UNIVERSITY
OF OSLO

Getting started with HPC facility in TSD

Login to the Linux VM and move the input data in the /cluster/projects location.

Access Colossus

By default, to submit jobs on the Colossus cluster, users must log on to the submit host assigned to the project. You can reach this host by using PUTTY from your Windows VM, or by using ssh from your Linux VM in your TSD-project:

```
$ ssh pXX-submit.tsd.usit.no
```

Data on the Colossus disk

When you get access to Colossus resources, you will have 1TiB of disk available on `/cluster/projects/pXX` (pXX = project number). In order to submit jobs you have to move all the input files and the data/repositories files needed in you workflow to the directory `/cluster/projects/pXX` using the submit host assigned to the project (as project login hosts can't access `/cluster`). This `/cluster/projects/pXX` directory is also where all your output files will be found at the end of the computation if the `sbatch` script is correctly written and the job completes successfully.

UNIVERSITY
OF OSLO

Queue System

Table of Contents

This page documents the queue system on the Colossus (HPC for TSD) cluster.

In TSD, each project has its own Linux Virtual Machine (VM) for submitting jobs to the cluster.

Submitting a job

To run a job on the cluster, you submit a job script into the *job queue*, and the job is started when one or more suitable *compute nodes* are available. The job queue is managed by a queue system (scheduler and resource manager) called Slurm (Slurm's documentation page).

Please note that jobscript names should not contain sensitive information.

Job scripts are submitted with the `sbatch` command:

```
sbatch YourJobscript
```

The `sbatch` command returns a *jobid*, an id number that identifies the submitted job. The job will be waiting in the job queue until there are free compute resources it can use. A job in that state is said to be *pending* (*PD*). When it has started, it is called *running* (*R*). Any output (stdout or stderr) of the job script will be written to a file called `slurm-jobid.out` in the directory where you ran `sbatch`, unless otherwise specified.

All commands in the job script are performed on the compute-node(s) allocated by the queue system. The script also specifies a number of requirements (memory usage, number of CPUs, run-time, etc.), used by the queue system to find one or more suitable machines for your job.

You can cancel running or pending (waiting) jobs with `scancel`:

```
scancel jobid # Cancel job with id jobid (as returned from sbatch)
scancel --user=MyUsername    # Cancel all your jobs
scancel --account=MyProject  # Cancel all jobs in MyProject
```

See `man scancel` for more details.

Project Quota

On Colossus, each user only has access to a single project, the name of which (p*NN*) is the prefix of the user name. Projects can have access to up to 3 allocations of computational resources in TSD:

- TSD allocation
- Sigma2 allocation
- Dedicated allocation

TSD allocation

Any TSD project with HPC access can use these resources for free with a 200k CPUh limit.

To use this resource, jobs should be submitted with the "`--account=pNN_tsd`" argument. Where pNN is your project number.

Sigma2 allocation

Only TSD projects with cpu hour quota from Sigma2 can use this pool. We advice any project with substantial computational needs to request CPU (and disk) quota from Sigma2 as described here. Sigma2 quota are valid for 6 month periods, starting April 1 and October 1.

To use this resource, jobs should be submitted with the "`--account=pNN`" argument. Where pNN is your project number. If you submit to this resource, but the project doesn't have a Sigma2 quota, jobs will remain pending (PD) with reason "AssocGrpBillingMinutes".

Dedicated allocation

All other compute and gpu nodes in Colossus were acquired by individual projects for privileged access but are maintained by TSD.

To use this resource, jobs should be submitted with the "`--account=pNN_reservationname`" argument. Where pNN is your project number and the reservationname the name of the dedicated resource.

Inspecting Quota

The command `cost` can be used to inspect the quota and see how much of it has been used and how much remains. This only applies to projects with an accountable Sigma2 quota. For projects *without* a Sigma2 quota, the cost command will list a zero quota and a corresponding message:

```
Report for account p11 on Colossus
Allocation period 2021.1 (2021-04-01 -- 2021-10-01)
Last updated on Mon Apr 12 10:14:02 2021
============================================================
Account  Description              Billing hours  % of quota
============================================================
p11      Used (finished)                   0.00          NA
p11      Reserved (running)                0.00          NA
p11      Pending (waiting)                 0.00          NA
p11      Available                         0.00          NA
p11      Quota                             0.00         100
============================================================
The project does not have a Sigma2 quota for the current period.
See https://www.uio.no/english/services/it/research/sensitive-data/use-tsd/hpc/
for information.
```

For the projects having a Sigma2 cpu hour quota, a typical output will look like:

```
Report for account p11 on Colossus
Allocation period 2021.1 (2021-04-01 -- 2021-10-01)
Last updated on Mon Apr 12 10:14:02 2021
================================================================
Account     Description             Billing hours     % of quota
================================================================
```

```
p11         Used (finished)                  0.95          0.0 %
p11         Reserved (running)               0.00          0.0 %
p11         Pending (waiting)              192.00          0.4 %
p11         Available                    49807.05         99.6 %
p11         Quota                        50000.00        100.0 %
============================================================
```

Notice that "Available" shows the "Quota" minus the "Used", the "Reserved" and the "Pending". So "Available" it is what is expected to be available if all running and pending jobs use the hours they specify. (Usually, jobs specify longer `--time` than they actually use, so "Available" will typically increase as jobs finish.)

One can also list the use per user within the project by adding "`--details`":

```
cost --details
```

This will append the quota usage per user:

```
Report for account p11 on Colossus
Allocation period 2020.2 (2020-10-01 -- 2021-04-01)
Last updated on Fri Oct  2 08:34:02 2020
============================================================
Account  Description              Billing hours  % of quota
============================================================
p11      Used (finished)                164.55       0.2 %
p11      Reserved (running)               0.00       0.0 %
p11      Pending (waiting)             1680.00       1.7 %
p11      Available                    98155.45      98.2 %
p11      Quota                       100000.00     100.0 %
============================================================
Account  User                Used billing hours  % of quota
============================================================
p11      p11-bartt                      162.32       0.2 %
p11      root                             2.22       0.0 %
============================================================
```

See `man cost` for details about this command.

Accounting

Accounting is done in terms of *billing units*, and the quota is in *billing unit hours*. Each job is assigned a number of billing units based on the requested CPUs, memory and GPUs. The number that is subtracted from the quota is the number of billing units multiplied with the (actual) wall time of the job.

The number billing units of a job is calculated like this:

1. Each requested CPU is given a cost of 1.

2. The requested memory is given a cost based on a *memory cost factor* (see below).
3. The requested GPU is given a cost based on a *GPU cost factor* (see below).
4. The number of billing units is the maximum of the CPU cost, memory cost and GPU cost.

The *memory cost factor* and the *GPU cost factor* vary between nodes.

- For regular compute nodes, the *memory cost factor* is 0.12749 units per GiB. Thus the memory cost of a job asking for all memory on a node will be 64, the number of CPUs on the node.
- For GPU nodes, the *memory cost factor* is 0.09775967 units per GiB, and the *GPU cost factor* is 24 per GPU. The means that a job asking for all memory, or all GPUs on a node, get a cost of 96, the number of CPUs on the node.
- For bigmem nodes, the *memory cost factor* is 0.0323641 per GiB. Thus the memory cost of a job asking for all memory on a node will be 128, the number of CPUs on the node.

When a project has exceeded the Quota limit, jobs will be left pending with the reason "AssocGrpBillingMinutes".

Inspecting Jobs

To get a quick view of the status of a job, you can use `squeue`:

```
squeue –j JobId
```

where *JobId* is the job id number that `sbatch` returns. To see more details about a job, use

```
scontrol show job JobId
```

See `man squeue` and `man scontrol` for details about these commands.

Inspecting the Job Queue

There are several available commands to inspect the job queue:

- `squeue`: list jobs in the queue
- `pending`: list the pending (waiting) jobs in the queue
- `qsumm`: show summary of queue usage

To see the list of running or pending jobs in the queue, use the command `squeue`. `squeue` will only show the jobs in your own project. Useful `squeue` options:

```
[-j jobids]   show only the specified jobs
[-w nodes]    show only jobs on the specified nodes
[-t states]   show only jobs in the specified states (pending, running,
             suspended, etc.)
[-u users]    show only jobs belonging to the specified users
```

All specifications can be comma separated lists. See `man squeue` for details. Examples:

```
squeue -j 14132,14133    # shows jobs 4132 and 4133
squeue -w c1-11         # shows jobs running on c1-11
squeue -u foo -t PD      # shows pending jobs belonging to user 'foo'
```

Squeue status (ST)

Status	**Text**
PD	Pending
R	Running
S	Suspended
CG	Completing
CDhttps://www.prosjektveiviseren.no/god-praksis/viktige-tema-i-alle-faser/interessenter/hva-er-en-interessent/interessentgrupper	Completed
CF	Configuring
CA	Cancelled
F	Failed
TO	Timeout

PR	Preempted
NF	Node failed

You can use the `pending` command to list only the pending jobs. It lists the jobs in descending priority order, and includes an estimate of when the job will start, when such an estimate is available. `pending` is simply a wrapper around `squeue`, and accepts all options that `squeue` takes. It will also just show the jobs belonging to your own project.

To see the resource situation of the cluster, use the command `qsumm`. It shows how many CPUs (or rather, Processor Equivalents) are used by running jobs and are requested by pending jobs. The output has two lines, one for your project, and one showing the total usage for all projects (including your project). An example output:

```
--------------------------------
Account        Limit  nRun nPend
--------------------------------
p11             1536    20    11
Total           1536  1550   200
--------------------------------
```

See `qsumm --help` for explanations of each column. The output is updated every 5 minutes, so it can take a couple of minutes after jobs are submitted/started/finished before it shows in the `qsumm` output.

General Job Limitations

Default values when nothing is specified

- 1 core (CPU)

The rest (time, mem per cpu, etc.) must be specified.

Limits

- The max wall time is 4 weeks, but do ***not*** submit jobs that will run for more than 7 days unless they implement checkpointing: ***None of the nodes on Colossus have dual power, and we reserve the right to shutdown any node for maintenance at any time with 7 days notice!***
- Max 4500 submitted jobs (running or pending) per project (`--account`) at the same time.

- Max size of job arrays: 4000. That also means that the largest job array index is 4000. Note that an job array of size *N* will count as *N* jobs wrt. the total number of submitted jobs per project.

Scheduling

Jobs are started by priority. Pending jobs are prioritized according to

- Queue time
- How many jobs each user has pending in the queue

Colossus starts jobs by priority + backfilling, so small, short jobs can start earlier than jobs with higher priority, as long as they do not delay the higher priority jobs. In addition, we have added a limit on how many jobs belonging to a user increase in priority over time, to avoid a single user preventing all other users from getting jobs run by submitting a large number of jobs at the same time. In this way, the priority will in effect increase for users running few jobs relative to users running many jobs. This is a trade-off, and we will adjust the limit (currently 10) if we see that the effect is too large/small.

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.

UNIVERSITY
OF OSLO

Modules

Introduction

To simplify setting up environments for compilers, MPI-versions and applications, Colossus has the Environment Modules package installed. This provides a means for dynamic modification of a user's environment via *modulefiles*.

Each modulefile contains the information needed to configure the shell for an environment. Typically, a modulefile modifies environment variables like `PATH`, `MANPATH`, `LD_LIBRARY_PATH`, `CPATH` and `PKG_CONFIG_PATH` to give access to an application or library. To invoke a module file you run the `module` command with arguments.

In TSD, the `module` command can be used from the command line on submit hosts and as part of a job script when submitting a job to Colossus. You can reach the submit host assigned to your project by connecting to `pXX-submit.tsd.usit.no` using PUTTY from your Windows VM, or by using `ssh` from your linux VM.

Listing available modules

The available modulefiles can be listed by `module avail`. Example output:

```
$ module avail
------------------ /cluster/software/MODULEFILES/custom ----------------------
   atlas-lapack/3.10.3            matlab/R2018b
   bcftools/1.8                   netcdf-fortran.gnu/4.4.4
   bedtools/2.17.0                netcdf.gnu/4.6.1
   bismark/0.22.1                 openmpi.gnu/1.8
   bowtie/1.2.2                   openmpi.gnu/3.1.3
[...]
------------------ /cluster/software/MODULEFILES/easybuild/all ----------------
   AFNI/18.3.00-foss-2018b-Python-3.6.6      libreadline/7.0-GCCcore-6.4.0
   Anaconda3/5.3.0                           libreadline/7.0-GCCcore-7.3.0
   arpack-ng/3.7.0-foss-2019a                libreadline/8.0-GCCcore-8.2.0
[...]
```

You'll see *custom* and *Easybuild* modules. The Easybuild modules are usually the most recent and should be preferred. When loading a custom module a notification will be displayed to clarify this.

You can also specify a name:

```
$ module avail MATLAB
------------------ /cluster/software/MODULEFILES/easybuild/all ----------------
   MATLAB/2017b    MATLAB/2018a    MATLAB/2018b    MATLAB/2019a
   MATLAB/2019b    MATLAB/2020a    MATLAB/2020b
```

Loading modules

Modulefiles can be used from most shells, as well as Perl and Python. Modulefiles can be loaded an unloaded on the command line or within your scripts by typing:

```
module load software/version
module unload software/version
```

If a version is not specified, no software will be loaded. For the purpose of reproducibility, we enforce users to load specific versions of modulefiles.

You will get a list of currently loaded modulefiles by:

```
$ module list
Currently Loaded Modulefiles:
 1) MATLAB/2020b
```

To view what exactly a modulefile is doing to your environment, type:

```
 module show modulefile/version
```

Switch to a different version of a module

Switching to another version is similar to loading a specific version. As an example, if you want to switch from the current loaded `R-bundle-Bioconductor` to an more recent one:

```
module switch R-bundle-Bioconductor/3.9-foss-2019a-R-3.6.0 R-bundle-Bioconducto
```

This, more compact syntax will fortunately also work:

```
module switch R-bundle-Bioconductor R-bundle-Bioconductor/3.11-foss-2020a-R-4.0
```

Loading multiple modules simultaneously

A module loads all necessary dependencies along with the parent module. This can lead to conflicts when different modules have conflicting dependencies, e.g. they depend on conflicting versions of the tool chains (`foss, GCC, GCCcore`). We recommend loading the modules sequentially and resolve any conflicts. When modules get reloaded with a version change this indicates a conflict and will break the dependencies of the previously loaded module(s), e.g.:

```
module load R-bundle-Bioconductor/3.11-foss-2020a-R-4.0.0
module load libGLU/9.0.0-foss-2018b

The following have been reloaded with a version change:
  1) FFTW/3.3.8-gompi-2020a => FFTW/3.3.8-gompi-2018b
  2) GCC/9.3.0 => GCC/7.3.0-2.30
  3) GCCcore/9.3.0 => GCCcore/7.3.0
  4) GMP/6.2.0-GCCcore-9.3.0 => GMP/6.1.2-GCCcore-7.3.0
```

Purge modules

We recommend using

```
module purge
```

prior to any `module load` commands in job scripts, to prevent inheriting environment variables set by `module` commands in the shell used when submitting the job.

Help

A help text exists for most of the modulefiles. To view it, type:

```
 module help modulefile/version
```

For further documentation, please refer to the Environment Modules homepage.

Personalize default modulefiles

You may add loading of modulefiles to your `.bash_login` file to make sure your favorite modulefile is always loaded when logging in. For instance, to always have the `MATLAB/2020b` modulefile load on login, add:

```
module load MATLAB/2020b
```

to the `.bash_login` file.

UNIVERSITY
OF OSLO

Managing data on Colossus

Available file systems on Colossus

File system	Path	Recommended use
Colossus project directory	`/cluster/projects/pXX`	Software, job configurations, input files, processing data in a job. DO NOT use it for long-term data storage unless you have backup enabled. 1 TiB * disk quota
Home directory	`/tsd/pXX/home/user` or `$HOME`	Software, job configurations. DO NOT use it for processing data in a job. 100 GiB disk quota
Scratch	`/cluster/work/jobs/`*`jobid`* or `$SCRATCH`	Processing data in a job, chkfile to retain output data.
Local disk space	`$LOCALTMP`	Processing data in a very high I/O job. 100-200 GiB disk quota

(*) Additional disk space can be requested.

Colossus project directory

Each project with access to Colossus has a project directory on the high-performance parallel filesystem: `/cluster/projects/pXX`, where `pXX` is your project number. By default, the disk quota of the project directory is 1 TiB.

By default, there is no backup of the data stored in `/cluster/projects/pXX`, but snapshots are available for the last 7 days in a `.snapshot` subdirectory. If you need to work on several TiB of data and moving the data between the `/tsd/pXX/data/durable` (HNAS) and the Colossus project directory is not convenient, then we strongly recommend to have backup of `/cluster/projects/pXX`. Backup is a service on demand. Project administrators can order it by contacting TSD. Pricing for the backup can be found here. A list of projects with backup enabled can be found here.

Serving and keeping track of a parallel cluster file system is a complicated task. While the hardware is high-end, once in a while the GPFS software will get in trouble. When this happens users might experience delay when doing simple commands as "ls" or even hangs. Usually these problems lasts for a very short time, but if there is a serious problem there will be an announcement on the TSD Operational Log and on the Colossus Users email list.

On the submit host

On the submit hosts this file system is available using NFSv4 with Kerberos authentication (see below) and mounted under `/cluster/p/pXX/cluster` with symlinks pointing to `/cluster/projects/pXX` for historical reasons. We advice to use references to `/cluster/projects/pXX` in your scripts.

Kerberos authentication

Access to the `/cluster/projects/pXX` file system over NFSv4 requires a valid Kerberos ticket. A valid ticket will grant you access to the `/cluster/projects/pXX`, whereas an expired, invalid ticket will deny access.

If you connect to the submit host (via ssh or PUTTY) you'll automatically be granted a ticket for a 10 hour session which is automatically renewed up to a week. If the ticket expires after a week, you'll have to log out and back in to restore access to `/cluster/projects/pXX`. This is the preferred method of obtaining a ticket.

You can also manually obtain a ticket using the `kinit` command. However, this ticket will **not** renew and expire after 10 hours. We advice you **not** to use this command. Obtaining automatic and manual tickets at the same time may result in `permission denied` errors if one of the tickets expires while the other is still valid.

Kerberos authentication requires password authentication, hence you'll **not** be given a ticket if you connect using ssh keys. Please disable ssh keys on the submit host and use password authentication instead.

You can list your current ticket status using:

```
klist
```

In the initial stage it may only list the entry for the Ticket Granting Ticket (TGT) indicating a successful password verification:

```
-bash-4.2$ klist
Ticket cache: FILE:/tmp/krb5cc_7927__Vx2FH
Default principal: p11-bartt@TSD.USIT.NO

Valid starting       Expires              Service principal
07/06/2020 14:50:43  07/07/2020 00:50:43  krbtgt/TSD.USIT.NO@TSD.USIT.NO
        renew until 07/13/2020 14:50:21
```

Once you access `/cluster/projects/pXX`, entries for access to `nfs/ess01.tsd.suit.no` will be added, indicating successful authorization to the nfs mount:

```
-bash-4.2$ klist
Ticket cache: FILE:/tmp/krb5cc_7927_Vx2FH
Default principal: p11-bartt@TSD.USIT.NO

Valid starting       Expires              Service principal
07/06/2020 14:50:43  07/07/2020 00:50:43  krbtgt/TSD.USIT.NO@TSD.USIT.NO
        renew until 07/13/2020 14:50:21
07/06/2020 14:52:44  07/07/2020 00:52:44  nfs/ess01.tsd.usit.no@TSD.USIT.NO
        renew until 07/13/2020 14:50:21
```

If your ticket expires, you'll need to re-authenticate to obtain a new ticket. Log out and back in. If you get `permission denied` or cannot list the contents when you access a directory for the first time, but you do have a valid ticket, Kerberos authentication may have been delayed and will succeed if you try again.

On Colossus

On Colossus the file system is concurrently available on all compute nodes using the global parallel file system (GPFS). On the compute nodes its mounted under `/gpfs`, with symlinks pointing to `/cluster/projects/pXX` to resemble the NFS mount points on the submit hosts. On the compute nodes, the communication with this storage-system is over ultra-fast 56 Gbps Infiniband.

Home directory

Each user has a home directory (`$HOME`) on a low-performance file system. By default, the disk quota for the home directory is 100 GiB (see below).

The home directory is backed up regularly (see below), but anything inside directories named *nobackup* is skipped. Backup is slow and expensive, so please put temporary files, files that can be downloaded again, installed software and other files that can easily be recreated or do not need to be backed up inside a *nobackup* directory.

Also note that one is not supposed to use the home directory as read/write area for jobs, especially not I/O intensive jobs. Use the scratch area for that (see below).

Scratch disk space

While a job runs, it has access to a temporary scratch directory on `/cluster/work/jobs/jobid` which resides on the high-performance GPFS filesystem. The directory is individual for each job, is automatically created when the job starts, and is deleted when the job finishes (or gets requeued). *There is no backup of this directory*. The name of the directory is stored in the environment variable `$SCRATCH`, which is set within the job script.

In general, jobs should copy their work files to `$SCRATCH` or `/cluster/projects/pXX` and run there. This is especially important for I/O intensive jobs. The scratch disk is faster than the home directory disk, and running I/O intensive jobs in `$HOME` slows down not only the job, but also interactive work for other users.

If you need to access the scratch directory from *outside* the job (for instance for monitoring a running job), the directory is `/cluster/work/jobs/jobid`, where `jobid` is the job id of the job in question.

Local disk space

For very intensive IO, it can be useful to use the local drives on the compute nodes. The path to the directory is stored in the environment variable `$LOCALTMP`. The compute and GPU nodes have 100 GB and 200 GB of local storage, respectively. Add the following to your batch script to request (e.g. 20 GB) of local temporary storage on the node:

```
#SBATCH --gres=localtmp:20
cleanup cp $LOCALTMP/outputfile $SLURM_SUBMIT_DIR
```

Disk quotas

All projects receive a 1 TiB Colossus project directory and a 100 GiB home directory disk space quota. Projects may apply for extra disk space from Sigma2.

On Colossus and the submit hosts, the UNIX `df` utility can be used to query disk usage on the GPFS file system. A disk will be full if either the space or the number of inodes (files) runs out. To query disk space, use:

```
$ df -h /cluster/projects/p11
Filesystem              Size  Used Avail Use% Mounted on
ess01.tsd.usit.no:/p11  1.0T  966G   59G  95% /cluster/p/p11
```

To query the number of inodes, use:

```
$ df -ih /cluster/projects/p11
Filesystem             Inodes IUsed IFree IUse% Mounted on
ess01.tsd.usit.no:/p11   1.1G   22M  1.1G    3% /cluster/p/p11
```

Data transfer to/from Colossus

To copy data from the the `/tsd/pXX/data/durable` project directory to the Colossus `/cluster/projects/pXX` project directory you'll have to connect to the submit host which has access to both.

Secure copy with `scp` is a common way of transferring files. However, if the communication fails half way through `scp` will not pick up where it stopped. In addition the transfer is not compressed, which is an advantage for bandwidth limited transfers. The `rsync` utility is the preferred way of moving large files.

```
rsync [OPTION]... SRC [SRC]... [USER@]HOST:DEST
```

Please see the `rsync` manual page (`man rsync`) for more information.

Data compression

Millions of small files pose a challenge for GPFS and is to be avoided. If possible pack the small files in archives making operations on them easy. One way is to copy the archive to `$SCRATCH` or `/cluster/projects/pXX`, unpacking them there and working on the local file tree.

A standard unix/linux utility is `gzip`. Read the man pages for more information.

```
gzip file.dta
```

This will produce a file `file.dta.gz`, hopefully a much smaller file. Not all types of data compress equally well. Text compresses well, jpg pictures not well at all. For files that are to be unpacked at Windows machines the `zip` utility can also be used. One limitation of `zip` is that neither the input files nor the resulting archive can be larger than 4 GB. For files larger that 4 GB use `gzip`. Giving it a numeric argument like `-9` forces higher compression at the expense of longer compression time. A more efficient alternative is `bzip2`.

To unpack the file :

```
gunzip file.dta.gz
```

This will result in the original file in it's uncompressed form.

Backup and restore

See here for more information on backup and restore options.

UNIVERSITY
OF OSLO

Dedicated Resources

You can find out if your project is entitled to dedicated resources by running the command:

```
$ qsumm
```

It will return a list of reservations and the number of cpus that is connected to the reservation. If it only shows something like this

```
--------------------------------
Account          Limit  nRun nPend
--------------------------------
p11               1536     0     0
Total             1536  1506     0
--------------------------------
Reservation foo:
p11_foo            416     0     0
Total              416   311  4712
--------------------------------
```

your project **has** access to dedicated resources in the reservation *foo*. If your project has access to several reservations, each of them will be listed in the output of `qsumm`.

You will be able to run in your dedicated nodes using

```
$ sbatch --account=pNN_reservationname ...
```

So, for instance, if a user in the p11 project wants to use the *foo* reservation, **instead** of using `--account=p11`, she should use `--account=p11_foo`. (Note: there is no need to add `--reservation=foo`; it will be added automatically when `--account=p11_foo` is used.)

Per April 1st 2021, projects that do not have a Sigma2 quota can submit to the `tsd` reservation using

```
$ sbatch --account=pNN_tsd ...
```

To prioritize among jobs on dedicated resources, please use "nice" settings. By default everyone gets top priority on the nice settings, so each project must internally agree on nice settings. See https://slurm.schedmd.com/sbatch.html.

UNIVERSITY
OF OSLO

R

Table of Contents

Description

R is a software environment for statistical computing and graphics. See R's home page and documentation for details.

Home page

http://www.r-project.org/

Documentation

http://cran.r-project.org/manuals.html

License

R is available as Free Software under the terms of the Free Software Foundation's GNU General Public License in source code form.

Usage

Use

```
module avail R
```

to see which versions of R are available. Use

```
module load R/version
```

to get access to R.

The best way of running R in a script, is to use

```
Rscript yourscript.R
```

or, if you do **not** want R to read and save .RData:

```
Rscript --no-save --no-restore yourscript.R
```

For details see:

```
Rscript --help
```

Installed packages

Several packages have been pre-installed on Colossus. Do

```
library()
```

in R to see what is available. If you start R with any of the switches `--vanilla` or `--no-environ`, you will *not* have access to the pre-installed packages.

Installing packages

We encourage you to install the packages you need yourself. That way you can upgrade packages whenever you need. You can install packages in R with:

```
> install.packages("PackageName")
```

In TSD, Linux hosts will automatically retrieve packages from our internal CRAN/Bioconductor mirrors as outlined here.

Checkpointing

If you run calculations that take more than a day, it is usually a good idea to use checkpointing, i.e, save the state of the calculations at intervals and use these saved values whenever the script is restarted. For an iteration based simulation, checkpointing can be implemented with something like this:

```
## Load any R packages
## ...

## Load the checkpointing file, if it exists:
checkFile <- "checkpoint.RData"
tempFile <- "tempCheckpoint.RData"
if (file.exists(checkFile)) {
 load(checkFile)
 cat("Resuming after iteration", iter, "\n")
 iter <- iter + 1                      # Important to avoid infinite loop
} else {
 ## Set up the simulation
 ## ...

 Nsims <- 1000
 iter <- 1                             # The first iteration
}

## Do the simulation
if (iter <= Nsims) {                    # In case we were interrupted after
                                       # the last iteration
 for (iter in iter:Nsims) {            # Note _iter_:Nsims
   cat("Iteration", iter, "\n")
   ## Do iteration iter
   ## ...

   ## Save the results of the iteration.  (By first saving to a temporary file
   ## and then renaming it into the checkpointing file, we guard against being
   ## interrupted while saving.)
   save.image(tempFile)
   file.rename(tempFile, checkFile)
 }
}

## Do final stuff and save results
## ...

## Clean up (_after_ saving the results)
if (file.exists(checkFile)) file.remove(checkFile)
```

Parallel jobs

There are many ways of parallelising jobs in R. Since version 2.14.0, R comes with a package called `parallel`, which provides perhaps the simplest interface for setting up parallel jobs. The following is a simple illustration of how to use the package to run a calculation as an MPI job on Colossus:

R script (test_parallel_MPI.R):

```
library(parallel)

## Start the MPI worker processes:
numWorkers <- as.numeric(Sys.getenv("SLURM_NTASKS")) - 1
myCluster <- makeCluster(numWorkers, type = "MPI")

## If needed, load any libraries, etc. on the workers:
# Run library(car) on all workers:
#clusterEvalQ(myCluster, library(car))
# Export variables var1 and var2 to all workers:
#clusterExport(myCluster, c("var1", "var2"))

## Define a worker function:
workerFunc <- function(n) {
    return(n^2)
}
## and a list or vector to use it on:
values <- 1:100

## Apply workerFunc to values in parallel:
results <- parLapply(myCluster, values, workerFunc)
print(unlist(results))

## Exit cleanly:
stopCluster(myCluster)
Rmpi::mpi.finalize()
# or Rmpi::mpi.quit(), which quits R as well
# (Note that "mpi.exit()" and "mpi.quit()" no longer works.)
```

Slurm script:

```
#!/bin/bash

#SBATCH --ntasks=NumberOfProcesses
#SBATCH --account=MyAccount --time=hh:mm:ss --mem-per-cpu=megabytes

module load R/4.0.0-foss-2020a

mpirun -n 1 R --slave < test_parallel_MPI.R
```

In order to learn more about the `parallel` package, or parallel R jobs in general, we suggest reading the vignette that comes with the `parallel` package, or the book *Parallel R* by Q. Ethan McCallum and Stephen Weston.