Reserve and Use

This method uses Dask to make the distribution easier. There are three basic scripts here. Two bash scripts reserve computing resources in MISTRAL, and the main script contains the parallelised code. The mind behind this method is Pavan Siligam from DKRZ.

Scheduler

The first script initialises a process that manages the resources. You initialise the job in MISTRAL with a script similar to the next template.

launch_scheduler_test.sh

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

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

srun dask-scheduler --scheduler-file /scratch/x/x000000/dask-
work/test/scheduler_test.json
```

The line before the `srun` command adds the binaries of a base `miniconda` distribution to the front of the path and then loads a given `conda` environment (here named `glamdring`). If you are not using a custom `miniconda` distribution, you should use the necessary `module` invocations, or any other code, and set the proper environment. The srun command executes the `dask-scheduler` script to create the scheduler node and stores its location in the file given after the option `–scheduler-file`. To run the script, you only need to issue.

```
sbatch launch_scheduler_test.sh
```

NB: You should create manually in your `scratch` space the directory `dask-work`. This is where all the logs will be stored. I strongly recommend that inside `dask-work` you further create a directory for

each project, such as `test` directory. Having this structure will allow you to better debug any problems with your scripts.

Workers

The second script initialises several processes called workers. As the name suggests, they are the processes that execute the tasks effectively.

launch_workers_test.sh

```
#!/bin/bash
#SBATCH --account=yy0000
#SBATCH --job-name=dask-worker-test
#SBATCH --workdir=/scratch/x/x000000/dask-work/test
#SBATCH --output=/scratch/x/x000000/dask-work/test/LOG_dask.%j.o
#SBATCH --error=/scratch/x/x000000/dask-work/test/LOG_dask.%j.o
#SBATCH --time=00:05:00
#SBATCH --partition=compute2
#SBATCH --nodes=8
#SBATCH --ntasks-per-node=1

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

srun dask-worker --scheduler-file /scratch/x/x000000/dask-
work/test/scheduler_test.json
```

In analogy to the scheduler case, the lines before the `srun` command should be used to set the proper environment for the workers. Now, the `srun` command executes the `dask-worker` script to create the worker processes. The option `–scheduler-file` gives the path to the scheduler file created with the successful execution of the first script. The result is that all the worker processes connect to the scheduler node.

The importance of this script is not in the `srun` command but in the environment and the SBATCH preamble. How you set the environment arranges the available software and the Python version in each worker. The SBATCH preamble will determine the resources that you reserve and how do you assign them to the workers. In this case, I reserve eight nodes in `compute2` partition for five minutes (40 node-minutes), with a worker running in each node. SBATCH parameters allow for a more detailed request. Even you can make a script for each worker with different resources. Here is a more advanced topic. Someone else can comment on it and show a minimal working example. Once the `launch_scheduler_test.sh` is running in MISTRAL, you can request to set up the workers by issuing.

```
sbatch launch_workers_test.sh
```

When they are running in the cluster, you can use the resources in your Python script. Just read the

next sections.

Accessing the resources from within the Python script

Apart from other packages, you should import the `Client` class from the `dask.distributed` module and the `subprocess` module.

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

With the `Client` class you can initialise a connection to the scheduler and make available the reserved resources with the following lines. The `subprocess` module allows you to kill the processes in the cluster once we ended our calculations, in case you overestimated the reserved time. The exact code to connect to the resources is

```
client=Client(scheduler_file='/scratch/x/x000000/dask-
work/test/scheduler_test.json')
print(client)
```

The second line prints some information about the scheduler and the workers, as well as the total reserved memory. Now, your script has access to the resources. The next step is to run your desired code.

Sample Python script

I present a minimal working example to show what is the methodology to parallelise. The problem is the matrix multiplication of a matrix **A** with eight columns and eight rows and a column vector *v* with eight components. We want to compute **A***v*

For the sake of the example, we use a random matrix. I assume that `numpy` package has been imported.

```
A=np.random.rand(8,8)
v=np.array([1,1,2,3,5,8,13,21])
```

Matrix multiplication will result in another column vector of eight elements, that come from the “dot product” of each row of the matrix with the vector *v*. Then each “dot product” can be done at the same time, that is, in parallel. In other words, we can calculate each of the eight components of the result independently of any of the other components. The elemental calculation for the first component of the result will be

```
first_component=A[0]*v
first_component=first_component.sum()
```

Where I used the operator * which multiplies the vectors component-wise. If I would not have that operator, then the elemental process should be

```
first_component=0
```

```
for j in range(v.size):
 first_component+=A[0,j]*v[j]
```

For-loops are slow. Then the real elemental process is the multiplication of components. Then we define a helper function.

```
def first_multip(j):
 return A[0,j]*v[j]
```

or, in a general setting

```
def multip(mat_elem,vec_elem):
 return mat_elem*vec_elem
```

Then the elemental process will be: multiply every element of the matrix by the corresponding component of the vector and then sum over rows to get the eight components of the result.

```
def multip(mat_elem,vec_elem):
 return mat_elem*vec_elem

task_multiply_sum=[ client.submit(sum,[ client.submit(multip,A[i,j],v[j])
for j in range(A.shape[1]) ]) for i in range(A.shape[0]) ]

result=np.array([task_multiply_sum[i].result() for i in range(A.shape[0])])
```

Here the code has two lines. First, in the `task_multiply_sum` variable, I send the elemental calculations to the cluster with the help of the `submit` method. This method has as first argument the name of the function to be applied. The following arguments are the arguments for the function. Then I say to the scheduler that, when the results of a row are ready, it should sum all of them (submit method with the function `sum`). Everything is done in the cluster workers and not in the script namespace. The second line gathers the final results from the cluster by issuing the `result` method on each component of the `task_multiply_sum` list.

Once the computation concludes, I disconnect from the cluster and confirm this by printing the client information

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

and then, with the following code, you can kill the processes in the cluster

```
command=["scancel","-u","m300556","--name","dask-workers-test"]
sp.run(command,check=True) # First kill workers
command=["scancel","-u","m300556","--name","dask-scheduler-test"]
sp.run(command,check=True) # Then the scheduler
```

We can check the result of the multiplication with the following lines

```
alt_result=np.dot(A,v)
```

```
print(result==alt_result)
print(result)
print(alt_result)
```

the first line makes the multiplication directly, the second checks the results element-wise. In the last two lines, I print out both resulting vectors. *NB: In my tests, I found that the element-wise comparison shows that some vector components are different. When you inspect them visually in the printouts, the elements are equal in both vectors, but when you look at the full representations of the numbers, you can see that some digits are different at the end of the numbers (Perhaps it is a consequence of how numpy operates on floats).*

The advantage in speed of this MWE is not evident since the problem is not large enough. With larger matrix dimensions we should see the difference. But for other tasks, the performance improvement should be apparent (as in the postprocessing of experiment output).

The code for the MWE is

matrix_multiplication.py

```
import numpy as np
import subprocess as sp
from dask.distributed import Client

# 0. Preparation

client=Client(scheduler_file=('/scratch/m/m300556/dask-work/test'+
                              '/scheduler_test.json'))
# Connect to the reserved resources.
print(client)
# Print out information about the resources.

A=np.random.rand(8,8)
# Define the matrix and the vector.
v=np.array([1,1,2,3,5,8,13,21])

def multip(mat_elem,vec_elem):
# Helper function.
 return mat_elem*vec_elem

# 1. Parallel calculation

task_multiply_sum=[ client.submit(sum,[
client.submit(multip,A[i,j],v[j]) for j in range(A.shape[1]) ]) for i
in range(A.shape[0]) ]

result=np.array([task_multiply_sum[i].result() for i in
range(A.shape[0])])     # This gathers the results.

# 2. Disconnection from the cluster resources

client.close()
```

```
# Disconnect the scheduler.
print(client)
# Shows information to verify the disconnection.

command=["scancel","-u","m300556","--name","dask-workers-test"]
sp.run(command,check=True)
# First kill workers
command=["scancel","-u","m300556","--name","dask-scheduler-test"]
sp.run(command,check=True)
# Then the scheduler

# 3. Check the results against an established library

alt_result=np.dot(A,v)
# Use np.dot to perform the matrix multiplication.
print(result==alt_result)
# Print comparison.
print(result)
# Print out both vectors.
print(alt_result)
```

NB: The method also works for jupyter notebooks without any issue.