncells')
```

Again nothing has been calculated yet, let's trigger the computation in the background on the cluster:

[55]:
```
fldmean = fldmean.persist()
progress(fldmean, notebook=True)
```

```
VBox()
```

The advantage is that the data is already loaded in the distributed memory. So any computations are fast. We can immediately plot the data:

[56]:
```
%matplotlib notebook
_ = fldmean['net_surf_energy'].plot.line(x='time')
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

Let's make this plot a little more pretty. The Seaborn library provides a nice interface for that:

[57]:
```
# Tell seaborn to style the plots:
col_blind = sns.color_palette("colorblind", 10)
sns.set_style('ticks')
sns.set_palette(col_blind)
sns.set_context("notebook", font_scale=1., rc={"lines.linewidth": 2.5})
```

Let's create a subplot with a time series and zonal averages of tropical ocean areas only. To do this we need a land-sea mask on a lat-lon grid which we have to create first.

[58]:
```
# Create an xarray dataset from the s_mask array
mask_sea = xr.DataArray(s_mask, name='sea', dims=('ncells',))
mask_sea
```

[58]:
```
<xarray.DataArray 'sea' (ncells: 20971520)>
dask.array<array, shape=(20971520,), dtype=float64, chunksize=(20971520,),␣
↪chunktype=numpy.ndarray>
Dimensions without coordinates: ncells
```

[59]:
```
# submit the remap job and get immediately the results
mask_sea_lonlat = dask_client.submit(remap,
                                     mask_sea,
                                     grid_file,
                                     griddes_file,
                                     weightfile,
                                     scratch_dir,
                                     attrs=global_attrs).result()
mask_sea_lonlat
```

[59]:
```
<xarray.Dataset>
Dimensions:  (exp: 0, lat: 1800, lon: 3600)
Coordinates:
  * lon      (lon) float64 -179.9 -179.8 -179.8 -179.6 ... 179.8 179.9 180.0
```

(continues on next page)

(continued from previous page)

```
  * lat      (lat) float64 -89.95 -89.85 -89.75 -89.65 ... 89.75 89.85 89.95
  * exp      (exp) float64
Data variables:
    sea      (lat, lon) float64 nan nan nan nan nan nan ... 1.0 1.0 1.0 1.0 1.0
Attributes:
    CDI:          Climate Data Interface version 1.9.8 (https://mpimet.mpg.de...
    Conventions:  CF-1.6
    history:      Sun Jun 14 00:51:01 2020: cdo -O remap,/scratch/m/m300765/g...
    source:       git@gitlab.dkrz.de:icon/icon-aes.git@6b5726d38970a46b3ff1ac...
    institution:  Max Planck Institute for Meteorology
    title:        ICON simulation
    references:   see MPIM/DWD publications
    comment:      Sapphire Dyamond (k203123) on m11338 (Linux 2.6.32-754.14.2...
    CDO:          Climate Data Operators version 1.9.8 (https://mpimet.mpg.de...
```

[60]:
```
zone_avg = (dset_remap*mask_sea_lonlat['sea'].data).sel({'lat':slice(-30, 30)}).
↪mean(dim='lon')
```

[61]:
```
%matplotlib notebook
fig, axs = plt.subplots(1, 2,figsize=(9, 4.5), sharey=False)
_ = fldmean['net_surf_energy'].plot.line(x='time', ax=axs[0], add_legend=False)
axs[0].set_ylabel('Net Surface Energy [W/m$^2$]')
_ = zone_avg['net_surf_energy'].plot.line(x='lat', ax=axs[1])
axs[-1].set_ylabel('')
axs[-1].set_xlabel('Latitude [$^\circ$N]')
fig.subplots_adjust(left=0.1, right=0.99, wspace=0.2, top=.95, bottom=0.3)
sns.despine()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

2.5.8 4.0 Working with 3D data

Working with 3D data can be a little more challenging because chances are high that the data wont fit into memory. So it is important to reduce the data as much as possible. But first lets open some 3D datasets and inspect them:

[62]:
```
# Define the file pattern for qc, qi, pressure and qv
glob_pattern_3d = ('atm_3d_1_ml',  'atm_3d_3_ml', 'atm_3d_4_ml', 'atm_3d_5_ml')
```

[63]:
```
datasets_3d = {exp: [] for exp in paths.keys()}
datasets_3d
# Open each file pattern for every experiment seperatly and merge the data into one␣
↪dataset
for exp in datasets_3d.keys():
    for glob_pattern in glob_pattern_3d:
        datasets_3d[exp].append(
            xr.open_mfdataset(f'{paths[exp] / exp}*{glob_pattern}*.nc',
                              parallel=True,
                              combine='by_coords',
                             chunks={'time': 1, 'height': 5} # This is important
                             ).sel({'time': slice('2020-01-21T00:00:00', '2020-01-
↪31T18:00:00')})
        )
```

(continues on next page)

(continued from previous page)

```
    # Merge the datasets (each for one variable) and select the time where both␣
↪experiments have data
    datasets_3d[exp] = xr.merge(datasets_3d[exp])
```

Note the chunks parameter. This tells xarray to try split up the data into chunks of 1 in time and 5 in height. When dealing with large datasets like here the chunk size is important because each data chunk will be distributed across the cluster memory. If the chunk size is to big (to little chunks) the worker nodes might run out of memory if on the other hand the there are to many chunks (to small chunk size) the cluster might die from communication overhead. In this case we have 77 levels so a chunk size of 5 could be a good choice. Read more about chunking on https://docs.dask.org/en/latest/array-chunks.html

```
[64]: # Check the total size of the datasets
      filesize(np.sum([dset.nbytes for dset in datasets_3d.values()]))
```

```
[64]: '2T'
```

We have 'only' 2.15 TB of memory available so the will barely fit into memory. Since we're just interested in tropical profiles we can reduce the number significantly averaging. The problem is during the data reduction process (averaging) much more memory will we consumed hence more memory. Let's do the average operation one experiment and 4 days at a time. This will take a while but we wont choke the cluster.

```
[66]: for exp, data in datasets_3d.items():
          # Split the data by day:
          days, daily_data = zip(*data.resample({'time': '4d'})) # Split data set into␣
      ↪chunks of 4 days
          print(f'{exp}: Averaging chunk 1/{len(days)}', flush=True)
          dset = daily_data[0].mean(dim='time').persist() # Push the 1st day to the cluster,
      ↪ wait unitl finished
          progress(dset, notebook=False)
          print('\b')
          wait(dset)
          for nn, data in enumerate(daily_data[1:]):
              print(f'{exp}: Averaging chunk {nn+2}/{len(days)}', flush=True)
              tmp_data = data.mean(dim='time').persist()
              wait(tmp_data)
              dset += tmp_data
              del tmp_data
          datasets_3d[exp] = (dset / len(days)).persist()
          del dset # Delete unsused arrays (just in case)
          wait(datasets_3d[exp])
```

```
dpp0015: Averaging chunk 1/3
[########################################] | 100% Completed |  5min 36.9s
dpp0015: Averaging chunk 2/3
[########################################] | 100% Completed |  0.5s
dpp0015: Averaging chunk 3/3
[########################################] | 100% Completed |  0.3s
dpp0018: Averaging chunk 1/3
[########################################] | 100% Completed |  1min 58.4s
dpp0018: Averaging chunk 2/3
[########################################] | 100% Completed |  0.4s
dpp0018: Averaging chunk 3/3
[########################################] | 100% Completed |  0.3s
```

This operation will take some time, so grab another coffee or tea and hope the cluster won't run out of memory.

Experiment dpp0016 has a file that describes the z coordinates lets load it for using it as the new height coordinate later:

```
[190]: z_file = Path('/work/mh0287/k203123/GIT/icon-aes-dyw_albW/experiments/dpp0016/dpp0016_
       ↪atm_vgrid_ml.nc')
       z_data = xr.open_mfdataset([z_file], combine='by_coords', chunks={'height_2': 5,
       ↪'height': 5})['zg']
       z_data.data = (z_data.data - z_data.isel({'height_2': -1}).data + 25)
       z_data = xr.Dataset(data_vars={'zg': z_data}).rename({'height_2':'height'}).persist()
       _ = wait(z_data) # Wait until loaded
```

```
[191]: height = np.round(z_data['zg'].isel({'ncells': 0}).values.round(0)[30:] / 10) * 10
       for exp in datasets_3d.keys():
           datasets_3d[exp]['zg'] = z_data['zg']
```

```
[176]: datasets_3d
```

```
[176]: <xarray.Dataset>
       Dimensions:  (exp: 2, height: 90, ncells: 20971520)
       Coordinates:
           clon     (ncells) float32 dask.array<chunksize=(20971520,), meta=np.ndarray>
           clat     (ncells) float32 dask.array<chunksize=(20971520,), meta=np.ndarray>
         * height   (height) float64 1.0 2.0 3.0 4.0 5.0 ... 86.0 87.0 88.0 89.0 90.0
         * exp      (exp) <U7 'dpp0015' 'dpp0018'
       Dimensions without coordinates: ncells
       Data variables:
           pfull    (exp, height, ncells) float32 dask.array<chunksize=(1, 5, 20971520),␣
       ↪meta=np.ndarray>
           ta       (exp, height, ncells) float32 dask.array<chunksize=(1, 5, 20971520),␣
       ↪meta=np.ndarray>
           wap      (exp, height, ncells) float32 dask.array<chunksize=(1, 5, 20971520),␣
       ↪meta=np.ndarray>
           cl       (exp, height, ncells) float32 dask.array<chunksize=(1, 5, 20971520),␣
       ↪meta=np.ndarray>
           hus      (exp, height, ncells) float32 dask.array<chunksize=(1, 5, 20971520),␣
       ↪meta=np.ndarray>
           clw      (exp, height, ncells) float32 dask.array<chunksize=(1, 5, 20971520),␣
       ↪meta=np.ndarray>
           cli      (exp, height, ncells) float32 dask.array<chunksize=(1, 5, 20971520),␣
       ↪meta=np.ndarray>
```

```
[80]: # Merge the dataset together
      datasets_3d = xr.concat([dset for dset in datasets_3d.values()], dim='exp').assign_
      ↪coords({'exp': dpp_runs})
      datasets_3d = datasets_3d.drop_vars({'height_bnds'}) # Get rid of height_bnds
```

```
[194]: shape = datasets_3d.coords['exp'].shape[0], len(height), datasets_3d.coords['ncells'].
       ↪shape[0]
       shape
```

```
[194]: (2, 60, 20971520)
```

Now we will use the stratify library to interpolate the data to constant z height.

```
[141]: import stratify
```

```
[199]: new_data = {}
       for varn in datasets_3d.data_vars:
           # Do the interpolation on the cluster with dask.delayed
```

(continues on next page)

(continued from previous page)

```
    tmp = dask.delayed(stratify.interpolate)(height, z_data['zg'].data, datasets_
↪3d[varn].data, axis=1)
    # Create an xarray data array from the delayed object
    new_data[varn] = xr.DataArray(dask.array.from_delayed(tmp, shape=shape, dtype=np.
↪float32),
                                  name=varn,
                                  dims=('exp', 'Z', 'ncells'),
                                  coords={'exp':datasets_3d.coords['exp'],
                                          'Z': height}).persist()
progress(new_data, notebook=True)
```

```
VBox()
```

[203]:
```
new_data = xr.Dataset(data_vars=new_data).persist()
```

Let's create some cloud-water/ice and relative humidity profiles for tropical ocean and land here we can also apply the masks.

[205]:
```
data  = {}
data['Land'] = (new_data * mask_tr * l_mask).mean(dim='ncells').persist()
wait(data) # Wait until done, not to choke clusters memory
data['Ocean'] = (new_data * mask_tr * s_mask).mean(dim='ncells').persist()
wait(data)
data['Land&Ocean'] = (new_data * mask_tr).mean(dim='ncells').persist()
progress(data, notebook=True)
```

```
VBox()
```

[206]:
```
# Let's lumb the datasets along a new dimension together
data = xr.concat([dset for dset in data.values()], dim='surf').assign_coords({'surf':
↪list(data.keys())})
data['Z'].attrs = {'standard_name': 'Z', 'units': 'km'}
data = data.load() # Load the data into local memory
```

We want calculate a relative humidity profile. The `metpy` packages offers a lot of calculation routines. You can check what type of calculation is available on their website https://unidata.github.io/MetPy/latest/api/generated/metpy.calc.html . Metpy is handy as it also takes care about units:

[207]:
```
# Calculate relative humidity
rh = metcalc.relative_humidity_from_specific_humidity(
    data['hus'].data * metunits('kg/kg'),
    data['ta'].data * metunits('K'),
    data['pfull'].data * metunits('Pa')
).magnitude
```

[225]:
```
data['rh'] = xr.DataArray(rh*100.,
                          name='rh',
                          coords=data['cl'].coords,
                          dims=data['cl'].dims,
                          attrs={'standard_name': 'RH',
                                 'units': '%'})
data['Z'].data = data['Z'].data / 1000.
```

[228]:
```
%matplotlib notebook
plot = data['rh'].isel({'Z':slice(10, 60)}).plot.line(y='Z',
                                                       col='surf',
```

(continues on next page)

(continued from previous page)

```
                                                        figsize=(8.5,5),
                                                        sharex=False)
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

[229]:
```
%matplotlib notebook
clx = data['clw'] + data['cli']
clx.attrs = {'standard_name': 'cli + clw' , 'units': 'kg/kg'}
plot = clx.isel({'Z':slice(10, 60)}).plot.line(y='Z', col='surf', figsize=(8.5,5),␣
↪sharex=False)
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

2.5.9 Conclusion

This notebook presented some techniques to analyze dyamond datasets in python. We basically utilized `xarray` a very powerful data processing library to work with multi dimensional data. `xarray` builds up on `dask` a library that makes distributed data processing easy.

Specifically we created a cluster of 10 workers with 420 cpu cores and nearly 2.5 TB of memory. This cluster was used to process the dyamond data across the workers. This setup allowed us to use the native grid with its high resolutions as much as possible. Only when plotting maps we had to involve cdo for remapping.

Altogether we have actively have utilized more than 400 GB of memory with peak usages of nearly 1 TB this would have never been possible with a single computer.

There are several pitfalls when it comes to distributed data processing. First it is essential to understand that computations are collected as much as possible rather than triggered immediately. This has the advantage that task streams can be optimized - this technique is called map reduce. Computations should be triggered only at the last moment. Another crucial part is data chunking this is especially true for 3D data. Choosing the right chunk size can be challenging at times but is important in order to keep the clusters memory intact.

The reader might have noticed that reading the data and especially plotting can be convoluted. Hence the next notebook will demonstrate some functionalities that have been added to this repository to make data processing and plotting a little more easy.

[ ]:
```

```

2.6 Basic Usage

This notebook should demonstrate how I try to access and process data more easily. As of now the `esm_analysis` library is just a concept and far from being ready to be deployed. As an example I will try to explain in functionality based on **CMORPH** satellite data:

```
# Import the library
import esm_analysis
```

2.6.1 Creating a Task Schedule

To involve a computing cluster we'll import the `SLURMCluster` from `dask_jobqueue`

```
from dask_jobqueue import SLURMCluster
```

The cluster cluster can be configured according to the desired queue and project

```
cluster = SLURMCluster(queue='gpu', project='mh0287', walltime='03:00:00')
```

```
/mnt/lustre01/work/mh0287/m300765/anaconda3/lib/python3.7/site-packages/
↪distributed/dashboard/core.py:74: UserWarning:
Port 8787 is already in use.
Perhaps you already have a cluster running?
Hosting the diagnostics dashboard on a random port instead.
  warnings.warn("n" + msg)
```

```
cluster
```

```
VBox(children=(HTML(value='<h2>SLURMCluster</h2>'), HBox(children=(HTML(value='n
↪<div>n  <style scoped>n    ...
```

```
%matplotlib notebook
from itertools import groupby
import getpass
import os
from pathlib import Path
from multiprocessing import cpu_count
import re

def etime():
    user, sys, _, _, _ = os.times()
    return user+sys
```

2.6.2 Creating the *RunDirectory* object

The `RunDirectory` class opens netcdf data and tries to fetch additional information on the data that might be stored in fortran name list files that might have been created by run-scripts. Creating an instance of the `RunDirecotry` object will create a **json** file where all important data is stored for faster accessing the run the next time.

To instantiate the `RunDirectory` object we need the path to the data an experiment name (or file name suffix) that is unique to the data files and which kind of model/dataset is considered. Model/Dataset type is useful as variable names will be conveniently translated by the library; the user will only have to know the variable names for the *ECHAM* model convection. Variable name translations to other datasets/models will be done by the library. The following translations are implemented at the moment: * ECHAM / ICON-MPI * ICON-DWD * CMORPH

Let's open a **CMORPH** satellite based rainfall dataset.

```
# First define the path where the data is stored
run_dir = Path('/home/mpim/m300765/Data/CMORPH/netcdf/years/')

suffix = 'Cmorph' # Experiment name (in this case the filename suffix)
data_type = suffix.upper() # No tell the module that it should convert variable names␣
↪from
                           # CMORPH to ECHAM names
# Instanciate the RunDirectory object
Run = esm_analysis.RunDirectory(run_dir, model_type=data_type, overwrite=True,␣
↪client=cluster)
```

If no client keyword is given then `RunDirectory` tries to start a local multiprocessing pool. But in our case we have started a cluster and can directly connect to that cluster

```
Run.dask_client
```

NOTE: The `overwrite` key word argument set to true for demonstrative purpose and will be explained in detail later.

At this time no files have been opened, let's see which files that belong to the dataset it has found. The `files` property will return a `pandas Series` with all filename that can be potentially opened. We know that the filenames follow a certain date pattern. Let's group all filenames for a certain months together:

```
Run.files.head()
```

```
0    /home/mpim/m300765/Data/CMORPH/netcdf/years/19...
1    /home/mpim/m300765/Data/CMORPH/netcdf/years/19...
2    /home/mpim/m300765/Data/CMORPH/netcdf/years/19...
3    /home/mpim/m300765/Data/CMORPH/netcdf/years/19...
4    /home/mpim/m300765/Data/CMORPH/netcdf/years/19...
dtype: object
```

Re-Gridding the Input Data

Let's remap the content of this run. The `run_directory` object also offers a method for this. The remapping method is applied by calling `remap`.

Remap will automatically try to run jobs in parallel. First see check how many parallel tasks are available on this machine:

```
print(f'Will run with {len(config.workers)} workers a {config.worker_cores} cores in␣
↪the background.')
```

```
Will run with 2 workers a 48 cores in the background.
```

Let's define the output directory and the grid description file. If no output directory is given the remapping will create a sub-directory in the work folder.

```
# Define the traget grid description and the output directory of the folder
griddes = '/home/mpim/m300385/precip/T63grid_TRMM.txt'
user = getpass.getuser()
out_dir = Path(f'/scratch/{user[0]}/{user}/tmp/CMORPH/echam_grid')
out_dir.mkdir(exist_ok=True, parents=True)
```

```
# Call the remap procedure
%time _ =  Run.remap(griddes, out_dir=out_dir, method='remapcon')
```

```
HBox(children=(IntProgress(value=0, description='Remapping: ', max=6574,␣
↪style=ProgressStyle(description_width...
CPU times: user 6min 5s, sys: 11.3 s, total: 6min 16s
Wall time: 34min 1s
```

Applying Custom Functions to the Run

The remapped data is stored in *daily* files. Suppose we want to merge the data into monthly files. We can take advantage of the fact that the filenames follow a the convention `PRODUCTNAME_YEAR_MONTH_DAY.nc`, hence is it easy to use some `regex` to group them by month.

```
search = lambda key : re.search('.*\d{4}_\d{2}_*', key, re.M).group(0)
groups = [list(g) for k, g in groupby(Run.files, key=search)]
print(f'There are {len(groups)} different months in the dataset')
```

```
There are 216 different months in the dataset
```

`mpi_data` comes with python bindings for cdo which can be accessed via `mpi_data.cdo`. Let's define a function that merges the daily data in *CMPORPH* into monthly data:

```
def mergetime(infiles, outdir, prefix):
    """Apply a cdo mergetime."""
    # Get the month and year from the first filename
    re_match = f'.*{prefix}_(?P<year>[^/]+)_(?P<month>[^/]+)_(?P<day>[^/]+).nc$'
    year, month = re.match(re_match, infiles[0]).group('year', 'month')
    outfile = Path(outdir)/Path(f'{prefix}_{year}_{month}.nc')
    return esm_analysis.cdo.mergetime(' '+' '.join(infiles), output=outfile.as_
↪posix())
```

```
# Define the out directory
out_dir = Path(f'/scratch/{user[0]}/{user}/tmp/CMORPH/echam_mon')
out_dir.mkdir(exist_ok=True, parents=True)
```

We have *216 months* to merge this operation can take quite some time when done serially. The `RunDirectory` object offers a static method (`apply_function`) that can apply a collection to any given function in parallel. Let's apply the above defined function `mergetime` in parallel and measure the speed of the application.

```
%time _ = Run.apply_function(mergetime, groups, args=(out_dir.as_posix(), 'Cmorph'),␣
↪label='Merge Time')
```

```
HBox(children=(IntProgress(value=0, description='Merge Time: ', max=216,␣
↪style=ProgressStyle(description_width...
```

```
CPU times: user 945 ms, sys: 68 ms, total: 1.01 s
Wall time: 5.7 s
```

Merging all months has finished in under just 10 seconds. Let's load the new monthly dataset :

```
MonthRun = esm_analysis.RunDirectory(out_dir, model_type=data_type, overwrite=True,␣
↪client=cluster)
MonthRun.files.iloc[-1]
```

```
'/scratch/m/m300765/tmp/CMORPH/echam_mon/Cmorph_2015_12.nc'
```

Up to now we did not load any dataset. Let's load the entire data by creating a virtual dataset that opens all files and virtually merges them along the *time* axis. This can be done by calling `load_data` method.

Loading only a subset can be specified by collections of filenames or glob patterns. If we wanted to load only data between 1999 and 2105 we could to the following:

```
t1 = etime()
MonthRun.load_data([f'*{year}*.nc' for year in range(1999, 2016)])
print(f'Fresh load took {etime() - t1} seonds')
```

```
Fresh load took 12.279999999999973 seonds
```

The loaded dataset can be accessed with help of the `dataset` property

```
MonthRun.dataset[MonthRun.variables['pr']]
```

```
<xarray.DataArray 'precip' (time: 49672, lat: 32, lon: 192)>
dask.array<shape=(49672, 32, 192), dtype=float32, chunksize=(248, 32, 192)>
Coordinates:
  * time     (time) datetime64[ns] 1999-01-01 ... 2015-12-31T21:00:00
  * lon      (lon) float64 0.0 1.875 3.75 5.625 7.5 ... 352.5 354.4 356.2 358.1
  * lat      (lat) float64 -28.91 -27.05 -25.18 -23.32 ... 25.18 27.05 28.91
Attributes:
    standard_name:  3_hourly_accumulated_precipitaion
    long_name:      3 hourly accumulated precipitaion
    units:          mm/3hr
    code:           142
    table:          128
    short_naem:     precip
    fill_value:     -999.0
```

Using Data-Caches for faster realoading

Loading the data can take, depending on the amount of files in the run, a significant up amount of time. This is where the `overwrite` keyword argument comes in. When creating an instance of the `RunDirectory` object meta data about the run is saved in a *json* file. Loading a dataset will dump a serialized image of that loaded dataset into the run directory and the path to the dump is stored in the *json* file. The next time the data is loaded this dump will be opened instead of the netcdf files. This speeds up significantly the reading process. If the user wishes not to stored meta data but created a 'fresh' read of the data from files the `RunDirectory` object can instantiated with the `overwrite` keyword argument set to `True`.

Let's do the same what we've done above but with the default behavior. No overwrite. You'll notice a significant speed-up for retrieving the dataset.

```
MonthRun = esm_analysis.RunDirectory(out_dir, model_type=data_type, client=cluser)
```

```
t1 = etime()
MonthRun.load_data()
print(f'Loading the serialized pickle took {etime() - t1} seonds')
```

```
Loading the serialized pickle took 0.43000000000006366 seonds
```

2.6.3 Interactive Data Visualisation

`mpi_data` comes with a very rudimentary plotting collection. For instances plotting on maps is still not supported. Yet the 2D rainfall field could be visualized by applying the `profile_2d` method of the `ProfilePlotter` class

```
from mpi_data.plot import ProfilePlotter
from matplotlib import pyplot as plt
```

```
P = ProfilePlotter.profile_2d(MonthRun.dataset, MonthRun.variables['pr'],
                              figsize=(9, 6),  data_dim='x', avg_dims=None,
                              vmin=0, vmax=20, apply_func=None ,
                              cbar_args={'cbar_size':'7%', 'cbar_pad': '5%'})
```

```
<IPython.core.display.Javascript object>
```

```
HBox(children=(BoundedFloatText(value=0.0, description='time', layout=Layout(height=
↪'30px', width='200px'), ma...
```

```
FloatRangeSlider(value=(0.0, 20.0), continuous_update=False, description='Range:',␣
↪layout=Layout(width='100%')...
```

As of now, this creates a `imshow` plot along with ipython widgets that can be used to change color bar range, color map and cycle through time steps in the plot. This makes a first exploration of the data very easy.

Since the dataset of *MonthRun* is an xarray dataset object you can do anything with it you would usually do with xarray datasets.

```
fig, ax = plt.subplots(1,1, figsize=(9, 3.2))
MonthRun.dataset[MonthRun.variables['pr']][0:8].sum(dim='time').plot(ax=ax)
```

```
<IPython.core.display.Javascript object>
```

```
<matplotlib.collections.QuadMesh at 0x2b54730c38d0>
```

2.7 Release Notes

2.7.1 0.0.1

- Initial commit

See also:

Xarray Dask Dask-Jobqueue Dask-Cleint

CHAPTER THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX