Using the Cirrus computing facility

P. Fowler-Wright

2023-03-19

Contents

1 Preliminaries

You will need to create an account on the SAFE website and ask Brendon to add you to project **d422 (Nanomaterials)** which runs until May 2023. Once added, this project will be listed under the `Projects` tab of the website and a user account created for you on Cirrus. An initial password for this account will be emailed to you. Your username is listed as `user@cirrus` under the `Login accounts` tab.

Further information (registration tips) During registration, you will be asked to select an institution from a bizarrely sorted[1] list (University of St Andrews is near the bottom) and optionally provide a default SSH public key used for new login accounts such as the one created

[1]The arrangement is something like: English universities sorted alphabetically by placename (i.e. ignoring 'University', 'The', 'of'), then Scottish, then Welsh, then Irish and finally other UK and non-UK research and commercial institutions.

for Cirrus, although this can easily be done after account creation (see below). I recommend staying subscribed to user emails which include important service changes and notifications; you can filter these optional mailings from `Your details` → `Update email settings`. Note that the login details for the SAFE website are entirely separate from those used to access the Cirrus facility itself.

2 Adding SSH credentials

Cirrus uses a combination of key–based and password authentication. In the following `user` refers to your username displayed as `user@cirrus` in the `Login accounts` tab of the SAFE website.

If you did not add a default SSH key when creating an account, or wish to add another e.g. for a second computer, navigate to `Login accounts` on SAFE and under `Account credentials` select `Add Credential`, `Credential Type: SSH public key` and upload the file containing your public key or paste its contents directly into the input box before pressing `Add`.

Further information (generating SSH keys) For those who have not used SSH before, the OpenSSH `ssh-keygen` tool is the standard way to create SSH private/public key pairs on Mac, Linux and Windows. Note the location to which the public key (`.pub`) is installed. The tool will prompt you for a password when creating the key (this is separate from the SAFE website and cirrus login passwords)—enter nothing to skip this addition layer of password protection.

3 First–time login

Having adding your public key to your login account you can connect to Cirrus using

```
$ ssh user@login.cirrus.ac.uk
```

If the SSH authentication is successful (`ssh -v` to debug) you will be prompted for the initial login password sent to you when creating an account (see above), and then to enter a new password for future logins. Thereafter this password may be changed whilst logged on to Cirrus using the `passwd` command. If you forget your password, a reset may be requested from the `Login accounts` page on safe. The commands `logout`, `exit` or `Ctrl+d` may be used to close the connection to Cirrus at any time.

In addition to interactive access to cirrus the `scp` command (or equivalent file transfer utility e.g. `rsync`) can be used to transfer data to and from Cirrus. For example, the command

```
$ scp -r test_scripts user@login.cirrus.ac.uk:remote_test_scripts
```

copies a local directory `test_scripts` to a directory called `remote_test_scripts` on in your user's home directory on Cirrus (more on the Cirrus filesystem structure below), and

```
$ scp -r user@login.cirrus.ac.uk:remote_test_scripts test_scripts
```

does the reverse. Note both of the above commands should be run in a local session; you can't `scp` from a Cirrus login node to your own machine unless you setup a `ssh` server on your machine.

Further information (SSH configuration) If you installed your private–public SSH key in a non–standard location, or have multiple keys installed on your system, it may be necessary to specify the path to the relevant private key using the -i option of `ssh` e.g. `ssh -i ~/my_keys/cirrus_rsa user@login.cirrus.ac.uk`. To streamline future connections you can add an entry to your local `ssh` configuration file (`~/.ssh/config`) such as

```
-----
Host cirrus
    HostName login.cirrus.ac.uk
    User user
    IdentityFile ~/my_keys/cirrus_rsa
-----
```

The command `ssh cirrus` (or `scp...cirrus:...`) can then be used to login (or copy) to Cirrus as `user`. A second useful snippet is

```
-----
Host *
    ControlMaster auto
    ControlPath ~/.ssh/sockets/%h-%p-%r
-----
```

This allows `ssh` connections to be reused, in particular saving time and the need to enter a password when using `scp` if you already have an `ssh` session to Cirrus running.

4 Python installation and environment management

4.1 Filesystem structure on Cirrus

Since March 2022 there are now two relevant locations on Cirrus you have write access to as `user`:

- Your user home directory, `/home/d422/d422/user`, for personal files and configuration. Since the March update the compute nodes do not have access to this directory, so you won't be using this much.
- Your user work directory, `/work/d422/d422/user`. The compute nodes have access to this directory and so this is the relevant location for your Python installation as well as scripts, input and output data.

4.2 Miniconda installation and activation

While users have access to the system Python and Anaconda distributions (run `module load anaconda/python3`), for the purpose of environment management and running jobs on the compute nodes—possibly requiring packages not available with anaconda—it is convenient to install Miniconda and create a custom environment.

Firstly, after logging in to Cirrus navigate to `/work/d422/d422/user` and download Miniconda3 using `wget`:

```
$ wget https://repo.continuum.io/miniconda/Miniconda3-latest-Linux-x86_64.sh
```

Run the installer with

```
$ bash Miniconda3-latest-Linux-x86_64.sh
```

Accept the license terms and enter `/work/d422/d422/user/miniconda3` as the install location. At the end be asked if you wish the installer to 'initialize Miniconda3.' This appends lines to your `~/.bashrc` to the effect of adding `miniconda3/bin/` to your path and loading the `base`

(default) environment on login. I have seen this fail to work properly in the past, so would suggest answering `no` unless you are particularly keen to have this 'convenience': there is little reason to use Python as a user on Cirrus other than to manage environments.

Miniconda environments will need to be loaded by the compute nodes when running jobs. To provide for this, create a script in your work directory called `miniconda-init.sh` (for example) containing the single line:

```
eval "$(/work/d422/d422/user/miniconda3/bin/conda shell.bash hook)"
```

where `user` must of course be replaced by your actual username. This script will be sourced at the start of any job we run on Cirrus and must be given executable permissions:

```
$ chmod +x miniconda-init.sh
```

4.3 Environment setup

To create and initialise a Python environment called `oqupy` (`#` indicates a comment):

```
# load base environment (does nothing if already loaded)
$ source miniconda3/bin/activate
# create a new envionrment using conda
$ conda create --name oqupy
# load the new environment to begin using it ('conda deactivate' to unload)
$ conda activate oqupy
```

Once activated, you can install packages into that environment. The simplest[2] way to do this is to install the `pip` package manager using `conda install pip` after which packages can be installed using `pip install numpy` (for example) or from a list in a file with `pip install -r requirements.txt`.

Further information (using Python) The Using Python page on the Cirrus documentation contains further details of the Cirrus Anaconda and Miniconda3 environments and using custom Python environments.

4.4 Using git

`git` is provided on cirrus and it is likely you will want to clone a repository with relevant code or data (`scp` is more appropriate for large files). For example, to clone the OQuPy repository using HTTPS,

```
$ git clone https://github.com/tempoCollaboration/OQuPy.git
```

or, using SSH,

```
$ git clone git@github.com:tempoCollaboration/OQuPy.git
```

Using SSH has the convenience of passwordless access to private repositories, but requires an SSH key–pair on your Cirrus home (or work) directory (use `ssh-keygen` as above) with the public added to your Github account or relevant host.

[2] `conda` can also be used to install packages, but it does not have access to the complete Python Package Index (PyPI) and has less simple syntax for e.g. installing packages from a file. If using both, it is recommended to install `conda` pacakges first before installing any packages that `conda` cannot provide with `pip`.

5 Running jobs on Cirrus

5.1 Overview

Suppose you wish to run a Python script `example.py` in your `work` directory on the Cirrus compute nodes. One way to go about this is:

1. Create a Bash script `run-example.sh` to load Miniconda, activate the relevant Python environment and invoke `example.py`
2. Create a submission script `submit-example.slurm` that defines a *job* for `run-example.sh`
3. Submit this job to the scheduler (Slurm) using `sbatch submit-example.slurm`

Examples for `run-example.sh` and `submit-example.slurm` are attached. Both scripts must be executable by `user` (use `chmod +x` as before). A few comments:

- The trickiest part is getting paths linking the three scripts (`.py`, `.sh`, `.slurm`) correct, something that will likely require trial an error (this is fine—a wrong path will cause the job to fail immediately and no resource will be wasted). For example, I normally place all submission scripts in one directory (e.g. `/work/d422/d422/user/submission`) and bash scripts in another, while the Python script will likely be in a subdirectory of a git repository also in my work directory such as `.../OQuPy/examples/...`.
- In addition to the `job-name` you will want to edit the `time` and `cpus-per-task` options in the submission script. `time` is the maximum length of time (`HH:MM:SS`) the job can run before being killed (`96` hours is the highest possible value for standard jobs), while `cpus-per-task` is the number of cores assigned to the job—in most cases you'll want to set this to `1`; see cpus-per-task below for further information
- Provided you place `"$@"` after the script names in the submission and Bash script (see example files), any additional arguments given to `sbatch` will be passed directly onto the Python script, which is useful in conjunction with the previous point e.g. `sbatch submit-example.slurm -T 100`, `sbatch submit-example.slurm -T 200` may run the simulation using two different temperatures.
- Resource usage of a job is measured in Core hours (`Corehrs`). That's `number of cores assigned` × `runtime in hours`. The **d422** project is allocated a set number of `Corehrs` per renewal period; you can ask Brendon for the details but this is of the order of 10^5 `Corehrs` so, as rough guidelines,
  - jobs totalling 10s of `Corehrs` or fewer—run liberally
  - jobs totalling 100s of `Corehrs` or fewer—think carefully before running, at your discretion
  - jobs totalling 1000s of `Corehrs` or more—be very sure before running these, consult Brendon and/or Jonathan

Once a job has been submitted and begins to run, any output that you would normally see in your terminal (i.e. `stdout`) is appended onto a newly created file `slurm-jobID.out` placed in the directory you issued the `sbatch` command from. This is useful to monitor the progress of jobs and debug errors. The job will also appear in the `squeue` listing; use `squeue -u user` to see your active jobs only. Finally, a job can be cancelled with `scancel jobID` where `jobID` is that shown in the `squeue` listing and used in the output filename.

Further information (login nodes) Note the login nodes themselves have slow I/O and are not suitable for running anything more than basic python scripts and tests!

5.2 `cpus-per-task`

Since vanilla Python (i.e. the standard library) does not multithread by default, there is little or no advantage to running scripts on multiple cores and you will typically want to set `cpus-per-task=1` to avoid using unnecessary `Corehrs`. There are two main exceptions to this:

1. You explicitly code for parallelism in Python using the `multiprocessing` module, in which case `cpus-per-task` can be set as high as desired, but note that jobs with low cpu counts are naturally allocated to compute nodes more quickly; during busy hours you will likely find yourself waiting a long time for a job with high CPU counts (e.g. `>8`).
2. You need to allocate more than ∼7 GB of memory to your job (see below)

Instead of parallelism within one script then the power of having access to the many processor cores provided by Cirrus is the ability to run many instances of a script at the same time e.g. a simulation with different parameters. This can be done manually by issuing multiple calls to `sbatch` or using another Bash script to execute this command in a for–loop.

Further information (job memory) On Cirrus memory aka RAM is allocated according to the number of CPUs used for a task, with approximately `7` GB of memory per CPU. If you need more memory, then you can increase the count `cpus-per-task` in your submission script, but be aware that this also increases the rate at which your job uses `Corehrs` (e.g. a two hour job with `cpus-per-task=8` costs `16 Corehrs`). Hence it is preferable to design your code such that large amounts of memory are not required e.g. by writing intermediary calculations to disk.

A script which can be used to monitor memory usage of a job is attached to this document.

Further information (numpy multithreading) A caveat of the above is that numpy can leverage multiple threads for more performant linear algebra using the OpenBLAS or MKL backends (whether this is worth the addition `Corehrs`: I doubt it). The number of threads can be controlled by setting the `OMP_NUM_THREADS` variable (e.g. `export OMP_NUM_THREADS=1` before a python script is run prevents numpy multi-threading entirely), but I recommend *you do not set this variable* as Cirrus appears to set it appropriately.[3] The exception I would guess (expert advice needed) is when using `multiprocessing`—there you want to restrict multithreading so each process is not assigned more threads then the corresponding CPU can support.[4] Note that other environmental variables may be relevant.

6 Troubleshooting

[3] Using `threadpoolctl` one can inspect `python -m threadpoolctl -i numpy` on the compute nodes (i.e. via a job).

[4] Since each Cirrus CPU supports two threads with hyperthreading enabled by default, my guess would be that `OMP_NUM_THREADS=2` is ideal.

Common error	Common solutions
`/var/.../slurm_script:...cd: ...No such file or directory`	The path used by the `cd` command in your submission script is wrong
`slurmstepd:...test-job.sh: No such file or directory`	The path in your submission script specifying the location of the Bash script is wrong (remember to account for any preceding `cd`)
`python: can't open file...No such file or directory`	The path in your Bash script specifying the location of your python script is wrong
`...No such file or directory`	You got a path wrong!
`slurmstepd:...test-job.sh: Permission denied`	Your Bash script doesn't have executable permissions, `chmod +x` it
`bash: cannot set terminal process group (-1)...no job control in this shell`	Not an issue, ignore this
`Matplotlib created a temporary config/cache...`	Create a directory `mkdir .cache` in your work directory and add `export MPLCONFIGDIR=/work/.../user/.cache/matplotlib` to your Bash script before the `python` command to avoid this warning
`slurmstepd: error:...Exec format error`	Your Bash script does not have `#!/bin/bash -i` as it's first line
`slurmstepd: error: Detected 1 oom-kill event...your job may have been killed by the out-of-memory handler`	Your job ran out of memory; increase `cpus-per-task` in the submission script or write your code in a way to use less memory
I can't (yet) see output from my script in `slrum-jobID.out`	Cirrus doesn't seem to continuously write output from jobs, so some patience may be required. Adding `flush=True` to your `print` statements in Python may sort this (otherwise, use the `logging` module to write messages to a different file entirely).

Table 1: My job won't run!

Common error	Common solutions
WARNING: UNPROTECTED PRIVATE KEY FILE!	The permissions on your private key (the one that doesn't end with `.pub`) are too lax. From a Linux Shell `chmod 600 KEY_FILE` will set the right permissions (`644` for the public key)
`: Permission denied (publickey)`	you didn't upload your `.pub` key to SAFE; your private/public key pair aren't in the correct place on your system (default `~/.ssh/` on Linux)
`: Broken pipe`	Your `ssh` session timed-out / was interrupted, start a new one
`...No entry for termainal type...using DUMB terminal settings...WARNING: terminal is not fully functional`	Your terminal emulator is not know/supported by cirrus, setting `TERM=xterm-256color` before making the connection should be fine in most cases
`FILE#-£$*[NAME])(:"sad'as#`	Stick to using alphanumerical characters and `.`, `-`, `_` in filenames. Using characters that have special meaning in bash e.g. `$`, `[`, `?` and spaces is bound to result in you having a very bad time when moving files around
`something else`	Use `-v`, `-vv` or `-vvv` to get verbose output to debug `ssh` issues

Table 2: Secure shell hell

7 Useful resources

- Cirrus help desk email (monitored Mon–Fri 08:30–18:00): support@cirrus.ac.uk — the service team is typically very quick to respond and helpful (best to contact them using the email associated with your account on SAFE)
- Cirrus documentation, in particular Connecting to Cirrus, Using Python and Running Jobs on Cirrus.
- Cirrus Service Status page for live information on all Cirrus subsystems

8 Example files

`submit-example.slurm`

```
#!/bin/bash
# (this script must be executable by user)

# Slurm job options (name, compute nodes, job time)
#SBATCH --job-name=test-job
#SBATCH --time=96:00:00
#SBATCH --nodes=1           # Leave as 1 to run on same node (each node has 18 cores)
#SBATCH --tasks-per-node=1  # Leave as 1
#SBATCH --cpus-per-task=1   # Leave as 1 unless want multiple cores assigned

# Replace [budget code] below with your project code (e.g. t01)
#SBATCH --account=d422
# We use the "standard" partition as we are running on CPU nodes
```

```
#SBATCH --partition=standard
# We use the "standard" QoS as our runtime is less than 4 days
#SBATCH --qos=standard

# Change to the submission directory - use your own username and desired working directory
cd /work/d422/d422/user/
# Launch the job - N.B. path is relative to that set above
srun --cpu-bind=cores ./run-example.sh "$@"
```

run-example.sh

```
#!/bin/bash -i
# (start an interactive shell)
# (this script must be executable by user)
# Initialise miniconda
source /work/d422/d422/pfw1/miniconda-init.sh
# load the required python environment
conda activate oqupy
# Additional arguments to sbatch are passed on here
python ./example.py "$@"
# note location of "./" was set in submit-example.slurm
```

monitor-memory.sh

```
#! /usr/bin/env bash
# usage:
#./monitor-memory.sh PID
# where PID is target process
# e.g. monitor memory of most recent spawned python command:
# python job-script.py &
# ./monitor=memory.sh $!
echo $0 "for" $(whoami)"@""$HOSTNAME"
for i in {1..3000}
do
                a=$(date +'%Y-%m-%d %H:%M:%S')
                # 'rss' is resident set size for current shell process
                b=$(ps -o rss= $1 | awk '{printf "%.1f\n", $1 / 1024}') # convert to MB
                if [[ -z "$b" ]]; then
                                # if $1 not an existing process, ps returns empty string
                                echo "process $1 not found, exiting $0"
                                break
                else
                                # otherwise print timestamp and memory usage of $1
                                echo "$a usage $b MB"
                fi
                # run every minute
                sleep 60
done
```