How to run a job on Curnagl

Overview

Suppose that you have finished writing your code, say a python code called <my_code.py>, and you want to run it on the cluster Curnagl. You will need to submit a job (a bash script) with information such as the number of CPUs you want to use and the amount of RAM memory you will need. This information will be processed by the job scheduler (a software installed on the cluster) and your code will be executed. The job scheduler used on Curnagl is called SLURM (Simple Linux Utility for Resource Management). It is a free open-source software used by many of the world's computer clusters.

The partitions

The clusters contain several partitions (sets of compute nodes dedicated to different means). To list them, type

```
sinfo
```

As you can see, there are three partitions:

- cpu - this is the main partition and includes the majority of the compute nodes
- gpu - this partition contains the GPUs equipped nodes
- interactive - this partition allows rapid access to resources but comes with a number of restrictions

Each partition is associated with a submission queue. A queue is essentially a waiting line for your compute job to be matched with an available compute resource. Those resources become available once a compute job from a previous user is completed.

Note that the nodes may be in different states: idle=not used, alloc=used, down=switch off, etc. Depending on what you want to do, you should choose the appropriate partition/submission queue.

The sbatch script

To execute your python code on the cluster, you need to make a bash script, say `my_script.sh`, specifying the information needed to run your python code (you may want to use nano, vim or emacs as an editor on the cluster). Here is an example:

```
#!/bin/bash -l

#SBATCH --job-name my_code
#SBATCH --output my_code.out
#SBATCH --partition cpu
#SBATCH --cpus-per-task 8
#SBATCH --mem 10G
#SBATCH --time 00:30:00

module load python

python3 /PATH_TO_YOUR_CODE/my_code.py
```

Here we have used the command `module load python` before `python3 /PATH_TO_YOUR_CODE/my_code.py` to load some libraries and to make several programs available.

To display the list of available modules or to search for a package:

```
module avail
module spider package_name
```

For example, to load bowtie2:

```
module load bowtie2/2.4.2
```

To display information of the sbatch command, including the SLURM options:

```
sbatch --help
sbatch --usage
```

Finally, you submit the bash script as follows:

```
sbatch my_script.sh
```

***Important:** We recommend to store the above bash script and your python code in your home folder, and to store your main input data in your work space. The data may be read from your python code.*
To show the state (R=running or PD=pending) of your jobs, type:

```
Squeue
```

If you realize that you made a mistake in your code or in the SLURM options, you may cancel it:

```
scancel JOBID
```

An interactive session

Often it is convenient to work interactively on the cluster before submitting a job. I remind you that when you connect to the cluster you are actually located at the front-end machine and your must NOT run any code there. Instead you should connect to a node by using the `Sinteractive` command as shown below.

```
[ulambda@login ~]$ Sinteractive -c 1 -m 8G -t 01:00:00

interactive is running with the following options:

-c 1 --mem 8G -J interactive -p interactive -t 01:00:00 --x11

salloc: Granted job allocation 172565
salloc: Waiting for resource configuration
salloc: Nodes dna020 are ready for job
[ulambda@dna020 ~]$  hostname
dna020.curnagl
```

You can then run your code.

Hint: *If you are having problems with a job script then copy and paste the lines one at a time from the script into an interactive session - errors are much more obvious this way.*

You can see the available options by passing the `-h` option.

```
[ulambda@login1 ~]$ Sinteractive -h
Usage: Sinteractive [-t] [-m] [-A] [-c] [-J]

Optional arguments:
   -t: time required in hours:minutes:seconds (default: 1:00:00)
   -m: amount of memory required (default: 8G)
   -A: Account under which this job should be run
   -R: Reservation to be used
   -c: number of CPU cores to request (default: 1)
   -J: job name (default: interactive)
   -G: Number of GPUs (default: 0)
```

To logout from the node, simply type:

```
exit
```

Embarrassingly parallel jobs

Suppose you have 14 image files in `path_to_images` and you want to process them in parallel by using your python code `my_code.py`. This is an example of embarrassingly parallel programming where you run 14 independent jobs in parallel, each with a different image file. One way to do it is to use a job array:

```
#!/bin/bash -l

#SBATCH --job-name my_code
#SBATCH --output=my_code_%A_%a.out
#SBATCH --partition cpu
#SBATCH --cpus-per-task 8
#SBATCH --mem 10G
#SBATCH --time 00:30:00

#SBATCH --array=0-13

module load python/3.9.13

FILES=(/path_to_configurations/*)

python /PATH_TO_YOUR_CODE/my_code.py ${FILES[$SLURM_ARRAY_TASK_ID]}
```

The above allocations (for example time=30 minutes) is applied to each individual job in your array.

Similarly, if your script takes integer parameters to control a simulation. You can do something like:

```
#!/bin/bash -l

#SBATCH --account project_id
#SBATCH --mail-type ALL
#SBATCH --mail-user firstname.surname@unil.ch
#SBATCH --job-name my_code
#SBATCH --output=my_code_%A_%a.out
#SBATCH --partition cpu
#SBATCH --cpus-per-task 8
```

```
#SBATCH --mem 10G
#SBATCH --time 00:30:00

#SBATCH --array=0-13

module load python/3.9.13

ARGS=(0.1 2.2 3.5 14 51 64 79.5 80 99 104 118 125 130 100)

python /PATH_TO_YOUR_CODE/my_code.py ${ARGS[$SLURM_ARRAY_TASK_ID]}
```

Another way to run embarrassingly parallel jobs is by using one-line SLURM commands. For example, this may be useful if you want to run your python code on all the files with bam extension in a folder:

```
for file in `ls *.bam`
do
  sbatch --job-name my_code --output my_code-%j.out --partition cpu
  --ntasks 1 --cpus-per-task 8 --mem 10G --time 00:30:00
  --wrap "module load gcc/9.3.0 python/3.8.8; python /PATH_TO_YOUR_CODE/my_code.py $file" &
done
```

MPI jobs

Suppose you are using MPI codes locally and you want to launch them on Curnagl.

The below example is a slurm script running an MPI code mpicode (which can be either of C, python, or fortran type...) on one single node (i.e. --nodes 1) using NTASKS cores without using multi-threading (i.e. --cpus-per-task 1). In this example, the memory required is 32Gb in total. To run an MPI code, the loading modules is mvapich2 only. You must add needed modules (depending on your code).

Instead of mpirun command, you must use srun command, which is the equivalent command to run MPI codes on a cluster. To know more about srun, go through srun --help documentation.

```
#!/bin/bash -l

#SBATCH --account project_id
#SBATCH --mail-type ALL
#SBATCH --mail-user firstname.surname@unil.ch
```

```
#SBATCH --chdir /scratch/<your_username>/
#SBATCH --job-name testmpi
#SBATCH --output testmpi.out

#SBATCH --partition cpu
#SBATCH --nodes 1
#SBATCH --ntasks NTASKS
#SBATCH --cpus-per-task 1
#SBATCH --mem 32G
#SBATCH --time 01:00:00

module purge
module load mvapich2/2.3.7

srun mpicode
```

For a complete MPI overview on Curnagl, please refer to compiling and running MPI codes

Good practice

- Put your file and data in the scratch and work folders only during the analyses that you are currently doing
- Do not keep important results in the scratch, but move them in the NAS data storage
- Clean your scratch folder after your jobs are finished, especially the large files
- Regularly clean your scratch folder for any unnecessary files

---

Révision #35
Créé 13 janvier 2020 11:37:28 par Philippe Jacquet
Mis à jour 6 mars 2025 15:56:21 par Cristian Ruiz