A Benchmark for processing Dyamond Data

This document is a jupyter notebook, it shows possible examples of how to process *big data* from high resolution model output. In the present case global storm resolving model simulations from the Dyamond++ project have been chosen as an example.

The output of Dyamond++ is based on fully couple Earth-Atmosphere-Ocean simulations with a horizontal resolution of roughly 5 km, 40 vertical layers and 6 hours in time. One month of simulation contains roughly 4.5 TB of data.

One of the first task of processing coupled global model output is the calculation of the global energy budgets at the top of the atmosphere and the surface. This processing will be demonstrated in this notebook. To calculate the surface and top of the atmosphere budgets we need long and short wave radiations at the upper and lower boundaries of the atmosphere as well as surface latent and sensible heat fluxes. Below is a list of the *nine* variables that need to be processed:

- Top of the atmosphere: `rsdt, rsut, rlut`
- Surface: `hfss, hfls, rsds, rsus, rlds, rlus`

This example will examine processing of 2D atmospheric output for *three months*. The amount of data that will be processed is roughly 0.6 TB. So semi big data. Yet this notebook should serve as an example of how *big climate data*, from dozens of TB to even PB scale, can be processed using distributed computing resources.

The given examples are ordered into three parts. First we will introduce `dask-distributed` and try analyzing the data using pure python. Then we will try a shell based approach using `cdo` with `slurm-array` jobs. Finally we will try to combine `cdo` and `dask-distributed`.

Mistral comes with a pre-installed ancaconda3 distribution. It is recommended to use this distribution by logging onto `mistral` and loading the anaconda3 modules:

```
$: module load anaconda3/bleeding_edge
```

This command will make most of the needed python libraries available. Additional packages can be installed into the user space. Once the module has been loaded and a `jupyter` notebook is launched we can begin the tutorial. More information on jupyter at dkrz is available on https://www.dkrz.de/up/systems/mistral/programming/jupyter-notebook.

First lets import some standard python libraries that we will use throughout this tutorial.

```
from datetime import datetime # Deal with datetime related things
import getpass # Getting user information on the system
from pathlib import Path # A very useful object oriented system path library
from subprocess import run, PIPE #Execute shell commands
from tempfile import TemporaryDirectory # Create temporary directories
import timeit # Measuring time

import pandas as pd # Powerfull data analysis library
from matplotlib import pyplot as plt # Pythons plotting libary
import numpy as np # N dim. Array library
```

```
import xarray as xr # Library that can deal with labled N dim. array
```

Using pure python with dask-distributed and jobqueue

In the first section we will analyze the data using pure python and distributed computing with help of the dask ecosystem. Let's load the dask libraries for distributed computing. They should be part of the anaconda distribution:

```
from dask.distributed import (Client, wait)
from dask_jobqueue import SLURMCluster
```

`dask_jobqueue` allows for an integration of the *slurm* workload manager and `dask`.

Let's define the paths to the data and the variables that need to be processed.

```
EXPDIR = '/work/mh0287/k203123/experiments/dpp0001'
EXPNAME = 'dpp0001'
VARNAMES = list({'rlut', 'hfls', 'hfss', 'rsut', 'rlut', 'rsds','rlds',
'rsus', 'rlus', 'rsdt'})
START, END = '2016-08-10T00:00', '2016-11-10T23:00:00'
GLOB_PATTERN = 'atm_2d' # The pattern to identify the filenames we are
interested in
```

The data should be stored in temporary directories which can be created with the `TemporaryDirectory` module. As mentioned above the tutorial consists of three parts, hence we create three temporary directories.

```
# Create some unique temporary directories where intermediate results can be
saved
dask_dir =
TemporaryDirectory(dir=f'/scratch/{getpass.getuser()[0]}/{getpass.getuser()}
', prefix='DaskBenchmark_')
cdo_dir =
TemporaryDirectory(dir=f'/scratch/{getpass.getuser()[0]}/{getpass.getuser()}
', prefix='CdoBenchmark_')
cdodask_dir =
TemporaryDirectory(dir=f'/scratch/{getpass.getuser()[0]}/{getpass.getuser()}
', prefix='CdoDask_')
```

Now it's time to create the `slurm cluster` that will execute the processing. For this purpose we use the *prepost* queue. Each computing node in this queue has 32 cpu's and a memory of 150 GB. 30 Minutes of wall time will be more than sufficient. We can create the cluster by using the imported `SLURMCluster` class. Under the hood this class creates a job-script that will be submitted to the slurm cluster via the `sbatch` command.

```
# Set up our cluster
Cluster = SLURMCluster(memory="150GB",
```

```
                        cores=32,
                        project='mh0731',
                        walltime='00:30:00',
                        queue='compute',
                        local_directory=dask_dir.name,
                        job_extra=['-J dask_benchmark', f'-D
{dask_dir.name}'])
```

Here I am using the *mh0731* account for processing. Additional slurm arguments can be posted with the `job_extra` keyword argument. A full documentation of the `das_jobqueue` library is available on https://jobqueue.dask.org/en/latest/. Lets inspect the job script that will be submitted.

```
print(Cluster.job_script())
```

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

#SBATCH -J lab-post
#SBATCH -p compute
#SBATCH -A mh0731
#SBATCH -n 1
#SBATCH --cpus-per-task=32
#SBATCH --mem=140G
#SBATCH -t 00:30:00
#SBATCH -J dask_benchmark
#SBATCH -D /scratch/m/m300765/DaskBenchmark_c2a5gmj0

JOB_ID=${SLURM_JOB_ID%;*}

/home/mpim/m300765/.anaconda3/bin/python -m distributed.cli.dask_worker
tcp://136.172.50.18:44106 --nthreads 32 --memory-limit 150.00GB --name name
--nanny --death-timeout 60 --local-directory
/scratch/m/m300765/DaskBenchmark_c2a5gmj0
```

The cluster can be created by using the `Cluster.scale` method with the number of nodes we want to use or interactively by just calling the `Cluster` in a notebook cell. 10 workers is a good number to begin with.

```
Cluster.scale(10)
```

Inspecting the state of the cluster on the command line gives:

```
$: squeue -u $USER854
```

```
             JOBID PARTITION     NAME     USER ST       TIME  NODES
NODELIST(REASON)
          19591955   compute dask_ben  m300765 PD       0:00      1
(Priority)
          19591956   compute dask_ben  m300765 PD       0:00      1
(Priority)
```

```
          19591957   compute dask_ben  m300765 PD       0:00      1
(Priority)
          19591958   compute dask_ben  m300765 PD       0:00      1
(Priority)
          19591959   compute dask_ben  m300765 PD       0:00      1
(Priority)
          19591960   compute dask_ben  m300765 PD       0:00      1
(Priority)
          19591961   compute dask_ben  m300765 PD       0:00      1
(Priority)
          19591962   compute dask_ben  m300765 PD       0:00      1
(Priority)
          19591963   compute dask_ben  m300765 PD       0:00      1
(Priority)
          19591964   compute dask_ben  m300765 PD       0:00      1
(Priority)
```

All jobs on the cluster can be canceled by using the `Cluster.close()` method. Now that all nodes are up and running in the cluster let's setup a `dask-distributed` client. This client will orchestrate the cluster and send commands to the cluster, get information and data back from the it. Setting up a client is easy, just create an instance of the imported Client class using the created slurm cluster:

```
client = Client(Cluster)
```

The client says we have 10 workers up and running using together 320 cores and 1.5 TB of memory. We can start processing the data. The first step is to identify the files that needs to be analyzed. Above was mentioned that the first three months of output data should be processed. Let's define a function that returns only all files between two given dates of a certain output type:

```
# Define a helper function that extracts the filenames we want to process
def get_filnames(out_dir, exp_name, glob_pattern, start, end):
    """Helper function that extracts filenames that ought to be processed.

    Arguments:
    ==========

    exp_dir  :
        Directory of where the experiment data resides
    exp_name :
        Name of the experiment
    glob_pattern :
        Collections of patterns that identify the filenames of interest
    start :
        First time step to process
    end :
        Last time step to proces

    Returns: list
        List of files that should be processed
```

```
    """
    out_dir = Path(out_dir)
    if not isinstance(glob_pattern, (list, set, tuple)):
        glob_pattern = glob_pattern,
    start, end = pd.DatetimeIndex([start, end]) # Convert to pandas datetime
index
    files = []
    for prefix in glob_pattern:
        search_string = f'{exp_name}*{prefix}*.nc'
        for f in out_dir.rglob(search_string):
            timeStep =
pd.DatetimeIndex([str(f.with_suffix('')).split('_')[-1].strip('Z')])[0]
            if timeStep >= start and timeStep <= end:
                files.append(str(f))
    return sorted(files)
# Get the filenames
filenames = get_filnames(EXPDIR, EXPNAME, GLOB_PATTERN, START, END)
print(f'{len(filenames)} files will be processed')
```

```
93 files will be processed
```

The authors of this tutorial are developing a small data analysis library that contains useful methods to make processing icon data a little easier. A documentation is available on https://esm-analysis.readthedocs.io/en/latest/. It can be installed in the userspace using pip:

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

After the module has been installed it can be imported:

```
from esm_analysis import (icon2datetime, progress_bar, RunDirectory,
progress_bar)
```

We can now use the library together with the identified filenames to load the Dyamond++ data:

```
# Open the experiment directory with the RunDirectory class
Run = RunDirectory(EXPDIR, prefix=EXPNAME, client=client)

# Experiment data can be loaded using the load_data method
dset = Run.load_data(filenames)[VARNAMES]
```

The `dset` object if of type `xarray.dataset`, this been we can immediately only select a subset of the variables. Lets inspect the data first:

```
dset
```

```
<xarray.Dataset>
Dimensions:  (ncells: 20971520, time: 372)
Coordinates:
```

```
  * time     (time) float64 2.016e+07 2.016e+07 ... 2.016e+07 2.016e+07
Dimensions without coordinates: ncells
Data variables:
    rsds     (time, ncells) float32 dask.array<chunksize=(4, 20971520),
meta=np.ndarray>
    rlut     (time, ncells) float32 dask.array<chunksize=(4, 20971520),
meta=np.ndarray>
    rsut     (time, ncells) float32 dask.array<chunksize=(4, 20971520),
meta=np.ndarray>
    rsus     (time, ncells) float32 dask.array<chunksize=(4, 20971520),
meta=np.ndarray>
    rsdt     (time, ncells) float32 dask.array<chunksize=(4, 20971520),
meta=np.ndarray>
    rlds     (time, ncells) float32 dask.array<chunksize=(4, 20971520),
meta=np.ndarray>
    rlus     (time, ncells) float32 dask.array<chunksize=(4, 20971520),
meta=np.ndarray>
    hfss     (time, ncells) float32 dask.array<chunksize=(4, 20971520),
meta=np.ndarray>
    hfls     (time, ncells) float32 dask.array<chunksize=(4, 20971520),
meta=np.ndarray>
Attributes:
    CDI:                 Climate Data Interface version 1.8.0rc7
(http://mpi...
    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@git.mpimet.mpg.de:icon-
aes.git@88370a120a6b2e91...
    history:             /home/dkrz/k203123/Dyamond_pp/icon-aes-dyamond-
pp.g...
    references:          see MPIM/DWD publications
    comment:             Sapphire Dyamond (k203123) on m21460 (Linux
2.6.32-...
```

There are 372 timesteps in the data. The data itself is already represented by `dask.array` types of a chunksize=(4, 20971520). With 6 hourly output data this means each chuck contains one day worth of output data. If you would like learn more on chunking please refer to https://docs.dask.org/en/latest/array-chunks.html.

Now it's time to do the actual calculation. The aim of this tutorial is creating time-series' of datasets in various ways and compare the processing time. The first step, and in this case last step, is applying an average over the spatial (ncells) dimension. Note that the ICON grid-cells have equal area so we do not have to apply weighted averaging.

Before applying the the average let's inspect each data array a little further:

```
dset['rsut'].data
```

```
dask.array<concatenate, shape=(372, 20971520), dtype=float32, chunksize=(4,
20971520), chunktype=numpy.ndarray>
```

We can see that the data has one dimension in for the horizontal axis (20971520 grid points) and one for the time axis (372). The `data` property gives access to the underlying `dask.array` object. We can see that this variable has a size of 31.21 GB and is split into 93 chunks of 335.54 MB each. `client.persist` will push the data onto the cluster - which means the dataset is read into memory *on* the cluster. But before pushing the data onto the cluster we want to define the steps for processing the data - or data reduction since we apply an average.

```
ts_future = dset.mean(dim='ncells')
ts_future
```

```
<xarray.Dataset>
Dimensions:  (time: 372)
Coordinates:
  * time     (time) float64 2.016e+07 2.016e+07 ... 2.016e+07 2.016e+07
Data variables:
    rsds     (time) float32 dask.array<chunksize=(4,), meta=np.ndarray>
    rlut     (time) float32 dask.array<chunksize=(4,), meta=np.ndarray>
    rsut     (time) float32 dask.array<chunksize=(4,), meta=np.ndarray>
    rsus     (time) float32 dask.array<chunksize=(4,), meta=np.ndarray>
    rsdt     (time) float32 dask.array<chunksize=(4,), meta=np.ndarray>
    rlds     (time) float32 dask.array<chunksize=(4,), meta=np.ndarray>
    rlus     (time) float32 dask.array<chunksize=(4,), meta=np.ndarray>
    hfss     (time) float32 dask.array<chunksize=(4,), meta=np.ndarray>
    hfls     (time) float32 dask.array<chunksize=(4,), meta=np.ndarray>
```

At this point nothing has been calculated. `ts_future` is only a representation of what the data *will* look like. Hence it is called a *future* or a *lazy* object. To initiate the calculation whe need to 'transfere', or better serialize, the `ts_future` onto the cluster. this can be done by with the `client.persit` method. On the cluster a process reading and averaging the data will start. The `esm_analysis.progress_bar` method gives us a feedback of what is going on:

```
# Start the calculation
progress_bar(client.persist(ts_future), notebook=False, label='Avg.')
# notebook=False prints a progress bar in ascii output
# this option has been chosen to make the output of
# the progress bar persistent.
```

```
Avg.: 100%|██████████| 837/837 [02:47<00:00, 4.99it/s]
```

`dask` was calculating that reading the dataset and averaging is 837 tasks. It created a task graph and distributed the tasks across the cluster. All together it took 2:47 Minutes to do the averaging.

This is a decent time. But there is catch, most of the time is spend reading the data - the file system

on mistral can be slow. If you want to do additional analysis like a time mean, or standard deviations, grouping, correlation it might be a better idea to push the whole dataset onto the cluster and do then the analysis. Once the data is on the cluster memory access is much faster. Although in our case this is not necessary lets demonstrate the difference. We first clean the clusters memory:

```
client.restart()
```

```
distributed.scheduler - ERROR - Not all workers responded positively: ['OK',
'OK', 'timed out', 'OK', 'OK', 'OK', 'OK', 'OK', 'OK', 'OK']
NoneType: None
distributed.client - ERROR - Restart timed out after 20.000000 seconds
```

```
<Client: 'tcp://136.172.50.18:44639' processes=9 threads=288, memory=1.35
TB>
```

We have 1.5 TB of memory, this means the whole dataset will fit comfortably into it. So lets push the dataset onto the cluster and watch the progress while the dataset is being read:

```
future_data = client.persist(dset)
progress_bar(future_data, notebook=False)
```

```
Progress: 100%|██████████| 837/837 [03:27<00:00, 4.04it/s]
```

Now the whole dataset has been loaded into the clusters memory. This time it took 03:27 Minutes, hence the majority of the processing time in the previous step was actually spend reading the data. But now that the data is on the clusters memory any operation will be fast:

```
future_ts = client.persist(future_data.mean(dim='ncells'))
```

```
progress_bar(future_ts, notebook=False)
```

```
Progress: 100%|██████████| 837/837 [00:16<00:00, 50.2it/s]
```

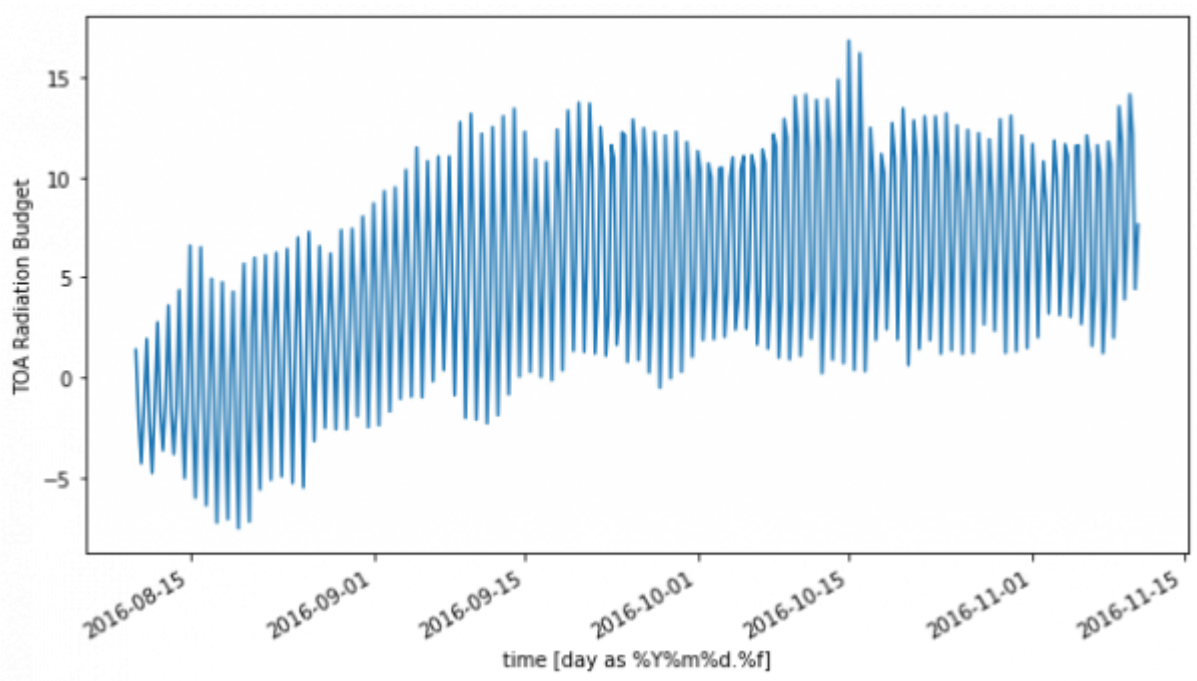
Now this time the calculation took just over 15 seconds. It is time to pull back the calculated data from the cluster:

```
ts_data = future_ts.compute()
```

```
toa_budget = (ts_data['rsdt'] - ts_data['rsut']) - ts_data['rlut']
```

```
%matplotlib inline
fig, ax = plt.subplots(1, 1, figsize=(10,5))
# Convert the time variable to a pandas datetime object using
esm_analysis.icon2datetime
toa_budget.time.values = icon2datetime(toa_budget.time.values)
toa_budget.time.attrs['uints'] = 'time'
toa_budget.attrs['long_name'] = 'TOA Radiation Budget'
# Plot the time-series
```

```
_ = toa_budget.plot(ax=ax)
```



Using pure bash

This section will discuss how to do the above described data processing routines using pure bash and cdo. The strategy here is to distribute the files that will be processed to a spawned slurm cluster using a so called *array-job* and a slurm *dependency* job that merges the results.

Before we start lets close the previously used cluster:

```
client.close()
Cluster.close()
```

```
$: squeue -u $USER
```

```
             JOBID PARTITION     NAME     USER ST       TIME  NODES
NODELIST(REASON)
```

The strategy here would be splitting the files which cdo will be working into a given number (number of nodes). And then distributing the filenames to each node. This can be done by an *array job*. Lets use 10 computing nodes and split up list filenames into 10 chunks.

```
filenames_split = np.array_split((list(filenames)), 10)

# Save this array for usage in script
np.savez(Path(cdo_dir.name) / 'filenames.npz',files=filenames_split)
filenames_split[:2] # Just check out the first two file names
```

```
[array(['/work/mh0287/k203123/experiments/dpp0001/201609/dpp0001_atm_2d_ml_2
0160925T000000Z.nc',
```

```
'/work/mh0287/k203123/experiments/dpp0001/201609/dpp0001_atm_2d_ml_20160919T
000000Z.nc',
'/work/mh0287/k203123/experiments/dpp0001/201609/dpp0001_atm_2d_ml_20160911T
000000Z.nc',
'/work/mh0287/k203123/experiments/dpp0001/201609/dpp0001_atm_2d_ml_20160918T
000000Z.nc',
'/work/mh0287/k203123/experiments/dpp0001/201610/dpp0001_atm_2d_ml_20161015T
000000Z.nc',
'/work/mh0287/k203123/experiments/dpp0001/201610/dpp0001_atm_2d_ml_20161001T
000000Z.nc',
'/work/mh0287/k203123/experiments/dpp0001/201610/dpp0001_atm_2d_ml_20161011T
000000Z.nc',
'/work/mh0287/k203123/experiments/dpp0001/201608/dpp0001_atm_2d_ml_20160813T
000000Z.nc',
'/work/mh0287/k203123/experiments/dpp0001/201608/dpp0001_atm_2d_ml_20160818T
000000Z.nc',
'/work/mh0287/k203123/experiments/dpp0001/201610/dpp0001_atm_2d_ml_20161024T
000000Z.nc'],
       dtype='<U85'),
array(['/work/mh0287/k203123/experiments/dpp0001/201610/dpp0001_atm_2d_ml_20
161006T000000Z.nc',
'/work/mh0287/k203123/experiments/dpp0001/201609/dpp0001_atm_2d_ml_20160917T
000000Z.nc',
'/work/mh0287/k203123/experiments/dpp0001/201609/dpp0001_atm_2d_ml_20160920T
000000Z.nc',
'/work/mh0287/k203123/experiments/dpp0001/201610/dpp0001_atm_2d_ml_20161023T
000000Z.nc',
'/work/mh0287/k203123/experiments/dpp0001/201609/dpp0001_atm_2d_ml_20160923T
000000Z.nc',
'/work/mh0287/k203123/experiments/dpp0001/201610/dpp0001_atm_2d_ml_20161007T
000000Z.nc',
'/work/mh0287/k203123/experiments/dpp0001/201610/dpp0001_atm_2d_ml_20161005T
000000Z.nc',
'/work/mh0287/k203123/experiments/dpp0001/201610/dpp0001_atm_2d_ml_20161020T
000000Z.nc',
'/work/mh0287/k203123/experiments/dpp0001/201608/dpp0001_atm_2d_ml_20160828T
000000Z.nc',
'/work/mh0287/k203123/experiments/dpp0001/201610/dpp0001_atm_2d_ml_20161009T
000000Z.nc'],
       dtype='<U85')]
```

Let's create the script that is submitted to the slurm cluster

```
worker_script = """#!/bin/bash

#SBATCH -J cdo_worker
#SBATCH -D {out_dir}
#SBATCH -p {queue}
#SBATCH -A {account}
```

```
#SBATCH -n 1
#SBATCH --cpus-per-task=32
#SBATCH --mem=140G
#SBATCH -t {walltime}
#SBATCH --array=1-{nworkers}

let id=$SLURM_ARRAY_TASK_ID

module purge
module load cdo/1.6.9-magicsxx-gcc48

cdo=/sw/rhel6-x64/cdo/cdo-1.9.8-magicsxx-gcc64/bin/cdo
python=/sw/rhel6-x64/python/python-3.5.2-gcc49/bin/python
varnames="{varnames}"
out_dir={out_dir}

# Get the filenames this worker should be processing
declare -a files=($($python -c "import numpy as np; print('
'.join(np.load('{lookup}')['files'][int($id)-1].tolist()))"))
date >> $out_dir/cdo_submit
if [ -z "$files" ];then
    exit 0
fi
out_id=$(echo $id |awk '{{printf("%04d", $1)}}')
# 1) Do spatial averaging averaging
mkdir -p ${{out_dir%/}}/tmp_${{out_id}}
for var in $varnames;do
   $cdo -O fldmean -mergetime -apply,selvar,$var [ ${{files[*]}} ]
$out_dir/tmp_${{out_id}}/fld_${{var}}.nc &
done
wait
$cdo -O merge $out_dir/tmp_${{out_id%/}}/fld_*.nc
$out_dir/fldmean_${{out_id}}.nc
""".format(
    out_dir=cdo_dir.name,
    queue='prepost',
    account='mh0731',
    walltime='01:00:00',
    nworkers=10,
    varnames=' '.join(VARNAMES),
    lookup=Path(cdo_dir.name) / 'filenames.npz',

          )
```

```
print(worker_script)
```

```
#!/bin/bash

#SBATCH -J cdo_worker
#SBATCH -D /scratch/m/m300765/CdoBenchmark_xp7w0mu1
#SBATCH -p prepost
```

```
#SBATCH -A mh0731
#SBATCH -n 1
#SBATCH --cpus-per-task=32
#SBATCH --mem=140G
#SBATCH -t 01:00:00
#SBATCH --array=1-10

let id=$SLURM_ARRAY_TASK_ID

module purge
module load cdo/1.6.9-magicsxx-gcc48

cdo=/sw/rhel6-x64/cdo/cdo-1.9.8-magicsxx-gcc64/bin/cdo
python=/sw/rhel6-x64/python/python-3.5.2-gcc49/bin/python
varnames="hfls rlus rsdt rsds hfss rsut rlut rsus rlds"
out_dir=/scratch/m/m300765/CdoBenchmark_xp7w0mu1

# Get the filenames this worker should be processing
declare -a files=($($python -c "import numpy as np; print('
'.join(np.load('/scratch/m/m300765/CdoBenchmark_xp7w0mu1/filenames.npz')['fi
les'][int($id)-1].tolist()))"))
date >> $out_dir/cdo_submit
if [ -z "$files" ];then
    exit 0
fi
out_id=$(echo $id |awk '{printf("%04d", $1)}')
# 1) Do spatial averaging averaging
mkdir -p ${out_dir%/}/tmp_${out_id}
for var in $varnames;do
   $cdo -O fldmean -mergetime -apply,selvar,$var [ ${files[*]} ]
$out_dir/tmp_${out_id}/fld_${var}.nc &
done
wait
$cdo -O merge $out_dir/tmp_${out_id%/}/fld_*.nc
$out_dir/fldmean_${out_id}.nc
```

The key here is the `#SBATCH --array=1-10` slurm directive. This tells slurm to launch 10 jobs. Each job can be accessed by the environment variable `$SLURM_ARRAY_TASK_ID`. The idea is then to open the saved numpy file and get the filenames each worker is supposed to work on. More information on *array jobs* can be found on https://slurm.schedmd.com/job_array.html.

Note: cdo will make use of the `apply` command, which is new in version 1.9.8. Now we will have 10 jobs working on a subset of the total data. Now let's create a job-script that merges back the results:

```
merge_script = """#!/bin/bash

#SBATCH -J cdo_merger
#SBATCH -p {queue}
#SBATCH -A {account}
```

```
#SBATCH -D {out_dir}
#SBATCH -n 1
#SBATCH --cpus-per-task=32
#SBATCH --mem=140G
#SBATCH -t {walltime}

module purge
module load cdo/1.6.9-magicsxx-gcc48
out_dir={out_dir}
n_workers={nworkers}
info_file={info_file}
cdo=/sw/rhel6-x64/cdo/cdo-1.9.8-magicsxx-gcc64/bin/cdo
python=/sw/rhel6-x64/python/python-3.5.2-gcc49/bin/python

fld_mean=$(find $out_dir -name "fldmean_*.nc" -type f|sort)
date_file=${{out_dir}}/cdo_submit
#1) Merge the files
$cdo -O mergetime ${{fld_mean[*]}} ${{out_dir}}/out_fldmean_out.nc
let now=$(date +%s)

min=$now
dates=$(date -f $date_file +%s)
for i in $(date -f $date_file +%s);do
    (( i < min )) && min=$i
done
let dt=$(date +%s)-$min
echo "runtime : $dt seconds" > $info_file

""".format(
    out_dir=cdo_dir.name,
    queue='prepost',
    account='mh0731',
    walltime='01:00:00',
    nworkers=10,
    varnames=' '.join(VARNAMES),
    info_file=Path('.').absolute() / 'runtime.txt')
```

```
print(merge_script)
```

```
#!/bin/bash

#SBATCH -J cdo_merger
#SBATCH -p prepost
#SBATCH -A mh0731
#SBATCH -D /scratch/m/m300765/CdoBenchmark_xp7w0mu1
#SBATCH -n 1
#SBATCH --cpus-per-task=32
#SBATCH --mem=140G
#SBATCH -t 01:00:00

module purge
```

```
module load cdo/1.6.9-magicsxx-gcc48
out_dir=/scratch/m/m300765/CdoBenchmark_xp7w0mu1
n_workers=10
info_file=/mnt/lustre01/pf/zmaw/m300765/workspace/dyamond_PostProc/docs/runtime.txt
cdo=/sw/rhel6-x64/cdo/cdo-1.9.8-magicsxx-gcc64/bin/cdo
python=/sw/rhel6-x64/python/python-3.5.2-gcc49/bin/python

fld_mean=$(find $out_dir -name "fldmean_*.nc" -type f|sort)
date_file=${out_dir}/cdo_submit
#1) Merge the files
$cdo -O mergetime ${fld_mean[*]} ${out_dir}/out_fldmean_out.nc
let now=$(date +%s)

min=$now
dates=$(date -f $date_file +%s)
for i in $(date -f $date_file +%s);do
    (( i < min )) && min=$i
done
let dt=$(date +%s)-$min
echo "runtime : $dt seconds" > $info_file
```

The clue is that this script should be submitted after all jobs in the array job have finished. This can be achieved by using the `--depend=afterok:job_id` command line argument of sbatch. Let's write the job-scripts to disk and submit them.

```
with open(str(Path(cdo_dir.name) / 'worker.sh'), 'w') as f:
    f.write(worker_script)
with open(Path(cdo_dir.name) / 'merger.sh', 'w') as f:
    f.write(merge_script)

(Path(cdo_dir.name) / 'worker.sh').chmod(0o755)
(Path(cdo_dir.name) / 'merger.sh').chmod(0o755)

# Submit the array job and get the job_id
res = run([f'sbatch',  f'{Path(cdo_dir.name) / "worker.sh"}'],
cwd=cdo_dir.name, stdout=PIPE, check=True)
job_id, _, _cluster = res.stdout.decode('utf-8').strip().partition(';')
job_id = job_id.split(" ")[-1]
# Now submit the job depending on the array job
res = run([f'sbatch', f'--depend=afterok:{job_id}', f'{Path(cdo_dir.name) /
"merger.sh"}'],
          cwd=cdo_dir.name, stdout=PIPE, check=True)
```

```
$: squeue -u $USER
```

```
             JOBID PARTITION     NAME     USER ST       TIME  NODES
NODELIST(REASON)
          19581787   prepost cdo_merg  m300765 PD       0:00      1
```

```
(Dependency)
        19581786_1   prepost cdo_work  m300765  R       0:03      1 m11542
        19581786_2   prepost cdo_work  m300765  R       0:03      1 m11546
        19581786_3   prepost cdo_work  m300765  R       0:03      1 m11547
        19581786_4   prepost cdo_work  m300765  R       0:03      1 m11556
        19581786_5   prepost cdo_work  m300765  R       0:03      1 m11514
        19581786_6   prepost cdo_work  m300765  R       0:03      1 m11516
        19581786_7   prepost cdo_work  m300765  R       0:03      1 m11517
        19581786_8   prepost cdo_work  m300765  R       0:03      1 m11518
        19581786_9   prepost cdo_work  m300765  R       0:03      1 m11520
       19581786_10   prepost cdo_work  m300765  R       0:03      1 m11521
```

The array job is already running while the job that depends on the outcome of the array job is still in the queue. After the array jobs have successfully finished it will start processing. Let's see how much time the whole procedure took. Once the all processing is finished its time is written into the file `runtime.txt`.

```
cat runtime.txt
```

```
runtime : 70 seconds
```

The runtime using slurm array jobs and cdo was this time only *70 Seconds*. Thats a huge improvement compared to the 2:43 Minutes using dask-distributed and xarray.

Combining Dask and CDO

Especially in interactive sessions, like this notebook using scripts is kind of awkward. The question is what would happen if we used cdo commands together with the dask-distributed scheduler?

To do this we'll have to use a python wrapper for cdo. It is already pre-installed in the anaconda environment. Let's import it and define some helper functions that will execute the cdo jobs on the cluster:

```
from cdo import Cdo
cdo = Cdo(cdo='/sw/rhel6-x64/cdo/cdo-1.9.8-magicsxx-gcc64/bin/cdo')


# Define the functions that will do the work
def _fldmean(files, var, out_dir):
    """Calculate fieldmean using cdo.

    Parameters
    ----------

    files : collection of str
        Input filenames

    var: str
        Variable to apply the averaging on
```

```
    out_dir: str
        Out put directory

    Returns
    -------
        str: output filename

    """

    out_dir = Path(out_dir)
    out_dir.mkdir(parents=True, exist_ok=True)
    return cdo.fldmean(f' -mergetime -apply,selvar,{var} [{" ".join(files)}
]',
                       output=f'{str(out_dir)}/tmp_{var}.nc')

def _merge(idx, in_dir, out_dir):
    """Merge datasets of different Variables.

    Parameters
    ----------

    idx: int
        Number of the distributed batch job

    in_dir: str
        Input directory
    out_dir: str
        Output directory

    Returns
    -------
    str: output filename

    """
    return cdo.merge(input=f'{Path(in_dir)/"tmp"}_*.nc',
output=f'{Path(out_dir)/"fldmean_"}{idx}.nc')

def _mergetime(in_dir):
    """Apply cdo mergetime."""
    return cdo.mergetime(input=f'{Path(in_dir)/"fldmean_*.nc"}',
returnXDataset=True).compute()
```

We follow the strategy from the slurm example above and split all filenames into 10 workers and then call the above defined functions for each slab of filenames. Now that we have defined the functions that will work on the cluster we still have to define a function that submits the tasks to the cluster:

```
# Define function that submits the tasks to the cluster
def create_ts(files, variables, out_dir):
```

```
    futures = []
    files = np.array_split(list(files), len(client.nthreads()))
    for idx, filenames in enumerate(files):
        for var in variables:
            # Use the submit method to submit a function to the cluster
            futures.append(client.submit(_fldmean, filenames, var,
Path(out_dir.name)/f'tmp_{idx}'))
    progress_bar(futures, notebook=False)
    wait(futures)
    futures = []
    # Merge the output
    for idx in range(len(files)):
        tmp_dir = Path(out_dir.name)/f'tmp_{idx}'
        futures.append(client.submit(_merge, idx, tmp_dir, out_dir.name))
    progress_bar(futures, notebook=False)
    wait(futures)
    futures = []
    # And finally apply the merge time
    futures.append(client.submit(_mergetime, out_dir.name))
    progress_bar(futures, notebook=False)
    # Return the result for processing
    return client.gather(futures)[0]
```

```
client.close()
Cluster.close()
```

```
# restart the slurm cluster

Cluster = SLURMCluster(memory="150GB",
                       cores=31,
                       project='mh0731',
                       walltime='02:00:00',
                       name='dask_benchmark',
                       queue='prepost')

Cluster.scale(10)
```

```
{"model_id":"48c7e0b2022f451ab9054e6ef6863966","version_major":2,"version_mi
nor":0}
```

```
client = Client(Cluster)
```

```
from datetime import datetime
start = datetime.now()
out = create_ts(filenames, VARNAMES, cdodask_dir)
dt = datetime.now() - start # Get the time difference
```

```
Progress: 100%|██████████| 90.0/90.0 [00:48<00:00, 1.84it/s]
Progress: 100%|██████████| 10.0/10.0 [00:01<00:00, 6.79it/s]
Progress: 100%|██████████| 1.00/1.00 [00:01<00:00, 1.95s/it]
```

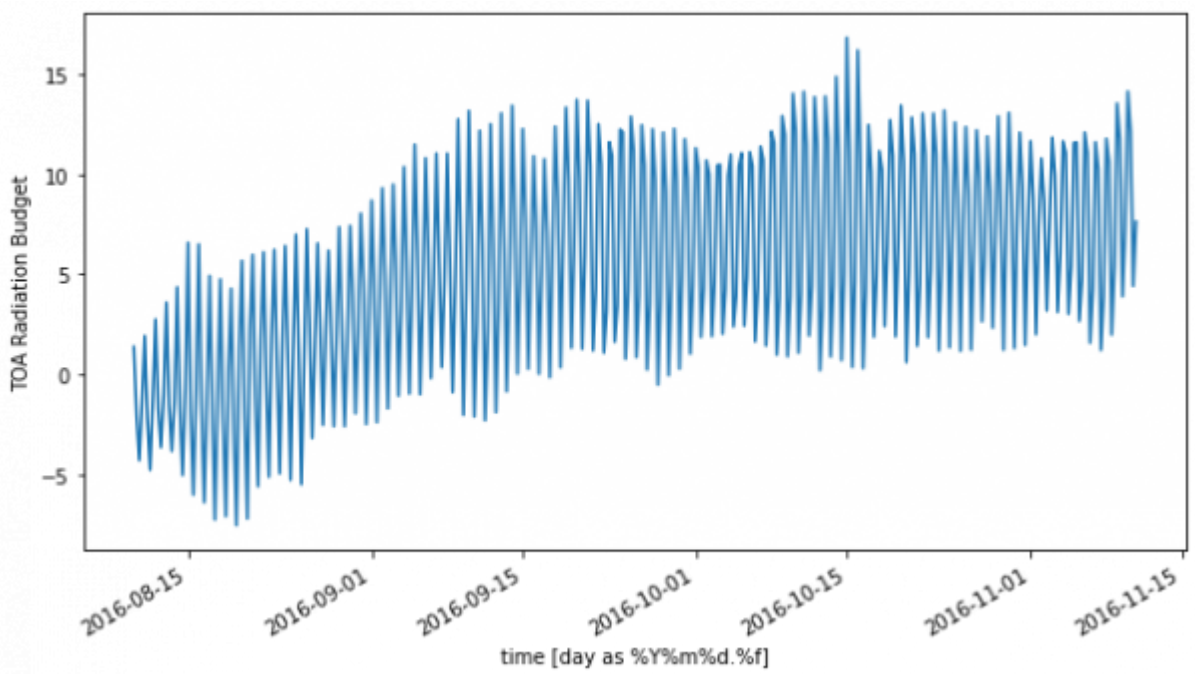
```
dt #Check how much time it took to process the data
```

```
datetime.timedelta(seconds=52, microseconds=521233)
```

It just took a little more than 50 seconds to do the computing. This is another speedup of nearly 30% - not bad. The reason is that the scheduler of dask distributes the work that needs to been done most efficient.

```
toa_budget = (out['rsdt'][:, 0, 0] - out['rsut'][:, 0, 0]) - out['rlut'][:,
0, 0]
```

```
%matplotlib inline
fig, ax = plt.subplots(1, 1, figsize=(10,5))
# Convert the time variable to a pandas datetime object using
esm_analysis.icon2datetime
toa_budget.time.values = icon2datetime(toa_budget.time.values)
toa_budget.time.attrs['uints'] = 'time'
toa_budget.attrs['long_name'] = 'TOA Radiation Budget'
# Plot the time-series
_ = toa_budget.plot(ax=ax)
```




```
client.close()
Cluster.close()
```

Conclusion

We have tested three different approaches to process output data from global storm resolving models. This data usually doesn't fit into the memory of a single machine. Hence distributed analysis

techniques have to be applied.

In this tutorial we used the `dask-distributed` to analyze data on a cluster of servers. Since parallel file systems like lustre or gpfs are slow compared to non parallel hard drives the bottlenecks in distributed data processing is reading the data.

Using `dask` and `xarray` we noticed that most of the time is spend reading the data. Our small task took roughly 2:30 Minutes using 10 working nodes. On the other hand once the dataset has been loaded into the distributed memory of the cluster any kind of data processing from applying simple data reduction like averaging to sophisticated machine learning is fast. Here lies the clear advantage of this approach. If the *whole* dataset fits into the distributed memory on the cluster - typically in the order of few TB - data access is fast and efficient. Users who want to do exploratory data analysis or sophisticated data processing like deep learning might want to go with this approach.

A completely different style is shell only based. The slurm workload manager can be used to split up similar tasks into an array of jobs. This can be combined with the *–depend* command line argument to wait for specific tasks. Hence it is possible to generate so called Directed Acyclic Graphs of tasks. In our example we prepared two bash scripts: one that is launched as an array job and works at a different files at a time and one job that waits upon successfully finishing the array job to merge the results. More information can be found on https://slurm.schedmd.com/job_array.html. This approach was significantly faster than the first approach mainly because cdo is very fast in reading data. Yet, there are limitations. For example interactive processing can be complicated when using bash scripts from within say a jupyter notebook session. Also it is often hard to do exploratory data analysis. Finally cdo has only limited to no capability to apply machine learning algorithms.

In a last step we combined the distributed processing capability of dask and cdo. This is possible because of a python wrapper that exists for cdo. Here, we could get even further speed gains. The advantage is that using cdo within a python session makes exploratory data analysis easier than using pure shell scripting. Yet limitations when it comes to machine learning remain.

There is no general answer which approach of the above presented ones are be best. For exploratory, interactive data processing or heavy use of machine learning `dask-distributed` in combination with `xarray` is clearly the way to go. If the data processing task is re-occurring a script based approach might be favorable. A compromise could be using the cdo python bindings together with dask distributed.