b. Python and Jupyter notebook

This tutorial shows how to open ICON output saved in multiple files, setup a distributed computing environment to process the files and read a subset of the output. As an example we will process data from the Dyamond Winter project. It presupposes that you have set up a Jupyter notebook (as explained under Computing Infrastructure).

Step 1 : Import all libraries that are needed

Note: Remember, to be able to import all libraries you'll have to use the *Python 3 unstable* kernel

```
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 dask_jobqueue import SLURMCluster # Setting up distributed memories via
slurm
from dask.utils import format_bytes
from distributed import Client, progress # Libaray to orchestrate
distributed resources
import numpy as np # Pythons standard array library
import xarray as xr # Libary to work with labeled n-dimensional data
```

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

Step 2: Setup a distributed computing cluster where we can process the output

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 computer memory. Therefore we're setting up a distributed cluster using the `dask_jobqueue` library. Specifically we will involve the Slurm workload manager to do that. More information on the `dask_jobqueue` library can be found here: https://jobqueue.dask.org/en/latest/ .

To create the slurm 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 queue but any other partition can be involved. More information on the hardware queues and their hardware specifications can be found on the dkrz user portal.

```
# Set some user specific variables
account_name = 'mh0731' # Account that is going to be 'charged' fore the
computation
queue = 'gpu' # Name of the partition we want to use
job_name = 'PostProc' # Job name that is submitted via sbatch
```

```
memory = "500GiB" # Max memory per node that is going to be used - this
depends on the partition
cores = 72 # Max number of cores per task that are reserved - also partition
dependend
walltime = '1:00:00' # Walltime - also partition dependent
```

```
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=job_name)
cluster = SLURMCluster(memory=memory,
                       cores=cores,
                       project=account_name,
                       walltime=walltime,
                       queue=queue,
                       name=job_name,
                       scheduler_options={'dashboard_address': ':12435'},
                       local_directory=dask_tmp_dir.name,
                       job_extra=[f'-J {job_name}',
                                  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')
```

Under the hood the `job_queue` library will create a job script that is going to be submitted via sbatch. Let's have a look at this job-script:

```
print(cluster.job_script())
```

```
#!/usr/bin/env bash

#SBATCH -J lab-post
#SBATCH -p gpu
#SBATCH -A mh0731
#SBATCH -n 1
#SBATCH --cpus-per-task=72
#SBATCH --mem=500G
#SBATCH -t 8:00:00
#SBATCH -J PostProc
#SBATCH -D /scratch/m/m300765/PostProc9pex9k_y
#SBATCH --begin=now
#SBATCH --output=/scratch/m/m300765/PostProc9pex9k_y/LOG_cluster.%j.o
#SBATCH --output=/scratch/m/m300765/PostProc9pex9k_y/LOG_cluster.%j.o
```

```
JOB_ID=${SLURM_JOB_ID%;*}

/home/mpim/m300765/.anaconda3/bin/python -m distributed.cli.dask_worker
tcp://10.50.32.30:46411 --nthreads 72 --memory-limit 536.87GB --name name --
nanny --death-timeout 60 --local-directory
/scratch/m/m300765/PostProc9pex9k_y --interface ib0
```

So far nothing has happened, lets order 2 nodes that will give us 1 TB of distributed memory and 144 cores to work on.

```
cluster.scale(jobs=2)
cluster
```

Now we have submitted 2 jobs that establish the distributed computing resources. After some queuing time, depending on how busy the machine is the resources will be available. This can be seen when the above interfaces changes from

Workers 0/2

to

Workers 2

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

```
! squeue -u $USER
```

```
             JOBID PARTITION     NAME     USER ST       TIME  NODES
NODELIST(REASON)
          23835542       gpu PostProc  m300765  R       1:02      1 mg200
          23835543       gpu PostProc  m300765  R       1:02      1 mg205
```

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

```
dask_client = Client(cluster)
dask_client
```

```
<Client: 'tcp://10.50.32.30:46411' processes=2 threads=144, memory=1.07 TB>
```

The distributed computing resources are ready, now we can do the actual task of this tutorial:

Step 3: Reading the data

Let's define the paths and the variables we are going to read. We want to read files from a coupled

atmosphere-ocean simulation. The data has a resolution of 5 km, the internal experiment ID is dpp0016. In this example we will make use of six hourly data which is store in <exp>_2d_atm_ml_<timestep>.nc file patterns. Let's define the paths and global pattern that will tell python which files to open:

```
# Define the paths and get all files
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 compression
file_names = sorted([str(f) for f in
data_path.rglob(f'*{glob_pattern_2d}*.nc')])[1:]
file_names[:10]
```

```
['/work/mh0287/k203123/GIT/icon-aes-
dyw_albW/experiments/dpp0016/dpp0016_atm_2d_ml_20200120T000000Z.nc',
 '/work/mh0287/k203123/GIT/icon-aes-
dyw_albW/experiments/dpp0016/dpp0016_atm_2d_ml_20200121T000000Z.nc',
 '/work/mh0287/k203123/GIT/icon-aes-
dyw_albW/experiments/dpp0016/dpp0016_atm_2d_ml_20200122T000000Z.nc',
 '/work/mh0287/k203123/GIT/icon-aes-
dyw_albW/experiments/dpp0016/dpp0016_atm_2d_ml_20200123T000000Z.nc',
 '/work/mh0287/k203123/GIT/icon-aes-
dyw_albW/experiments/dpp0016/dpp0016_atm_2d_ml_20200124T000000Z.nc',
 '/work/mh0287/k203123/GIT/icon-aes-
dyw_albW/experiments/dpp0016/dpp0016_atm_2d_ml_20200125T000000Z.nc',
 '/work/mh0287/k203123/GIT/icon-aes-
dyw_albW/experiments/dpp0016/dpp0016_atm_2d_ml_20200126T000000Z.nc',
 '/work/mh0287/k203123/GIT/icon-aes-
dyw_albW/experiments/dpp0016/dpp0016_atm_2d_ml_20200127T000000Z.nc',
 '/work/mh0287/k203123/GIT/icon-aes-
dyw_albW/experiments/dpp0016/dpp0016_atm_2d_ml_20200128T000000Z.nc',
 '/work/mh0287/k203123/GIT/icon-aes-
dyw_albW/experiments/dpp0016/dpp0016_atm_2d_ml_20200129T000000Z.nc']
```

The above syntax uses the `rglob` method of the data_path object to find all files with a name according to a specified global pattern. We were also using list compression here. That is combining a `for-loop` into one line. Which makes the execution of that loop much faster.

Now we use `xarray.mfdataset` to open all the data and inspect their content:

```
dset = xr.open_mfdataset(file_names, combine='by_coords', parallel=True)
dset
```

```
<xarray.Dataset>
Dimensions:  (ncells: 20971520, time: 437)
Coordinates:
  * time     (time) datetime64[ns] 2020-01-20 2020-01-20T06:00:00 ...
```

```
2020-12-05
Dimensions without coordinates: ncells
Data variables:
    ps       (time, ncells) float32 dask.array<chunksize=(4, 20971520),
meta=np.ndarray>
    psl      (time, ncells) float32 dask.array<chunksize=(4, 20971520),
meta=np.ndarray>
    rsdt     (time, ncells) float32 dask.array<chunksize=(4, 20971520),
meta=np.ndarray>
    rsut     (time, ncells) float32 dask.array<chunksize=(4, 20971520),
meta=np.ndarray>
    rsutcs   (time, ncells) float32 dask.array<chunksize=(4, 20971520),
meta=np.ndarray>
    rlut     (time, ncells) float32 dask.array<chunksize=(4, 20971520),
meta=np.ndarray>
    rlutcs   (time, ncells) float32 dask.array<chunksize=(4, 20971520),
meta=np.ndarray>
    rsds     (time, ncells) float32 dask.array<chunksize=(4, 20971520),
meta=np.ndarray>
    rsdscs   (time, ncells) float32 dask.array<chunksize=(4, 20971520),
meta=np.ndarray>
    rlds     (time, ncells) float32 dask.array<chunksize=(4, 20971520),
meta=np.ndarray>
    rldscs   (time, ncells) float32 dask.array<chunksize=(4, 20971520),
meta=np.ndarray>
    rsus     (time, ncells) float32 dask.array<chunksize=(4, 20971520),
meta=np.ndarray>
    rsuscs   (time, ncells) float32 dask.array<chunksize=(4, 20971520),
meta=np.ndarray>
    rlus     (time, ncells) float32 dask.array<chunksize=(4, 20971520),
meta=np.ndarray>
    ts       (time, ncells) float32 dask.array<chunksize=(4, 20971520),
meta=np.ndarray>
    sic      (time, ncells) float32 dask.array<chunksize=(4, 20971520),
meta=np.ndarray>
    sit      (time, ncells) float32 dask.array<chunksize=(4, 20971520),
meta=np.ndarray>
    clt      (time, ncells) float32 dask.array<chunksize=(4, 20971520),
meta=np.ndarray>
    prlr     (time, ncells) float32 dask.array<chunksize=(4, 20971520),
meta=np.ndarray>
    prls     (time, ncells) float32 dask.array<chunksize=(4, 20971520),
meta=np.ndarray>
    pr       (time, ncells) float32 dask.array<chunksize=(4, 20971520),
meta=np.ndarray>
    prw      (time, ncells) float32 dask.array<chunksize=(4, 20971520),
meta=np.ndarray>
    cllvi    (time, ncells) float32 dask.array<chunksize=(4, 20971520),
meta=np.ndarray>
    clivi    (time, ncells) float32 dask.array<chunksize=(4, 20971520),
meta=np.ndarray>
```

```
    qgvi     (time, ncells) float32 dask.array<chunksize=(4, 20971520),
meta=np.ndarray>
    qrvi     (time, ncells) float32 dask.array<chunksize=(4, 20971520),
meta=np.ndarray>
    qsvi     (time, ncells) float32 dask.array<chunksize=(4, 20971520),
meta=np.ndarray>
    hfls     (time, ncells) float32 dask.array<chunksize=(4, 20971520),
meta=np.ndarray>
    hfss     (time, ncells) float32 dask.array<chunksize=(4, 20971520),
meta=np.ndarray>
    evspsbl  (time, ncells) float32 dask.array<chunksize=(4, 20971520),
meta=np.ndarray>
    tauu     (time, ncells) float32 dask.array<chunksize=(4, 20971520),
meta=np.ndarray>
    tauv     (time, ncells) float32 dask.array<chunksize=(4, 20971520),
meta=np.ndarray>
    sfcwind  (time, ncells) float32 dask.array<chunksize=(4, 20971520),
meta=np.ndarray>
    uas      (time, ncells) float32 dask.array<chunksize=(4, 20971520),
meta=np.ndarray>
    vas      (time, ncells) float32 dask.array<chunksize=(4, 20971520),
meta=np.ndarray>
    tas      (time, ncells) float32 dask.array<chunksize=(4, 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@b582fb87edbd30...
    history:               /work/mh0287/k203123/GIT/icon-aes-
dyw_albW/bin/icon...
    references:            see MPIM/DWD publications
    comment:               Sapphire Dyamond (k203123) on m21322 (Linux
2.6.32-...
```

Let's inspect this data a little further:

```
dset['tas']
```

```
<xarray.DataArray 'tas' (time: 437, ncells: 20971520)>
dask.array<concatenate, shape=(437, 20971520), dtype=float32, chunksize=(4,
```

```
20971520), chunktype=numpy.ndarray>
Coordinates:
  * time     (time) datetime64[ns] 2020-01-20 2020-01-20T06:00:00 ...
2020-12-05
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
```

```
dset['tas'].data
```

```
dask.array<concatenate, shape=(437, 20971520), dtype=float32, chunksize=(4,
20971520), chunktype=numpy.ndarray>
```

An important concept of xarray and dask is that the data hasn't been read into memory yet. We only see a representation of what the data *will* look like. This 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 314 chunks. Reading the array would take 942 tasks the total dataset would take 36.66 GB of memory. We can also ask xarray how much memory the whole dataset would consume:

```
format_bytes(dset.nbytes)
```

```
'1.32 TB'
```

This means that the total dataset (both experiments) would need 1.32 TB of memory. Way too much for a local computer but we do *only* have 1 TB of distributed memory. Let's reduce the data further by creating daily averages and sub setting by only selecting data until April 2020.

```
timesf = pd.date_range(start='2020-01-20', periods=dset.time.size
,freq='6H')
dset['time'] = timesf
dset_subset = dset.sel(time=slice('2020-01-20',
'2020-02-29')).resample(time='1D').mean()
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:
    ps       (time, ncells) float32 dask.array<chunksize=(1, 20971520),
meta=np.ndarray>
    psl      (time, ncells) float32 dask.array<chunksize=(1, 20971520),
```

```
meta=np.ndarray>
    rsdt     (time, ncells) float32 dask.array<chunksize=(1, 20971520),
meta=np.ndarray>
    rsut     (time, ncells) float32 dask.array<chunksize=(1, 20971520),
meta=np.ndarray>
    rsutcs   (time, ncells) float32 dask.array<chunksize=(1, 20971520),
meta=np.ndarray>
    rlut     (time, ncells) float32 dask.array<chunksize=(1, 20971520),
meta=np.ndarray>
    rlutcs   (time, ncells) float32 dask.array<chunksize=(1, 20971520),
meta=np.ndarray>
    rsds     (time, ncells) float32 dask.array<chunksize=(1, 20971520),
meta=np.ndarray>
    rsdscs   (time, ncells) float32 dask.array<chunksize=(1, 20971520),
meta=np.ndarray>
    rlds     (time, ncells) float32 dask.array<chunksize=(1, 20971520),
meta=np.ndarray>
    rldscs   (time, ncells) float32 dask.array<chunksize=(1, 20971520),
meta=np.ndarray>
    rsus     (time, ncells) float32 dask.array<chunksize=(1, 20971520),
meta=np.ndarray>
    rsuscs   (time, ncells) float32 dask.array<chunksize=(1, 20971520),
meta=np.ndarray>
    rlus     (time, ncells) float32 dask.array<chunksize=(1, 20971520),
meta=np.ndarray>
    ts       (time, ncells) float32 dask.array<chunksize=(1, 20971520),
meta=np.ndarray>
    sic      (time, ncells) float32 dask.array<chunksize=(1, 20971520),
meta=np.ndarray>
    sit      (time, ncells) float32 dask.array<chunksize=(1, 20971520),
meta=np.ndarray>
    clt      (time, ncells) float32 dask.array<chunksize=(1, 20971520),
meta=np.ndarray>
    prlr     (time, ncells) float32 dask.array<chunksize=(1, 20971520),
meta=np.ndarray>
    prls     (time, ncells) float32 dask.array<chunksize=(1, 20971520),
meta=np.ndarray>
    pr       (time, ncells) float32 dask.array<chunksize=(1, 20971520),
meta=np.ndarray>
    prw      (time, ncells) float32 dask.array<chunksize=(1, 20971520),
meta=np.ndarray>
    cllvi    (time, ncells) float32 dask.array<chunksize=(1, 20971520),
meta=np.ndarray>
    clivi    (time, ncells) float32 dask.array<chunksize=(1, 20971520),
meta=np.ndarray>
    qgvi     (time, ncells) float32 dask.array<chunksize=(1, 20971520),
meta=np.ndarray>
    qrvi     (time, ncells) float32 dask.array<chunksize=(1, 20971520),
meta=np.ndarray>
    qsvi     (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>
    evspsbl  (time, ncells) float32 dask.array<chunksize=(1, 20971520),
meta=np.ndarray>
    tauu     (time, ncells) float32 dask.array<chunksize=(1, 20971520),
meta=np.ndarray>
    tauv     (time, ncells) float32 dask.array<chunksize=(1, 20971520),
meta=np.ndarray>
    sfcwind  (time, ncells) float32 dask.array<chunksize=(1, 20971520),
meta=np.ndarray>
    uas      (time, ncells) float32 dask.array<chunksize=(1, 20971520),
meta=np.ndarray>
    vas      (time, ncells) float32 dask.array<chunksize=(1, 20971520),
meta=np.ndarray>
    tas      (time, ncells) float32 dask.array<chunksize=(1, 20971520),
meta=np.ndarray>
```

The `resample` method splits up the data into daily chunks while `mean` applies an average over the chunks. Consequently we get daily data. The same way we could calculate monthly or any kind of arbitrary averages.

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 and the data will be copied into memory and the averaging will be done. If no distributed memory is available persist uses the local memory. Please refer to https://distributed.readthedocs.io/en/latest/manage-computation.html#dask-collections-to-futures for more information.

We can subset the data a little further by only selecting certain variables:

```
var_names = ['ts', 'rsus', 'rlus', 'rlds', 'rsdt']
dset_subset = dset_subset[var_names]
format_bytes(dset_subset.nbytes)
```

```
'30.20 GB'
```

Now let's read the data from files and push it to the cluster with the `persist` method:

```
dset_subset = dset_subset.persist()
# Let's inspect the progress
progress(dset_subset, notebook=True)
```

```
[########################################] | 100% Completed |  3 min 36 s
```

We can see the tasks that are worked on in the background on the cluster. We can continue working with the data. Setting the `notebook=False` keyword will block further executions until all calculations are done. The paradigm is that all calculations are collected and not executed until we explicitly instruct xarray / dask to trigger computations.