Using the Cirrus computing facility

P. Fowler-Wright

2023-10-24

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 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 Accessing Cirrus

2.1 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.

2.2 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.

[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.

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.

3 Working with files on Cirrus

3.1 Filesystem structure

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 appropriate location for any software installation as well as scripts, input and output data.

3.2 Transferring files

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` in your user's *home* (not work) directory on Cirrus, 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 have a `ssh` server running on your machine.

3.3 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.

4 Using python on Cirrus

4.1 Overview

Early May 2023 there was a significant upgrade to the Python installations on Cirrus.[2] There are now a number of versioned Miniconda/Anaconda modules available on the login and compute nodes providing the common suite of scientific packages—`numpy`, `scipy`, `matplotlib` etc.—plus the ability to install additional packages from PyPI or the conda repositories. For those who want complete control over their python environments i.e. installed packages and their versions, or to install packages not on PyPI or Conda repositories, a local installation of Miniconda is still recommended, see Section 4.3. Otherwise, the centrally-installed python should be sufficient and Section 4.2 is for you.

4.2 Using the Cirrus python modules (easy)

In addition to the base python-2 and -3 executables,[3] Cirrus provides several different python installations or 'modules' with pre-installed packages that you will likely need to run your code.

In order to use a python module it firstly needs to be *loaded*. To list available modules, enter `module avail python`. At the time of writing, this produces

```
$ module avail python
------------------- /mnt/lustre/indy2lfs/sw/modulefiles --------------------
python/3.7.16  python/3.8.16-gpu  python/3.9.13  python/3.9.13-gpu ...
```

You will typically want to use the latest non-gpu specific installation, in this case `python/3.9.13`. To load the module, run `module load python/X.X.XX`:

```
$ module load python/3.9.13
```

After loading, the command `python` is aliased to a specific python version and you will have access to all packages provided by the module, e.g.

[2] Previously, python was provided by a single Anaconda distribution and the advice was to use a local Miniconda installation (Section 4.3) if additional packages were required. Alongside improving ease of use and reliability I expect performance was a target of the upgrade: the multiprocessing package `mpi4py` and machine learning frameworks `pytorch`, `tensorflow` may benefit from being built specifically for Cirrus hardware (GPUs). This is unlikely to matter to you (but if it sounds like it might, see Using Python).

[3] These are available immediately from when you login as `python2` and `python3`, but are without additional packages.

```
$ python
Python 3.9.12 (main, Apr  5 2022, 06:56:58) # Looks slightly out-dated!
>>> import numpy as np # no ImportError
>>> exit
```

To list all the packages provided by a module, run `pip list` *after* loading it. If this includes all the packages you need, then you are done—you simply need to add `module load python/X.X.XX` in your submission script before any python code is run (Section 5). If not, you will need to create a virtual environment.

4.2.1 Virtual environments

To create a virtual environment based on a Cirrus python module, firstly load the module e.g. `module load python/3.9.13` and then use

```
$ python -m venv --system-site-packages /work/d442/d422/user/my-env
```

Here `user` should of course be replaced by your own login and `my-env` a sensible name for your environment in your work directory (if you plan on having multiple environments you may wish to create a subdirectory). Next, run `extend-venv-activate` on the new environment:

```
$ extend-venv-activate /work/d422/d422/user/my-env
```

This ensures the selected Cirrus python module is loaded and unloaded whenever the environment is activated or deactivated. To use the environment in the current session, activate it using `source`:

```
$ source /work/d422/d422/user/my-env
```

You should notice `(my-env)` appears at the prompt. You can now install additional packages i.e. those not provided by the base set using `pip install` as follows:

```
$ pip install PACKAGE_NAME
```

Once you have finished installing packages, you can leave the virtual environment with `deactivate`. All that needs to be done now is to place the above `source .../my-env` command in your submission script before any python invocation (`module load` is *not* required), see Section 5.

4.3 Local Miniconda installation for python (advanced)

While the central python modules provide a wide array of python packages suitable for many projects, if you need complete control over your development environment it is best to start from an empty python environment. A local Miniconda installation, detailed below, is a reliable way to do this.[4]

4.3.1 Installation

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
```

[4] This *can* be done using the system-wide python (run `python3 -m venv` without loading a module), but I wouldn't recommend it—changes to the system python will likely break everything.

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
```

4.3.2 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[5] 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`.

5 Running jobs on Cirrus

5.1 Overview

The simplest workflow for running jobs on Cirrus requires two files:

1. A python script (run-me.py) containing the code you wish to run. This must be executable (`chmod +x run-me.py`) and contain the python shebang `#!/usr/bin/env python` on the first line.

2. A submission script (submit-me.slurm) containing job options and a command to run the python script. This does not need to be executable, but should contain commands to change to an appropriate working directory and load any required python modules or environments.

[5] `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`.

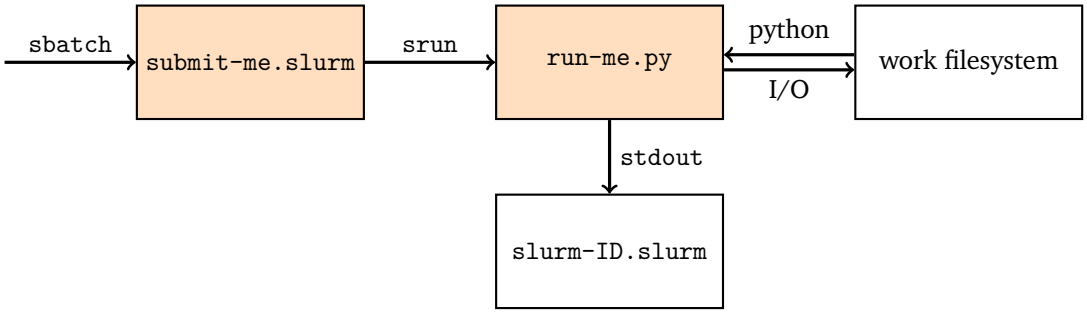

Figure 1: Job workflow

Examples for `submit-me.slurm` and `run-me.py` are attached to the end of this document. Both *must* be placed in your user's work directory (or a subdirectory thereof) in order to be accessible by the compute nodes. You should be able to modify these examples for your own purpose. Note `run-me.slurm` is a Shell script in which you can use for loops, redirects and generally any Bash shenanigans to e.g. start `run-me.py` multiple times with different arguments.

Once these scripts are in place, you can submit the job to the Cirrus scheduler using `sbatch`:

```
$ sbatch submit-me.slurm
Submitted batch job ID
```

Here `ID` is a 7 digit identifier assigned to the job. The job will be placed in a system-wide queue job and run when sufficient resource becomes available (if not immediately). `squeue -u user` can be used to view your submitted jobs with the status indicated in the `ST` column by '`PG`' for queued, '`R`' for running and '`CG`' for cancelling. The last one may be brought about by the command `scancel ID` which tells the schedule to cancel a job you submitted with identifier ID.

Once a job begins to run, any output that you would normally see in your terminal (i.e. `stdout`) is appended to a new file `slurm-ID.out` in the directory you issued the `sbatch` command from. This is useful to monitor the progress of jobs and debug errors. Of course, your python script may also read and write to other files whilst running.

The most common issue running jobs on Cirrus is getting paths linking the submission script, python script and any input/output data correct. Don't worry if things proceed a little by trial and error—in the first two cases at least (i.e. getting the python code running) an incorrect path will cause the job to fail immediately and no resource will be wasted. Personally, I place all my submission scripts in one directory (e.g. `/work/d422/d422/user/submission`), whilst the python script will likely be in a git repository within my work directory (e.g. `.../OQuPy/examples/...`) and I make sure the submission script `cd`s to that repository before any python is run.

A summary of the job workflow is given in Fig. 1. Finally, you may wish to edit options for your job in `submit-me.slurm`. The main ones are discussed in Section 5.2 below. The remainder of this section covers some useful tips and information. See also Section 6 which lists some common errors you may encounter and how to fix them.

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 Job options

The one option you will almost certainly want to set is the `job-name` (best to call it something sensible as this will be visible in the system-wide queue :). Beyond that, there are two main options you might consider changing:

1. `time` set 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 you will have access to.
2. `cpus-per-task` is the number of cores assigned to the job—in most cases you'll want to set this to `1`; see Section 5.5 for further discussion.

5.3 Passing arguments to your python scripts

Arguments can be passed to your python script from within the submission script as usual, e.g.

```
for T in 0 100 200
do
    echo "Running script at Temperature" $T "K"
    srun --cpu-bind=cores run-simulation.py $T
    sleep 1
done
```

You can also pass arguments to the submission script, referenced using Bash variables `"$1"`, `"$2"`,...or the convenient catch-all `"$@"`. For example, `sbatch submit-me.slurm -mass 70 -freq 123` in conjunction with

```
srun --cpu-bind=cores run-me.py "$@"
```

inside the submission script would result in `"-mass 70 -freq 123"` being passed directly to `run-me.py` (in which these arguments could be accessed via `sys.argv`).

5.4 Resource on Cirrus

Resource usage of a job is measured in Core hours (`Corehrs`). That's `number of cores assigned` $\times$ `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

5.5 `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 $\sim$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.[6] 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.[7] Note that other environmental variables may be relevant.

6 Troubleshooting

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

[7] 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:...run-me.py: No such file or directory` or `slurmstepd:...: Not a directory`	The path in your submission script to your python script is wrong (remember to account for any preceding `cd`)
`FileNotFoundError: [Errno 2] No such file or directory:...`	A path in your python script is wrong
`/usr/bin/env: 'python': No such file or directory`	You didn't load a python module or environment
`...No such file or directory`	You got a path wrong!
`slurmstepd:...run-me.py: Permission denied`	Your pythonscript doesn't have executable permissions, `chmod +x` it
`bash: cannot set terminal process group (-1)...no job control in this shell`	Innocuous error if seen when using `srun` with a Bash script; ignore
`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 python script does not have `#!/usr/bin/env python` 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

# Slurm job options
#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
#
# Specify project code
#SBATCH --account=d422
# Use the "standard" partition as running on CPU nodes
```

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

# Change to a directory under /work - use your own username and desired working directory
cd /work/d422/d422/user/

# Load a relevant python module or environment - uncomment ONE of the following lines
module load python/3.9.13 # System python module, no special environment (set version as required)
#source /work/d422/d422/user/my-env # System python module with environment (edit user/my-env as required)
#source /work/d422/d422/user/miniconda-init.sh; conda activate oqupy # Local Miniconda installation

# Avoid Matplotlib warning (edit user)
#export MPLCONFIGDIR='/work/d422/d422/user/.cache/matplotlib
#
# Launch the job with any arguments passed to this script
srun --cpu-bind=cores ./run-me.py "$@"
# N.B. path is relative to that set above (cd...) and --cpu-bind=cores option should be used
```

run-example.sh

```
#!/usr/bin/env python

# This script must be executable as user

import sys
print(f'Running {sys.argv[0]} with arguments {sys.argv[1:]}')
```

monitor-memory.sh

```
#! /usr/bin/env bash
# usage:
#./monitor-memory.sh PID
# where PID is target process
# e.g. monitor memory of most recent srun command:
# srun --cpu-bind=cores run-me.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
```