Setting up Dask processes

Tutorial for batch computing with SLURMCluster object and all 3 processes worker/scheduler /client launched with in the same SLURM file: https://ulhpc-tutorials.readthedocs.io/en/latest /python/advanced/dask-ml/

Introduction to Dask processes

In Dask, we use three types of processes:

- The Scheduler: The scheduler receives the computation graph and assigns tasks to the workers. The scheduler does not perform heavy computations and can run on a single core.
- The Workers: Workers are associated with heavy computation resources. They receive tasks from the scheduler and execute them. The number of workers can vary during the computation.
- The Client: The client process designs the computation graph using Python and sends it to the scheduler. It does not directly execute the graph but manages the communication between the user and the Dask cluster.

All communications and connections between the client, scheduler, and workers are done via TCP/IP.

Launch the scheduler through SLURM

```
#!/bin/sh -l
#SBATCH --job-name=tuto_dask_scheduler # Name of the job
#SBATCH --nodes=1                          # Number of nodes
#SBATCH --ntasks-per-node=1          # Number of tasks (processes) per node
#SBATCH --cpus-per-task=1              # Number of CPU cores per task
#SBATCH --time=2:00:00                   # Wall clock time limit (hh:mm:ss)
#SBATCH --output=scheduler.log     # Output file name

# Customize the path to the scheduler network configuration file. The file allows to info
DASK_SCHEDULER_FILE=~/scheduler.json

# Load your environement
source /home/users/ppochelu/program/miniconda/scripts/env.sh

dask-scheduler ---interface "ib0" --scheduler-file $DASK_SCHEDULER_FILE
```

To launch the scheduler, you can submit the SLURM job using the sbatch command:


```
sbatch scheduler.sh
```

You can check the status and output of the scheduler by inspecting the *scheduler.log* file.

Launch the workers through SLURM

```
#!/bin/sh -l
#SBATCH --job-name=tuto_dask_workers # Name of the job
#SBATCH --nodes=2                    # Number of nodes
#SBATCH --cpus-per-task=4            # Number of CPU cores per task
#SBATCH --time=2:00:00               # Wall clock time limit (hh:mm:ss)

DASK_SCHEDULER_FILE=/home/users/ppochelu/project/bigdata_tuto/dask/scheduler.json

source /home/users/ppochelu/program/miniconda/scripts/env.sh

srun --nodes=$SLURM_JOB_NUM_NODES --output worker-%j.log   dask-worker --interface "ib0"
```

To launch the workers, you can submit the SLURM job using the sbatch command:

```
sbatch more_workers.sh
```

The workers will automatically connect to the scheduler specified in the *scheduler.json* file. You can adjust the number of nodes dynamically and other parameters based on your requirements.

Launch the client

```
import sys
import numpy as np
import time
from dask.distributed import Client

# Connection to the Scheduler
client = Client(scheduler_file=sys.argv[1])

def square(x:float):
    time.sleep(1) # simulate longer runtime by sleeping 1 second
    return x ** 2.

start_time=time.time()

input_vector = np.arange(16,dtype=float)              # We random input data [0,1,2,3,...
distributed_result = client.map(square, input_vector) # Spread the float numbers on all w
result = client.gather(distributed_result)            # Gather floats and put them in the

print(f"Result: {result} Computation time: {time.time()-start_time}")
```

The code basically call square() on each number from 0 to 15 and wait 1 second each time. Sequential execution last 16 seconds, but parallel one may divide this time.

You can launch the DAG from access node. The code is send to the scheduler, and the scheduler will distribute on all workers. Finally, workers will answer back the squared floats.

```
(base) 130 [ppochelu@access1 dask]$ python client.py scheduler.json
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225]
Result: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225] Computation ti
```

The computation is done in 2 seconds showing that the worload have been well parallized on 8 cpus (2 nodes, 4 cpus per node).

Exercice for the reader - Dynamic scaling

Scaling up:

1. Launch the scheduler and wait for it to be ready.
2. Launch a worker pool with "sbatch more_workers.sh" (8 CPUs) and wait for it to be ready.
3. Launch the Python code with "python client.py scheduler.json" and record the computing time.
4. Launch "sbatch more_workers.sh" again with 8 new CPUs to add more workers.
5. Launch the Python code again with "python client.py scheduler.json" and compare the computing time with the previous run.

Scaling-down & resiliency:

1. Launch the scheduler.
2. Launch two worker pools by executing the "sbatch more_workers.sh" command twice.
3. Use the command "squeue -u $USER" to check if all processes are ready.
4. Launch the Python code for computation.
5. Before the computation ends, choose one worker pool and kill it using "scancel {{JOBID-FROM-SQUEUE}}" command.

Killing some workers to simulate scenarios like HPC/Cloud disconnections or node breakdowns does not cause the entire computation to crash but negatively impacts performance.