# Running jobs

## Writing a job script

We are now ready to put together a toy example on a batch job. First we need to create a plain text file containing a valid shell script that executes the calculation we want to perform on the cluster.

To work with text files on the cluster you can use any of the installed text editors, desktop logins provide simple text editors from the menu, or vim, emacs, nano etc.. Pick one and learn how to use it, in this example we will use nano.

First we create a *job script file* that is going to specify our batch job, lets simply call it `jobscript`

```
[emilia@vera ~]$ nano jobscript
```

In the job script file we need to specify all information needed by the scheduler to execute the job. Below are minimal examples for Vera and Alvis.

Vera example:

```
#!/usr/bin/env bash
#SBATCH -A NAISS1234-5-67 -p vera
#SBATCH -n 64
#SBATCH -t 0-00:10:00

echo "Hello cluster computing world!"
sleep 60
```

Alvis example (Note the necessity of launching at least one GPU):

```
#!/usr/bin/env bash
#SBATCH -A NAISS1234-5-67 -p alvis
#SBATCH -N 2 --gpus-per-node=T4:8  # We're launching 2 nodes with 8 Nvidia T4
GPUs each
#SBATCH -t 0-00:10:00

echo "Hello cluster computing world!"
sleep 60

#Here you should typically call your GPU-hungry application
```

write it in the editor (or copy and paste it) and save (using `Ctr-o` in `nano`) and exit (`Ctr-x`). But more importantly is to figure out what the job file content means.

The first row is just a bash script "Shebang".

The rows that start with `#SBATCH` are special commands given directly to the scheduler. The most important ones are:

1. Specification of under which *project* `-A` the job is to be accounted, here `NAISS1234-5-67`
2. The *queue* or *partition* `-p` that the job should be scheduled to.
3. Job size; some combination of `--ntasks/-n`, `--ntasks-per-node`, `--nodes/-N`, `--cpus-per-task/-c`.
4. Job walltime, here we have to specify the maximum time that the job should be let to run. If the job does not end within the specified time it will be killed by the scheduler. Here we requested 0 days, 10 minutes.
5. Constrain nodes options (see below)
6. Request GPUs (see below)
7. Request some special setup for the job with e.g. `--gres=ptmpdir:1` for parallel `$TMPDIR`

A few things to note regarding the above points:

1. On Alvis, you must request a GPU.
2. We don't allow node-sharing for multi node jobs. The flag `--exclusive` is automatically added when `-N` is larger than 1.
3. Requesting `-N 1` does *not* mean 1 full node.

Everything after the scheduler information is the actual script that will be executed on the compute nodes. In the above examples we simply write some output and wait for 60 seconds.

There are many more flags you can give to `sbatch` (https://slurm.schedmd.com/sbatch.html).

## Binding job to cores

Binding the jobs processes to cores are usually beneficial for performance but is somewhat complicated to manually get right. The case where this is most important is for hybrid MPI+OpenMP jobs when both -n (and/or --ntasks-per-node) and -c has been used in the job specification. The easiest solution is simply to let the batch system handle this by using `srun` instead of `mpirun`.

```
#!/usr/bin/env bash
#SBATCH -A NAISS1234-5-67 -p alvis
#SBATCH -N 2 --gpus-per-node=T4:8  # We're launching 2 nodes with 8 Nvidia T4
GPUs each
#SBATCH -n 16  # One MPI task per GPU
#SBATCH -c 4   # 4 cores per MPI task
#SBATCH -t 0-00:10:00

srun mpi_openmp_program
```

## Submitting and monitoring jobs

So now we've managed to write a job script and store it to a file called `jobscript`. To run the job you now want to send it to the Scheduler. For this we use the command `sbatch`:

```
[emilia@vera1 ~]$ ls jobscript
[emilia@vera1 ~]$ sbatch jobscript
Submitted batch job 123456
```

The job is now sent to the Scheduler that will put in the job queue of the cluster. The number that was printed when we *submitted* the job is called the **JobID** and is a unique identifier for each job. To monitor the status of our job we ask for information from the queue using another command:

```
[emilia@vera1 ~]$ squeue -u emilia
  JOBID PARTITION     NAME     USER  ST       TIME  NODES NODELIST(REASON)
 123456      vera  JobTest   emilia   R       0:07      1 vera06-2
```

Note that even more useful information can be obtained using the command `jobinfo -u emilia` instead of `squeue`.

Decoding the information we see that the job is already running, as seen from the *status* flag that is set to `R`, looking at the *elapsed time* the job was actually started 7 seconds ago. In general there will not be available compute nodes at the time you submit a job and then the job will be put in *queued status* until it is started by the Scheduler. At any point it is possible to *kill* the job (it is not uncommon to realise a mistake only after submitting a job). To *kill* a job you just have to know its **JobID** (that we know from above) and run the command:

```
[emilia@vera1 ~]$ scancel 123456
[emilia@vera1 ~]$
```

Try this out a couple of times on your own, For information on the usage of the login-node, and GPUs idle, used and queued, look at:

1. Submit the job

2. Look at the queue and the job status on a node (`vera06-2` in the above example)

This is slightly more *advanced*, but you may need it at times. When your job starts executing ssh into that node (`ssh vera06-2`) to monitor the performance and/or usage of the resource.

en our job is done executing and has disappeared from the queue listing we are now ready to look at the results.

```
[emilia@vera1 ~]$ ls jobscript
slurm-123456.out
[emilia@vera1 ~]$ cat slurm-123456.out
Hello cluster computing world!
[emilia@vera1 ~]$
```

Our job has now created a new file `slurm-123456.out`. We could redirect the output to a different file using `#SBATCH -o xxx`.

Our little toy example is just the necessary steps to get started, yet to do some real computational work, all you need to do is to replace the line that echos a message and waits for 60 sec. with the command that you use to run your program.

If you are interested or need to know more, e.g. what exactly the output of the queue listing means, this can be found in the *manual pages* of each command. The *manual pages* are available directly on the command-line by running `man command`, just replace `command` with e.g. `squeue`.

## Monitoring jobs

To get information on how a job is running on the node(s), you can generate a link to a Grafana status page by running `job_stats.py JOBID` (replace JOBID with your actual jobid). The status page gives an overview of what the node(s) in your job are doing. Check e.g. GPU and CPU utilisation, disk usage, etc.

## Memory and other node features

In order to request nodes with special features, for example nodes with more memory or limit multi-node jobs to run within the same Infiniband-switch, you can use the `--constraint` or equivalently `-C` flag with `sbatch`. E.g:

```
#SBATCH -C MEM512           # a 512GB node will be allocated
#SBATCH -C MEM512|MEM1024   # either 512GB or 1024GB RAM node will be
allocated
```

**Note:** Not all combinations of constraints are available. Though rarely ever used, you can use complex logic and node counts when specifying constraints. See https://slurm.schedmd.com/sbatch.html for details on the `-C` flag.

The most relevant features you may need to pick from as listed:

| Resource | | | | | |
|---|---|---|---|---|---|
| Vera | MEM512 | MEM768 | MEM1024 | MEM1536 | (MEM2048) |
| Alvis | MEM512 | MEM1536 | | | |

Of course, not all combinations of constraints might exist, some only exists in private partitions. You can get a full breakdown of all available nodes and their partition, number of cores, features, and generic resources with the command:

```
sinfo -o "%20N  %9P %4c  %24f  %50G"
# or only view a specific partition:
sinfo -p vera -o "%20N  %4c  %24f  %50G"
```

**Note**

Unless you know you have special requirements, do not specify any constraints on your job.

Requesting more memory or other features does not cost any more core-hours than other nodes, but you will have to wait longer in the queue for the particular nodes to become available.

## GPUs

**Important**

If you allocate GPU:s, you need to make sure that you use all of the GPU:s that you request. If the GPU:s you allocate are not utilized, your job may be automatically terminated.

You can request GPU nodes using the right `#SBATCH` flags. On Vera, you should request it using SLURMs generic resource system. On Alvis, you are strongly recommended to instead use the `#SBATCH --gpus-per-node=xxx` flag.

On Vera, you can request the following NVidia GPUs:

```
#SBATCH --gpus-per-node=V100:1   # up to 2
#SBATCH --gpus-per-node=T4:1     # up to 1
#SBATCH --gpus-per-node=A40:1    # up to 4
#SBATCH --gpus-per-node=A100:1   # up to 4
```

On Alvis, you can request the following NVidia GPUs:

```
#SBATCH --gpus-per-node=V100:1      # up to 4
#SBATCH --gpus-per-node=T4:1        # up to 8
#SBATCH --gpus-per-node=A40:1       # up to 4
#SBATCH --gpus-per-node=A100:1      # up to 4
#SBATCH --gpus-per-node=A100fat:1   # up to 4
```

#### Important configurations via gres flags (currently only applicable to Alvis)

Note the following two gres-flags that can be useful in certain cases:

1. **Nvidia MPS**: This feature allows for multiple processes (e.g. MPI ranks) to simultaneously access GPU(s). It can be enabled using gres flags in your job script: `#SBATCH --gres=nvidiamps:1`. This feature is useful when the workload assigned to a single process is **not** sufficient to keep the entire computational capacity of the GPU busy during execution. Using MPS, the under-utilised capacity can be used by another processes at the same time. Nvidia MPS mitigates context switches when multiple processes need to share the GPU device and is therefore crucial for performance. For that, the MPS server must be the only process that communicates with the GPU device. Therefore, you should activate the exclusive compute mode when using this feature (see below).
2. **Exclusive compute mode**: In contrast to the default mode which allows for multiple contexts per device, **exclusive** mode allows only one context (process) per device. To enable this feature, use `#SBATCH --gres=gpuexcl:1`. Note that it must be activated when using the Nvidia MPS feature.

### GPU utilisation statistics

A management daemon collects GPU utilisation statistics for every job and creates a log of the data after the job ends. The log file includes power and memory usage as well as the fraction of the GPU's streaming multiprocessors used (each V100 device has 80 SMs while that of a T4 device is 40 and A100 128).

## Running job-arrays

There is often a need to run a series of similar simulations with varying inputs. For this purpose, there exist a job-array feature in SLURM. Using the `--array` flag for sbatch introduces a new environment variable `$SLURM_ARRAY_TASK_ID` which is used by the script to determine what simulation to run.

You **must** use this feature if you need to submit a list of jobs to the queue. Do **not** write your own scripts that executes a ton of induvidual `sbatch jobscript` commands.

This offer several great advantages:

1. It's less work for you (no need to generate and modify tons of different job-scripts that are almost identical).
2. The squeue command is more readable for everyone (the whole array is only 1 entry)
3. It's easier for support to help you.
4. The scheduler isn't overloaded. For safety, there is a max-size of the queue, so very large submissions *must* use arrays to avoid hitting this limit. It also affects the backfill algorithm when scheduling jobs.
5. Convenient to cancel every job in a given array if you discover some mistake.
6. Email notifications can be customised to be sent only when all jobs have finished.
7. It's vastly simpler to re-run an aborted simulation (for example when a `NODE_FAIL` occurs). E.g. if job 841342_**3** dies, then:

```
sbatch --array=3 my_jobscript.sh
```

### Example 1: Numbered files

We have input files named `data_0.in`, `data_1.in`, ..., `data_10.in`, which we want to run on 2 nodes each on Vera. Our input file might then look something like this:

```
#!/usr/bin/env bash
#SBATCH -A NAISS1234-5-67 -p vera
#SBATCH -J CrashBenchmark
#SBATCH -N 2 --ntasks-per-node 32 -c 2
#SBATCH -t 0-04:00:00

module load intel

echo "Running simulation on data_${SLURM_ARRAY_TASK_ID}.in"

mpirun ./my_crash_sim data_${SLURM_ARRAY_TASK_ID}.in
```

### Example 2: Named input files

We have directories that are not enumerated, e.g: "At", "Bi", "Ce", etc. located in `input_data/` We need to run a 1 core simulation in each directory, so we could do:

```
#!/usr/bin/env bash
#SBATCH -A NAISS1234-5-67 -p vera
#SBATCH -J CobaltDiffusion
#SBATCH -N 1 --exclusive
#SBATCH -t 0-20:00:00

module load intel

# Create a list of each directory:
DIRS=($(find input_data/))
# (we could have also specified the list directly if we wanted: DIRS=(At Bi
Ce)

# Fetch one directory from the array based on the task ID (index starts from
0)
CURRENT_DIR=${DIRS[$SLURM_ARRAY_TASK_ID]}

echo "Running simulation $CURRENT_DIR"

# Go to folder
cd $CURRENT_DIR

./diffusion_sim cobalt_data.inp
```

These scripts can both be submitted using the syntax

```
sbatch --array=0-10 my_script.sh
```

### Example 3: Unstructured parameters as flags

We have an unstructured parameters (perhaps from an optimization). While we could just generate input files and just do it like example 2, another neat trick if we need to pass flags to a software is to write all bash variables directly to file `inputs-XYZ.txt` (space seperated):

```
temperature=78.3 pressure=114.5 outputdir="output/78.3/114.5/"
temperature=79.0 pressure=115.5 outputdir="output/79.0/115.5/"
temperature=80.1 pressure=117.7 outputdir="output/80.1/117.7/"
...
```

and then grab a line and evaluate it in bash:

```
#!/usr/bin/env bash
#SBATCH -A NAISS1234-5-67 -p vera
#SBATCH -N 1 -n 8
```

```
#SBATCH -t 0-20:00:00

input_file=$1  # opting to also take the file as an input argument

# Read the given line from the input file and evaluate it:
eval `head -n $SLURM_ARRAY_TASK_ID $input_file | tail -1`

module load Python/3.9.6-GCCcore-11.2.0

echo "[`date`] Running tempoerature=${temperature} pressure=$pressure
seed=$seed"

python3 pressure_cooker.py --temperature=$temperature --pressure=$pressure --
output_dir=$outputdir
```

which we could submit as:

```
sbatch --array=1-1000 -J XYZ_analysis my_script.sh inputs-XYZ.txt
```

For more examples and details, see the SLURM manual on job arrays:
https://slurm.schedmd.com/job_array.html

If you are unsure how to make use of an job-array, please contact the support for help writing a suitable jobscript.

### Job-arrays in other workflow managers

- submitit
- NextFlow

## Running interactive jobs

With the Open Ondemand portals

- Vera:https://vera.c3se.chalmers.se
- Alvis: https://alvis.c3se.chalmers.se

you can conveniently start a full interactive desktop or use webapps for Jupyter, MATLAB and more. These interactive sessions are not tied to any login node, and will continue to run interrupted until their walltime runs out, even if you disconnect. You should **always only** use the portals for any long running interactive work.

You are allowed to use RDP on login nodes for light/moderate tasks that require interactive input. Heavy load for hours is to much and you must use jobs. It is also possible to launch interactive `srun` session against a compute node, for light, short term work with

```
srun -A C3SE2019-1-2 -n 10 -t 00:30:00 --pty bash
```

you are *eventually* presented with a shell on the node:

```
[ohmanm@vera12-3]#
```

You can also directly start software, and add the `--x11` flags for X forwarding (3D acceleration will not be supported; for this you must use the portals).

- Not good for anything long running. Prefer the portal, or make batch jobs.
- Somewhat useful for **short** debugging sessions on a node, application problems, longer compilations.
- Not useful when there is a long queue (you still have to wait like all jobs).
- You will be tied to the login node and when we need to restart login nodes it will **kill** all `srun` jobs. We can not wait for individual jobs to finish if we have security updates, so you must **save often** and be ready for your jobs to die when the login node is rebooted. Using the portals avoids this.