SLURM usage guide

If you are completely new to scientific computing, read this:

Please keep in mind that simply throwing a serial program without provisions for parallelization at the cluster will not make it run faster. In fact, it will often run SLOWER, since cpu cores on machines specialized in scientific computing usually run at lower clock frequencies than workstation cpus.

There are many commercial and open source software packages that are already very well parallelized, **but you'll definitely need to know how to use the parallel capabilities of your software or programming language**. Requesting lots of nodes, cpu cores, memory and time for a program that will use only one single cpu will only keep you and all other users waiting.

So **start small and simple**. There's no use waiting for a 10-node-job just to find out it immeadiately crashes, so test with a 1-node-1-task-job that requests only 10 minutes first, then moderately make it bigger and check that e.g. your parallel program delivers reasonable results. When you are sure you understand what you do and when your test case works reliably, when you see a decrease in run time when you add ressources — then go for it.

Why use a cluster at all?

The reason you want to use the cluster is probably the computing resources it provides. With about several hundred people using the compute cluster 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", in the LUH-cluster: "SLURM"). The batch system sorts the ressource requests ("batch jobs") it gets from the users according to ressource availability and priority rules. When the priority of a job is sufficiently high to start, it gets scheduled on the requested ressources (usually some compute node(s)) and starts. Requests to the batch system are usually made by submitting a text file containing (bash-) instructions about what to do (the "batch job"), or by requesting an "interactive" batch job from the command line that puts you directly into a command shell on a compute node (or a set of compute nodes) to work with, or, as a third option, by the Open OnDemand portal ("OOD") that is available on https://login.cluster.uni-hannover.de which translates the settings you enter via the web browser gui into a text file to again submit as a batch job to the batch system.

Parallelization Basics

Batch jobs can roughly be divided into several categories:

- serial (also: single-threaded, single-task) batch jobs; they run just like normal programs/scripts. Usually, no changes/adaptions are needed. The time needed to complete such a job almost entirely depends on the speed of a single cpu core, and no scaling is achieved. As mentioned above, starting a serial program on a larger machine will NOT make it run faster, and since the compute servers in the cluster usually combine lots of cpu cores on each cpu socket, they actually may run even slower than cores on a smaller workstation that has fewer cores, due to the total heat generated by each chip that needs to be dissipated. So think about the size and characteristics of your workload. In the "real world", smaller loads over short distances are better transported using a fast car, while large ones that travel around the world need a big container ship. Similar considerations are valid for workloads on computers, in particular, if the software is not parallelized.
- parallel jobs; these in turn can be distinguished by the kind of problems they treat and how they attempt to achieve scaling ("reaching results quicker"). Problem are either "trivially parallel computations" or not:
  - trivially parallel problems can be solved by simply starting as many tasks as needed / as possible, and each task will run happily minding its own business, until it achieves a partial result in the end, which is then combined with all the other results of the other tasks to get the result of the whole job. Luckily, many problems can be computed in this way.
  - non-trivial problems are those that usually need to exchange data *during* computation, for example when finishing a simulation time step to update the (so-called) "ghost"-borders of all the simulation subdomains the complete simulation region has been decomposed into.

- The other main distinction of parallel jobs is *how* they do it. The two main variants here are:
  - OpenMP jobs (shared-memory-processing, SMP, single-node, multi-threaded, multi-processing, can typically scale to the larges compute nodes available) usually run on ONE node only, using multiple threads or processes, but sharing memory. So the software needs some logic that specifies which parts of the program (e.g. loops) should run in parallel, but synchronization between threads is automatic. These programs typically are linked to an OpenMP library during compilation, so the node itself usually needs to have some libraries installed, but parallelization is relatively easy and low-effort. The software must ensure that only one process updates a specific memory location at the same time, but it can rely on having the same memory contents. Beware, though, that while many application programs are parallelized this way, and nowadays compute nodes may contain many cpu cores, scaling still may be quite limited depending on the specific kind of problem and how much effort has been spent to parallelize the regions of the software that should do parallel computations. So, while some software packages achieve very good scaling up to over 100 cpu cores on big servers, others already hit their limits when using more than 4 or 8 cpus. There's many hardware-factors limiting performance, too, like the number of cache levels, CPU cache sizes, memory bandwidth, NUMA architecture etc, and of course I/O.
  - MPI (message-passing-interface, possible multi-node, multi-processing, can scale up to millions of cpus); software parallelized using MPI is usually able to achieve the highest scaling, but the cost is that the software must explicitly specify which parts of the program run in parallel and how data / results are updated, which task does what, and

how the simulation is kept in sync. So the scaling here also depends on the genius of the person writing the software and their knowledge about specific hardware features. Each MPI-task (called a “rank”) is highly independent of the others, and it needs to explicitly communicate its results to the other tasks whenever there's a need for that, since each task uses their own memory that only they can access.

Hint: to avoid an easy understanding of complicated things, someone thought it would be a good idea to name one of the several MPI libraries available “OpenMPI”. Do not fall for this trap. OpenMPI is one specific implementation of the MPI programming interface (there are many others called IntelMPI, MVAPICH, IBMMPI, …) that uses message-passing between independent tasks to achieve a high degree of parallelization. OpenMP, on the other hand, is a general term for a completely different programming interface that is using compiler-directives and which has NOT much to do with MPI. So OpenMP is NOT MPI, while OpenMPI is MPI, but NOT OpenMP.

The SLURM Workload Manager

The software that decides which job to run when and where in the cluster is called SLURM. 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 information 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	48 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 when still able to maintain a fair balance between all users' needs, we may sometimes also consider requests for a higher priority for a short time, which may be submitted to cluster-help@luis.uni-hannover.de. You should include an explanation for what period of time you need which kind of priority, and of course why we should consider your request regarding the fact that usually all other users want priority, too.

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
```

The `clusterinfo` command (Python script) retrieves real-time information about node and partition configurations, resource (CPU/GPU) usage, and user access rights to resources through native SLURM commands and displays the data in a structured format for easier interpretation. It shows which nodes are accessible by all users and which are reserved for specific research groups with exclusive access during configured times (see Forschungscluster-Housing). By executing `clusterinfo -l`, your configured SLURM limits (such as the maximum number of running and pending jobs, maximum wall clock time, etc.) will also be displayed. For a list of available options and their descriptions, run `clusterinfo -h`.

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.

Note: In case you get an error message like `srun: Warning: can't honor --ntasks-per-node set to X which doesn't match the requested tasks YY with the number of requested nodes ZZ. Ignoring`, check (using `set | grep SLURM_N` within the job shell, for example) that your request has been honored despite the message, and then ignore the message. 🙂

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

```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 $SHELL -l
 srun: slurm_job_submit: set partition of submitted job to amo,tnt,gih
[user@amo-n004 ~]$
```

At this point, we would like to note that SLURM differentiates between `--ntasks`, which may roughly be translated into the number of (independent) MPI-ranks or instances of a job, and `--cores-per-task`, which may translate into the number of (OpenMP) threads. MPI-jobs usually request `--ntasks` larger than one, while OpenMP-jobs may request `--ntasks=1` and `--cores-per-task` higher than one.

If you want to run your jobs on nodes with a specific CPU type, you can request them using the SLURM option `--constraint=CPU_ARCH:<cpu_arch>`, where `<cpu_arch>` can currently have the following values: `sse`, `avx`, `avx2`, and `avx512`.

To check the available CPU architectures for different SLURM partitions and nodes, you can use the command `clusterinfo -n -i`.

If your job can run on nodes with any of multiple CPU types, you can specify them using the following syntax: `--constraint=[CPU_ARCH:<cpu_arch1>,CPU_ARCH:<cpu_arch1>,…].`

Submitting a batch script

A SLURM job submission file for your job (a “batch script”) is a shell script with a set of additional directives that are only interpreted by the batch system (Slurm) at the beginning of the file. These directives are marked by starting the line with the string #SBATCH, so the batch system knows that the following parameters and commands are not just a comment (which the # character otherwise would imply). The shell (the command line interpreter of Unix) usually ignores everything that follows a # character. But at the beginning of your file, the Slurm commands used to submit a batch script will also check whether the # character is immediately followed by SBATCH. If that is the case, the batch system will interpret the following characters as directives. Processing of these directives stops once the first non-comment non-whitespace line has been reached in the script. The very first line of your script usually should read `#!/bin/bash` - ask Wikipedia for the meaning of “Shebang (Unix)” in case you want to understand what this is for.

Valid directives can be found using the command `man sbatch`. In principle, you may write almost any option that you could feed to sbatch at the command line as a #SBATCH-line in your script.

A suitable batch script is usually 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=[CPU_ARCH:avx512|CPU_ARCH:avx2]
```

```
#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
# SLURM_SUBMIT_DIR is an environment variable that automatically gets
# assigned the directory from which you did submit the job. A batch job
# is like a new login, so you'll initially be in your HOME directory.
# So it's usually a good idea to first change into the directory you
did
# submit your job from.
cd $SLURM_SUBMIT_DIR


# Load the modules you need, see corresponding page in the cluster
documentation
module load my_modules

# Start my serial app
# srun is needed here only to create an entry in the accounting system,
# but you could also start your app without it here, since it's only
serial.
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 oversteps the resource limits that you have defined in your #SBATCH directives, the job will automatically be killed by the SLURM server. This is particularly the case when you try to use more memory than you allocated, which results in an OOM (out-of-memory) -event.

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
`--tasks=<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

Options	Default Value	Description
`--mem=<mem>`	partition dependent	memory per node in MB
`--gres=gpu:<type>:<N>`	-	Request nodes with GPUs; <type> may be omitted (thus: `--gres=gpu:<N>`)
`--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; to find out the features assigned to a specific node, use e.g. `scontrol show nodes <nodename>`
`--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 configured in SLURM for each node allocated.

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=[CPU_ARCH:avx512|CPU_ARCH:avx2]
#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.

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=[CPU_ARCH:avx512|CPU_ARCH:avx2]
#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 300. The index number must be smaller than 1 million.

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

GPU jobs on the cluster

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

Currently, 4 nodes containing 2 NVIDIA Tesla V100 and 3 nodes containing 4 A100 cards (each) are available for general use in the partition `gpu` (named `euklid-n00x`). So the ressources regularly available to all users are still relatively scarce, and respecting the tip in this page's introduction to start small to first test your jobs really is important — or you will possibly just wait for a long time, only to see your job instantly crashing. So the more pressure you have, the more thorough you should test your job first.

There's also some additional ressources available that you might have a fair chance to use. Several institutes have entrusted us with running their nodes in the so-called FCH service (“Forschungscluster-Housing”, consult the LUIS-website for details). These nodes are usually reserved during daytime Mo-Fr 08:00-20:00 for the respective institute, but during the night and on weekends, they participate in the common queue. Whether you can run a job there will mainly depend on the respective institute's own usage, and of course your job has to request a walltime shorter than 12 hours during the week to squeeze in. To try your luck on FCH nodes, omit the `--partition=gpu` request.

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

```
sinfo --partition gpu -Node --
Format="nodelist:15,memory:8,disk:10,cpusstate:15,gres:30,gresused:40"
NODELIST       MEMORY  TMP_DISK  CPUS(A/I/O/T)  GRES
GRES_USED
euklid-n001    125000  291840    2/38/0/40      gpu:v100:2(S:0-1)
gpu:v100:2(IDX:0-1)
euklid-n002    125000  291840    2/38/0/40      gpu:v100:2(S:0-1)
gpu:v100:2(IDX:0-1)
euklid-n003    125000  291840    2/38/0/40      gpu:v100:2(S:0-1)
gpu:v100:2(IDX:0-1)
euklid-n004    125000  291840    2/38/0/40      gpu:v100:2(S:0-1)
gpu:v100:2(IDX:0-1)
euklid-n005    1025000 3600000   4/44/0/48      gpu:a100m40:4(S:0-1)
gpu:a100m40:4(IDX:0-3)
euklid-n006    1025000 3600000   4/44/0/48      gpu:a100m40:4(S:0-1)
gpu:a100m40:4(IDX:0-3)
euklid-n007    1025000 3600000   4/44/0/48      gpu:a100m40:4(S:0-1)
gpu:a100m40:4(IDX:0-3)
```

To inquire about *all* nodes that have at least one gpu, including those reserved during daytime for FCH, use

```
sinfo --Node --
Format="nodelist:15,memory:8,disk:10,cpusstate:15,gres:30,gresused:30" |
grep -v null
```

To ask for GPU resources, you need to add the directive `#SBATCH --gres=gpu:<type>:n` to your job script, or on the command line, respectively, “n” being the number of GPUs requested. The type of GPU can be omitted. Thus, `#SBATCH --gres=gpu:n` will give you a wider selection of potential nodes to run the job. 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.

Note: the maximum runtime for jobs in the gpu partition is limited to 48 hours.

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 `sacct` command is primarily designed to display job data from the SLURM accounting database, specifically for jobs that have exited the queue (e.g., completed, failed, or canceled). If a job is still running, tools like `sstat` or `squeue` might provide more current metrics. 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 accounting 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`

Analyzing Job Efficiency

Monitoring job efficiency helps reduce queue waiting times by identifying resource allocation mismatches (e.g., over-requesting CPUs or memory). Efficient resource utilization not only ensures faster job completion but also increases overall system throughput, enabling more users to benefit from the cluster.

Seff

`seff` is a command-line tool used to display resource utilization efficiency for completed jobs.

Note: The job must be completed; `seff` does not work for running jobs.

Syntax

```
seff <job_id>
```

Sample Output

```
Job ID: 12345
Cluster: luis
User/Group: user/group
State: COMPLETED
Nodes: 1
Cores: 8
CPU Utilized: 02:00:00
CPU Efficiency: 25% of 08:00:00 core-walltime
Memory Utilized: 4 GB
Memory Efficiency: 50% of 8 GB requested
```

- **CPU Efficiency**: Calculated as the ratio of CPU time used to the total CPU time allocated (cores × walltime). Low efficiency indicates under-utilized cores.
- **Memory Efficiency**: Indicates how much of the requested memory was actually consumed. Over-requesting memory can lead to wasted resources.

Reportseff

The `reportseff` command is a tool available on the cluster to help users analyze the efficiency of their Slurm jobs. While `seff` focuses on a single job, allowing you to evaluate resource utilization per job ID, reportseff provides broader capabilities, including:

- Analyzing jobs over a specified time period (e.g., --`since` and --`until` options)
- Filtering jobs based on partition, job state, or multiple job IDs simultaneously
- Generating efficiency details for a single or all array job elements

The tool reads accounting data via `sacct` and is particularly helpful for identifying how effectively resources are being used, enabling users to adjust job submissions for optimal performance.

This example generates a report for jobs completed in the last 3 days, including additional fields for the requested time limit (`timelimit`), CPUs (`reqcpus`), and memory (`reqmem`)

```
reportseff --since now-3days --until now --state COMPLETED --format
+timelimit,reqcpus,reqmem

JobID              State         Elapsed  TimeEff   CPUEff   MemEff
Timelimit    ReqCPUS   ReqMem
4043949          COMPLETED    6-01:43:06   97.1%    23.0%    15.3%
6-06:00:00     64        512G
4048121          COMPLETED    5-00:21:51   60.2%    20.8%    100.0%
8-08:00:00     48        128G
4056804          COMPLETED    4-21:07:01   61.0%    87.7%    15.0%
8-00:00:00     12         48G
4059203          COMPLETED    5-05:39:14   65.4%    13.6%     9.2%
8-00:00:00      8        128G
4059230          COMPLETED    4-19:06:34   60.0%     8.9%    58.3%
8-00:00:00      8        128G
4059298          COMPLETED    5-07:27:45   66.4%    13.2%    25.3%
8-00:00:00      8        128G
4059303          COMPLETED    4-16:25:54   58.6%    13.0%    23.2%
8-00:00:00      8        128G
4059317          COMPLETED    4-23:42:34   62.3%    12.9%    20.9%
8-00:00:00      8        128G
4066049          COMPLETED    3-10:06:40   85.5%    18.4%     3.0%
4-00:00:00     16         64G
4067800          COMPLETED    3-05:45:30   46.3%    99.5%    12.0%
7-00:00:00     36        108G
...
```

The tool is pre-installed and ready for use on the cluster. For further information, refer to the reportseff GitHub page.

How the scheduler works

The scheduler has to consider many constraints, rules, priorities and limits until a particular job gets scheduled. The definitive guide can be found on SLURM's website. Some of the factors to consider when you want to ask “why does my job not run?” are:

→ First, there's priorities. Compare your job's priority against that of others (e.g. with a command like `squeue –states=pending,configuring –sort=-p,-i –priority –Format=“JobID:11`

,UserName:9 ,StateCompact:3 ,NumNodes:.3 ,NumCPUs:.4 ,MinCPUs:.5 ,MinMemory:.5 ,TimeLimit:.11 ,SubmitTime:.20 ,StartTime:.20 ,PriorityLong:8 ,TRES-per-node:20 ,Partition:15”). If other jobs have higher priority, they will get considered first. Only if jobs with higher priority have resource requests that currently can not be fulfilled, the next jobs in order will get considered. Jobs that do not yet have priority, but could instantly run on currently free resources and finish before any prioritized job could use them, may get run instantly via a mechanism called “backfill”. The important thing is that they will ONLY run if they block no priority job in ANY of their respective dimensions, like walltime, cpu count, memory, …

→ Next, check whether your job may have resource requests that are difficult to fulfill. Compare the resources listed in our Available Hardware Table and ask on which partitions your job could run at all. If you want to configure your jobs to use a particular partition, e.g. because you know your software can use cpu-specific features like AVX-512, try the commands scontrol show partition <partitionname> and scontrol show node <nodename-n001> to show which resources every node in the cluster exactly provides.

→ If you are in doubt whether any of your jobs would run, you could try to submit a very small and short (!) job (ex. 1 cpu core, 2 GB mem, 10 minutes, like salloc –nodes=1 –ntasks=1 –mem=2GB –time=10:00). Even with such a job, the “problem” could just be that the ressources of the cluster are not infinitely large. A future reservation for somebody else may prevent your job from starting immediately. Or, in case some nodes have not been used for a while, they will have been powered-down automatically to save energy. That means that if you just see a message “queued and waiting for resources”, it may also mean that in the background, a node has begun to boot just for your job, and in about 10 minutes time, your job and any of the same kind you would care to submit afterwards will start almost instantly.

→ We take only moderate influence on how many jobs our users may submit. The scheduler will try to at least run at least one job of every user before turning to the next job of the same user, provided the resources for other jobs are available. In case the cluster is really empty, up to 64 jobs of one user will run at the same time, and if they all request 200 hours of wall time, that may in extreme cases mean that others will wait for a correspondingly long time until new resources become free.

Generally speaking, the cluster is a shared tool for hundreds of users. We often get requests (“the resources are free, why does my job not start?”) that show a lack of awareness that at any instant, there possibly are many users on the system waiting for many jobs' execution.

So the conclusion is: just because something matching the requirements of your job currently appears to be free, this does NOT automatically mean that the scheduler will pick or even consider exactly *your* job as the next to run. Scheduling is a little bit like a more complicated version of stock trading, in case the comparison helps someone…

From:
https://docs.cluster.uni-hannover.de/ - **Cluster Docs**

Permanent link:
https://docs.cluster.uni-hannover.de/doku.php/guide/slurm_usage_guide

Last update: **2025/07/24 09:44**