UG2.6.1: How to submit the job - Batch Scheduler SLURM

In this page:

SLURM Workload Manager (or simply SLURM, which stands for "Simple Linux Utility for Resource Management") is an open source and highly scalable job scheduling system.

SLURM has three key functions. Firstly, it allocates exclusive and/or non-exclusive access to resources (compute nodes) to users for some duration of time, so they can perform their work. Secondly, it provides a framework for starting, executing, and monitoring work (normally a parallel job) on the set of allocated nodes. Finally, it arbitrates contention for resources by managing the queue of pending jobs.

Important: the node you are logged in is a login node and **cannot be used** to execute parallel programs. Any command in the login nodes is limited up to 10 minutes. For longer runs, you need to use SLURM scheduler in **"batch"** mode or "**interactive**" mode.

Currently, SLURM is the scheduling system of of MARCONI, MARCONI100 and GALILEO100. Comprehensive documentation is on this portal, as well as on the original SchedMD site.

Running applications using SLURM

With SLURM you can specify the tasks that you want to be executed; the system takes care of running these tasks and returns the results to the user. If the resources are not available, then SLURM holds your jobs and runs them when they will become available.

With SLURM you normally **create a batch job** which you **submit to the scheduler**. A batch job is a file (a shell script under UNIX) containing the set of commands that you want to run. It also contains the directives that specify the characteristics (attributes) of the job and the resource requirements (e.g. number of processors and CPU time) that your job needs.

Once you create your job, you can reuse it if you wish. Or, you can modify it for subsequent runs.

For example, here is a simple SLURM job script to run a user's application by setting a limit (one hour) to the maximum wall time, requesting 1 node with 32 cores:

```
#!/bin/bash

#SBATCH --nodes=1                    # 1 node
#SBATCH --ntasks-per-node=32         # 32 tasks per node
#SBATCH --time=1:00:00               # time limits: 1 hour
#SBATCH --error=myJob.err            # standard error file
#SBATCH --output=myJob.out           # standard output file
#SBATCH --account=<account_no>       # account name
#SBATCH --partition=<partition_name> # partition name
#SBATCH --qos=<qos_name>             # quality of service

./my_application
```

Check for the maximum value allowed for ntasks per node for the cluster you are using (i.e. the number of cores per node, hyperthreading included when enabled). More in general, SLURM has been configured differently on the various systems reflecting the different system features. Please refer to the system specific guidesfor more detailed information.

Basic SLURM commands

The main user's commands of SLURM are reported in the table below: please consult the man pages for more information.

sbatch, srun, salloc	Submit a job

squeue	Lists jobs in the queue
sinfo	Prints queue information about nodes and partitions
sbatch <batch script>	Submits a batch script to the queue
scancel <jobid>	Cancel a job from the queue
scontrol hold <jobid>	Puts a job on hold in the queue.
scontrol release	Releases a job from hold
scontrol update	Change attributes of submitted job.
scontrol requeue	Requeue a running, suspended, or finished Slurm batch job into pending state.
scontrol show job <jobid>	Produce a very detailed report for the job.
sacct -k, --timelimit-min	Only send data about jobs with this time limit.
sacct -A account_list	Display jobs when a comma separated list of accounts are given as the argument.
sstat	Display information about CPU, Task, Node, Resident Set Size, and Virtual Memory
sshare	Display information about shared for a user, a repo, a job, a partition, etc.
sprio	Display information about a job's scheduling priority from multi-factor priority components.

Submit a job:

```
> sbatch [opts] job_script
> salloc [opts] <command>  (interactive job)
```

```
where:

[opts] --> --nodes=<nodes_no> --ntasks-per-node=<tasks_per_node_no> --account=<account_no> --partition=<name> ...
```

job_script is a SLURM batch job.

The second command is related to a so-called **"Interactive job"**: with salloc the user allocates a set of resources (nodes, cores, etc). The job is queued and scheduled as any SLURM batch job, but when executed with srun, the standard input, output, and error streams of the job are connected to the terminal session in which salloc is running. When the job begins its execution, all the input to the job is taken from the terminal session. You can use CTRL-D or "exit" to close the session.
If you specify a command at the end of your salloc string (like "./myscript"), the job will simply execute the command and close, prompting the standard output and error directly on your working terminal.

```
salloc -N 1 --ntask-per-node 8 # here I'm asking for a compute node with 1 GPU and 8 core
squeue -u $USER  # can be used to check remote allocation is ready
hostname # will run on the front-end NOT ON ALLOCATED RESOURCES
srun hostname # will run on allocated resources showing the name of remote compute node
exit # ends the salloc allocation
```

WARNING: interactive jobs with SLURM are quite delicate. With salloc, your prompt won't tell you that you are working on a compute node, so it can be easy to forget that there is an interactive job running. Furthermore, deleting the job with "scancel" while inside the job itself will not boot you out of the nodes, and will invalid your interactive session because every command is searching for a jobid that doesn't exist anymore. If you are stuck in this situation, you can always revert back to your original front-end session with "CTRL-D" or "exit".

WARNING: interactive jobs may also be created launching the command

```
srun -N 1 --ntask-per-node 8 ... --pty /bin/bash
```

but be careful because in this case SLURM will allocate all the requested resources to the interactive step job. Therefore any srun comman launched inside the interactive job will be stuck due to the absence of available resources. We suggest you to use salloc to create interactive jobs. As alternative you can use the flag --overlap on the srun commands inside the interactive job allowing all the steps to share resources each other. See also our FAQ.

Displaying Job Status:

```
> squeue                          (lists all jobs, default format)
> squeue --format=...             (lists all jobs, more readable format)
> squeue -u $USER                 (lists only jobs submitted by you)
> squeue --job <job_id>           (only the specified job)
> squeue --job <job_id> -l        (full display of the specified job)
> scontrol show job <job_id>      (detailed informations about your job)
```

Displaying Queue Status:

The command sinfo displays information about nodes and partitions (queues).

It offers several options - here is a template that you may find useful.

```
> sinfo -o "%20P %10a %10l %15F %10z"
```

Display a straight-forward summary: available partitions, their status, timelimit, node information with A/I/O/T (allocated, idle, other, total) and specifications S:C:T (sockets:cores:threads)
Numbers represent field length and should be used to properly accommodate the data.

Other useful options are:

```
> sinfo
> sinfo -p <partition>            (Long format of the specified partition, eg gll_usr_prod)
> sinfo -d                        (Information about the offline nodes. The list of available partition is also
easier to read)
> sinfo --all                     (Displays more details)
> sinfo -i <n>                    (Top-like display, iterates every "n" seconds)
> sinfo -l or --long              (Displays several additional information, such as the reason why specific
nodes are down/drained. Usually used together with -N)
> sinfo -n <node>                 (Shows information about a specific node, eg sinfo -N -n r033c01s01)
```

To view a complete list of all options and their descriptions, use man sinfo, or access the SchedMD webpage.

Delete a job:

```
> scancel <jobID>
```

More information about these commands is available with the **man** command.

The User Environment

There are a number of environment variables provided to the SLURM job. Some of them are taken from the user's environment and carried with the job. Others are created by SLURM.

All SLURM-provided environment variable names start with the characters **SLURM**_.

Below are listed some of the more useful variables, and some typical values taken as an example:

```
SLURM_JOB_NAME=job
SLURM_NNODES (or SLURM_JOB_NUM_NODES)=2
SLURM_JOBID (or SLURM_JOB_ID)=453919
SLURM_JOB_NODELIST=node1,node2,...
SLURM_SUBMIT_DIR=/marconi_scratch/userexternal/username
SLURM_SUBMIT_HOST=node1
SLURM_CLUSTERNAME=cluster1
SLURM_JOB_PARTITION=partition1
```

There are a number of ways that you can use these environment variables to make more efficient use of SLURM. For example, **SLURM_JOB_NAME** can be used to retrieve the SLURM jobname. Another commonly used variable is **SLURM_SUBMIT_DIR** which contains the name of the directory from which the user submitted the SLURM job.

WARNING: $SLURM_JOB_NODELIST will display the node names in contracted forms, meaning that for consecutive nodes you will get their range instead of the full list. You will see in square brackets the ID of the first and the last node of the chunk, meaning that all the nodes between them are also part of the actual node list.

Job TMPDIR:

When a job starts, a temporary area is defined on the storage local to each compute node:

TMPDIR=/scratch_local/slurm_job.$SLURM_JOB_ID

which can be used exclusively by the job's owner. During your jobs, you can access the area with the (local) variable $TMPDIR. The directory is removed at the end of the job, hence remember to save the data stored in such area to a permanent directory. Please note that the area is located on local disks, so it can be accessed only by the processes running on the node. For multinode jobs, if you need all the processes to access some data, please use the shared filesystems $HOME, $WORK, $CINECA_SCRATCH.

SLURM Resources

A job requests resources through the SLURM syntax; SLURM matches requested resources with the available ones, according to the rules defined by the administrator. When the resources are allocated to the job, the job can be executed.

There are different types of resources, i.e. server level resources, like walltime, chunk resources, like number of cpus or nodes, and generic resource (GRES) like GPUs on the systems that have them.

Other resources may be added to manage access to software resources, for example when resources are limited and the lack of availability leads to jobs abort when they are scheduled for execution. More details may be found in the module help of the application you are trying to execute.

The syntax of the request depends on the type of resource:

```
#SBATCH --<resource>=<value>          (server level resources, e.g. walltime)
#SBATCH --<chunk_resource>=<value>    (chunk resources, e.g. cpus, nodes,...)
#SBATCH --gres=gpu:<value>            (generic resources, e.g. gpus)
```

For example:

```
#SBATCH --time=10:00:00
#SBATCH --ntasks-per-node=1
#SBATCH --gres=gpu:2
```

Resources can be required either:

1) using SLURM directives in the job script

2) using options of the **sbatch/salloc** command

SLURM job script and directives

A SLURM job script consists of:

- An optional shell specification
- SLURM directives
- Tasks -- programs or commands to be executed

Once ready, the job must be submitted to SLURM:

```
> sbatch [options] <name of script>
```

The shell to be used by SLURM is defined in the first line of the job script (mandatory!):

```
#!/bin/bash (or #!/bin/sh)
```

The SLURM directives are used to request resources or set attributes. A directive begins with the default string #SBATCH. One or more directives can follow the shell definition in the job script.

The tasks can be programs or commands. This is where the user specifies the application to run.

SLURM directives: resources

The type of resources required for a serial or parallel MPI/OpenMP/mixed job must be specified with a SLURM directive:

```
#SBATCH --<chunk_resource>=<value>
```

where <chunk_resource> can be one of the following:

- **--nodes=NN** number of nodes
- **--ntasks-per-node=CC** number of tasks/processes *per node*
- **--cpus-per-task=TT** number of threads/cpus per task

For example for a MPI or MPI/OpenMP mixed job (2 MPI processes and 8 threads):

```
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=8
```

For a serial job for example:

```
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
```

SLURM directives: processing time

Resources such as computing time must be requested by this syntax:

```
#SBATCH --time=<value>
```

where <value> expresses the actual elapsed time (wall clock) in the format **hh:mm:ss**

for example:

```
#SBATCH --time=1:00:00 (one hour)
```

Please note that there are specific limitations on the maximum walltime on a system, also depending on the partition. Check the system specific guide for more information.

SLURM directives: memory allocation

The default memory depends on the partition/queue you are working with. Usually we set it as the Total Memory of the Node divided by the total number of cores in a single node that we call here memory-per-core. So if you request 3 cores by default you would get the equivalent of 3 times the memory-per-core. Alternatively, you can specify the requested memory with the --mem=<value> directive up to maximum memory available on the nodes.

```
#SBATCH --mem=10000
```

The default measurement unit for memory requests is the **Megabyte** (in the example above, we are requesting for 10000MB per node). It is possible to ask for an amount of memory expressed in GB, like this:

```
#SBATCH --mem=10GB
```

However, the default request method in MB is preferable, since the memory limits defined for any partition are expressed in these terms. For example, Marconi SkyLake partition has 182000MB as a limit, corresponding to approx. 177GB.

Please note: if you are requiring a larger memory with respect to the "main amount" for the system, the number of "effective cores" and the cost of your job could increase. For more information check the accounting section.

SLURM directives: MPI tasks/OpenMP threads affinity

You may have to modify the default affinity, in order to ensure optimal performances on A3 Marconi.

The slurm directives that concern the processes binding are the following:

```
--cpu-bind=<cores|threads>
```

```
--cpus-per-task=<physical or logical cpus number to allocate for single task>
```

The value of --cpus-per-task defines the SLURM_CPUS_TASK variable. When launching with "srun" (whenever possible), the variable is not inherited by srun and needs to be exported with

```
export SRUN_CPUS_PER_TASK=$SLURM_CPUS_PER_TASK
```


```
export SRUN_CPUS_PER_TASK=$SLURM_CPUS_PER_TASK
```


In order to modify them correctly, we suggest to follow our guidelines.

Other SLURM directives

```
#SBATCH --account=<account_no>          --> name of the project to be accounted to ("saldo -b" for a list of
projects)
#SBATCH --job-name=<name>               --> job name
#SBATCH --partition=<destination>       --> partition/queue destination. For a list and description of available
partitions, please refer to the specific cluster description of the guide.
#SBATCH --qos=<qos_name>                --> quality of service. Please refer to the specific cluster description
of the guide.
#SBATCH --output=<out_file>             --> redirects output file (default, if missing, is slurm-<Pid> containing
merged output and error file)
#SBATCH --error=<err_file>              --> redirects error file (as above)
#SBATCH --mail-type=<mail_events>       --> specify email notification (NONE, BEGIN, END, FAIL, REQUEUE, ALL)
#SBATCH --mail-user=<user_list>         --> set email destination (email address)
```

Directives in contracted form

Some SLURM directives can be written with a contracted syntax. Here are all the possibilities:

```
#SBATCH -N <NN>                         --> #SBATCH --nodes=<NN>
#SBATCH -c <TT>                         --> #SBATCH --cpus-per-task=<TT>
#SBATCH -t <value>                      --> #SBATCH --time=<value>
#SBATCH -A <account_no>                 --> #SBATCH --account=<account_no>
#SBATCH -J <name>                       --> #SBATCH --job-name=<name>
#SBATCH -p <destination>                --> #SBATCH --partition=<destination>
#SBATCH -q <qos_name>                   --> #SBATCH --qos=<qos_name>
#SBATCH -o <out_file>                   --> #SBATCH --output=<out_file>
#SBATCH -e <err_file>                   --> #SBATCH --error=<err_file>
```

Note: the directives --mem, --mail-type, --mail-user and --ntasks-per-node can't be contracted. About the latter, it exists a SLURM directive "-n" for the number of tasks, but it can be misleading since it is used to indicate the TOTAL number of tasks and not the number of tasks per node. Therefore, it is not recommended since it can lead to confusion and unexpected behaviour. Use of the uncontracted --ntasks-per-node is recommended instead.

Using sbatch attributes to assign job attributes and resource request

It is also possible to assign the job attributes using the **sbatch** command options:

```
> sbatch [--job-name=<name>]  [--partition=<queue/partition>]  [--out=<out_file>] [--err=<err_file>] [--mail-type=<mail_events>] [--mail-user=<user_list>] <name of script>
```

And the resources can also be requested using the **sbatch** command options:

```
> sbatch  [--time=<value>] [-ntasks=<value>] [--account=<account_no>] <name of script>
```

The sbatch command options override script directives if present.

Examples

Serial job script

For a typical serial job you can take the following script as a template, and modify it depending on your needs.

The script asks for 10 minutes wallclock time and runs a serial application (R). The input data are in file "data", the output file is "out.txt"; job.out will contain the std-out and std-err of the script. The working directory is $CINECA_SCRATCH/test/.

The account number (#SBATCH --account) is required to specify the project to be accounted for. To find out the list of your account number/s, please use the "**saldo -b**" command.

```
#!/bin/bash
#SBATCH --time=00:10:00
#SBATCH --nodes=1 --ntasks-per-node=1 --cpus-per-task=1
#SBATCH --mem=10000
#SBATCH --out=job.out
#SBATCH --account=<account_no>

cd $CINECA_SCRATCH/test/
module load autoload r
R < data > out.txt
```

Serial job script with specific queue request

This script is similar to the previous one but asks for the explicit serial partition on Marconi (that is the default).

```
#!/bin/bash
#SBATCH --out=job.out
#SBATCH --time=00:10:00
#SBATCH --nodes=1 --ntasks-per-node=1 --cpus-per-task=1
#SBATCH --account=<my_account>
#SBATCH --partition=bdw_all_serial
#
cd $CINECA_SCRATCH/test/
cp /gss/gss_work/DRES_my/* .
```

MPI job script

For a typical MPI job you can take one of the following scripts as a template, and modify it depending on your needs.

In this example we ask for 8 tasks, 2 SKL nodes and 1 hour of wallclock time, and runs an MPI application (myprogram) compiled with the intel compiler and the mpi library. The input data are in file "myinput", the output file is "myoutput", the working directory is where the job was submitted from. Through “–cpus-per-task=1” istruction each task will bind 1 physical cpu (core). This is a default option.

```
#################################
```

```
#!/bin/bash
#SBATCH --time=01:00:00

#SBATCH --nodes=2
#SBATCH --ntasks-per-node=4
#SBATCH --ntasks-per-socket=2
#SBATCH --cpus-per-task=1

#SBATCH --mem=<mem_per_node>
#SBATCH --partition=<partion_name>
#SBATCH --qos=<qos_name>
#SBATCH --job-name=jobMPI
#SBATCH --err=myJob.err
#SBATCH --out=myJob.out
#SBATCH --account=<account_no>

module load intel intelmpi
srun myprogram < myinput > myoutput

###################################
```

For SKL users an useful option may be to consider --cpu-bind=cores when the number of tasks requested is less than 48 cores per node. More details can be found at this page.

OpenMP job script

For a typical OpenMPI job you can take one the following scripts as a template, and modify it depending on your needs.

Nodes without hyperthreading

Here we ask for a single node and a single task, thus allocating 48 physical cpus for MARCONI SKL and GALILEO100 or 32 (128 logical cpus) for MARCONI100. With the export of OMP_NUM_THREADS we are setting 48 or 32 OperMP threads for the single task.

```
###################################

#!/bin/bash
#SBATCH --time=01:00:00

#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=48 or 32

#SBATCH --partition=<partition_name>
#SBATCH --qos=<qos_name>
#SBATCH --mem=<mem_per_node>
#SBATCH --out=myJob.out
#SBATCH --err=myJob.err
#SBATCH --account=<account_no>

module load intel

export SRUN_CPUS_PER_TASK=$SLURM_CPUS_PER_TASK
export OMP_NUM_THREADS=48 or 32 srun myprogram < myinput > myoutput

###################################
```

MPI+OpenMP job script

For a typical hybrid job you can take one the following scripts as a template, and modify it depending on your needs.

Nodes without hyperthreading

For example the script asks for 8 MPI tasks, 2 nodes and 4 OpenMP threads for task, 1 hours of wallclock time. The application (myprogram) was compiled with the intel compiler and the openmpi library. The input data are in file "myinput", the output file is "myoutput", the working directory is where the job was submitted from.

```
###################################
```

```
#!/bin/bash
#SBATCH –time=01:00:00

#SBATCH --nodes=2
#SBATCH --ntasks-per-node=4
#SBATCH --ntasks-per-socket=2
#SBATCH --cpus-per-task=4

#SBATCH --mem=<mem_per_node>
#SBATCH --partition=<partition_name>
#SBATCH --qos=<qos_name>
#SBATCH --job-name=jobMPI
#SBATCH --err=myJob.err
#SBATCH --out=myJob.out
#SBATCH --account=<account_no>

module load intel intelmpi


export SRUN_CPUS_PER_TASK=$SLURM_CPUS_PER_TASK
export OMP_NUM_THREADS=4
export OMP_PLACES=cores
export OMP_PROC_BIND=true

srun myprogram < myinput > myoutput

###################################
```

For SKL users an useful option may be to consider --cpu-bind=cores when the number of tasks requested is less than 48 cores per node. More details can be found at this page.

Running Hybrid MPI/OpenMP code with pure MPI job

If you would like to run a MPI code compiled with OpenMP flags as a pure MPI code, OMP_NUM_THREADS needs to be set to 1 explicitly. Otherwise, it will run with 4 OpenMP threads, since the default behavior for Intel and GNU compilers is to use max available threads.

```
###################################

#!/bin/bash
#SBATCH –time=01:00:00

#SBATCH --ntasks-per-node=4
#SBATCH --cpus-per-tasks=1
#SBATCH --nodes=2

#SBATCH --partition=<partition_name>
#SBATCH --qos=<qos_name>
#SBATCH --mem=86000
#SBATCH --out=myJob.out
#SBATCH --err=myJob.err
#SBATCH --account=<account_no>

module load intel

export SRUN_CPUS_PER_TASK=$SLURM_CPUS_PER_TASK
export OMP_NUM_THREADS=1

srun myprogram < myinput > myoutput

###################################
```

Chaining multiple jobs

In some cases, you may want to chain multiple jobs together, for example, so that the output of a run can be used as input of the next run. This is typical when you perform Molecular Dynamics Simulations and you want to obtain a long trajectory from multiple simulation runs.

In order to exploit this feature you need to submit your jobs using the sbatch option "-d" or "--dependency" to submit dependent jobs using SLURM. In the following lines we will show an example when the second job will run only when the first job runs successfully:

```
> sbatch job1.cmd
 submitted batch job 100
> sbatch -d afterok:100 job2.cmd
 submitted batch job 101
```

Alternatively:

```
> sbatch job1.cmd
 submitted batch job 100
> sbatch --dependency=afterok:100 job2.cmd
 submitted batch job 102
```

The available options for -d or --dependency are:
afterany:job_id[:jobid...], afternotok:job_id[:jobid...], afterok:job_id[:jobid...], ... etc..
See the sbatch man page for more detail.

High throughput Computing with SLURM

Array jobs are an efficient way to perform multiple similar runs, either serial or parallel, by submitting a unique job. The maximum allowed number of runs in an array job depends on the cluster. Job arrays are only supported for batch jobs and the array index values are specified using the "--array" or "-a" option of the sbatch command. The option argument can be specific array index values, a range of index values, and optional step size.

In the following examples, 20 serial runs with index values between 0 and 20 are submitted, and job.cmd is a SLURM batch script:

```
>sbatch --array=0-20 -N1 job.cmd
```

(-N1 is the equivalent of "--nodes=1")

Alternatively, to submit a job array with index values of 1, 3, 5 and 8:

```
>sbatch --array=1,3,5,8 -N1 job.cmd
```

To submit a job array with index values in the range 1 and 7 with a step size of 2 (i.e. 1,3,5, and 7):

```
>sbatch --array=1-7:2 -N1 job.cmd
```

When submitting a job array using SLURM you will have five additional environment variables set:

SLURM_ARRAY_JOB_ID will be set to the first job ID of the array.

SLURM_ARRAY_TASK_ID will be set to the job array index value.

SLURM_ARRAY_TASK_COUNT will be set to the number of tasks in the job array.

SLURM_ARRAY_TASK_MAX will be set to the highest job array index value.

SLURM_ARRAY_TASK_MIN will be set to the lowest job array index value.

As an exemple, let's assume a job submission like this:

```
>sbatch --array=1-3 -N1 job.cmd
```

This will generate a job array consisting of three jobs. If you submit the command above and assuming the sbatch command returns:

```
> Submitted batch job 100
```

(where 100 is an example of a job_id)

Then you will have the following environment variables:

SLURM_JOB_ID=100
SLURM_ARRAY_JOB_ID=100
SLURM_ARRAY_TASK_ID=1
SLURM_ARRAY_TASK_COUNT=3
SLURM_ARRAY_TASK_MAX=3
SLURM_ARRAY_TASK_MIN=1

SLURM_JOB_ID=101
SLURM_ARRAY_JOB_ID=100
SLURM_ARRAY_TASK_ID=2
SLURM_ARRAY_TASK_COUNT=3
SLURM_ARRAY_TASK_MAX=3
SLURM_ARRAY_TASK_MIN=1

SLURM_JOB_ID=102
SLURM_ARRAY_JOB_ID=100
SLURM_ARRAY_TASK_ID=3
SLURM_ARRAY_TASK_COUNT=3
SLURM_ARRAY_TASK_MAX=3
SLURM_ARRAY_TASK_MIN=1

All SLURM commands and APIs recognize the SLURM_JOB_ID value. Most commands also recognize the SLURM_ARRAY_JOB_ID plus SLURM_ARRAY_TASK_ID values separated by an underscore as identifying an element of a job array. Using the example above, "101" or "100_2" would be equivalent ways to identify the second array element of job 100.

Two additional options are available to specify a job's stdin, stdout, and stderr file names:
%A will be replaced by the value of SLURM_ARRAY_JOB_ID (as defined above) and **%a** will be replaced by the value of SLURM_ARRAY_TASK_ID (as defined above). The default output file format for a job array is "slurm-%A_%a.out". An example of explicit use of the formatting is:

```
>sbatch -o slurm-%A_%a.out --array=1-3 -N1 tmp
```

Some useful commands to manage job arrays

scancel

If the job ID of a job array is specified as input of the scancel command, then all elements of that job array will be canceled. Alternately an array ID, optionally using regular expressions, may be specified for job cancellation.
To cancel array ID 1 to 3 from job array 100:

```
> scancel 100_[1-3]
```

To cancel array ID 4 and 5 from job array 100:

```
> scancel 100_4 100_5
```

To cancel all elements from job array 100:

```
> scancel 100
```

scontrol

The use of the scontrol show job option shows two new fields related to job array support. The JobID is a unique identifier for the job. The ArrayJobID is the JobID of the first element of the job array. The ArrayTaskID is the array index of this particular entry, either a single number of an expression identifying the entries represented by this job record (e.g. "5-1024").

The scontrol command will operate on all elements of a job array if the job ID specified is ArrayJobID. Individual job array tasks can be modified using the ArrayJobID_ArrayTaskID as shown below. A few examples below:

```
> scontrol update JobID=100_2 name=my_job_name
> scontrol suspend 100
> scontrol resume 100
> scontrol suspend 100_3
> scontrol resume 100_3
```

squeue

When a job array is submitted to SLURM, only one job record is created. Additional job records will only be created when the state of a task in the job array changes, typically when a task has allocated resources or its state is modified using the scontrol command. By default, the squeue command will report all of the tasks associated with a single job record on one line and use a regular expression to indicate the "array_task_id" values.
An option of "--array" or "-r" can also be added to the squeue command to print one job array element per line.
The squeue --step/-s and --job/-j options can accept job or step specifications of the same format:

```
> squeue -j 100_2,100_3
> squeue -s 100_2.0,100_3.0
```

Further documentation

More specific information about partitions and qos (quality of service), limits and available features are described on the "system specific" pages of this Guide, for MARCONI and GALILEO100, as well as "man" pages about SLURM commands:

```
> man sbatch

> man squeue

> man sinfo

> man scancel

> man ...
```