Parallel Analysis with CDO

Search
Search only in this Namespaces below. For a global search use the field in the upper right corner.
More tipps: how_to_use_the_wikisearch

Python and CDO

In working with the DYAMOND output a combination of shell/python scripting and the CDOs was found to be effective. The general syntax is shown in the pseudo-code snippet below.

```
import glob
from   subprocess import call
from   concurrent.futures import ProcessPoolExecutor, as_completed, wait

nprocs = 12 # specify number of processors to use (depends on node)
pool   = ProcessPoolExecutor(nprocs)
tasks  = []  # list for references to submitted jobs

vname  = 'ATHB_T'
for file in glob.glob('*.nc'):
    arg  = 'cdo -P 2 selname,%s %s %s'%(vname, file, file[:-3]+vname+'.nc')
    task = pool.submit(call, arg, shell=True)
    tasks.append(task)

for task in as_completed(tasks): # helps see as tasks are completed
    print(task.result())

wait(tasks) # useful if further tasks depend on completion of all tasks
above
```

The above logic is applied to a real, and more complex, processing task in the script below. This script was executed on the post-processing nodes of Mistral, and was used to extract, re-map, and re-compose files of single variables. A special step was required from some output variables whose *means* had been inadvertently accumulated and had to be de-accumulated to be useful.

get_fields_icon.py

```
# Author: Bjorn Stevens
# Date: 27 Jan 2019
# Purpose:  post processes 2D ICON DYAMOND output to create 2D fields
at all time-steps
# Radiation fields which were accumulated as means in the 2d_avg_ml
files are de-accumulated.
#
import pandas as pd
```

```
import numpy as np
import xarray as xr
import os
from   cdo import Cdo
from   subprocess import call, check_output
from   concurrent.futures import ProcessPoolExecutor, as_completed,
wait

cdo      = Cdo()
cdo.setCdo('/sw/rhel6-x64/cdo/cdo-1.9.5-gcc64/bin/cdo')
print('Using',cdo.version())

nprocs = 10
vnames =
['ASOU_T','ATHB_T','ASOB_T','ASOB_S','ASWDIFU_S','ATHB_S','ATHU_S','ATH
D_S']

vers   = '5.0km_2'
pool   = ProcessPoolExecutor(nprocs)
tasks  = []

# this is the variable dictionary and it indicates where a variable
lives
# variables in 'atm_2d_avg_ml_' must be deaccumulated
#
vardict ={'ASOU_T'   : 'atm_2d_avg_ml_'
         ,'ATHB_T'   : 'atm_2d_avg_ml_'
         ,'ASOB_T'   : 'atm_2d_avg_ml_'
         ,'ASOB_S'   : 'atm_2d_avg_ml_'
         ,'ASWDIFU_S': 'atm_2d_avg_ml_'
         ,'ATHB_S'   : 'atm_2d_avg_ml_'
         ,'ATHU_S'   : 'atm_2d_avg_ml_'
         ,'ATHD_S'   : 'atm_2d_avg_ml_'
         ,'TQV_DIA'  : 'atm1_2d_ml_'
         ,'TQC_DIA'  : 'atm1_2d_ml_'
         ,'TQI_DIA'  : 'atm1_2d_ml_'
         ,'TQG'      : 'atm1_2d_ml_'
         ,'TQS'      : 'atm1_2d_ml_'
         ,'TQR'      : 'atm3_2d_ml_'
         ,'PMSL'     : 'atm2_2d_ml_'
         ,'LHFL_S'   : 'atm2_2d_ml_'
         ,'SHFL_S'   : 'atm2_2d_ml_'
         ,'TOT_PREC' : 'atm2_2d_ml_'
         ,'CLCT'     : 'atm2_2d_ml_'
         ,'U_10M'    : 'atm3_2d_ml_'
         ,'V_10M'    : 'atm3_2d_ml_'
         ,'T_2M'     : 'atm3_2d_ml_'
         ,'QV_2M'    : 'atm3_2d_ml_'
         ,'UMFL_S'   : 'atm4_2d_ml_'
```

```
          ,'VMFL_S'    : 'atm4_2d_ml_'
          ,'TG'        : 'atm4_2d_ml_'
          ,'QV_S'      : 'atm4_2d_ml_'
          ,'CIN_ML'    : 'atm4_2d_ml_'
          ,'CAPE_ML'   : 'atm2_2d_ml_'
          }

scr    = '/scratch/m/m219063/'
grid   = '/work/ka1081/Hackathon/GrossStats/0.10_grid.nc'

for vname in vnames:
    sffx   = vardict[vname]
    fnm    = '/work/mh0492/m219063/DYAMOND/ICON-%s_%s.nc'%(vers,vname)
    if (vers=='5.0km_1'):
        wght   =
'/work/ka1081/Hackathon/GrossStats/ICON_R2B09_0.10_grid_wghts.nc'
        stem   =
'/work/ka1081/DYAMOND/ICON-5km/nwp_R2B09_lkm1006_%s'%(sffx,)
    elif (vers=='5.0km_2'):
        wght   = '/work/ka1081/Hackathon/GrossStats/ICON_R2B09-
mpi_0.10_grid_wghts.nc'
        stem   =
'/work/ka1081/DYAMOND/ICON-5km_2/nwp_R2B09_lkm1013_%s'%(sffx,)
    elif (vers=='2.5km'):
        wght   =
'/work/ka1081/Hackathon/GrossStats/ICON_R2B10_0.10_grid_wghts.nc'
        stem   =
'/work/ka1081/DYAMOND/ICON-2.5km/nwp_R2B10_lkm1007_%s'%(sffx,)
    else:
        print ('version not supported')
        exit

    if (os.path.isfile(fnm)):
        print (fnm+' exists, not overwriting')
    else:
            time= pd.date_range('2016-08-01','2016-09-09', freq='1D')
            for i in np.arange(time.size):
                ddat = stem  +
time[i].strftime("%Y%m%d")+'T000000Z.grb'
                tmp1 = scr+vname + '_' + time[i].strftime("%m%d")+'.nc'
            arg  = 'cdo -O -P 2 -f nc4 remap,%s,%s -selname,%s %s
%s'%(grid,wght,vname,ddat,tmp1)
            task = pool.submit(call, arg, shell=True)
            tasks.append(task)

        for task in as_completed(tasks):
            print(task.result())
        wait(tasks)

    if vardict[vname] == 'atm_2d_avg_ml_':
        fur     = '/work/mh0492/m219063/DYAMOND/ICON-
```

```
%s_%s_deacc_0.10deg.nc'%(vers,vname)
        if (os.path.isfile(fur)):
            print (fur+' exists, not overwriting')
        else:
            cdo.mergetime(input = '%s%s_????.nc'%(scr,vname), output =
fnm, options =  '-P 4')
            nt_max = xr.open_dataset(fnm,autoclose=True).time.size
            for k in range(2,nt_max):
                tmp2 = '%s%s_step-%4.4i.nc'%(scr,vname,k,)
                arg  = 'cdo -O -P 2 sub -mulc,%i -seltimestep,%i %s -
mulc,%i -seltimestep,%i %s %s'%(k-1,k,fnm,k-2,k-1,fnm,tmp2)
                task = pool.submit(call, arg, shell=True)
                tasks.append(task)

            for task in as_completed(tasks):
                print(task.result())
            wait(tasks)

            cdo.mergetime(input = '%s%s_step-????.nc'%(scr,vname),
output =  fur, options =  '-P 4')
    else:
        fur    = '/work/mh0492/m219063/DYAMOND/ICON-
%s_%s_0.10deg.nc'%(vers,vname)
        if (os.path.isfile(fur)):
            print (fur+' exists, not overwriting')
        else:
            cdo.mergetime(input = '%s%s_????.nc'%(scr,vname), output =
fur, options =  '-P 4')

    os.system('rm %s%s_????.nc'%(scr,vname))
    os.system('rm %s%s_step-????.nc'%(scr,vname))
```

Scheduling parallel jobs from Python

In this example a function *show_timestamp* is defined and executed in parallel over a list of files. Here resources in the forms of cores and processors are explicitly allocated. To use this approach for more general tasks one would need to define the appropriate function that is being executed concurrently, i.e., redefine *show_timestamp*. In addition, the user specific paths and project accounts would have be set appropriately for the system you are working on.,

```
import os
import glob
import subprocess as sp

# for distributed computing
```

```
from distributed import Client
from dask_jobqueue import SLURMCluster

files = sorted(glob.glob("/work/bm0834/k203095/ICON_LEM_DE/20130502-default-readccn_v2/DATA/2*.nc"))

def show_timestamp(filename):
    return sp.check_output(['cdo', '-s', 'showtimestamp', filename])

slurm_options = {
    'project': 'bm0834',
    'cores': 8,  # no. of threads
    'processes': 4,
    'walltime': '01:00:00',
    'queue': 'compute2,compute',
    'memory': '64GB',
    'python': '/work/bm0834/k202101/anaconda3/bin/python',
    'interface': 'ib0',
    'local_directory': '/home/dkrz/k202101/dask-worker-space',
}

cluster = SLURMCluster(**slurm_options)
cluster.start_workers(4)
client = Client(cluster)
%%time
res = client.gather(futures)

futures = [client.submit(show_timestamp, f) for f in files]
cluster.close()
client.close()
```

Bash (shell script) and CDO

A *poor-person's* parallelization can also be accomplished by looping over cdo commands in a shell script, and putting them in the background. This is best done when subsequent commands are not dependent on the completion of the jobs sent to the background and when one knows how many processors are available on the node, and matches this to the length of the loop. But in reality this stub has been introduced in the hope that someone picks up on it and suggests parallel cdo implementation with bash similar to the example for Python.

DASK and Xarray

For many tasks that are not too computationally intensive and not performed over a grid – in which case preprocessing with CDOs may be more effective – parallel post processing can be efficiently performed using DASK and xarray.

For this Pavan Siligam of DKRZ has been developing use cases as part of the HD(CP)2 project. These are maintained here:

https://nbviewer.jupyter.org/urls/gitlab.dkrz.de/k202101/hdcp2_postprocessing_notebooks/raw/master/notebooks/Index.ipynb

Guido Cioni provided an in-depth tutorial here which may also be applicable to tasks usually reserved for CDOs.