Submitting a job with `sbatch`

HPC workloads are usually better suited to batch processing than interactive working. A batch job is sent to the system (submitted) with `sbatch`. Comments at the start of the script, which match a special pattern (`#SBATCH`) are read as Slurm options.

Batch vs interactive

The working pattern we are all familiar with is interactive: I type (or click) something, and the computer performs the associated action. Then I type (or click) the next thing.

You may recall this from the first tutorial.

The trouble with interactive environments

There is another reason why GUIs are less common in HPC environments: point-and-click is necessarily interactive. In HPC environments (as we'll see in session 3) work is scheduled in order to allow exclusive use of the shared resources. On a busy system there may be several hours wait between when you submit a job and when the resources become available, so a reliance on user interaction is not viable. In Unix, commands need not be run interactively at the prompt, you can write a sequence of commands into a file to be run as a script, either manually (for sequences you find yourself repeating frequently) or by another program such as the batch system.

The job might not start immediately, and might take hours or days, so we prefer a batch approach:

- plan the sequence of commands which will perform the actions we need
- write them into a script

I can now run the script interactively, which is a great way to save effort if I frequently use the same workflow, or ...

- submit the script to a batch system, to run on dedicated resources when they become available

Where does the output go?

- The batch system writes stdout and stderr from a job to a file named "slurm-12345.out"
  - Which you can change, using `sbatch` options
- While a job is running, it caches the stdout and stderr in the job working directory
- You can use redirection (See Tutorial 1) to send output of a specific command into a file

Writing and Submitting a job

There are two aspects to a batch job script:

- A set of `SBATCH` directives describing the resources required and other information about the job
- The script itself, comprised of commands to setup and perform the computations without additional user interaction

A simple example

A typical batch script on an NYU Prince cluster looks something like these:

<table>
<thead>
<tr>
<th>Using precompiled third-party software</th>
<th>Using self-developed or built</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=1
#SBATCH --time=5:00:00
#SBATCH --mem=2GB
#SBATCH --job-name=myTest
#SBATCH --mail-type=END
#SBATCH --mail-user=bob.smith@nyu.edu
#SBATCH --output=slurm_%j.out
module purge
module load stata/14.2
RUNDIR=$SCRATCH/my_project/run-${SLURM_JOB_ID/.*}
mkdir -p $RUNDIR
DATADIR=$SCRATCH/my_project/data
cd $RUNDIR
stata -b do $DATADIR/data_0706.do

We'll work through them more closely in a moment.

You submit the job with sbatch:

```
$ sbatch myscript.s
```

And monitor its progress (as is discussed further in here) with:

```
$ squeue -u $USER
```

What just happened? Here's an annotated version of the first script:

```
#!/bin/bash
# This line tells the shell how to execute this script, and is unrelated
# to SLURM.

# at the beginning of the script, lines beginning with "#SBATCH" are read by
# SLURM and used to set queueing options. You can comment out a SBATCH
# directive with a second leading #, eg:
#SBATCH --nodes=1

# we need 1 node, will launch a maximum of one task and use one cpu for the task:
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=1

# we expect the job to finish within 5 hours. If it takes longer than 5
# hours, SLURM can kill it:
#SBATCH --time=5:00:00

# we expect the job to use no more than 2GB of memory:
#SBATCH --mem=2GB
```
# we want the job to be named "myTest" rather than something generated
# from the script name. This will affect the name of the job as reported
# by squeue:
#SBATCH --job-name=myTest

# when the job ends, send me an email at this email address.
#SBATCH --mail-type=END
#SBATCH --mail-user=bob.smith@nyu.edu

# both standard output and standard error are directed to the same file.
# It will be placed in the directory I submitted the job from and will
# have a name like slurm_12345.out
#SBATCH --output=slurm_%j.out

# once the first non-comment, non-SBATCH-directive line is encountered, SLURM
# stops looking for SBATCH directives. The remainder of the script is executed
# as a normal Unix shell script

# first we ensure a clean running environment:
module purge
# and load the module for the software we are using:
module load stata/14.2

# next we create a unique directory to run this job in. We will record its
# name in the shell variable "RUNDIR", for better readability.
# SLURM sets SLURM_JOB_ID to the job id, ${SLURM_JOB_ID/.*} expands to the job
# id up to the first '.' We make the run directory in our area under $SCRATCH, because
# at NYU HPC
# $SCRATCH is configured for the disk space and speed required by HPC jobs.
RUNDIR=$SCRATCH/my_project/run-${SLURM_JOB_ID/.*}
mkdir $RUNDIR

# we will be reading data in from somewhere, so define that too:
DATADIR=$SCRATCH/my_project/data

# the script will have started running in $HOME, so we need to move into the
# unique directory we just created
cd $RUNDIR
# now start the Stata job:
stata -b do $DATADIR/data_0706.do

The second script has the same SBATCH directives, but this time we are using code we compiled ourselves. Starting after the SBATCH directives:

```bash
# first we ensure a clean running environment:
module purge

# and ensure we can find the executable:
SRCDIR=$HOME/my_project/code

# create a unique directory to run this job in, as per the script above
RUNDIR=$SCRATCH/my_project/run-${SLURM_JOB_ID/.*}
mkdir $RUNDIR

# By default the script will have started running in the directory we ran sbatch from.
# Let's assume our input file is in the same directory in this example. SLURM
# sets some environment variables with information about the job, including
# SLURM_SUBMIT_DIR which is the directory the job was submitted from. So let's
# go there and copy the input file to the run directory on /scratch:
cd $SLURM_SUBMIT_DIR
cp my_input_params.inp $RUNDIR

cd $RUNDIR

# load whatever environment modules the executable needs:
module load fftw/intel/3.3.5

# run the executable (sending the contents of my_input_params.inp to stdin)
$SRCDIR/my_exec.exe < my_input_params.inp
```

Submitting a job

Jobs are submitted with the sbatch command:

```bash
$ sbatch options job-script
```

The options tell SLURM information about the job, such as what resources will be needed. These can be specified in the job-script as SBATCH directives, or on the command line as options, or both (in which case the command line options take precedence should the two contradict each other). For each option there is a corresponding SBATCH directive with the syntax:

`#SBATCH option`

For example, you can specify that a job needs 2 nodes and 4 cores on each node (by default one CPU core per task) on each node by adding to the script the directive:

```bash
#!/bin/bash
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=4
```
or as a command-line option to `sbatch` when you submit the job:

```
$ sbatch --nodes=2 --ntasks-per-node=4 my_script.s
```

### Options to manage job output:

- `-J jobname`
  - Give the job a name. The default is the filename of the job script. Within the job, `$SBATCH_JOB_NAME` expands to the job name
- `-o path/for/stdout`
  - Send stdout to path/for/stdout. The default filename is slurm-$({SLURM_JOB_ID}).out, e.g. slurm-12345.out, in the directory from which the job was submitted
- `-e path/for/stderr`
  - Send stderr to path/for/stderr.
- `--mail-user=my_email_address@nyu.edu`
  - Send email to my_email_address@nyu.edu when certain events occur.
- `--mail-type=
  - Valid type values are NONE, BEGIN, END, FAIL, REQUEUE, ALL...

### Options to set the job environment:

- `--export=VAR1,VAR2=\"some value\",VAR3`
  - Pass variables to the job, either with a specific value (the `VAR=\"some value\"`) or from the submitting environment (without `\"\``)
- `--get-user-env=[timeout][mode]
  - Run something like `su - <username> -c /usr/bin/env` and parse the output. Default timeout is 8 seconds. The mode value can be "S", or "L" in which case "su" is executed with "-l" option

### Options to request compute resources:

- `-t, --time=time`
  - Set a limit on the total run time. Acceptable formats include "minutes", "minutes:seconds", "hours:minutes:seconds", "days-hours", "days-hours:minutes" and "days-hours:minutes:seconds"
- `--mem=MB`
  - Memory required per node in MegaBytes
- `--mem-per-cpu=MB`
  - Memory required per allocated CPU in MegaBytes
- `-N, --nodes=num`
  - Number of nodes are required. Default is 1 node
- `-n, --ntasks=num`
  - Maximum number tasks will be launched. Default is one task per node
- `--ntasks-per-node=ntasks`
  - Request that ntasks be invoked on each node
- `-c, --cpus-per-task=ncpus`
  - Require ncpus number of CPU cores per task. Without this option, allocate one core per task

Requesting the resources you need, as accurately as possible, allows your job to be started at the earliest opportunity as well as helping the system to schedule work efficiently to everyone's benefit.

### Options for running interactively on the compute nodes with `srun`:

- `-n num`
  - Specify the number of tasks to run, e.g. `-n4`. Default is one CPU core per task. Don't just submit the job, but also wait for it to start and connect stdout, stderr and stdin to the current terminal
- `-t time`
  - Request job running duration, e.g. `-t1:30:00`
- `--mem=MB`
  - Specify the real memory required per node in MegaBytes, e.g. `--mem=4000`
- `--pty`
  - Execute the first task in pseudo terminal mode, e.g. `--pty /bin/bash,` to start a bash command shell
- `--x11`
  - Enable X forwarding, so programs using a GUI can be used during the session (provided you have X forwarding to your workstation set up)

To leave an interactive batch session, type `exit` at the command prompt.

### Options for delaying starting a job:

- `-d, --dependency=dependency_list`
  - For example, `--dependency=afterok:12345`, to delay starting this job until the job 12345 has completed successfully.
- `--begin=time`
  - Delay starting this job until after the specified date and time, e.g. `--begin=9:42:00`, to start the job at 9:42:00 am.

### Options for running many similar jobs:
Submit an array of jobs with array ids as specified. Array ids can be specified as a numerical range, a comma-separated list of numbers, or as some combination of the two. Each job instance will have an environment variable SLURM_ARRAY_JOB_ID and SLURM_ARRAY_TASK_ID. For example:

--array=1-11, to start an array job with index from 1 to 11
--array=1-7:2, to submit an array job with index step size 2
--array=1-9%4, to submit an array job with simultaneously running job elements set to 4

The srun command is similar to pbsdsh. It launches tasks on allocated resources.