UNIVERSITY
OF OSLO

Job Scripts

Table of Contents

This page documents how to write job scripts to submit jobs on the Colossus (HPC for TSD) cluster.

To run a *job* on the cluster involves creating a shell script called a *job script*. The job script is a plain-text file containing any number of commands, including your main computational task, i.e., it may copy or rename files, cd into the proper directory, etc., all before doing the "real" job. The lines in the script file are the commands to be executed, in the given order. Lines starting with a "#" are ignored as comments, except lines that start with a "#SBATCH" which are not executed, but contain special instructions to the queue system.

If you are not familiar with shell scripts, they are simply a set of commands that you could have typed at the command line. You can find more information about shell scripts here: Introduction to Bash shell scripts.

A job script consists of a couple of parts:

- Instructions to the queue system
- Commands to set up the execution environment
- The actual commands you want to be run

Instruction parameters to the queue system may be specified on the `sbatch` command line and/or in `#SBATCH` lines in the job script. There can be as many `#SBATCH` lines as you need, and you can combine several parameters on the same line. If a parameter is specified both on the command line and in the jobscript, the parameter specified on the command line takes precedence. The `#SBATCH` lines should precede any commands in the script. A couple of parameters are compulsory. If they are not present, the job will not run:

- **--account**: Specifies the *project* the job will run in. If your project has a Sigma quota, this will be *project* (e.g. *p11*). If your project does not have a Sigma quota you'll need to use the tsd reservation, *project_tsd* (e.g. *p11_tsd*).
- **--time**: Specifies the maximal *wall clock time* of the job. Do not specify the limit too low, because the job will be killed if it has not finished within this time. On the other hand, shorter jobs will be started sooner, so do not specify longer than you need. The default maximum allowed **--time** specification is 1 week; see details here.
- **--mem-per-cpu**: Specifies how much RAM each task (default is 1 task; see Parallel Jobs) of the job needs. Technically, this limit specifies the amount of resident memory + swap the job can use. If the job tries to use more than this setting, it will be killed. The maximum that can be requested it the available memory on the node (~500GB on the compute nodes). For instance:

  ```
  #SBATCH --mem-per-cpu=2000M
  ```

  If the job tries to use more than 2000 MiB resident memory requested in this example, it will be killed. Note that *--mem-per-cpu* is specified per requested core.

The commands to set up the environment should include

```
module load SomeProgram/SomeVersion
```

`jobsetup` will set up needed environment variables and shell functions, and must be the first command in the script. `module` will set up environment variables to get access to the specified program. It is recommended to specify the explicit version in the `module load` command. We also encourage you to use

```
module purge
```

prior to the `module load` statements, to avoid inheriting unknown environment variable settings from the shell you use to submit the job. We also advice using

```
set -o errexit
```

early in the script. This makes the script exit immediately if any command below it fails, instead of simply going on to the next command. This makes it much easier to find out whether anything went wrong in the job, and if so, where it happened.

A Simple Serial Job

This is a script for running a simple, serial job

```
#!/bin/bash

# Job name:
#SBATCH --job-name=YourJobname
#
# Project:
#SBATCH --account=YourProject
#
# Wall clock limit:
#SBATCH --time=hh:mm:ss
#
# Max memory usage:
#SBATCH --mem-per-cpu=Size

## Set up job environment:
module purge   # clear any inherited modules
set -o errexit # exit on errors

## Copy input files to the work directory:
cp MyInputFile $SCRATCH

## Make sure the results are copied back to the submit directory (see Work Dire
chkfile MyResultFile

## Do some work:
cd $SCRATCH
YourCommands
```

Substitute real values for *YourJobname*, *YourProject*, *hh:mm:ss*, *Size* (with 'M' for megabytes or 'G' for gigabytes; e.g. *200M* or *1G*), *MyInputFile*, *MyResultFile* and *YourCommands*.

Work Directory

Each job has access to a separate scratch space directory on the shared file system `/cluster`. The name of the directory is stored in the environment variable `$SCRATCH` in the job script. As a general rule, all jobs should use the scratch directory (`$SCRATCH`) as its work directory. This is especially important if the job uses a lot of files, or does much random access on the files (repeatedly read and write to different places of the files).

There are several reasons for using `$SCRATCH`:

- `$SCRATCH` is on a faster file system than user home directories.
- There is less risk of interfering with running jobs by accidentally modifying or deleting the jobs' input or output files.
- Temporary files are automatically cleaned up, because the scratch directory is removed when the job finishes.
- It avoids taking unneeded backups of temporary and partial files, because `$SCRATCH` is not backed up.

The directory where you ran `sbatch` is stored in the environment variable `$SLURM_SUBMIT_DIR`. If you want automatic copying of files or directories back to `$SLURM_SUBMIT_DIR` when the job is terminated, mark them with the command `chkfile` in the job script:

```
chkfile OneFile AnotherFile SomeDirectory
```

- The `chkfile` command should be placed early in the script, before the main computational commands: The files will be copied back even if the script crashes (even when you use `set -o errexit`), but not if it is terminated before it got to the `chkfile` command.
- If you want to use shell-metacharacters, they should be quoted. I.e., use `chkfile "Results*"` instead of `chkfile Results*`.
- Do not use absolute paths or things like `".*"` or `..` in the `chkfile` command; use paths relative to `$SCRATCH`.
- The files/directories are copied to `$SLURM_SUBMIT_DIR` with `rsync`, which means that if you use `chkfile ResultDir`, the contents of `ResultDir` will be copied into `$SLURM_SUBMIT_DIR/ResultDir`, but if you use `chkfile ResultDir/` (i.e., with a trailing `/`), the contents of `ResultDir` will be copied directly into `$SLURM_SUBMIT_DIR`.

We recommend using `chkfile` (or `cleanup`; see below) instead of explicitly copying the result files with `cp`.

For instance:

```
#!/bin/bash
```

```
#SBATCH --job-name=YourJobname --account=YourProject
#SBATCH --time=hh:mm:ss --mem-per-cpu=Size

module purge   # clear any inherited modules
set -o errexit # exit on errors

## Copy files to work directory:
cp YourDatafile $SCRATCH

## Mark outfiles for automatic copying to $SLURM_SUBMIT_DIR:
chkfile YourOutputfile

## Run command
cd $SCRATCH
YourProgram YourDatafile > YourOutputfile
```

The `$SCRATCH` directory is removed upon job exit (after copying back `chkfile`d files).

If you want more flexibility than what `chkfile` gives, you can use the command `cleanup` instead. It is used to specify commands to run when your job exits (before the $SCRATCH directory is removed). Just like `chkfile`, the `cleanup` commands are run even if your script crashes (even when you use `set -o errexit`), but not if it crashes before reaching the `cleanup` command, so place the command early in the script.

For instance:

```
cleanup "cp $SCRATCH/outputfile /some/other/directory/newName"
```

Note: do not use *single* quotes (') if the commands contain variables like $SCRATCH and $SLURM_SUBMIT_DIR.

Splitting a Job into Tasks (Array Jobs)

To run many instances of the same job, use the `--array` switch to `sbatch`. This is useful if you have a lot of data-sets which you want to process in the same way with the same job-script:

```
sbatch --array=from-to [other sbatch switches] YourScript
```

You can also put the `--array` switch in an `#SBATCH` line inside the script. *from* and *to* are the first and last task number. Each instance of *YourScript* can use the environment variable `$SLURM_ARRAY_TASK_ID` for selecting which data set to use, etc. For instance:

```
sbatch --array=1-100 MyScript
```

will run 100 copies of *MyScript*, setting the environment variable `$SLURM_ARRAY_TASK_ID` to 1, 2, ..., 100 in turn.

It is possible to specify the task ids in other ways than *from-to*: it can be a single number, a range (*from-to*), a range with a step size (*from-to*:*step*), or a comma separated list of these. Finally, adding *%max* at the end of the specification puts a limit on how many tasks will be allowed to run at the same time. A couple of examples:

```
Specification   Resulting TASK_IDs
1,4,42        # 1, 4, 42
1-5           # 1, 2, 3, 4, 5
0-10:2        # 0, 2, 4, 6, 8, 10
32,56,100-200 # 32, 56, 100, 101, 102, ..., 200
1-200%10      # 1, 2, ..., 200, but maximum 10 running at the same time
```

Note: spaces, decimal numbers or negative numbers are not allowed.

The instances of an array job are independent, they have their own `$SCRATCH` and are treated like separate jobs. (This means that they count against the limit of the total number of jobs a user can have in the job queue, currently 400.) You may also specify a parallel environment in array-jobs, so that each instance gets e.g. 8 processors. To cancel all tasks of an array job, cancel the jobid that is returned by the *sbatch* command.

See `man sbatch` for details.

An extended example:

```
$ cat JobScript
#!/bin/bash
#SBATCH --account=YourProject
#SBATCH --time=1:0:0
#SBATCH --mem-per-cpu=1G
#SBATCH --array=1-200

module purge   # clear any inherited modules
set -o errexit # exit on errors

DATASET=dataset.$SLURM_ARRAY_TASK_ID
OUTFILE=result.$SLURM_ARRAY_TASK_ID

cp $DATASET $SCRATCH
cd $SCRATCH
chkfile $OUTFILE
YourProgram $DATASET > $OUTFILE
```

```
$ sbatch JobScript
```

This job will process the datasets dataset.1, dataset.2, ... dataset.200 and leave the results in result.1, result.2, ... result.200.

Parallel Jobs

Since the clusters consist of multi-cpu and multi-core compute nodes, there are several ways of parallelising code and running jobs in parallel. One can use MPI, OpenMP and threading. (For SIMD jobs and other jobs with very independent parallel tasks, array jobs is a good alternative.) In this section we will explain how to run in parallel in either of these ways.

Note: A parallel job will get one, shared scratch directory (`$SCRATCH`), not separate directories for each node! This means that if more than one process or thread write output to disk, they must use different file names, or put the files in different subdirectories.

OpenMP or Threading

In order to run in parallel on one node using either threading or OpenMP, the only thing you need to remember is to tell the queue system that your task needs more than one core, all on the same node. This is done with the *--cpus-per-task* switch. It will reserve the needed cores on the node and set the environment variable `$OMP_NUM_THREADS`. For example:

```
#!/bin/bash
# Job name:
#SBATCH --job-name=YourJobname
#
# Project:
#SBATCH --account=YourProject
#
# Wall clock limit:
#SBATCH --time=hh:mm:ss
#
# Max memory usage per core (MB):
#SBATCH --mem-per-cpu=MegaBytes
#
# Number of cores:
#SBATCH --cpus-per-task=NumCores

## Set up job environment:
module purge   # clear any inherited modules
set -o errexit # exit on errors

## Copy files to work directory:
cp YourDatafile $SCRATCH

## Mark outfiles for automatic copying to $SLURM_SUBMIT_DIR:
chkfile YourOutputfile

## Run command
cd $SCRATCH
## (For non-OpenMP-programs, you must control the number of threads manually, u
YourCommand YourDatafile > YourOutputfile
```

The *--cpus-per-task* will ensure all cores are allocated on a single node. If you ask for more cores than is available on any node, you will not be able to submiut the job. Most nodes on Colossus have 64 cores.

MPI

To run MPI jobs, you must specify how many tasks to run (i.e., cores to use), and set up the desired MPI environment. The symbol generation for different fortran compilers differ, hence versions of the MPI fortran interface for GNU, Intel, Portland and Open64. All OpenMPI libraries are built using gcc.

You need to use `module load` both in order to compile your code and to run it. You also should compile and run in the same MPI-environment. Although some MPI-versions may be compatible, they usually are not.

If you need to compile C MPI code with `icc` please see OpenMPI's documentation about environment variables to set in order to force `mpicc` to use `icc`.

The simplest way to specify the number of tasks (cores) to use in an MPI job, is to use the `sbatch` switch `--ntasks`. For instance

```
#SBATCH --ntasks 10
```

would give you 10 tasks. The queue system allocates the tasks to nodes depending on available cores and memory, etc. A simple MPI jobscript can then be like:

```
#!/bin/bash
# Job name:
#SBATCH --job-name=YourJobname
#
# Project:
#SBATCH --account=YourProject
#
# Wall clock limit:
#SBATCH --time=hh:mm:ss
#
# Max memory usage per task:
#SBATCH --mem-per-cpu=Size
#
# Number of tasks (cores):
#SBATCH --ntasks=NumTasks

## Set up job environment:
module purge   # clear any inherited modules
set -o errexit # exit on errors

module load openmpi.gnu
```

```
## Set up input and output files:
cp InputFile $SCRATCH
chkfile OutputFile

cd $SCRATCH
srun YourCommand
```

Queue System Options

Using the queue option *--ntasks* in the previous example, we have assumed that it doesn't matter how your tasks are allocated to nodes. The queue system will run your tasks on nodes as it sees fit (however, it will try to allocate as many tasks as possible on each node). For small jobs, that is usually OK. Sometimes, however, you might need more control. Then you can use the switches `--nodes` and `--ntasks-per-node` instead of `--ntasks`:

- `--nodes`: How many nodes to use
- `--ntasks-per-node`: How many tasks to run on each node.

For instance, to get 1 task on each of 4 nodes, you can use

```
#SBATCH --nodes=4 --ntasks-per-node=1
```

Or, to use 4 task on each of 2 nodes:

```
#SBATCH --nodes=2 --ntasks-per-node=4
```

There are more advanced options for selecting cores and nodes, as well. See `man sbatch` for the gory details.

TCP/IP over InfiniBand for MPI

If you have MPI jobs hard linked to use TCP/IP we have some tricks to use InfiniBand even for these. It is possible to run the TCP/IP over InfiniBand with far better performance than over Ethernet. However this only apply to communications between the compute nodes. Please contact us if you have such an application or want to use TCP/IP over InfiniBand. All nodes have two IP numbers one for the Ethernetnet and one for InfiniBand.

Useful Commands in MPI Scripts

If you need to execute a command once on each node of a job, you can use

```
srun --ntasks=$SLURM_JOB_NUM_NODES command
```

Combining MPI with OpenMP or threading

You can combine MPI with OpenMP or threading, such that MPI is used for launching multi-threaded processes on each node. The best way to do this is:

```
#!/bin/bash
# Job name:
#SBATCH --job-name=YourJobname
#
# Project:
#SBATCH --account=YourProject
#
# Wall clock limit:
#SBATCH --time=hh:mm:ss
#
# Max memory usage per task:
#SBATCH --mem-per-cpu=Size
#
# Number of tasks (MPI ranks):
#SBATCH --ntasks=NumTasks
#
# Number of threads per task:
#SBATCH --cpus-per-task=NumThreads

## Set up job environment:
module purge   # clear any inherited modules
set -o errexit # exit on errors
module load openmpi.gnu

## Set up input/output files:
cp InputFile $SCRATCH
chkfile OutputFile

## Run command
## (If YourCommand is not OpenMP, use $OMP_NUM_THREADS to control the number of
srun YourCommand
```

This makes `srun` start *NumTasks* ranks (processes), each of which having *NumThreads* threads. If you are not using OpenMP, your program must make sure not to start more than *NumThreads* threads.

Just as with single-threaded MPI jobs, you can get more than one rank (MPI process) on each node. If you need more control over how many ranks are started on each node, use `--ntasks-per-node` and `--nodes` as above. For instance:

```
#SBATCH --nodes=3 --ntasks-per-node=2 --cpus-per-task=4
```

will start 2 MPI ranks on each of 3 machines, and each process is allowed to use 4 threads.

Large Memory Jobs

Most nodes on Colossus are equipped with 64 CPU cores and 502 GiB of RAM usable for jobs. There are also a couple of special *bigmem* nodes with 128 cores and 4 TiB memory, of which about 3955 GiB can be used for jobs.

If you need more than 502 GiB RAM *on a single node*, you must specify `--partition=bigmem` to get access to the nodes with more RAM. For instance

```
#SBATCH --ntasks-per-node=16
#SBATCH --mem-per-cpu=50G --partition=bigmem
```

Accounting of Large Memory Jobs

To ensure maximal utilisation of the cluster, memory usage is accounted as well as cpu usage. Memory specifications are converted to "Processor Equivalents" (PE) using a conversion factor of *approximately* 8 GiB / core(*). If a job specifies more than 8 GiB RAM per task, i.e., `--mem-per-cpu=`*M*, where *M* > 8G, each task will count as *M* / 8G cores instead of 1 core. For instance, a job with `--ntasks=2 --mem-per-cpu=16G` will be counted as using 4 cores instead of 2.

The reason for this is that large memory jobs make the "unused" cores on the same nodes inaccessible to other jobs. For instance, a job on a 502 GiB node using `--ntasks=1 --mem-per-cpu=502G` will in practice use all cores on the node, and should be accounted as such.

Note that only jobs which specify more than 8 GiB per core will be affected by this; all other jobs will be accounted with the number of tasks specified.

(*) The exact value of the factor depends on the total amount of RAM per core in the cluster, and is currently about 8 GiB / core on Colossus.

GPU jobs

Colossus has GPU nodes with 2 (Nvidia Tesla V100; TSD allocation) or 4 (Nvidia Tesla A100; Sigma2 allocation) GPUs each. You can run jobs on the GPU nodes like this in the TSD queue, where N is the number of GPUs:

```
$ sbatch --account=YourProject_tsd --partition=accel --gres=gpu:N
```

or like this in the Sigma2 queue:

```
$ sbatch --account=YourProject --partition=accel --gres=gpu:N
```

Checkpointing

Checkpointing a job means that the job can be stopped and started somewhere else, and continues where it left off.

Long-running jobs should implement some form of checkpointing, by saving intermediate results at intervals and being able to start with the latest intermediate results if restarted.

Useful sbatch parameters

Parameter	**Description**
--account=*project*	Specify the project to run under. This parameter is required.
--begin=*time*	Start the job at a given time
--constraint=*feature*	Request nodes with a certain feature. Currently supported features include `amd`m `intel`, `ib`, `rack`*N*. If you need more than one feature, they must be combined with `&` in the same `--constraint` specification, e.g. `--constraint=ib&rack21`. Note: If you try to use more than one `--constraint` specification, the last one will override the earlier.
--cpus-per-task=*cores*	Specify the number of cpus (actually: cores) to allocate for each task in the job. See --ntasks and --ntasks-per-node, or *man sbatch*.
--dependency=*dependency list*	Defer the start of this job until the specified dependencies have been satisfied. See *man sbatch* for details.
--error=*file*	Send 'stderr' to the specified file. Default is to send it to the same file as 'stdout'. (Note: `$HOME` or ~ cannot be used. Use absolute or relative paths instead.)
--input=*file*	Read 'stdin' from the specified file. (Note: `$HOME` or ~ cannot

	be used. Use absolute or relative paths instead.)
--job-name=*jobname*	Specify job name
--mem-per-cpu=*size*	Specify the memory required *per allocated core*. This is the normal way of specifying memory requirements (see Large Memory Jobs). *size* should be an integer followed by 'M' or 'G'.
--partition=hugemem	Run on a *hugemem* node (see Large Memory Jobs).
--nodes=*nodes*	Specify the number of nodes to allocate. *nodes* can be an integer or a range (min-max). This is often combined with `--ntasks-per-node`.
--ntasks=*tasks*	Specify the number of tasks (usually cores, but see --cpus-per-task) to allocate. This is the usual way to specify cores in MPI jobs.
--ntasks-per-node=*tasks*	Specify the number of tasks (usually cores, but see --cpus-per-task) to allocate within each allocated node. Often combined with `--nodes`.
--output=*file*	Send 'stdout' (and 'stderr' if not redirected with `--error`) to the specified file instead of slurm-%j.out. Note: 1. `$HOME` or ~ cannot be used. Use absolute or relative paths instead. 2. The jobid should always be included in the filename in order to produce a separate file for each job (i.e. --output=*custom-name*%j.out). This is necessary for error tracing.
--time=*time*	Specify a (wall clock) time limit for the job. *time* can be *hh:mm:ss* or *dd-hh:mm:ss*. This parameter is required.
--nice=*value*	Run the job with an adjusted scheduling priority within your project. A negative nice value increases the priority, otherwise decreases it. The adjustment range is +/-

	2147483645.

Type to search

Contact support

Chat

Register case

Video call

Phone

Did you find what you were looking for?

Give us feedback

Published June 21, 2021 10:35 AM - Last modified June 17, 2022 3:05 PM

Share by
E-mail

Share on
Facebook

Share on
Twitter

UNIVERSITY
OF OSLO