Slurm Job Submission How-to

Helen Wang - 2025-12-12 - Research Systems

How to Create Job Scripts with R, Python, Bash

In this tutorial we will write a job submission script for SLURM. Assume that you have an account on SOMHPC or other campus HPC system running SLURM and understand the jobs submission procedure introduced in SLURM User Guide.

Scenario

HPC cluster with a job manager such as SLURM is a great way to scale your jobs for data analysis and other large computation work! In this tutorial, we will walk through a very simple method to do this using different strategies.

1. Write an executable script in R / Python
2. Organize your inputs, output location, and scripts.
3. Loop over some set of variables and submit a SLURM job to use your executable to process each one.

We will cover each of these steps in detail.

Write an Executable Script

You first have some script in R or Python. It likely reads in data, processes it, and creates a result. You will need to turn this script into an executable, meaning that it accepts variable arguments.

Using R

A simple parse is to retrieve your script inputs with just a few lines:

```
args = commandArgs(TRUE)

input1 = args[1]

input2 = args[2]

input3 = args[3]

# Your code here!
```

if I saved this in a script called “run.R” I could then execute:

```
$ Rscript run.R bird1 bird2 bird3
```

and `input1` would be assigned to “bird1,” and “bird2” and “bird3” to `input2` and `input3`, respectively.

Using Python

In Python, we use the `sys` module. The same would look like this:

```
import sys

input1 = sys.argv[1]
input2 = sys.argv[2]
input3 = sys.argv[3]

# Your code here!
```

Calling would then look like:

```
$ python run.py bird1 bird2 bird3
```

`sys.argv` is actually just a list of your calling script and the input arguments following it. Python starts indexing at 0, and we are skipping over the value at `sys.argv[0]`. This would actually coincide to the name of your script.

A Little About Executables

When you write your executable, it’s good practice to **not hard code any variables** For example, if my script is going to process an input file, I could take in just a subject identifier and then define the full path in the script, but If you change a location, your script breaks. Instead, assign this path duty to the calling script. This means that your executable should instead expect a *general* `input_path`:

```
## R Example DO THIS
...
input_path = args[3]

(!file.exists(input_path)){
   cat('Cannot find', input_path, 'exiting!\n')
   stop()
}

## Python Example DO THIS
input_path = sys.argv[3]
if not os.path.exists(input_path)
    print('Ruhroh, %s doesn't exist!' %input_path)
    sys.exit 1
```

Notice that for both, as a sanity check we check that `input_path` exists.

Loop submission using your executable:

You then want to loop over some set of input variables (for example, csv files with data.) You can imagine doing this on your computer - each of the inputs would be processed in serial. That can take many hours if you have a few hundred inputs each taking 10 minutes, and it's totally unfeasible if you have thousands of simulations, each of which might need 30 minutes to an hour.

Strategy 1: Submit a Job File

A submission script looks like this:

```
#!/bin/bash

#SBATCH --job-name=MyHPC.job

#SBATCH --output=.out/MyHPC.out

#SBATCH --error=.out/MyHPC.err

#SBATCH --time=2-00:00

#SBATCH --mem=12000

#SBATCH --qos=debug

#SBATCH --mail-type=ALL

#SBATCH --mail-user=$USER@vcu.edu

Rscript $HOME/myHPCJOBS/run.R bird1 bird2 bird3
```

Importantly, notice the last line! It's just a single line that calls our script to run our job. In fact, look at the entire file, and the interpreter at the top - `#!/bin/bash` - it's **just** a bash script! The only thing that makes it a little different is all of the `#SBATCH` commands. What's that about? This is actually the syntax that SLURM understands as a configuration argument for your job. It just corresponds with the way that you submit the job to slurm using the `sbatch` command. In fact, you are free to write whatever lines that you need after the `SBATCH` lines. You can expect that the node running the job will have all the same information that you have on a login node. This means it will source your bash profile, and know the locations of your $HOME and $SCRATCH. It also means that you can run the same commands like `module load` if you need special software for the job.

What do all the different variables mean?

Some of the above are obvious, like `--mem` corresponds to memory in GB, `--time` in the format above means 2 days, and the output and error correspond to file paths to write output and error to. For full descriptions of all the options, the best source is the man pages (linux manual) which you can read via:

```
$ man sbatch
```

If you just had the one job file above, let's say it were called `MyHPC.job`, you would submit it like this to slurm:

```
sbatch MyHPC.job
```

If it falls within the node limits and accounting, you will see that the job is submit. If you need a helper tool to generate just one template, check out the Job maker that I put together a few years ago.

Strategy 2: Submit Directly to sbatch

What if you had the script RScript, and you didn't want to create a job file? You can do the exact same submission using sbatch directly:

```
sbatch --job-name=MyHPC.job \

       --output=.out/MyHPC.out \

       --error=.out/MyHPC.err
       --time=2-00:00 \
       --mem=12000 \
       --qos=debug \
       Rscript $HOME/MYHPCJOBS/run.R bird1 bird2 bird3
```

and then of course you would need to reproduce that command to run it again.

Write a Loop Submission Script

Here I will show you very basic example in each of R, Python, and bash to loop through a set up input variables (the subject identifier to derive an input path) to generate job files, and then submit them. We can assume the following:

- **the number of jobs to submit is within our accounting limits**, so we will submit them all at once (meaning they get added to the queue).
- **at the start of the script, you check for existence of directories.** Usually you will need to create a top level or subject level directory somewhere in the loop, given that it doesn't exist.
- **you have permission to write to where you need to write.** This not only means that you have write permission, but if you are writing to a shared space, you make sure that others will too.

Bash Submission

Bash scripting is the most general solution, and we will start with it. Here is a basic template to generate the SBATCH script above. Let's call this script `run_jobs.sh`

```
#!/bin/bash

# We assume running this from the script directory
job_directory=$PWD/.job
data_dir="${SCRATCH}/project/birdsfly”

birds=("bird1" "bird2")

for bird in ${birds[@]}; do

    job_file="${job_directory}/${bird}.job"

    echo "#!/bin/bash
#SBATCH --job-name=${bird}.job
#SBATCH --output=.out/${bird}.out
#SBATCH --error=.out/${bird}.err
#SBATCH --time=2-00:00
#SBATCH --mem=12000
#SBATCH --qos=debug
```

```
#SBATCH --mail-type=ALL

#SBATCH --mail-user=$USER@vcu.edu

Rscript $HOME/birdfly /run.R A B C" > $job_file

    sbatch $job_file

done
```

Notice that we are echoing the job into the `$job_file`. We are also launching the newly created job with the last line `sbatch $job_file`.

Good Practices:

Finally, here are a few good job submission practices.

- **Always use full paths**. For the scripts, it might be reasonable to use relative paths, and this is because they are run in context of their own location. However, in the case of data and other files, you should always use full paths.

- **Don’t run large computation on the login nodes!** It negatively impacts all cluster users. Grab a development node with `sdev`.

- **Think about how much memory you actually need**. You want to set a bit of an upper bound so a spike doesn’t kill the job, but you also don’t want to waste resources when you (or someone else) could be running more jobs.

- **You should generally not run anything massive before it is tested.** This means that after you write your loop script, you might step through it manually, submit the job, ensure that it runs successfully, and inspect the output.

- And as a reminder, here are some useful SLURM commands for checking your job.

```
# Show the overall status of each partition

sinfo

# Submit a job

sbatch .jobs/jobFile.job

# See the entire job queue

squeue
```

```
# See only jobs for a given user
squeue -u username

# Count number of running / in queue jobs
squeue -u username | wc -l

# Get estimated start times for your jobs
squeue --start -u username

# Show the status of a currently running job
sstat -j jobID

# Show the final status of a finished job
sacct -j jobID

# Kill a job with ID $PID
scancel $PID

# Kill ALL jobs for a user
scancel -u username

# Kill all pending jobs
scancel -u username --state=pending

# Run interactive node with 16 cores (12 plus all memory on 1 node) for 4 hours:
srun -n 12 -N 1 --mem=64000 --time 4:0:0 --pty bash

# Claim interactive node for exclusive use, 8 hours
srun --exclusive --time 8:0:0 --pty bash

# Same as above, but with X11 forwarding
srun --exclusive --time 8:0:0 --x11 --pty bash

# Same as above, but with priority over your other jobs
srun --nice=9999 --exclusive --time 8:0:0 --x11 --pty -p dev -t 12:00 bash

# Check utilization of group allocation
sacct

# Running jobs in the group allocation
srun -p groupid
sbatch -p groupid

# Stop/restart jobs interactively
# To stop:
scancel -s SIGSTOP job id

# To restart (this won't free up memory):
scancel -s SIGCONT job id

# Get usage for file systems
```

```
df -k

df -h $HOME

# Get usage for your home directory

du

# Counting Files

find . -type f | wc -l
```