b. Python and Jupyter notebook

Although methods exist that can plot unstructured ICON data from its native data layout (https://psyplot.github.io/examples/maps/example_ugrid.html), these methods become very slow with increasing resolution. The most common approach is to process the data as much as possible and remap the processed field variables to regular latitude-longitude mesh for displaying the data. This tutorial shows how this can be achieved using distributed computing resources.

Again import the libraries we are going to use. Remember to use the *Python 3 unstable* kernel to be able to import all libraries:

```
from getpass import getuser # Libaray to copy things
from pathlib import Path # Object oriented libary to deal with paths
import os
from tempfile import NamedTemporaryFile, TemporaryDirectory # Creating
temporary Files/Dirs
from subprocess import run, PIPE
import sys

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
import xarray as xr # Libary to work with labeled n-dimensional data and
dask
```

```
import warnings
warnings.filterwarnings(action='ignore')
```

Step 0. Load the data

This step is covered in great detail in the Open/Read section.

```
# Set some user specific variables
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_tmp_dir = TemporaryDirectory(dir=scratch_dir, prefix='PostProc')
cluster = SLURMCluster(memory='500GiB',
                       cores=72,
                       project='mh0731',
                       walltime='1:00:00',
                       queue='gpu',
                       name='PostProc',
                       scheduler_options={'dashboard_address': ':12435'},
```

```
                        local_directory=dask_tmp_dir.name,
                        job_extra=[f'-J PostProc',
                                   f'-D {dask_tmp_dir.name}',
                                   f'--begin=now',
                                   f'--
output={dask_tmp_dir.name}/LOG_cluster.%j.o',
                                   f'--
output={dask_tmp_dir.name}/LOG_cluster.%j.o'
                                  ],
                        interface='ib0')
cluster.scale(jobs=2)
dask_client = Client(cluster)
dask_client.wait_for_workers(18)
data_path = Path('/work/mh0287/k203123/GIT/icon-aes-
dyw_albW/experiments/dpp0016/')
glob_pattern_2d = 'atm_2d_ml'

# Collect all file names with pathlib's rglob and list compressions
file_names = sorted([str(f) for f in
data_path.rglob(f'*{glob_pattern_2d}*.nc')])[1:]
dset = xr.open_mfdataset(file_names, combine='by_coords', parallel=True)
dset_subset = dset.sel(time=slice('2020-01-20',
'2020-11-30')).resample(time='1D').mean()
var_names = ['ts', 'rsus', 'rlus', 'rlds', 'rsdt', 'hfls', 'hfss']
dset_subset = dset_subset[var_names].persist()
dset_subset
```

```
<xarray.Dataset>
Dimensions:  (ncells: 20971520, time: 72)
Coordinates:
  * time     (time) datetime64[ns] 2020-01-20 2020-01-21 ... 2020-03-31
Dimensions without coordinates: ncells
Data variables:
    ts       (time, ncells) float32 dask.array<chunksize=(1, 20971520),
meta=np.ndarray>
    rsus     (time, ncells) float32 dask.array<chunksize=(1, 20971520),
meta=np.ndarray>
    rlus     (time, ncells) float32 dask.array<chunksize=(1, 20971520),
meta=np.ndarray>
    rlds     (time, ncells) float32 dask.array<chunksize=(1, 20971520),
meta=np.ndarray>
    rsdt     (time, ncells) float32 dask.array<chunksize=(1, 20971520),
meta=np.ndarray>
    hfls     (time, ncells) float32 dask.array<chunksize=(1, 20971520),
meta=np.ndarray>
    hfss     (time, ncells) float32 dask.array<chunksize=(1, 20971520),
meta=np.ndarray>
```

```
progress(dset_subset, notebook=False)
```

```
[########################################] | 100% Completed | 51.6s
```

To quickly recap we've gotten all 6 hourly single level data files and opened them in a combined virtual view using `xarray`. We then created daily averages using the `resample` method. We then pushed the data to the distributed memory, where it is available for processing.

Step 1. Creating averages:

Before regridding we are going to create averages along the field and time dimensions for later visualization use. Averages along the field dimensions will result in a time series. We obviously don't need to regrid this data. But the averages along the time dimensions will have to be regridded. Since ICON data is stored on a equal area triangular mesh we do not have to care about weighting when averaging on the native grid. Applying averages is easy to achieve using the `mean` method

```
# Create averges along the time and field dimensions
time_mean = dset_subset.mean(dim='time').persist()
field_mean = dset_subset.mean(dim='ncells').persist()
```

We have immediately triggered the calculation on the cluster using the `persist` method. We can watch the status of the calculation with `progress`.

```
progress([time_mean, field_mean], notebook=False)
```

```
[########################################] | 100% Completed | 29.5s
```

Since the data is already on the distributed memory the calculation is quite fast in our case just under 30 seconds.

Step 2. Defining the remap commands

Since we are going to use distance weighted remapping with CDO, we need a target grid description and a weight file. The target grid description is a text file that contains the information of the regular latitude-longitude grid. Let's assume that we want to create a grid of 0.1 x 0.1 degrees. The target grid file would look likes this:

```
#
# 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
```

instead of hard coding this grid information we can create a function that calculates the grid information:

```
def get_griddes(y_res, x_res, x_first=-180, y_first=-90):
    """Create a description for a regular global grid at given x, y
resolution."""

    xsize = 360 / x_res
    ysize = 180 / y_res
    xfirst = -180 + x_res / 2
    yfirst = -90 + x_res / 2

    return f'''
#
# gridID 1
#
gridtype  = lonlat
gridsize  = {int(xsize * ysize)}
xsize     = {int(xsize)}
ysize     = {int(ysize)}
xname     = lon
xlongname = "longitude"
xunits    = "degrees_east"
yname     = lat
ylongname = "latitude"
yunits    = "degrees_north"
xfirst    = {xfirst}
xinc      = {x_res}
yfirst    = {yfirst}
yinc      = {y_res}


    '''
```

The best way to remap unstructured data 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 `dask.delayed` to tell the code it should be executed remotely. We will create another function `run_cmd` that executes a shell command.

```
@dask.delayed
def gen_dis(dataset, xres, yres, gridfile):
    '''Create a distance weights using cdo.'''
    scratch_dir = Path('/scratch') / getuser()[0] / getuser() # Define the
users scratch dir
```

```
    with TemporaryDirectory(dir=scratch_dir, prefix='Weights_') as td:
        in_file = Path(td) / 'in_file.nc'
        weightfile = Path(td) / 'weight_file.nc'
        griddes = Path(td) / 'griddes.txt'
        with griddes.open('w') as f:
            f.write(get_griddes(xres, yres))
        dataset.to_netcdf(in_file, mode='w') # Write the file to a temorary
netcdf file
        cmd = ('cdo', '-O', f'gendis,{griddes}', f'-setgrid,{gridfile}',
str(in_file), str(weightfile))
        run_cmd(cmd)
        df = xr.open_dataset(weightfile).load()
        wait(df)
        return df

def run_cmd(cmd, path_extra=Path(sys.exec_prefix)/'bin'):
    '''Run a bash command.'''
    env_extra = os.environ.copy()
    env_extra['PATH'] = str(path_extra) + ':' + env_extra['PATH']
    status = run(cmd, check=False, stderr=PIPE, stdout=PIPE, env=env_extra)
    if status.returncode != 0:
        error = f'''{' '.join(cmd)}: {status.stderr.decode('utf-8')}'''
        raise RuntimeError(f'{error}')
    return status.stdout.decode('utf-8')
```

Dask collects the future results before executing them. This way we can setup a task tree. For example we could now define a function that gets the output of `gen_dis` and applies the remapping.

```
@dask.delayed
def remap(dataset, x_res, y_res, weights, gridfile):
    """Perform a weighted remapping.

    Parameters
    ==========

    dataset : xarray.dataset
        The dataset that will be regridded
    griddes : Path, str
        Path to the grid description file
    weights : xarray.dataset
        Distance weights

    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})
    scratch_dir = Path('/scratch') / getuser()[0] / getuser() # Define the
users scratch dir
```

```
    with TemporaryDirectory(dir=scratch_dir, prefix='Remap_') as td:
        infile = Path(td) / 'input_file.nc'
        weightfile = Path(td) / 'weight_file.nc'
        griddes = Path(td) / 'griddes.txt'
        outfile = Path(td) / 'remaped_file.nc'
        with griddes.open('w') as f:
            f.write(get_griddes(x_res, y_res))
        dataset.to_netcdf(infile, mode='w') # Write the file to a temorary
netcdf file
        weights.to_netcdf(weightfile, mode='w')
        cmd = ('cdo', '-O', f'remap,{griddes},{weightfile}', f'-
setgrid,{gridfile}',
               str(infile), str(outfile))
        run_cmd(cmd)
        return xr.open_dataset(outfile).load()
```

Step 3: Applying the remapping in parallel

Calling only the`gen_dis` function defined in step 2 will not trigger any computation. But first we need a grid description file of our dpp0016 simulation:

```
grid_file =
'/pool/data/ICON/grids/public/mpim/0015/icon_grid_0015_R02B09_G.nc'
weights_future = gen_dis(time_mean, 0.5, 0.5, grid_file)
weights_future
```

```
Delayed('gen_dis-379f0fee-deb2-480e-8527-82914b9de2a8')
```

```
remap_futures = []
# Process each variable in parallel.
for var_name in time_mean.data_vars:
    remap_futures.append(remap(time_mean[var_name], 0.5, 0.5,
weights_future, grid_file))
remap_futures
```

```
[Delayed('remap-bf89d25d-1bdc-4eb9-b0ad-3a40ac2c328d'),
 Delayed('remap-d8576790-d46a-4d3c-b4d9-05041720c17c'),
 Delayed('remap-76e0316f-0ab5-473c-b353-df99a69647a3'),
 Delayed('remap-7787725d-98f4-4b45-9edf-b3a019c91401'),
 Delayed('remap-aaee97d7-3cc8-41c6-81ec-b6e7c22d2438'),
 Delayed('remap-8d0c091f-92d4-4705-8e0d-a4b2d14de6b1'),
 Delayed('remap-f4d38530-1477-460d-9e7f-98ab22dbeefc')]
```

The remap futures list if a collection of task that *will* be executed, nothing has been executed yet. The execution of each task depends on the outcome of `gen_dis`. Dask will take care of this when we execute the tasks. Calling the `dask.persist` will trigger the computation in the cluster in parallel:

```
remap_jobs = dask.persist(remap_futures)
progress(remap_jobs, notebook=False)
```

```
[########################################] | 100% Completed |  30.5s
```

Now we just have to merge the results back together into one dataset

```
time_mean_remap = xr.merge(list(dask.compute(*remap_futures)))
time_mean_remap
```

```
<xarray.Dataset>
Dimensions:  (lat: 360, lon: 720)
Coordinates:
  * lon      (lon) float64 -179.8 -179.2 -178.8 -178.2 ... 178.8 179.2 179.8
  * lat      (lat) float64 -89.75 -89.25 -88.75 -88.25 ... 88.75 89.25 89.75
Data variables:
    ts       (lat, lon) float32 232.50095 232.49681 ... 237.39883 237.39789
    rsus     (lat, lon) float32 130.34554 130.34743 ... 2.7430184 2.7431903
    rlus     (lat, lon) float32 165.59589 165.58539 ... 180.20906 180.20578
    rlds     (lat, lon) float32 120.99219 120.99084 ... 157.13567 157.13335
    rsdt     (lat, lon) float32 225.16904 225.16885 ... 7.7745104 7.774646
    hfls     (lat, lon) float32 -0.0069637517 -0.006892924 ... -1.5673956
    hfss     (lat, lon) float32 5.974602 5.947811 ... 1.8291237 1.8200014
```

Step 4: Saving the data to file:

Finally we are going to save the remapped data along with the time series (field_mean) to a new netcdf file for further processing in the next session about visualizing. To do this we will use two different groups one for the time series and one for the remapped data:

```
# 1 Save the time-series
out_file = Path(scratch_dir) / 'dpp0016_PreProc.nc'
field_mean.to_netcdf(out_file, mode='w', group='field_mean')
time_mean_remap.to_netcdf(out_file, mode='a', group='time_mean')
```

From:
https://wiki.mpimet.mpg.de/ - **MPI Wiki**

Permanent link:
https://wiki.mpimet.mpg.de/doku.php?id=analysis:postprocessing_icon:regridding:python:start

Last update: **2024/05/05 21:11**