Scripts for Output

I show you how to use Python for compressing MPI-ESM output. You only need to think in two things: what do you want to store and how much MISTRAL resources you will use for that end. For the distribution of the compression, I use the ideas given in the Reserve and Use method.

Python script

What do I want to store?

MPI-ESM have several streams of output that come from four submodels: ECHAM, JSBACH, MPIOM and HAMOCC. Depending on your application, the model output can be massive. For storing in HPSS tape system, DKRZ recommends that each file you want to upload should be larger than 10 Gb but lesser than 500 Gb. If they are less than 1 Gb, they count as 1 Gb for your project's quota. Thus, you should be careful of what you want to store.

Therefore, before any compression, you should think about what you need (or what you will need) from the output data. Once you know that, you can check the size of your dataset by using commands such as `du` or `ls`. Knowing that enables you to prepare the data for storage. I offer you the following scripts to make the task as simple as possible.

The main script is extensible yet straightforward. I describe the different sections in the following. In the end, I offer the full script. First, we need to import the following modules

```
import os
import subprocess as sp
from dask.distributed import Client
```

The `os` module allows us to perform some operative and file system tasks, such as directory changing. The `subprocess` module makes possible to launch subprocesses to run programs outside the python session, in particular, shell programs. The `dask.distributed` module enables the use of MISTRAL to distribute and parallelise computations. About this last point, the script is a direct application of the Reserve and Use method.

Once we have imported the modules, we first connect to the computing resources of MISTRAL (section 0 of the code). In the following sections, I will talk about it. Thus, for now, let us give it for granted.

```
# 0. Acquire the reserved resources in the cluster

client=Client(scheduler_file=('/scratch/x/x000000/dask-work/Storage3d'+
                              '/scheduler_Storage3d.json'
                             )
```

```
              )
print(client)
```

With the resources secured, we now define the paths to the model output and to the temporary space where we keep the compressed files (section I). I provide dummy paths. I strongly recommend that the temporary space is in `/scratch`, because of the large amounts of disk space that the compressed files occupy.

```
# I. Definition of paths to the data and the scratch space

exps_path=os.path.abspath(("/work/yy0000/x000000/"+
                           "mpiesm-1.2.01p1/experiments"
                          )
                         )+"/"
outs_path=os.path.abspath("/scratch/x/x000000")+"/foobar/experiments/"
```

Now, in section II, we set some variables that will describe which experiments we will process, which streams of the output, and which substreams of files. The assumption here is that the experiments directory has the following structure: `<experiment>/outdata/<stream>/<experiment>_<stream>_<substream>_<date>.<extension>`. `exps` is the list of experiment names. `strms` define the output stream list. An output stream corresponds to the output corresponding to each component of the model. The substreams, defined in `sstrms` dictionary, define subsets of the output. In this example, I have only sixteen experiments. I have four streams, but I commented all but `mpiom` stream. For the `substream` dictionary, I registered all the substreams of the `echam6`, `jsbach` and `mpiom` streams, but I only want to process the "`data_3d_mm`" substream and, thus, I commented all of the other `mpiom` substreams.

```
# II. Definition of experiments, streams and substreams

exps=["djc_cntrl_stnd","djc_cnt4x.0.1_stnd"] # Experiment names
for i in range(7):
 exps.append("djc_cld4x.{0}.1_stnd".format(i))
 exps.append("djc_cldfc.{0}_stnd".format(i))

strms=[#"echam6",
       #"jsbach",
       #"hamocc",
       "mpiom"
      ] # Streams of output

sstrms={
 "echam6" : ["ATM_mm",
             "BOT_mm",
             "LOG_mm",
             "echam",
             "accw",
             "co2",
             "co2_mm"
            ],
 "jsbach" : ["jsbach_mm",
             "land_mm",
```

```
             "nitro_mm",
             "surf_mm",
             "veg_mm",
             "yasso_mm"
            ],
 "mpiom"   : [#"data_2d_3h",
             #"data_2d_dmax",
             #"data_2d_dm",
             #"data_2d_mm",
             #"data_2d_mmax",
             #"data_2d_mmin",
             #"data_2d_ym",
             "data_3d_mm"#,
             #"data_3d_ym",
             #"data_moc_mm",
             #"monitoring_ym",
             #"timeser_mm",
             #"fx",
             #"map"
            ]
} # Dictionary with substreams for each output

# III. Strings with evaluable fields used in the helper function

path="{0}{1}/{2}"
file="{0}_{1}.tar.gz"
```

In section III, we define two strings with evaluable fields that we will use in the helper function, which is section IV of the code. The helper function is the core of the script. It is the function that will process the files. That is why I called it `tar_and_zip_output`. It has five inputs, the path to the experiments, the path to the compressed output, the name of the experiment, and the name of the stream-substream pair to be processed. The first step is to construct the string that contains the path to the `outdata` directory where we store the data, as well as the `outdata` directory where the compressed files will be stored. Then, we make the latter directory if it does not exist yet. The third step is to add the file name to the path of the compressed output with the following structure: `<output_path>/<experiment>/outdata/<stream>_<substream>.tar.gz`. The fourth step begins with changing to the experiment output directory, apply a `find` operation and some filters to obtain the list of files to compress. After that we apply a `tar` to them and pipe the result to a `pigz -9` command to compress with the highest compression rate. The function returns, after a successful compression, the path to the output.

```
# IV. Helper function

def tar_and_zip_output(experiments_path,
                       output_path,
                       experiment,
                       stream,
                       substream
                      ):

 # 1 Path construction
```

```
 path_to_files=path.format(experiments_path,experiment,"outdata")
 path_to_output=path.format(output_path,experiment,"outdata")

 # 2 Construct output directory if missing

 command=["mkdir","-p",path_to_output]
 sp.run(command,check=True)

 # 3 Add the file name to the output path

 path_to_output+="/"
 path_to_output+=file.format(stream,substream)

 # 4 tar and zip all files of the stream-substream pair

 os.chdir(path_to_files)
 with open(path_to_output,"w") as out:
  pattern=experiment+"_"+stream+"_"+substream+"_*" # Pattern to find
  c0=["find",stream,"-type","f","-name",pattern] # Find command
  c1=["tar","cvf","-"] # Tar command
  c2=["pigz","-9"] # Parallel gzip command with the maximum compression
level
  r0=sp.run(c0,check=True,capture_output=True,text=True).stdout # Find
  r0=r0.split("\n")[:-1] # Convert output to list and cut the last element
  r0.sort() # Order the list
  r0=r0[-500:] # Cut the list to only up to 500 files
  [ c1.append(i) for i in r0 ] # Append the file list to the tar command
  r1=sp.Popen(c1,stdout=sp.PIPE) # Execute tar and pipe the output...
  r2=sp.Popen(c2,stdin=r1.stdout,stdout=out) # ...to the pigz command
  r1.stdout.close() # When pigz ends, close stdout of tar
  r3=r2.communicate()[0] # Check whether pigz have ended or not

 return path_to_output
```

How do I want to store it using MISTRAL power?

The next step depends heavily on i) the MISTRAL resources that your group have and ii) How fast do you want to have the result. If you answer these two questions, then you know how many resources you will ask for and how you will set section V of the following code. In the example that I provide, I process only one substream of `mpiom` for each of the sixteen experiments. Then, I can afford to reserve sixteen nodes of MISTRAL (later I show you how) and process the sixteen substreams in parallel. For doing that, the code first defines two dictionaries, one of the tasks and one of the results: `list_tasks` and `list_results`. Then I cycle throughout the experiments, create the subdictionaries `list_tasks[exp]` and `list_results[exp]`. Then I cycle on the streams and create a list for each stream. The list submits the helper function with the corresponding arguments for the experiment, stream and substream. With this for-loop, we will send concurrently sixteen tasks to the cluster. There is another for-loop that waits for the sixteen tasks to be completed. If you want another order, you should plan where to wait for the results.

```
# V. Parallel compilation of files

# In this case, the settings of the workers are for sixteen nodes. One node
for
# each experiment. There is only a single substream for each stream and only
a
# stream for each experiment. Then we parallelise at the level of
experiments.
# For that reason, we gather the results in a separate for-loop.

list_tasks={} # Dictionary of tasks
list_results={} # Dictionary of results
for exp in exps: # Cycle throughout experiments...
 print("Concatenating experiment {}...".format(exp))
 list_tasks[exp]={} # Initialize the dictionary of the exp-stream pair
 list_results[exp]={}
 for strm in strms: # Cycle throughout the streams...
  print("     Concatenating stream {}...".format(strm))
  list_tasks[exp][strm]=[ client.submit(tar_and_zip_output,
                                        exps_path,
                                        outs_path,
                                        exp,
                                        strm,
                                        sstrm,
                                        pure=False
                                       )
                          for sstrm in sstrms[strm]
                        ]
for exp in exps:
 for strm in strms:
  list_results[exp][strm]=[ i.result() for i in list_tasks[exp][strm] ]
```

The last two sections of the code (VI and VII) are only for closing the connection to the MISTRAL resources and cancelling the processes.

```
# VI. Relinquish control of cluster reserved resources

client.close()
print(client)

# VII. Cancel the processes in the cluster

command=["scancel","-u","x000000","--name","dask-workers-Storage3d"]
sp.run(command,check=True)
command=["scancel","-u","x000000","--name","dask-scheduler-Storage3d"]
sp.run(command,check=True)
```

The complete script

Storage3d.py

```
import os
import subprocess as sp
from dask.distributed import Client

# 0. Acquire the reserved resources in the cluster

client=Client(scheduler_file=('/scratch/x/x000000/dask-work/Storage3d'+
                              '/scheduler_Storage3d.json'
                             )
              )
print(client)

# I. Definition of paths to the data and the scratch space

exps_path=os.path.abspath(("/work/yy0000/x000000/"+
                           "mpiesm-1.2.01p1/experiments"
                          )
                         )+"/"
outs_path=os.path.abspath("/scratch/x/x000000")+"/foobar/experiments/"

# II. Definition of experiments, streams and substreams

exps=["djc_cntrl_stnd","djc_cnt4x.0.1_stnd"] # Experiment names
for i in range(7):
 exps.append("djc_cld4x.{0}.1_stnd".format(i))
 exps.append("djc_cldfc.{0}_stnd".format(i))

strms=[#"echam6",
       #"jsbach",
       #"hamocc",
       "mpiom"
      ] # Streams of output

sstrms={
 "echam6" : ["ATM_mm",
             "BOT_mm",
             "LOG_mm",
             "echam",
             "accw",
             "co2",
             "co2_mm"
            ],
 "jsbach" : ["jsbach_mm",
             "land_mm",
             "nitro_mm",
             "surf_mm",
             "veg_mm",
             "yasso_mm"
            ],
```

```
 "mpiom"  : [#"data_2d_3h",
             #"data_2d_dmax",
             #"data_2d_dm",
             #"data_2d_mm",
             #"data_2d_mmax",
             #"data_2d_mmin",
             #"data_2d_ym",
             "data_3d_mm"#,
             #"data_3d_ym",
             #"data_moc_mm",
             #"monitoring_ym",
             #"timeser_mm",
             #"fx",
             #"map"
            ]
} # Dictionary with substreams for each output

# III. Strings with evaluable fields used in the helper function

path="{0}{1}/{2}"
file="{0}_{1}.tar.gz"

# IV. Helper function

def tar_and_zip_output(experiments_path,
                       output_path,
                       experiment,
                       stream,
                       substream
                      ):

 # 1 Path construction

 path_to_files=path.format(experiments_path,experiment,"outdata")
 path_to_output=path.format(output_path,experiment,"outdata")

 # 2 Construct output directory if missing

 command=["mkdir","-p",path_to_output]
 sp.run(command,check=True)

 # 3 Add the file name to the output path

 path_to_output+="/"
 path_to_output+=file.format(stream,substream)

 # 4 tar and zip all files of the stream-substream pair

 os.chdir(path_to_files)
 with open(path_to_output,"w") as out:
  pattern=experiment+"_"+stream+"_"+substream+"_*" # Pattern to find
```

```
  c0=["find",stream,"-type","f","-name",pattern] # Find command
  c1=["tar","cvf","-"] # Tar command
  c2=["pigz","-9"] # Parallel gzip command with the maximum compression
level
  r0=sp.run(c0,check=True,capture_output=True,text=True).stdout # Find
  r0=r0.split("\n")[:-1] # Convert output to list and cut the last
element
  r0.sort() # Order the list
  r0=r0[-500:] # Cut the list to only up to 500 files
  [ c1.append(i) for i in r0 ] # Append the file list to the tar
command
  r1=sp.Popen(c1,stdout=sp.PIPE) # Execute tar and pipe the output...
  r2=sp.Popen(c2,stdin=r1.stdout,stdout=out) # ...to the pigz command
  r1.stdout.close() # When pigz ends, close stdout of tar
  r3=r2.communicate()[0] # Check whether pigz have ended or not

 return path_to_output

# V. Parallel compilation of files

# In this case, the settings of the workers are for sixteen nodes. One
node for
# each experiment. There is only a single substream for each stream and
only a
# stream for each experiment. Then we parallelise at the level of
experiments.
# For that reason, we gather the results in a separate for-loop.

list_tasks={} # Dictionary of tasks
list_results={} # Dictionary of results
for exp in exps: # Cycle throughout experiments...
 print("Concatenating experiment {}...".format(exp))
 list_tasks[exp]={} # Initialize the dictionary of the exp-stream pair
 list_results[exp]={}
 for strm in strms: # Cycle throughout the streams...
  print("     Concatenating stream {}...".format(strm))
  list_tasks[exp][strm]=[ client.submit(tar_and_zip_output,
                                        exps_path,
                                        outs_path,
                                        exp,
                                        strm,
                                        sstrm,
                                        pure=False
                                       )
                          for sstrm in sstrms[strm]
                        ]
for exp in exps:
 for strm in strms:
  list_results[exp][strm]=[ i.result() for i in list_tasks[exp][strm] ]

# VI. Relinquish control of cluster reserved resources
```

```
client.close()
print(client)

# VII. Cancel the processes in the cluster

command=["scancel","-u","x000000","--name","dask-workers-Storage3d"]
sp.run(command,check=True)
command=["scancel","-u","x000000","--name","dask-scheduler-Storage3d"]
sp.run(command,check=True)
```

Slurm scripts

To reserve the resources, you need to follow the steps described in the Reserve and Use method. In this case, I reserved one node in the `shared` partition for the scheduler. I also reserve sixteen nodes in the `compute2` partition. In both cases, I only allow one task to run in each node. The scheduler node manages the sixteen worker nodes and sends your requests to them. Then, when I submit the requests in section V of the Python code, I am sending the instructions to the workers. As I said, if you want more details, you should look at the Reserve and Use method. In the following, I provide the scripts to reserve the nodes.

I recommend that you have a specific dask-work directory (possibly in scratch) as a place to store all the log files of the `dask.distributed` operations. The directory should also have several subdirectories for each dask processing task. In case there is a problem, you will debug it better and faster.

launch_scheduler_Storage3d.sh

```
#!/bin/bash
#SBATCH --account=yy0000
#SBATCH --job-name=dask-scheduler-Storage3d
#SBATCH --workdir=/scratch/x/x000000/dask-work/Storage3d
#SBATCH --output=/scratch/x/x000000/dask-work/Storage3d/LOG_dask.%j.o
#SBATCH --error=/scratch/x/x000000/dask-work/Storage3d/LOG_dask.%j.o
#SBATCH --time=08:00:00
#SBATCH --partition=shared
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1

rm -rf worker-*
rm *.o
rm scheduler*.json
rm global.lock
rm purge.lock

module purge
module load pigz
export PATH=/work/yy0000/x000000/miniconda3/bin:$PATH
```

```
conda activate glamdring

srun dask-scheduler --scheduler-file /scratch/x/x000000/dask-
work/Storage3d/scheduler_Storage3d.json
```

launch_workers_Storage3d.sh

```
#!/bin/bash
#SBATCH --account=yy0000
#SBATCH --job-name=dask-workers-Storage3d
#SBATCH --workdir=/scratch/x/x000000/dask-work/Storage3d
#SBATCH --output=/scratch/x/x000000/dask-work/Storage3d/LOG_dask.%j.o
#SBATCH --error=/scratch/x/x000000/dask-work/Storage3d/LOG_dask.%j.o
#SBATCH --time=08:00:00
#SBATCH --partition=compute2
#SBATCH --nodes=16
#SBATCH --ntasks-per-node=1

module purge
module load pigz
export PATH=/work/yy0000/x000000/miniconda3/bin:$PATH
conda activate glamdring

srun dask-worker --scheduler-file /scratch/x/x000000/dask-
work/Storage3d/scheduler_Storage3d.json
```

How to run these scripts?

The first thing is that you would want to first use `screen` (or `tmux`) to create a shell session that you can detach without killing your long-lived processing. Inside your `screen` session, you submit to Slurm your request for the scheduler node.

```
sbatch launch_scheduler_Storage3d.sh
```

After some time it should be online, then you run the request for workers.

```
sbatch launch_workers_Storage3d.sh
```

Once the workers are online, they will connect to the scheduler. Then you can run your script

```
python Storage3d.py
```

Then you can wait and do other things. As an example, for the task that I described, it took one hour to compress 16 Tb of data (1 Tb for each experiment). Finally, I only remark that this script is only the solution when you have already answered the two fundamental questions!

Remark: In the Slurm scripts there are lines for putting in the search path a `miniconda`

distribution and initialising an environment called `glamdring`. If you are using the anaconda3 from the system, just delete those lines and use `module` command to load anaconda3. If you are using your own `miniconda` distribution, you should change those lines and load the proper environment.