SLURM usage guide

The reason you want to use the cluster is probably the computing resources it provides. With around 400 people using the cluster system for their research every year, there has to be an instance organizing and allocating these resources. This instance is called the batch system (“scheduler”, “resource manager”), and compute nodes are only accessible when the system grants a request for resources. These requests are made by either submitting a text file containing (bash-) instructions about what to do, or you can request an “interactive” batch job that puts you directly into a command shell on a compute node (or a set of compute nodes) to work with, as soon as the resources you requested are free.

The SLURM Workload Manager

SLURM (**S**imple **L**inux **U**tility for **R**esource **M**anagement) is a free open-source batch scheduler and resource manager that allows users to run their jobs on the LUIS compute cluster. It is a modern, extensible batch system that is installed on many clusters of various sizes around the world. This chapter describes the basic tasks necessary for submitting, running and monitoring jobs under the SLURM Workload Manager on the LUIS cluster. Detailed information about SLURM can be found on the official SLURM website.

Here are some of the most important commands to interact with SLURM:

- **sbatch** - submit a batch script
- **salloc** - allocate compute resources
- **srun** - allocate compute resources and launch job-steps
- **squeue** - check the status of running and/or pending jobs
- **scancel** - delete jobs from the queue
- **sinfo** - view intormation abount cluster nodes and partitions
- **scontrol** - show detailed information on active and/or recently completed jobs, nodes and partitions
- **sacct** - provide the accounting information on running and completed jobs
- **slurmtop** - text-based view of cluster nodes' free and in-use resources and status of jobs

Some usage examples for these commands are provided below. As always, you can find out more using the manual pages on a terminal/console on the system (like `man squeue`) or on the SLURM manuals' website.

Partitions

Compute nodes with similar hardware attributes (like e.g. the same cpu) in the cluster are usually grouped in partitions. Each partition can be regarded as somewhat independent from others. A batch job can be submitted in such a way that it can run on one of several partitions, and a compute node may also belong to several partitions simultaneously to facilitate selection. Jobs are allocated resources like cpu cores, memory and time within a single partition for executing tasks on the cluster. A concept called “job steps” is used to execute several tasks simultaneously or sequentially within a job using the `srun` command.

The table below lists the currently defined partitions and their parameters/constraints:

Part of cluster	Max Job Runtime	Max Nodes Per Job	Max CPUs per User	Default Runtime	Default Memory per CPU	Shared Node Usage
amo, dumbo, haku, lena, taurus, … (generic)	200 hours		800	24 hours	4000 MB	yes
gpu nodes	24 hours	1		1 hour	1600 MB	yes

To keep things fair, control job workload and keep SLURM responsive, we enforce some additional restrictions:

SLURM limits	Max number of jobs running	Max number of jobs submitted
per user	64	500
cluster-wide	10000	20000

Based on available resources and in keeping with maintaining a fair balance between all users, we may sometimes be able to accommodate special needs for a limited time. In that case, please submit a short explanation to cluster-help@luis.uni-hannover.de.

To list job limits relevant for you, use the `sacctmgr` command:

```
sacctmgr -s show user
sacctmgr -s show user format=user,account,maxjobs,maxsubmit,maxwall,qos
```

Up-to-date information on ALL available nodes:

```
sinfo -Nl
scontrol show nodes
```

Information on partitons and their configuration:

```
sinfo -s
scontrol show partitions
```

Interactive jobs

Please note: when you have a *non-interactive* (standard) reservation/running job on a node or a set of nodes, you may *also* directly open additional shell(s) to that node(s) coming from a login node, e.g. for watching/debugging/changing what happens. But beware: you will get kicked out as soon as your job finishes.

Batch submission is the most common and most efficient way to use the computing cluster. Interactive jobs are also possible; they may be useful for things like:

- working with an interactive terminal or GUI applications like R, iPython, ANSYS, MATLAB, etc.
- software development, debugging, or compiling

You can start an interactive session on a compute node using the SLURM `salloc` command. The following example submits an interactive job that requests 12 tasks (this corresponds to 12 MPI ranks) on two compute nodes and 4 GB memory per CPU core for an hour:

```
[user@login02 ~]$ salloc --time=1:00:00 --nodes=2 --ntasks=12 --mem-per-cpu=4G --x11
 salloc: slurm_job_submit: set partition of submitted job to amo,tnt,gih
 salloc: Pending job allocation 27477
 salloc: job 27477 queued and waiting for resources
 salloc: job 27477 has been allocated resources
 salloc: Granted job allocation 27477
 salloc: Waiting for resource configuration
 salloc: Nodes amo-n[001-002] are ready for job
[user@amo-n001 ~]$
```

The option `--x11` sets up X11 forwarding on the first(master) compute node enabling the use of graphical applications.

Note: Unless you specify a cluster partition explicitly, all partitions that you have access to will be available for your job.

Note: If you do not explicitly specify memory and time parameters for your job, the corresponding default values for the cluster partition to which the job will be assigned will be used. To find out the default time and memory settings for a partition, e.g. amo, look at the `DefaultTime` and `DefMemPerCPU` values in the `scontrol show partitions amo` command output.

Once the job starts, you will get an interactive shell on the first compute node (`amo-n001` in the example above) that has been assigned to the job, where you can interactively spawn your applications. The following example compiles and executes the MPI `Hello World` program (save the source code to the file `hello_mpi.c`):

hello_mpi.c

```
#include "mpi.h"
```

```
#include <stdio.h>

int main (int argc, char** argv) {
  int  ntasks, taskid, len;
  char hostname[MPI_MAX_PROCESSOR_NAME];

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD,&ntasks);
  MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
  MPI_Get_processor_name(hostname, &len);

  printf ("Hello from task %d of %d on %s\n", taskid, ntasks, hostname);

  MPI_Finalize();
}
```

```
[user@amo-n001 ~]$ module load GCC/9.3.0 OpenMPI/4.0.3
[user@amo-n001 ~]$ mpicc hello_mpi.c -o hello_mpi
[user@amo-n001 ~]$ srun --ntasks=6 --distribution=block hello_mpi
 Hello from task 0 of 6 on amo-n001
 Hello from task 1 of 6 on amo-n001
 Hello from task 2 of 6 on amo-n001
 Hello from task 3 of 6 on amo-n001
 Hello from task 4 of 6 on amo-n001
 Hello from task 5 of 6 on amo-n002
```

Note: If you want to run a parallel application using Intel MPI Library (e.g by loading the module `impi/2020a`) then provide the `srun` command with an additional option `--mpi=pmi2`

Note: Environment variables set on the login node from which the job was submitted are not passed to the job.

The interactive session is terminated by typing `exit` on the shell:

```
[user@amo-n001 ~]$ exit
 logout
 salloc: Relinquishing job allocation 27477
```

Alternatively you can use the `srun –pty $SHELL -l` command to interactively allocate compute resources, e.g.

```
[user@login02 ~]$ srun --time=1:00:00 --nodes=2 --ntasks=12 --mem-per-cpu=4G --x11 --pty
$SHELL -l
 srun: slurm_job_submit: set partition of submitted job to amo,tnt,gih
[user@amo-n004 ~]$
```

Submitting a batch script

An appropriate SLURM job submission file for your job is a shell script with a set of directives at the beginning of the file. These directives are issued by starting a line with the string #SBATCH. A suitable batch script is then submitted to the batch system using the `sbatch` command.

An example of a serial job

The following is an example of a simple serial job script (save the lines to the file `test_serial.sh`).

Note: change the #SBATCH directives to your use case where applicable.

example_serial_slurm.sh

```
#!/bin/bash -l
#SBATCH --job-name=test_serial
#SBATCH --ntasks=1
#SBATCH --mem-per-cpu=2G
#SBATCH --time=00:20:00
#SBATCH --constraint=[skylake|haswell]
#SBATCH --mail-user=user@uni-hannover.de
#SBATCH --mail-type=BEGIN,END,FAIL
#SBATCH --output test_serial-job_%j.out
#SBATCH --error test_serial-job_%j.err

# Change to my work dir
cd $SLURM_SUBMIT_DIR

# Load modules
module load my_module

# Start my serial app
srun ./my_serial_app
```

To submit the batch job, use

```
sbatch example_serial_slurm.sh
```

Note: as soon as compute nodes are allocated to your job, you can establish an `ssh` connection from the login machines to these nodes.

Note: if your job uses more resources than defined with the #SBATCH directives, the job will automatically be killed by the SLURM server.

The table below shows frequently used sbatch options that can either be specified in your job script with the #SBATCH directive or on the command line. Command line options override options in the script. The commands `srun` and `salloc` accept the same set of options. Both long and short options are listed.

Options	**Default Value**	**Description**
–nodes=<N> or -N <N>	1	Number of compute nodes
–ntasks=<N> or -n <N>	1	Number of tasks to run
–cpus-per-task=<N> or -c <N>	1	Number of CPU cores per task
–ntasks-per-node=<N>	1	Number of tasks per node
–ntasks-per-core=<N>	1	Number of tasks per CPU core
–mem-per-cpu=<mem>	partition dependent	memory per CPU core in MB
–mem=<mem>	partition dependent	memory per node in MB
–gres=gpu:<type>:<N>	-	Request nodes with GPUs
–time=<time> or -t <time>	partition dependent	Walltime limit for the job
–partition=<name> or -p <name>	none	Partition to run the job
–constraint=<list> or -C <list>	none	Node-features to request
–job-name=<name> or -J <name>	job script's name	Name of the job
–output=<path> or -o <path>	slurm-%j.out	Standard output file
–error=<path> or -e <path>	slurm-%j.err	Standard error file
–mail-user=<mail>	your account mail	User's email address
–mail-type=<mode>	-	Event types for notifications
–exclusive	nodes are shared	Exclusive acccess to node

To obtain a complete list of parameters, refer to the sbatch man page: `man sbatch`

Note: if you submit a job with `–mem=0`, it gets access to the complete memory of each allocated node.

By default, the stdout and stderr file descriptors of batch jobs are directed to `slurm-%j.out` and `slurm-%j.err` files, where `%j` is set to the SLURM batch job ID number of your job. Both files will be found in the directory in which you launched the job. You can use the options –output and –error to specify a different name or location. The output files are created as soon as your job starts, and the output is redirected as the job runs so that you can monitor your job's progress. However, due to SLURM performing file buffering, the output of your job will not appear in the output files immediately. To override this behaviour (**this is not recommended in general, especially when the job output is large**), you may use -u or –unbuffered either as an #SBATCH directive or directly on the sbatch command line.

If the option –error is not specified, both stdout and stderr will be directed to the file specified by –output.

Example of an OpenMP job

For OpenMP jobs, you will need to set –cpus-per-task to a value larger than one and explicitly define the OMP_NUM_THREADS variable. The example script launches eight threads, **each** with 2 GiB of memory and a maximum run time of 30 minutes.

example_openmp_slurm.sh

```
#!/bin/bash -l
#SBATCH --job-name=test_openmp
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=8
#SBATCH --mem-per-cpu=2G
#SBATCH --time=00:30:00
#SBATCH --constraint=[skylake|haswell]
#SBATCH --mail-user=user@uni-hannover.de
#SBATCH --mail-type=BEGIN,END,FAIL
#SBATCH --output test_openmp-job_%j.out
#SBATCH --error test_openmp-job_%j.err

# Change to my work dir
cd $SLURM_SUBMIT_DIR

# Bind your OpenMP threads
export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK
# Intel compiler specific environment variables
export KMP_AFFINITY=verbose,granularity=core,compact,1
export KMP_STACKSIZE=64m

## Load modules
module load my_module

# Start my application
srun ./my_openmp_app
```

The srun command in the script above sets up a parallel runtime environment to launch an application on multiple CPU cores, but on **one** node. For MPI jobs, you may want to use multiple CPU cores on **multiple** nodes. To achieve this, have a look at the following example of an MPI job:

Note: srun should be used in place of the "traditional" MPI launchers like mpirun or mpiexec.

Example of an MPI job

This example requests 10 compute nodes on the lena cluster with 16 cores each and 320 GiB of memory **in total** for a maximum duration of 2 hours. WARNING: this currently is a preview of things to come, Lena is still managed by PBS!

example_mpi_slurm.sh

```
#!/bin/bash -l
#SBATCH --job-name=test_mpi
#SBATCH --partition=lena
```

```
#SBATCH --nodes=10
#SBATCH --ntasks-per-node=16
#SBATCH --mem-per-cpu=2G
#SBATCH --time=02:00:00
#SBATCH --mail-user=user@uni-hannover.de
#SBATCH --mail-type=BEGIN,END,FAIL
#SBATCH --output test_mpi-job_%j.out
#SBATCH --error test_mpi-job_%j.err

# Change to my work dir
cd $SLURM_SUBMIT_DIR

# Load modules
module load foss/2018b

# Start my MPI application
#
# Note: if you use Intel MPI Library provided by modules up to intel/2020a, execute
srun as
#
# srun --mpi=pmi2 ./my_mpi_app
#
# For all Intel MPI libraries set the environment variable
I_MPI_PMI_LIBRARY=/usr/lib64/libpmi.so before executing srun

srun --cpu_bind=cores --distribution=block:cyclic ./my_mpi_app
```

As mentioned above, you should use the `srun` command instead of `mpirun` or `mpiexec` in order to launch your parallel application.

Within the same MPI job, you can use `srun` to start several parallel applications, each utilizing only a subset of the allocated resources. However, the preferred way is to use a Job Array (see section). The following example script will run 3 MPI applications simmultaneously, each using 64 tasks (4 nodes with 16 cores each), thus totalling to 192 tasks:

example_mpi_multi_srun_slurm.sh

```
#!/bin/bash -l
#SBATCH --job-name=test_mpi
#SBATCH --partition=lena
#SBATCH --nodes=12
#SBATCH --ntasks-per-node=16
#SBATCH --mem-per-cpu=2G
#SBATCH --time=00:02:00
#SBATCH --constraint=[skylake|haswell]
#SBATCH --mail-user=user@uni-hannover.de
#SBATCH --mail-type=BEGIN,END,FAIL
#SBATCH --output test_mpi-job_%j.out
#SBATCH --error test_mpi-job_%j.err

# Change to my work dir
cd $SLURM_SUBMIT_DIR

# Load modules
module load foss/2018b

# Start my MPI application
srun --cpu_bind=cores --distribution=block:cyclic -N 4 --ntasks-per-node=16 --exclusive
./my_mpi_app_1 &
srun --cpu_bind=cores --distribution=block:cyclic -N 4 --ntasks-per-node=16 --exclusive
./my_mpi_app_1 &
srun --cpu_bind=cores --distribution=block:cyclic -N 4 --ntasks-per-node=16 --exclusive
```

```
./my_mpi_app_2 &
wait
```

Note the `wait` command in the script; it results in the script waiting for all previously commands that were started with $&$ (execution in the background) to finish before the job can complete. We kindly ask to take care that the time necessary to complete each subjob is not too different in order not to waste too much valuable cpu time

Job arrays

Job arrays can be used to submit a number of jobs with the same resource requirements. However, some of these requirements are subject to changes after the job has been submitted. To create a job array, you need to specify the directive `#SBATCH –array` in your job script or use the option `–array` or `-a` on the `sbatch` command line. For example, the following script will create 12 jobs with array indices from 1 to 10, 15 and 18:

example_jobarray_slurm.sh

```
#!/bin/bash -l
#SBATCH --job-name=test_job_array
#SBATCH --ntasks=1
#SBATCH --mem-per-cpu=2G
#SBATCH --time=00:20:00
#SBATCH --mail-user=user@uni-hannover.de
#SBATCH --mail-type=BEGIN,END,FAIL
#SBATCH --array=1-10,15,18
#SBATCH --output test_array-job_%A_%a.out
#SBATCH --error test_array-job_%A_%a.err

# Change to my work dir
cd $SLURM_SUBMIT_DIR

# Load modules
module load my_module

# Start my app
srun ./my_app $SLURM_ARRAY_TASK_ID
```

Within a job script like in the example above, the job array indices can be accessed by the variable `$SLURM_ARRAY_TASK_ID`, whereas the variable `$SLURM_ARRAY_JOB_ID` refers the the job array's master job ID. If you need to limit (e.g. due to heavy I/O on the BIGWORK file system) the maximum number of simultaneously running jobs in a job array, use a % separator. For example, the directive `#SBATCH –array 1-50%5` will create 50 jobs, with only 5 jobs active at any given time.

Note: the maximum number of jobs in a job array is limited to 100.

SLURM environment variables

SLURM sets many variables in the environment of the running job on the allocated compute nodes. Table 7.4 shows commonly used environment variables that might be useful in your job scripts. For a complete list, see the "OUTPUT ENVIRONMENT VARIABLES" section in the sbatch man page.

SLURM environment variables

$SLURM_JOB_ID	Job id
$SLURM_JOB_NUM_NODE	Number of nodes assigned to the job
$SLURM_JOB_NODELIST	List of nodes assigned to the job
$SLURM_NTASKS	Number of tasks in the job
$SLURM_NTASKS_PER_CORE	Number of tasks per allocated CPU

$SLURM_NTASKS_PER_NODE	Number of tasks per assigned node
$SLURM_CPUS_PER_TASK	Number of CPUs per task
$SLURM_CPUS_ON_NODE	Number of CPUs per assigned node
$SLURM_SUBMIT_DIR	Directory the job was submitted from
$SLURM_ARRAY_JOB_ID	Job id for the array
$SLURM_ARRAY_TASK_ID	Job array index value
$SLURM_ARRAY_TASK_COUNT	Number of jobs in a job array
$SLURM_GPUS	Number of GPUs requested

Creating a "classic" node list / node file for certain programs

In case you used the variable $PBS_NODEFILE up to now, you will note that SLURM does things in a slightly different way. Whereas $PBS_NODEFILE points to the name of a file containing the list of nodes allocated to your job line by line, $SLURM_JOB_NODELIST contains a pattern matching the allocated nodes, and not all software can handle that. So in case you are looking for a way to get the traditional nodelist, the following example shows how to do the trick:

Within the job script, insert a line like e.g.

```
NODEFILE=$(scontrol show hostnames | tr '[:space:]' ',')
```

Sample call for ANSYS/Fluent:

```
fluent [...] -cnf=${NODEFILE}
```

GPU jobs on the cluster

The LUIS cluster has a number of nodes that are equipped with NVIDIA Tesla GPU Cards.

Currently, 4 Dell nodes containing 2 NVIDIA Tesla V100 each are available for general use in the partition gpu.

Use the following command to display the actual status of all nodes in the gpu partition and the computing resources they provide, including type and number of installed GPUs:

```
$ sinfo -p gpu -NO nodelist:15,memory:8,disk:10,cpusstate:15,gres:20,gresused:20
 NODELIST      MEMORY  TMP_DISK  CPUS(A/I/O/T)  GRES                GRES_USED
 euklid-n001   128000  291840    32/8/0/40      gpu:v100:2(S:0-1)   gpu:v100:2(IDX:0-1)
 euklid-n002   128000  291840    16/24/0/40     gpu:v100:2(S:0-1)   gpu:v100:1(IDX:0)
 euklid-n003   128000  291840    0/40/0/40      gpu:v100:2(S:0-1)   gpu:v100:0(IDX:N/A)
 euklid-n004   128000  291840    0/40/0/40      gpu:v100:2(S:0-1)   gpu:v100:0(IDX:N/A)
```

To ask for a GPU resource, you need to add the directive `#SBATCH –gres=gpu:<type>:n` to your job script or on the command line, respectively. Here, "n" is the number of GPUs you want to request. The type of requested GPU (<type>) can be skipped. The following job script requests 2 Tesla V100 GPUs, 8 CPUs in the gpu partition and 30 minutes of wall time:

example_gpu_slurm.sh

```
#!/bin/bash -l
#SBATCH --job-name=test_gpu
#SBATCH --partition=gpu
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=8
#SBATCH --gres=gpu:v100:2
#SBATCH --mem-per-cpu=2G
#SBATCH --time=00:30:00
#SBATCH --mail-user=user@uni-hannover.de
#SBATCH --mail-type=BEGIN,END,FAIL
#SBATCH --output test_gpu-job_%j.out
#SBATCH --error test_gpu-job_%j.err
```

```
# Change to my work dir
cd $SLURM_SUBMIT_DIR

# Load modules
module load fosscuda/2018b

# Run GPU application
srun ./my_gpu_app
```

When submitting a job to the gpu partition, you **must** specify the number of GPUs. Otherwise, your job will be rejected at the submission time.

Note: on the Tesla V100 nodes, you may currently only request up to 20 CPU cores for each requested GPU.

Job status and control commands

This section provides an overview of commonly used SLURM commands that allow you to monitor and manage the status of your batch jobs.

Query commands

The status of your jobs in the queue can be queried using

```
$ squeue
```

or – if you have array jobs and want to display one job array element per line –

```
$ squeue -a
```

Note that the symbol $ in the above commands and all other commands below represents the shell prompt. The $ is NOT part of the specified command, do NOT type it yourself.

The `squeue` output should look more or less like the following:

```
$ squeue
  JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
    412       gpu     test username PD       0:00      1 (Resources)
    420       gpu     test username PD       0:00      1 (Priority)
    422       gpu     test username  R      17:45      1 euklid-n001
    431       gpu     test username  R      11:45      1 euklid-n004
    433       gpu     test username  R      12:45      1 euklid-n003
    434       gpu     test username  R       1:08      1 euklid-n002
    436       gpu     test username  R      16:45      1 euklid-n002
```

ST shows the status of your job. `JOBID` is the number the system uses to keep track of your job. `NODELIST` shows the nodes allocated to the job, `NODES` the number of nodes requested and – for jobs in the pending state (PD) – a `REASON`. `TIME` shows the time used by the job. Typical job states are `PENDING(PD)`, `RUNNING(R)`, `COMPLETING(CG)`, `CANCELLED(CA)`, `FAILED(F)` and `SUSPENDED(S)`. For a complete list, see the “JOB STATE CODES” section in the squeue man page.

You can change the default output format and display other job specifications using the option `–format` or `-o`. For example, if you want to additionally view the number of CPUs and the walltime requested:

```
$ squeue --format="%.7i %.9P %.5D %.5C %.2t %.19S %.8M %.10l %R"
  JOBID PARTITION NODES  CPUS TRES_PER_NODE ST MIN_MEMORY     TIME TIME_LIMIT
NODELIST(REASON)
    489       gpu     1    32         gpu:2 PD         2G     0:00      20:00 (Resources)
    488       gpu     1     8         gpu:1 PD         2G     0:00      20:00 (Priority)
    484       gpu     1    40         gpu:2  R          1G    16:45      20:00 euklid-n001
    487       gpu     1    32         gpu:2  R          2G    11:09      20:00 euklid-n004
    486       gpu     1    32         gpu:2  R          2G    12:01      20:00 euklid-n003
    485       gpu     1    16         gpu:2  R          1G    16:06      20:00 euklid-n002
```

Note that you can make the `squeue` output format permanent by assigning the format string to the environment variable `SQUEUE_FORMAT` in your `$HOME/.bashrc` file:

```
$ echo 'export SQUEUE_FORMAT="%.7i %.9P %.5D %.5C %.13b %.2t %.19S %.8M %.10l %R"'>>
~/.bashrc
```

The option `%.13b` in the variable assignment for `SQUEUE_FORMAT` above displays the column `TRES_PER_NODE` in the squeue output, which provides the number of GPUs requested by each job.

The following command displays all job steps (processes started using `srun`):

```
squeue -s
```

To display estimated start times and compute nodes to be allocated for your pending jobs, type

```
$ squeue --start
 JOBID PARTITION NAME     USER ST          START_TIME  NODES SCHEDNODES   NODELIST(REASON)
   489       gpu test username PD 2020-03-20T11:50:09      1 euklid-n001  (Resources)
   488       gpu test username PD 2020-03-20T11:50:48      1 euklid-n002  (Priority)
```

A job may be waiting for execution in the pending state for a number of reasons. If there are multiple reasons for the job to remain pending, only one is displayed.

- **Priority** - the job has not yet gained a high enough priority in the queue
- **Resources** - the job has sufficient priority in the queue, but is waiting for resources to become available
- **JobHeldUser** - job held by user
- **Dependency** - job is waiting for another job to complete
- **PartitionDown** - the queue is currently closed for new jobs

For the complete list, refer to the `squeue` man page the section “JOB REASON CODES”.

If you want to view more detailed information about each job, use

```
$ scontrol -d show job
```

If you are interested in the detailed status of one specific job, use

```
$ scontrol -d show job <job-id>
```

Replace <`job-id`> by the ID of your job.

Note that the command `scontrol show job` will display the status of jobs for up to 5 minutes after their completion. For batch jobs that finished more than 5 minutes ago, you need to use the `sacct` command to retrieve their status information from the SLURM database (see section).

The `sstat` command provides real-time status information (e.g. CPU time, Virtual Memory (VM) usage, Resident Set Size (RSS), Disk I/O, etc.) for running jobs:

```
# show all status fields
sstat -j <job-id>

# show selected status fields
sstat --format=AveCPU,AvePages,AveRSS,AveVMSize,JobID -j <job-id>
```

Note: the above commands only display your own jobs in the SLURM job queue.

Job control commands

The following command cancels a job with ID number `<job-id>`:

```
$ scancel <job-id>
```

Remove all of your jobs from the queue at once using

```
$ scancel -u $USER
```

If you want to cancel only array ID `<array_id>` of job array `<job_id>`:

```
$ scancel <job_id>_<array_id>
```

If only job array ID is specified in the above command, then all job array elements will be canceled.

The commands above first send a `SIGTERM` signal, then wait 30 seconds, and if processes from the job continue to run, issue a `SIGKILL` signal.

The `-s` option allows you to issue any signal to a running job which means you can directly communicate with the job from the command line, provided that it has been prepared for this:

```
$ scancel -s <signal> <job-id>
```

A job in the pending state can be held (prevented from being scheduled) using

```
$ scontrol hold <job-id>
```

To release a previously held job, type

```
$ scontrol release <job-id>
```

After submitting a batch job and while the job is still in the pending state, many of its specifications can be changed. Typical fields that can be modified include job size (amount of memory, number of nodes, cores, tasks and GPUs), partition, dependencies and wall clock limit. Here are a few examples:

```
# modify time limit
scontrol update JobId=279 TimeLimit=12:0:0

# change number of tasks
scontrol update jobid=279 NumTasks=80

# change node number
scontrol update JobId=279 NumNodes=2

# change the number of GPUs per node
scontrol update JobId=279 Gres=gpus:2

# change memory per allocated CPU
scontrol update Jobid=279 MinMemoryCPU=4G

# change the number of simultaneously running jobs of array job 280
scontrol update ArrayTaskThrottle=8 JobId=280
```

For a complete list of job specifications that can be modified, see section "SPECIFICATIONS FOR UPDATE COMMAND, JOBS" in the `scontrol` man page.

Job accounting commands

The command `sacct` displays the accounting data for active and completed jobs which is stored in the SLURM database.

Here are a few usage examples:

```
#  list IDs of all your jobs since January 2019
sacct -S 2019-01-01 -o jobid

# show brief accounting data of the job with <job-id>
sacct -j <job-id>

# display all job accountig fields
sacct -j <job-id> -o ALL
```

The complete list of job accounting fields can be found in section “Job Accounting Fields” in the `sacct` man page. You could also use the command `sacct –helpformat`