Dask on curnagl

In order to use Dask in Curnagl you have to use the following packages:

- dask
- dask-jobqueue

> Note: please make sure to use version 2022.11.0 or later. Previous versions have some bugs on worker-nodes that make them very slow when using several threads.

Dask makes easy to parallelize computations, you can run computational intensive methods on parallel by assigning those computations to different CPU resources.

For example:

```
def cpu_intensive_method(x, y , z):
    # CPU computations
    return x + 1



futures = []
for x,y,z in zip(list_x, list_y, list_z):
future = client.submit(cpu_intensive_method, x, y, z)
    futures.append(future)

result = client.gather(futures)
```

This documentation proposes two types of use:

- LocalCluster: this mode is very simple and can be used to easily parallelize computations by submitting just one job in the cluster. This is a good starting point
- SlurmCluster: this mode handle more parallelisim by distributing work on several machines. It can handle load and submit automatically new jobs for increasing paralellisim

Local cluster

Python script looks like:

```
import dask
from dask.distributed import Client, LocalCluster

def compute(x):
  ""CPU demanding code"



if __name__ == "__main__":

cluster = LocalCluster()
client = Client(address=cluster)
    parameters = [1, 2, 3, 4]
    for x in parameters:
      future = client.submit(inc, x)
      futures.append(future)

    result = client.gather(futures)
```

> Call to LocalCluster and Client should be put inside the block if __name__ == "__main__". For more information, you can check the following link:
>
> https://docs.dask.org/en/stable/scheduling.html

The method LocalCluster() will deploy N workers, each worker using T threads such that NxT is equal to the number of cores reserved by SLURM. Dask will balance the number of workers and the number of threads per worker, the goal is to take advantage of GIL free workloads such as Numpy and Pandas.

SLURM script:

```
#SBATCH --job-name dask_job
#SBATCH --ntasks 16
#SBATCH -N 1
#SBATCH --partition cpu
#SBATCH --cpus-per-task 1
#SBATCH --time 01:00:00
#SBATCH --output=dask_job-%j.out
#SBATCH --error=dask_job%j.error



python script.py
```

Make sure to include the parameter `-N 1` otherwise SLURM will allocate tasks on different nodes and it will make Dask local cluster fail. You should adapt the parameter `--ntasks`, as we are using just one machine we can choose between 1 and 48. Just have in mind that the smallest the number the faster your job will start. You can choose to run with less processes but for a longer time.

Slurm cluster

The python script can be launched directly from the frontend but you need to keep you session open with tools such as `tmux` or `screen` otherwise your jobs will be cancelled.

In your Python script you should put something like:

```
import dask
from dask.distributed import Client
from dask_jobqueue import SLURMCluster

def compute(x):
  ""CPU demanding code"



if __name__ == "__main__":

cluster = SLURMCluster(cores=8, memory="40GB")
    client = Client(cluster)

    cluster.adapt(maximum_jobs=5, interval="10000 ms")
    for x in parameters:
      future = client.submit(inc, x)
      futures.append(future)

    result = client.gather(futures)
```

In this case DASK will launch jobs with 8 cores and 40GB of memory. The parameters `memory` and `cores` are mandatory. There are two methods to launch jobs: adapt and scale. `adapt` will launch/kill jobs by taking into account the load of your computation and how many computations in parallel you can run. You can put a limit on the number of jobs that will be launched. The parameter `interval` is necessary and needs to be set to `10000 ms` to avoid killing jobs too early.

`scale` will create a static infrastructure composed of a fix number of jobs, specified with the parameters jobs. Example

`scale(jobs=10)`

This will launch 10 jobs independent from the load and the amount of computation you generate.

Some facts about Slurm jobs and DASK

You need to have in mind that the computation will depend on the availability of resources, if jobs are not running your computation will not start. So if you think that your computation is stuck, please verify first that jobs have been submitted and that they are running using the command: `squeue -u $USER`.

By default the walltime is set to 30 min, you can use the parameter: **`walltime`** if you think that each individual computation will last more than the default time.

Slurm files will be generated under the same directory where you launch your python command.

Jobs will killed by Dask when there is no more computation to be done. If you see the message:

`slurmstepd: error: *** JOB 25260254 ON dna051 CANCELLED AT 2023-03-01T11:00:19 ***`

It is completely normal and it does not mean that there was an error in your computation.

Optimal number of workers

Both LocalCluster or SLURMCluster, will automatically balance the number of workers and the number of threads per worker. You can choose the number of workers using the parameter `n_workers`. If most of the computation relies on Numpy or Pandas, it is preferable to have only one worker `n_workers=1`. If most of the computation is pure Python code you should use as much workers as possible. Example:

Local cluster:

`LocalCluster(n_workers=int(os.environ['SLURM_NTASKS']))`

Slurm cluster:

`SLURMCluster(cores=8, memory="40GB", n_workers=8)`

Example

Here, it is an example code which illustrates the use of Dask. The code runs 40 multiplications of random matrices of size NXN, each computation returns the sum of all the elements of the result matrix:

```
import os
import time
import numpy as np
```

```
from dask.distributed import Client, LocalCluster
from dask_jobqueue import SLURMCluster

SIZE = 9192

def compute(tag):
    np.random.seed(tag)
    A = np.random.random((SIZE,SIZE))
    B = np.random.random((SIZE,SIZE))
    start = time.time()
    C = np.dot(A,B)
    end = time.time()
    elapsed = end-
start

    return elapsed, np.sum(C)

if __name__ == "__main__":

#    cluster =
LocalCluster(n_workers=int(os.environ['SLURM_NTASKS']))

    cluster = SLURMCluster(memory="40GB",
n_workers=8)
    client = Client(cluster)

    cluster.adapt(maximum_jobs=5, interval="10000
ms")
    N_ITER = 40

    futures = []
    for i in range(N_ITER):
        future = client.submit(compute, i)
        futures.append(future)

    results =
client.gather(futures)

    print(results)
```

Révision #15
Créé 2 mars 2023 15:24:19 par Cristian Ruiz
Mis à jour 6 juin 2023 10:08:03 par Emmanuel Jeanvoine