Exercises

a) Log in to Taito either with your training or CSC user account, either from a terminal (with X11 forwarding) or using NoMachine client

b) Go to working directory and download the exercises file

```
$ cd $WRKDIR
$ curl https://...path-from-course-page -o input-data.tar
```

b) Uncompress the exercises file

```
$ tar xvf input-data.tar
```

Note. The not bolded dollar sign "$" at the beginning of some lines is not to be typed, but marks the command prompt. What follows is to be typed on the command prompt. Text in `italics` is not a command and you can choose what to put there (but you need to be consistent later).

1. Simple batch job script

a) Create a batch job script that prints the compute node on which it is running.

Using nano editor (use whichever editor you like):

```
$ nano test_hostname.sh
```

Put this into the file.

```
#!/bin/bash -l
#SBATCH -J print_hostname
#SBATCH -o output_%j.txt
#SBATCH -e errors_%j.txt
#SBATCH -t 00:01:00
#SBATCH -p test
#SBATCH -n 1
#SBATCH --mem-per-cpu=500
echo "This job runs on the host: "; hostname
```

To exit from the nano editor:

```
CTRL+O (enter); CTRL+X (confirm save)
```

Submit the batch script to Taito

```
$ sbatch test_hostname.sh
```

b) Check the job status.

In the following command replace *<your username>* with the training account or your CSC username, which ever you used to log in to Taito. If you are not sure which it is, you can check it with **`whoami`** or with this command **`echo $USERNAME`**).

```
$ squeue -u your username
```

c) What and where did the job print out?

Note, here you need to replace the *JOBID* with the ID your job was given.

```
$ less output_JOBID.txt (type q to quit)
$ less errors_JOBID.txt (type q to quit)
```

2. Simple R job

Run a simple R job from a script. The script will fit a straight line through a file containing some x,y value pairs.

a) Create a script containing the R commands.

Go to the directory "r-job", where you have the data (a file called **`data.csv`**). In that directory, create the following R script file (R commands to be executed). Name the file "**`fit.R`**"

```
mydata <- read.csv("data.csv")
fit <- lm(y~x,mydata)
fit$coefficients
```

b) Set up the R environment with the module command

```
$ module load r-env
```

c) Run the script with

```
$ R --no-save --no-restore -f fit.R
```

d) Did the job succeed? What are the fit coefficients?

3. Simple R job as a batch job

Now run the previous R script as a batch job.

a) Create a batch job script, which will submit the job to the queue.

Start with the batch script file from the first exercise. Copy it to the current folder and edit it as follows. Add to the batch script file the **`module load ...`** command (after the #SBATCH lines) and then the command to run the R-script (the commands you gave in the command line in the previous exercise, you can remove the hostname command if you like). The other SLURM requirements can be as in the previous example.

b) Submit the batch script with

```
$ sbatch your_script_name
```

c) Check the job results.

Did the job succeed? Where are the fit constants?

4. Run tens of R batch jobs as an array job

In this example, we will repeat the previous fitting job for 20 datasets using the array job functionality of SLURM. Note, that for such short jobs it would not make sense to run them as separate batch jobs, but you could loop over them in one job or better inside the R script.

a) Prepare a list of files to process.

Go to the folder named **r-array**. Create there a file called **datanames.txt**. This file will contain the names of all those files that will be used as input to the fitting. Run the following commands to create it.

```
$ cd data_dir
$ ls * > ../datanames.txt
```

b) Write the R script, that will do the fitting.

Go back to the r-array folder, create a script named **modelscript.R** and put the following commands to it (you can copy the previous script and edit that, or start from scratch).

```
dataname <- commandArgs(trailingOnly = TRUE)
mydata <- read.csv(paste0("data_dir/",dataname))
fit <- lm(y~x,mydata)
write(fit$coefficients,
     file=paste0("result_dir/",dataname,"_result.txt"))
```

The first line will extract from the batch command the name of the dataset to be fitted. The next line reads that data into the variable mydata. Then we fit, like in the previous example, and finally write the coefficients into a file.

c) Create a batch script to submit the job.

Name it **R_array.sh**. Copy the contents from the previous example. Add the following line among the other lines starting #SBATCH:

```
#SBATCH -a 1-20
```

It will ask SLURM to run an array of 20 jobs. Edit the output and error files to go to their own directories and files by editing/adding:

```
#SBATCH -o out/output%a.txt
#SBATCH -e err/errors%a.txt
```

After the line with module `load r-env`, add the following line

```
dataname=$(sed -n "$SLURM_ARRAY_TASK_ID"p datanames.txt)
```

and replace the line to run the R command into:

```
R --no-save --no-restore -f modelscript.R --args $dataname
```

You should now have:

1) `datanames.txt`, which has the names of your datafiles
2) `modelscript.R`, which contains the R code to do the fitting
3) `R_array.sh`, which is the batch script to submit the job
4) (and the folders out, err, data_dir, result_dir which were there already)

d) run the batch script with

```
$ sbatch R_array.sh
```

You should get the fit coefficients in separate files in the **result_dir**. Let's now use interactive R to look at the results.

e) Initialise R and run it

Go back to the directory where you have the analysis script (**analyse.R**). Then

```
$ module load r-env #(as you did this already, there is no need to repeat it unless you're in a new shell.  How do you know if it was already loaded?)
$ R
```

f) Collect the results and plot them.

In the R shell that opens, write

```
> source("analyse.R")
```

This will run (source) the script contents. The original data was created by calculating the y values by y=2x + some random noise.

e) How do the fit coefficients match that?

5. Copying files to hpc_archive

You can access hpc_archive only from Taito and Sisu. You can access IDA also from your computer after installing the required tools.

Log in to Taito. Check the contents of your hpc_archive:

```
$ ils
```

Show the directory that you're in in hpc_archive:

```
$ ipwd
```

Show the directory that you're in in taito:

```
$ pwd
```

Create a directory in hpc_archive

```
$ imkdir test
```

Move to test directory in hpc_archive

```
$ icd test
```

Confirm where you are in hpc_archive

```
$ ipwd
```

Copy (put) a file to the test directory in hpc_archive:

```
$ iput <filename>
```

Confirm that the file is in hpc_archive

```
$ ils
```

Copy the file back from hpc_archive, but to a local folder called `localtest`

```
$ mkdir localtest
$ cd localtest
$ iget <filename>
```

6. Archive a file

Make a new directory in hpc_archive home directory (not under test) called `mysafe`. Copy there your files, e.g., `test_hostname.sh` and `R_array.sh` from previous exercises, but not as two separate files. Compress and archive them first to a single file, e.g., `fit.tar.gz` using a command `tar -zcvf files to include`, and copy it to `mysafe`.

7. Batch job with thread parallelization

Some applications can be run in parallel to speed them up. In this example you run the HMMER software to describe and analyze related or similar sequence areas both in serial and parallel to see if the jobs speed up.

HMMER uses a database that is already installed, but the protein sequences you want to study need to be copied first to be used as input:

```
$ cp /appl/bio/hmmer/example.fasta .
```

Let's first run the job with just one core. Copy one of the old batch scripts to current directory, and change / add the following items in it:

1) Output to ***out_%j.txt***
2) error to ***err_%j.txt***
3) run time 10 minutes

4) load the **hmmer** -module
5) run command:
```
hmmscan $HMMERDB/Pfam-A.hmm example.fasta >
example_1.result
```

Submit the job with:

```
$ sbatch jobscript.sh
```

Submitting the job echoes the SLURM job id number to the screen, but that is also shown in the output and error filenames (**out_<SLURM_JOBID>.out**). Check if the job is running with

```
$ squeue -u <your username>
```

Or

```
$ squeue -j <SLURM_JOBID>
```

Once the job is finished you can check how much memory and time it used:

```
$ sacct -j <SLURM_JOBID> -o elapsed,reqmem,maxrss
```

Did you reserve a good amount of memory? (not excessively too much, but enough to not be close to memory running out and terminating the job).

Now, let's try with 4 cores. At this point we'll also switch to using the environment variable **$SLURM_CPUS_PER_TASK** to avoid mistakes and the need to change that in many places. Add this line to the batch script:

```
#SBATCH --cpus-per-task=4
```

Change the run command to:

```
hmmscan --cpu $SLURM_CPUS_PER_TASK $HMMERDB/Pfam-A.hmm \
     example.fasta > example_$SLURM_CPUS_PER_TASK.result
```

As you've asked for 4 cpus per task, the environment variable **`$SLURM_CPUS_PER_TASK`** evaluates to 4 when the script is run, and you only need to change the number on the **`#SBATCH`** line.

Submit the job and check with the **`sacct`** command how long it took to run the hmmer job and how did the memory usage change and try to answer these questions:

a) Does it make sense to use 4 cores instead of 1?

b) Was the memory reservation ok?

c) Does it make sense to use more than 4 cores?

d) How to speed up the job?

8. Batch job memory consumption

Create a new R-script (like in exercise 3) named **`mem-test.R`**. It should have the following contents:

```
dim=10
dim_end=1000
while (dim < dim_end) {
 mat <- matrix(rnorm(dim*dim), dim)
 print("passed dimension")
 print(dim)
 dim=dim*2
}
print("all done")
```

Variable `dim` is the dimension of the square matrix, which will be filled with normal distributed random numbers. The script increases the dim until it exceeds `dim_end`. Make a new batch script to run your R-script `mem-test.R`. In addition to the resource requests your script needs to load the R-environment and then run the R-script (as in exercise 3). Submit the script with the `sbatch` command.

Once the job has been completed (how can you check if it is running or queuing?), check with `sacct` or `seff` how much memory was used as in the previous exercise. If the job completed successfully, increase the `dim_end` variable in your script *i.e.* make a bigger matrix and rerun the job. Note also the time it takes to run the job. How does the time and memory needed by the job depend on the number of elements in the array?

max(dim)	# of elements	Time [s] (Elapsed)	Memory used (MaxRSS)

a) How big matrix is needed to exceed the default allowed memory of the batch job?

9. Scaling test for an MPI parallel job

Before running parallel jobs it is important to determine how the job scales. It does not make sense to use many cores, if this does not speed up the job. The speedup depends on the application but also input. In this tutorial we'll use CP2K which can use thousands of cores for certain job types and model systems (but not for this one). Therefore, it is important to test the scaling for each different job type (or model system type). The idea is not to run production simulations, but quick short simulations (i.e. using the actual production system) but only for like 1-5 minutes, which will be enough to reveal the performance.

First copy the input file to your working directory:

```
$ cp /appl/chem/cp2k/tests/QS/benchmark/H2O-32.inp .
```

Then create the following batch script and submit it with **`sbatch`**

```
#!/bin/bash -l
#SBATCH -t 00:15:00
#SBATCH -J cp2k-1
#SBATCH -p serial
#SBATCH -o ocp2k_%j.txt
#SBATCH -e ecp2k_%j.txt
#SBATCH --mem-per-cpu=1000
#SBATCH -n 1

module load cp2k-env/5.1
```

```
srun cp2k.popt H2O-32.inp > H2O-32_$SLURM_NPROCS.out
```

After running the job with one core, edit the batch script to use more cores/mpitasks (e.g. 2,4,8,16, ... this is the `-n` flag) and rerun the job. The output files will be named with the number of cores used to run them `($SLURM_NPROCS)`. Instead of the serial partition you can also use the `test` partition. If you ask for more than 24 cores, you need to switch to `parallel` partition. In that case it also makes sense to limit the number of Nodes so `#SBATCH -N 2-2` which asks for a minimum and maximum of 2 nodes, so that the job is not spread onto more nodes than necessary (creates unwanted communication overhead and fragments the allocations on compute nodes).

With the following command you can sum the time spent at different steps for each job.

```
$ grep "CPU TIME" H2O-32_1.out | awk '{a+=$5;print a}'
```

Check with `seff JOBID` how much memory the simulation used (compare `sacct` and `seff` output!) and fill in the data in the following table. If your code does not print out either the performance or used time, you can use the `sacct` command (`sacct -j JOBID -o elapsed,alloc,maxrss`)
A rule of thumb for good scaling is that when you double the resources, the job should speed up at least 1.5 fold. Ideally it would speed up linearly with resources i.e. 2 fold.
The speedup at some core counts may be off the trend. There may be variation due to load on the system or because the code/system does not parallelize well or be able to distribute the computational load for that particular core count. Sometimes rerun helps to sort out an outlier. If you know the code parallelizes well, there is no point to start testing from 1 core, but where you think the code runs well. Bad speedup will show if optimal number of cores is less in any case.

# cores	Time [s] (Elapsed)	Speedup	Ideal speedup	Memory used
		-	-	

a) How many cores can you use efficiently? (i.e. how far does the job scale)

b) How does the required memory depend on the number of cores?

c) Why are the elapsed times reported by sacct slightly different to the sum of "CPU TIME" lines?

d) Why did we limit the acceptable nodes to Sandy Bridge? What else could we have done?

e) If we want to run a different cp2k system do we need to rerun the scaling test?