Running molecular dynamics simulations with GROMACS on LUMI

Authors: Szilárd Páll and Andrey Alekseenko (adapted from doi:10.6084/m9.figshare.22303477 (https://doi.org/10.6084/m9.figshare.22303477))

Introduction

🎯 **Learning goals**

- Get familiar with the GROMACS tools used in the exercises.
- Understand common features of the `mdrun` command line.
- Understand key parts of the `mdrun` log file structure.

Software: GROMACS 2023

The GROMACS simulation engine

GROMACS (https://www.gromacs.org) is a molecular simulation package which comes with a molecular dynamics simulation engine (`mdrun`), a set of analysis tools, and the gmxlib Python API.

GROMACS is highly flexible and can be built in various ways depending on the target hardware architecture and the parallelization features enabled. The GROMACS features and dependencies enabled at compile-time are shown in the *version header* which is listed by invoking the `gmx -version` command as well as at the top of the simulation log outputs. All functionalities of the GROMACS package, the simulation engine and tools, are provided in the `gmx` program through subcommands. The program can have suffixes, e.g. MPI builds are typically installed as `gmx_mpi` .

In this tutorial, we will use a version of GROMACS that has already been built on the LUMI-G cluster, but if you wish to install GROMACS on your own system, instructions for many different hardware configurations are available at GROMACS documentation (https://manual.gromacs.org/current/install-guide/index.html).

GROMACS parallelization overview

Parallelization of MD simulation requires *expressing* concurrent work (multiple computations happening at the same time) and *exposing* it using an implementation with the help of a parallel programming model. To express concurrency within a single simulation in GROMACS we can divide the work using data (e.g. spatial decomposition algorithms), task (e.g. rank specialization for the "separate PME ranks" feature), or ensemble decomposition. The exposed concurrent work can then be mapped to various processing units, like CPU cores or GPU accelerators. GROMACS relies on a hierarchical heterogeneous parallelization using MPI, OpenMP multi-threading, CUDA/SYCL/OpenCL for asynchronous GPU execution, and SIMD for low-level CPU and GPU algorithms.

The data parallelism is used for implementing spatial decomposition (that consist of dividing the simulation system into parts that are as independent as possible) and takes place across MPI ranks using multi-threading on CPUs and fine-grained SIMD-style algorithms (Single Instruction Multiple Data). At the same time, task parallelism is at the heart of the heterogeneous GPU engine and it is also what enables scaling the PME algorithm efficiently by employing rank specialization (https://manual.gromacs.org/documentation/current/user-guide/mdrun-performance.html#separate-pme-ranks).

MD simulation studies can be classified into two major flavors: those that use a single (or a few) long trajectory, and those realying on a larger set of trajectories. Due to the timescale of some biological processes, a single/few very long trajectories might not be enough (and/or it is inefficient) to sample the conformational space. Then, an alternative is to use an ensemble of simulations.

A wide range of algorithms, from free energy perturbation to replica exchange to the accelerated weight histogram method (AWH), rely on (or require) multiple MD simulations which form an *ensemble*. An *ensemble simulation* refers to a set of simulations, where each individual simulation is referred to as *ensemble member* (called "replica" in replica-exchange and walker in AWH). These algorithms provide a source of concurrent work, which simulation workflows can use to parallelize over, and require different levels of coupling between the ensemble members. E.g., standard free-energy calculations (with a pre-determined simulation length) require no communication across the ensemble members, whereas replica-exchange and AWH require exchange of information at regular time intervals. The latter class of methods is referred to as *coupled* ensembles. Depending on the frequency of data exchange, ensembles can be *weakly* or *strongly* coupled (with infrequent or frequent data exchange, resp.). Coupled ensembles are more performance sensitive, hence more prone to be influenced by imbalance (e.g. member simulations running with different throughput). The stronger the coupling the more sensitive the ensemble simulation is to performance bottlenecks.

The `mdrun` simulation tool

In GROMACS, the primary way to run simulations across CPUs and GPUs is to use the command line program `mdrun` . The simulation tool `mdrun` can be invoked as a subcommand of the main program, e.g. `gmx mdrun` . The `mdrun` functionalities available in a specific build depend on the options GROMACS was configured with and can be seen in the version header.

The following list contains key performance-related command line options used in this tutorial:

- `-g LOGFILE` set a custom name for the log file (default `md.log`);
- `-pin on` enable `mdrun` internal thread affinity setting (might override externally set affinities). Note on LUMI externally set affinities are recommended or the LUMI documentation (https://docs.lumi-supercomputer.eu/runjobs/scheduled-jobs/distribution-binding/);
- `-tunepme` / `-notunepme` enable PME task load balancing;
- `-nsteps N` set the number of simulations steps for the current run to `N` (N=-1 means infinitely long runs, intended to be combined with `-maxh`);
- `-maxh H` stop the simulation after `0.99*H` hours;
- `-resethway` reset performance counters halfway through the run;
- `-nb` / `-pme` / `-bonded` / `-update` task assignment options used to select tasks to run on either CPU or GPU.
- `-npme N` set the number of separate ranks to be used for PME (N=-1 is a guess)

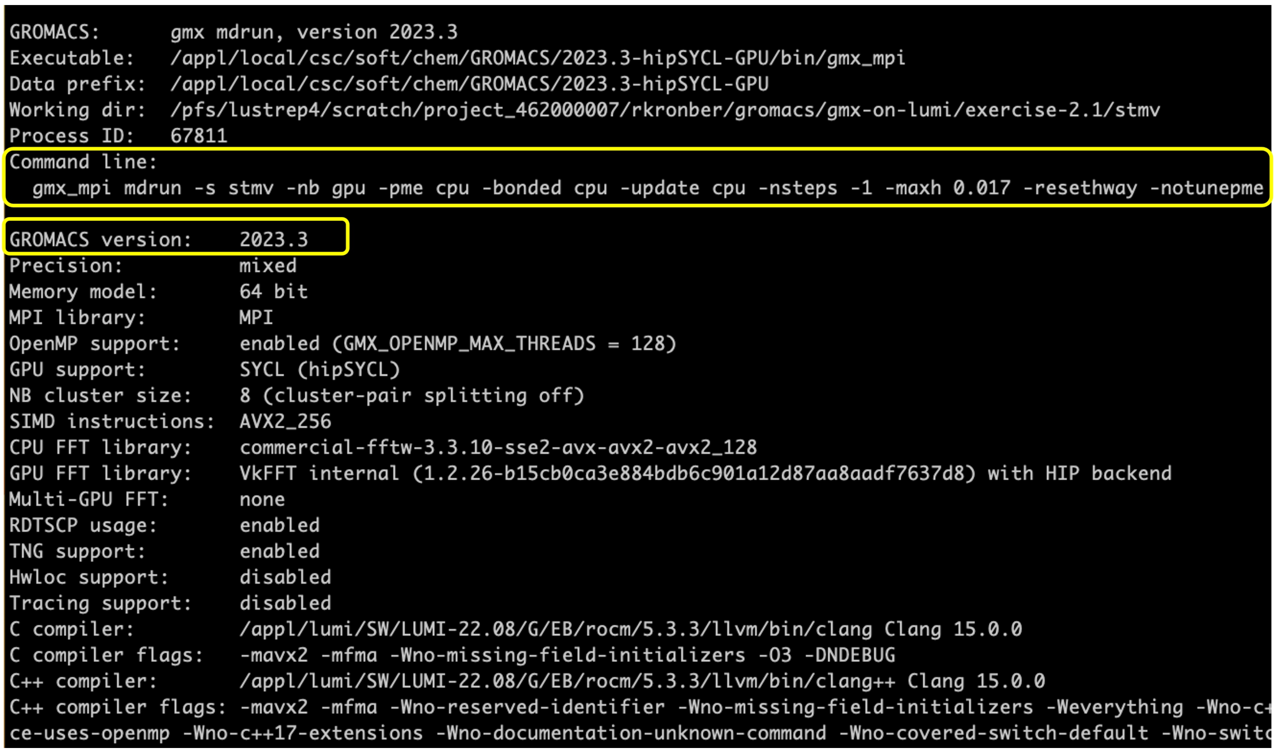
Note that some performance features require using environment variables; see examples in exercise 3.2. Documentation for these can be found in the GROMACS user guide (https://manual.gromacs.org/current/user-guide/environment-variables.html).

For further information on the `mdrun` simulation tool command line options and features, see the online documentation (https://manual.gromacs.org/current/onlinehelp/gmx-mdrun.html).

The `mdrun` log file

The log file of the `mdrun` simulation engine contains extensive information about the GROMACS build, hardware detected at runtime, complete set of simulation setting, diagnostic output related to the run and its performance, as well as physics and performance statistics.

The version header in a GROMACS `mdrun` log:

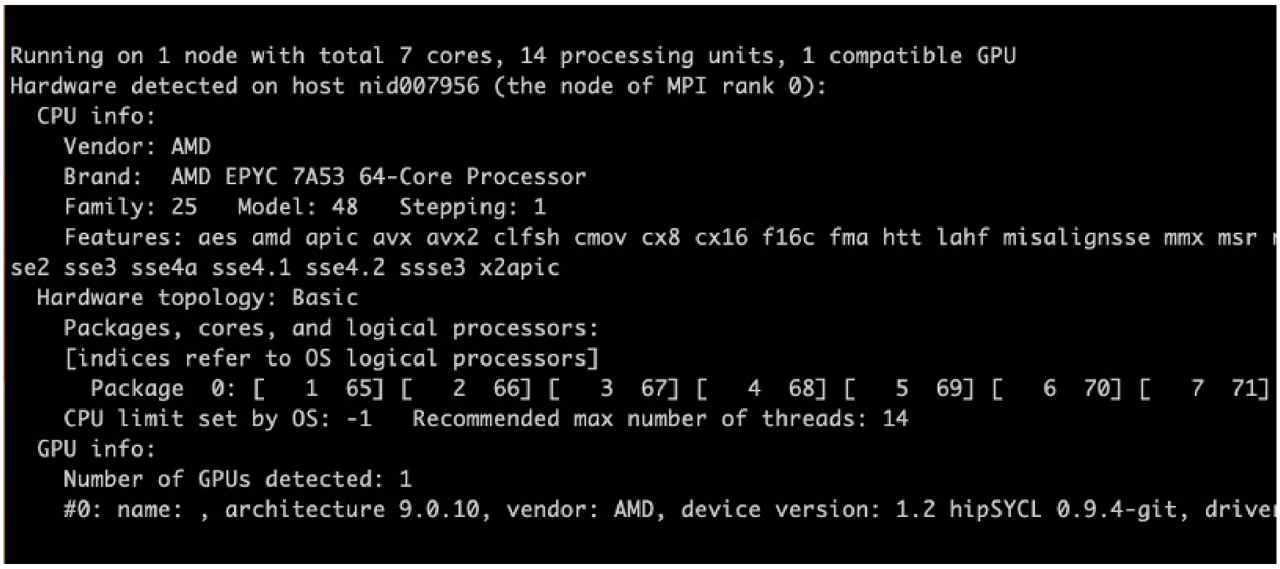


```
GROMACS:      gmx mdrun, version 2023.3
Executable:   /appl/local/csc/soft/chem/GROMACS/2023.3-hipSYCL-GPU/bin/gmx_mpi
Data prefix:  /appl/local/csc/soft/chem/GROMACS/2023.3-hipSYCL-GPU
Working dir:  /pfs/lustrep4/scratch/project_462000007/rkronber/gromacs/gmx-on-lumi/exercise-2.1/stmv
Process ID:   67811
Command line:
  gmx_mpi mdrun -s stmv -nb gpu -pme cpu -bonded cpu -update cpu -nsteps -1 -maxh 0.017 -resethway -notunepme

GROMACS version:    2023.3
Precision:          mixed
Memory model:       64 bit
MPI library:        MPI
OpenMP support:     enabled (GMX_OPENMP_MAX_THREADS = 128)
GPU support:        SYCL (hipSYCL)
NB cluster size:    8 (cluster-pair splitting off)
SIMD instructions:  AVX2_256
CPU FFT library:    commercial-fftw-3.3.10-sse2-avx-avx2-avx2_128
GPU FFT library:    VkFFT internal (1.2.26-b15cb0ca3e884bdb6c901a12d87aa8aadf7637d8) with HIP backend
Multi-GPU FFT:      none
RDTSCP usage:       enabled
TNG support:        enabled
Hwloc support:      disabled
Tracing support:    disabled
C compiler:         /appl/lumi/SW/LUMI-22.08/G/EB/rocm/5.3.3/llvm/bin/clang Clang 15.0.0
C compiler flags:   -mavx2 -mfma -Wno-missing-field-initializers -O3 -DNDEBUG
C++ compiler:       /appl/lumi/SW/LUMI-22.08/G/EB/rocm/5.3.3/llvm/bin/clang++ Clang 15.0.0
C++ compiler flags: -mavx2 -mfma -Wno-reserved-identifier -Wno-missing-field-initializers -Weverything -Wno-c
ce-uses-openmp -Wno-c++17-extensions -Wno-documentation-unknown-command -Wno-covered-switch-default -Wno-swit
```

The version header contains GROMACS version and the command line used for the current run (highlighted). It also contains additional information, like where GROMACS is installed and how it was compiled.

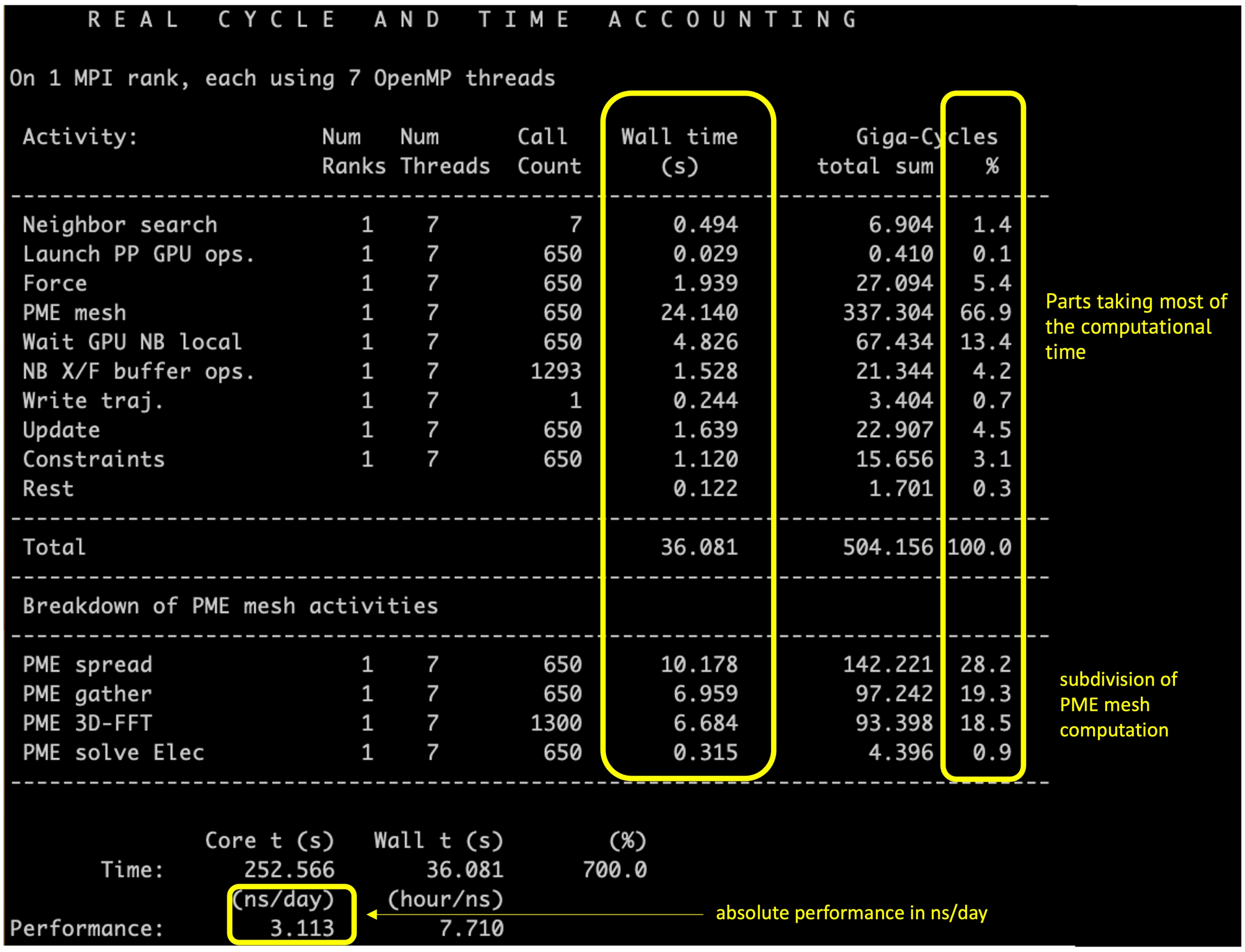
The hardware detection section of the `mdrun` log:



```
Running on 1 node with total 7 cores, 14 processing units, 1 compatible GPU
Hardware detected on host nid007956 (the node of MPI rank 0):
  CPU info:
    Vendor: AMD
    Brand:  AMD EPYC 7A53 64-Core Processor
    Family: 25   Model: 48   Stepping: 1
    Features: aes amd apic avx avx2 clfsh cmov cx8 cx16 f16c fma htt lahf misalignsse mmx msr
se2 sse3 sse4a sse4.1 sse4.2 ssse3 x2apic
  Hardware topology: Basic
    Packages, cores, and logical processors:
    [indices refer to OS logical processors]
      Package  0: [   1  65] [   2  66] [   3  67] [   4  68] [   5  69] [   6  70] [   7  71]
    CPU limit set by OS: -1   Recommended max number of threads: 14
  GPU info:
    Number of GPUs detected: 1
    #0: name: , architecture 9.0.10, vendor: AMD, device version: 1.2 hipSYCL 0.9.4-git, drive
```

This section contains the detailed information about the hardware GROMACS is running on. The first line is a brief summary of the available resources (number of nodes, CPU cores and GPUs), followed by additional details: CPU architecture, CPU topology, and the list of the GPUs.

The performance accounting in the `mdrun` file:



```
     R E A L   C Y C L E   A N D   T I M E   A C C O U N T I N G

On 1 MPI rank, each using 7 OpenMP threads

 Activity:              Num   Num      Call    Wall time         Giga-Cycles
                        Ranks Threads  Count      (s)         total sum    %
-----------------------------------------------------------------------------
 Neighbor search           1    7          7       0.494          6.904   1.4
 Launch PP GPU ops.        1    7        650       0.029          0.410   0.1
 Force                     1    7        650       1.939         27.094   5.4
 PME mesh                  1    7        650      24.140        337.304  66.9
 Wait GPU NB local         1    7        650       4.826         67.434  13.4
 NB X/F buffer ops.        1    7       1293       1.528         21.344   4.2
 Write traj.               1    7          1       0.244          3.404   0.7
 Update                    1    7        650       1.639         22.907   4.5
 Constraints               1    7        650       1.120         15.656   3.1
 Rest                                              0.122          1.701   0.3
-----------------------------------------------------------------------------
 Total                                            36.081        504.156 100.0
-----------------------------------------------------------------------------
 Breakdown of PME mesh activities
-----------------------------------------------------------------------------
 PME spread                1    7        650      10.178        142.221  28.2
 PME gather                1    7        650       6.959         97.242  19.3
 PME 3D-FFT                1    7       1300       6.684         93.398  18.5
 PME solve Elec            1    7        650       0.315          4.396   0.9
-----------------------------------------------------------------------------

               Core t (s)   Wall t (s)        (%)
       Time:    252.566       36.081        700.0
                 (ns/day)    (hour/ns)
Performance:      3.113        7.710
```

Parts taking most of the computational time
subdivision of PME mesh computation
absolute performance in ns/day

This section is printed at the end of the run. The table contains breakdown of the total run time per different kinds of activity. One can see the "Wall time" taken by each activity, as well as its percentage with regard to the total simulation time. Under the table one can find the absolute simulation performance (ns/day).

Simulation input systems

In the following exercises, we will use two different simulation systems:

- large-sized satellite tobacco mosaic virus, STMV (https://en.wikipedia.org/wiki/Tobacco_virtovirus_1) (~ 1 milion atoms) system solvated in a box of TIP3P water molecules, using the CHARMM27 force field.
- medium-sized aquaporin membrane protein (https://en.wikipedia.org/wiki/Aquaporin), a tetrameric ion channel (~110000 atoms) embedded in a lipid bilayer and solvated in a box of TIP3P water using the CHARMM36 force field. We will use the Accelerated Weight Histogram (AWH) algorithm with 32 walkers.

Both systems have previously been used to benchmark GROMACS heterogeneous parallelization and acceleration (https://doi.org/10.1063/5.0018516 (https://doi.org/10.1063/5.0018516)).

The simulation input files (including tpr) can be obtained from:

- Aquaporin (https://a3s.fi/gmx-lumi/aqp-240122.tar.gz)
- STMV (https://a3s.fi/gmx-lumi/stmv-240122.tar.gz)
- or from the folder `/projappl/project_465000934` on LUMI

In exercises 1-3, start with the STMV input, while for exercise 4 Aquaporin. If time allows, feel free to experiment with the other input too.

1. Running your first jobs on LUMI-G

In this first exercise, we will submit our initial jobs on LUMI-G and explore key features and peculiarities of the LUMI system, scheduler, and GROMACS' `mdrun` simulation tool. As simulation system we use STMV input.

We will start with a basic job submission script (batch script) and successively build on it to explore how to correctly request resources using the SLURM job scheduler and to finally arrive to have a script that correctly requests resources on LUMI-G nodes.

💡 The LUMI-G hardware partition consists of 2978 nodes with 4 AMD MI250X GPUs and a single 64 cores AMD EPYC "Trento" CPU. Each MI250X is a multi-chip module with two GPU dies named "AMD Graphics Compute Die" (GCD). For further details on the LUMI architecture, see the LUMI documentation (https://docs.lumi-supercomputer.eu/hardware/lumig/) or how to use GROMACS on LUMI (https://docs.csc.fi/apps/gromacs/).

🎯 **Learning goals**

- Know how to submit GROMACS jobs.
- Get familiar with common SLURM scheduling and `mdrun` command line options.
- Get familiar with the `mdrun` console and log outputs.
- *Bonus*: Understand the impact of using multiple CPU cores/threads on parts of the MD computation.

💡 **The GROMACS log file** Before starting, take a look at the introduction on the GROMACS `mdrun` simulation tool (https://hackmd.io/qvLmXFLCQGScOHhdjS5uQw#The-mdrun-simulation-tool) and at the description of log files.

Exercise 1.1: Launching a first GROMACS simulation on LUMI-G

Now, we will launch a first test simulation on LUMI-G. Make sure you have copied the necessary topol.tpr file into a working directory of your choice under `/scratch/project_465000934` (this is also where your output files will end up, so it's good to keep it organized!), and then create a batch file (with suffix .sh) with the following content:

```
#!/bin/bash
#SBATCH --partition=small-g            # partition to use
#SBATCH --account=project_465000934    # project for billing
#SBATCH --reservation=gromacs_wednesday # workshop reservation
#SBATCH --time=00:10:00                # maximum execution time of
#SBATCH --nodes=1                      # we use 1 node
#SBATCH --ntasks-per-node=1            # 1 MPI rank
#SBATCH --cpus-per-task=1              # 1 CPU core

module use /appl/local/csc/modulefiles
module load gromacs/2023.3-gpu

export OMP_NUM_THREADS=${SLURM_CPUS_PER_TASK}

srun gmx_mpi mdrun \
    -g ex1.1_${SLURM_NTASKS}x${OMP_NUM_THREADS}_jID${SLURM_JOB_ID}
    -nsteps -1 -maxh 0.017 -resethway -notunepme
```

Note the four benchmarking flags used above (`-nsteps` , `-maxh` , `-resethway` and `-[no]tunepome`) are described in the introduction (https://hackmd.io/qvLmXFLCQGScOHhdjS5uQw#The-mdrun-simulation-tool). We use time limited runs here by passing `-maxh 0.017` which sets the run time limit in hours (~one minute); we do that as the simulation throughput significantly changes and we want to avoid either very long wait time or unreliable benchmark measurements due to just a few seconds of runtime.

Submit the job (use the `sbatch` command) and wait until it finishes.

- Take a look at log file (.log) and find the hardware detection and performance table. What are the resources detected and used?

Now try to enable multithreading. To do that, we need to request multiple CPU cores. Edit the job script, change the number of CPU cores and submit a new job.

💡 `SBATCH` arguments provided in the job script header can also be passed on the command line (e.g. `sbatch --cpus-per-task N`) overriding the setting in the job script header. Doing so can allow varying submission parameters without having to edit the job script.

```
#!/bin/bash
#SBATCH --partition=small-g               # partition to use
#SBATCH --account=project_465000934       # project for billing
#SBATCH --reservation=gromacs_wednesday # reservation for 24 Januar
#SBATCH --time=00:10:00                   # maximum execution time of
#SBATCH --nodes=1                         # we run on 1 node
#SBATCH --ntasks-per-node=1               # 1 MPI rank
#SBATCH --cpus-per-task=...               # number cpus-per-task

module use /appl/local/csc/modulefiles
module load gromacs/2023.3-gpu

export OMP_NUM_THREADS=${SLURM_CPUS_PER_TASK}

srun gmx_mpi mdrun \
    -g ex1.1_${SLURM_NTASKS}x${OMP_NUM_THREADS}_jID${SLURM_JOB_ID}
    -nsteps -1 -maxh 0.017 -resethway -notunepme
```

- Compare the log file hardware detection and performance table of the two above runs. What has changed in terms of resources detected and used? Is there anything still missing?

LUMI-G has relatively few CPUs cores per GPU, so making the best use of these is important and can have a strong impact on performance. We will explore this further in Exercise 1.3.

Exercise 1.2: Launching a simple GROMACS GPU run

Note in the log files of previous exercise that GPUs are **not** detected. Now we learn how to request GPUs. Use the job script below to submit a job using one GPU.

```
#!/bin/bash
#SBATCH --partition=small-g           # partition to use
#SBATCH --account=project_465000934    # project for billing
#SBATCH --reservation=gromacs_wednesday # reservation for 24 Januar
#SBATCH --time=00:10:00                # maximum execution time of
#SBATCH --nodes=1                      # we run on 1 node
#SBATCH --ntasks-per-node=1            # 1 MPI rank
#SBATCH --cpus-per-task=7              # number cpus-per-task
#SBATCH --gpus-per-node=1              # New line! Get 1 GPU device

module use /appl/local/csc/modulefiles
module load gromacs/2023.3-gpu

export OMP_NUM_THREADS=${SLURM_CPUS_PER_TASK}

srun gmx_mpi mdrun \
    -g ex1.2_${SLURM_NTASKS}x${OMP_NUM_THREADS}_jID${SLURM_JOB_ID}
    -nsteps -1 -maxh 0.017 -resethway -notunepme
```

- Look at the log file hardware detection and performance table: what are the resources detected and used?
- Compare the log file of Ex 1.1 and 1.2. Has the performance changed?

Bonus Exercise 1.3: Explore the use of CPUs and OpenMP multi-threading

In this exercise, we will use only the CPUs of the LUMI-G nodes to explore how the different computational tasks perform with OpenMP multi-threading. Use the job script below to submit a job using only CPUs. Note that you have to fill in a value for `--cpu-per-task`.

```
#!/bin/bash
#SBATCH --partition=small-g            # partition to use
#SBATCH --account=project_465000934    # project for billing
#SBATCH --reservation=gromacs_wednesday # reservation for 24 Januar
#SBATCH --time=00:10:00                # maximum execution time of
#SBATCH --nodes=1                      # we run on 1 node
#SBATCH --ntasks-per-node=1            # 1 MPI rank
#SBATCH --cpus-per-task=...            # number cpus-per-task

module use /appl/local/csc/modulefiles
module load gromacs/2023.3-gpu

export OMP_NUM_THREADS=${SLURM_CPUS_PER_TASK}

srun gmx_mpi mdrun \
    -g ex1.3_${SLURM_NTASKS}x${OMP_NUM_THREADS}_jID${SLURM_JOB_ID}
    -nsteps -1 -maxh 0.017 -resethway -notunepme
```

- Modify the script varying the number of CPU cores used (`--cpus-per-task`) and submit runs with each new setting.

- Look at the `mdrun` log file output (the files will be named `ex1.3_1xN_jIDXXXXXX.log`).
- How does the absolute performance (ns/day) change when increasing the number of cores used?
- How does the wall-time of various computations change with the thread count (e.g. "Force", "PME mesh", "Update" tasks)?

▼ Help with the solution

Sample log files for the exercise session.:

- ex 1.1 (https://github.com/Lumi-supercomputer/gromacs-on-lumi-workshop/tree/main/Exercise-1.1/STMV)
- ex 1.2 (https://github.com/Lumi-supercomputer/gromacs-on-lumi-workshop/tree/main/Exercise-1.2/STMV)
- ex 1.3 (https://github.com/Lumi-supercomputer/gromacs-on-lumi-workshop/tree/main/Exercise-1.3/STMV)

2. GPU accelerated simulations

🎯 **Learning goals**

- Understand how the GROMACS heterogeneous parallelization allows moving tasks between CPU and GPU and how that impacts performance.
- Understand the difference between force-offload and GPU-resident modes.
- *Advanced*: Explore the effects of load balancing.

The GROMACS MD engine uses heterogeneous parallelization which can flexibly utilize both CPU and GPU resources. As discussed in the lecture, there are two offload modes:

- In the *force offload* mode, some or all forces are computed on the GPU, but are transferred to the CPU every iteration for integration;
- In the GPU-resident mode, the integration happens on the GPU allowing the simulation state to reside on the GPU for tens or hundreds of iterations. Further details can be found in the GROMACS users guide (https://manual.gromacs.org/current/user-guide/mdrun-performance.html#running-mdrun-with-gpus) and DOI:10.1063/5.0018516 (https://aip.scitation.org/doi/full/10.1063/5.0018516).

In the following exercises, we will learn how moving tasks between the CPU and GPU impacts performance. As simulation system we use the STMV input.

We will be using LUMI-G GPU nodes for submitting single-GPU device jobs (hence using one of the eight in the full compute node); for further details on the architecture and usage see the GPU nodes - LUMI-G (https://docs.lumi-supercomputer.eu/hardware/lumig/).

Exercise 2.1: GPU offloading force computations

The tasks corresponding to the computation of bonded, short and long-range non-bonded forces can be offloaded to a GPU in GROMACS. The assignment of these tasks is controlled by the following `mdrun` command line options:

- (short-range) nonbonded: `-nb ASSIGNMENT`
- particle mesh Ewald: `-pme ASSIGNMENT`
- bonded: `-bonded ASSIGNMENT`

The possible " `ASSIGNMENT` " values are `cpu` , `gpu` , or `auto` .

We use one GPU with CPU cores (an eighth of the node) in a simulation and assess how the performance changes with offloading different force calculations. As a baseline, launch a run first with assigning all tasks to the CPU (as below).

```
#!/bin/bash
#SBATCH --partition=small-g           # partition to use
#SBATCH --account=project_465000934   # project for billing
#SBATCH --reservation=gromacs_thursday # reservation for 25 January
#SBATCH --time=00:10:00               # maximum execution time of
#SBATCH --nodes=1                     # we run on 1 node
#SBATCH --gpus-per-node=1             # we use 1 GPU device
#SBATCH --ntasks-per-node=1           # 1 MPI rank
#SBATCH --cpus-per-task=7             # number cpus-per-task

module use /appl/local/csc/modulefiles
module load gromacs/2023.3-gpu

export OMP_NUM_THREADS=${SLURM_CPUS_PER_TASK}

srun gmx_mpi mdrun \
    -nb cpu -pme cpu -bonded cpu -update cpu \
    -g ex2.1_${SLURM_NTASKS}x${OMP_NUM_THREADS}_jID${SLURM_JOB_ID}
    -nsteps -1 -maxh 0.017 -resethway -notunepme
```

- Next submit jobs by incrementally offloading various force tasks (non-bonded `-nb`, PME `-pme`, bonded `-bonded`) to the GPU.

- How does the performance (ns/day) change with offloading more tasks?
- Look at the performance table in the log and observe how the fraction wall-time spent in the tasks left on the CPU change.

💡 **Note** that the log file performance report will only contain timings of tasks executed on the CPU, not those offloaded to the GPU, as well as timings of the CPU time spent launching GPU work as well as waiting for GPU results.

Exercise 2.2: GPU-resident mode

Continuing from the previous exercise, we will now explore using the GPU-resident mode.

```
#!/bin/bash
#SBATCH --partition=small-g           # partition to use
#SBATCH --account=project_465000934    # project for billing
#SBATCH --reservation=gromacs_thursday # reservation for 25 January
#SBATCH --time=00:10:00                # maximum execution time of
#SBATCH --nodes=1                      # we run on 1 node
#SBATCH --gpus-per-node=1              # we use 1 GPU device
#SBATCH --ntasks-per-node=1            # 1 MPI rank
#SBATCH --cpus-per-task=7              # number cpus-per-task

module use /appl/local/csc/modulefiles
module load gromacs/2023.3-gpu

export OMP_NUM_THREADS=${SLURM_CPUS_PER_TASK}

srun gmx_mpi mdrun \
    -nb gpu -pme gpu -bonded gpu -update gpu \
    -g ex2.2_${SLURM_NTASKS}x${OMP_NUM_THREADS}_jID${SLURM_JOB_ID}
    -nsteps -1 -maxh 0.017 -resethway -notunepme
```

- Submit a fully offloaded GPU-resident job using the `-update gpu` option (as above).
- Since we moved most computation to the GPU, the CPU cores are left unused. The GROMACS heterogeneous engine allows moving work back to the CPU. Now, let's try to utilize CPU cores for potential performance benefits. First, try moving the PME tasks back to the CPU, then the bonded tasks.

- How does the GPU-resident mode perform compared to the best performing force-offload run from ex 2.1?
- How did the performance change when force tasks were moved back to the CPU?
- *Bonus*: Enable (PME) task load balancing by replacing `-notunepme` with `-tunepme`. Assign the PME task to the CPU and GPU and observe how the performance changes compared to the earlier similar run without load balancing.
- *Bonus*: The frequency of neighbor search (`nstlist`) is a free parameter and can impact performance. Observe the default in the log files, try other values (e.g. double and triple) and observe how the performance changes.

▼ help with the results

Sample log files for the exercise session.:

- ex 2.1 (https://github.com/Lumi-supercomputer/gromacs-on-lumi-workshop/tree/main/Exercise-2.1/STMV)

- ex 2.2 (https://github.com/Lumi-supercomputer/gromacs-on-lumi-workshop/tree/main/Exercise-2.2/STMV)

3. Scaling GROMACS across multiple GPUs

🎯 **Learning goals**

- Understand how task- and domain decomposition is used in the `mdrun` simulation engine.
- Explore the multi-GPU strong scaling of GROMACS simulations and the effect of using different decomposition and communication schemes.

In these exercises, we will explore the use of multiple GPUs to improve the simulation performance. As simulation system we use the satellite tobacco mosaic virus, STMV.

Exercise 3.1: Separate PME rank

In the previous exercise series, we have learnt how to offload some *tasks* from CPU to GPU: short-range nonbonded, PME and bonded forces, update and constraints. When scaling across **two** GPU devices, the same approach can be applied.

MPI ranks coordinate their work by exchanging data over a special protocol and, unlike OpenMP threads, can run on different *nodes* in the cluster. Each MPI rank can only use one GPU, but different MPI ranks can use different GPUs (or the same one).

With two MPI ranks, `mdrun` can do non-bonded, bonded, and integration tasks on the first rank and PME on the second rank. This is a basic version of run mode called *separate PME rank*. Furthermore, each task can be run either on CPU or GPU. When running on CPU, the work can be distributed across multiple OpenMP threads within a rank; when running on GPU, the task is automatically mapped to the GPU's compute resources.

Try running on two GPUs using two ranks, one particle-particle (PP) and one PME:

```
#!/bin/bash
#SBATCH --partition=small-g           # partition to use
#SBATCH --account=project_465000934    # project for billing
#SBATCH --reservation=gromacs_thursday # reservation for 25 January
#SBATCH --time=00:10:00               # maximum execution time of
#SBATCH --nodes=1                     # we run on 1 node
#SBATCH --gpus-per-node=..            # fill in number of  GPU dev
#SBATCH --ntasks-per-node=...         # fill in number of MPI rank
#SBATCH --cpus-per-task=7             # number cpus-per-task

export OMP_NUM_THREADS=${SLURM_CPUS_PER_TASK}

srun gmx_mpi mdrun -npme 1  \
                 -nb gpu -pme gpu -bonded gpu -update gpu \
                 -g ex3.1_${SLURM_NTASKS}x${OMP_NUM_THREADS}_jID${S
                 -nsteps -1 -maxh 0.017 -resethway -notunepme

```

- Try changing which tasks are offloaded to GPUs and CPUs by varying `-bonded` and `-update` flags (keeping `-nb gpu -pme gpu`).

- Look at the absolute performance in the log files. Note that the communication overhead when using two GPUs outweighs the gains.
- *Advanced*: Compare log files for GPU-resident run with Ex. 2.2. How many times is "Wait GPU state copy" called in each case (compared to the total number of steps)?

Exercise 3.2: Separate PME rank with direct GPU communication.

By default, GROMACS uses *staged communication*: data is copied from GPU to CPU on one rank, then sent between CPUs, and finally copied to the target GPU. On LUMI, it is more efficient to use *direct GPU communication* (also called "GPU-aware"). This is achieved by setting the following environment variables:

```
export MPICH_GPU_SUPPORT_ENABLED=1
export GMX_ENABLE_DIRECT_GPU_COMM=1
export GMX_FORCE_GPU_AWARE_MPI=1
```

- Add these three variables to the script from Exercise 3.1 (before calling `srun`) and repeat the runs of the previous exercise, but now with GPU-direct communication enabled.

- Look at how the absolute performance change in the log files.
- *Advanced*: Compare the performance counters in log files in this and previous exercise. What do you observe? (Hint: check "Send X to PME" and "Wait GPU state copy").

Exercise 3.3: Domain-decomposition with a separate PME rank

Distributing tasks between GPUs only allows limited parallelism. If we wish to scale to more than two GPUs, *domain decomposition* should be employed.

Recall from the lecture that the domain decomposition allows *spatially* decomposing simulation data into *domains*. Most interactions in molecular dynamics are short-range, and thus suitable for the domain decomposition approach. However, long-range electrostatic interactions are less amenable to decomposition and therefore require some special handling. Typically, when scaling over *N* ranks (1 GPU per rank, *N* > 2), we use one rank to compute long-range electrostatics (PME) for the whole system and use domain decomposition to distribute the other tasks (short-range non-bonded, bonded, integration) between the remaining *N*-1 ranks.

Now we try running on 4, 6, 8 ranks (including one separate PME rank) by changing the values in the script below (ensure that the values of `--gpus-per-node` and `--ntasks-per-node` are equal). To dedicate one rank for PME we use the option `-npme 1`.

Recall from the lecture on LUMI architecture that there is an intricate interconnection between CPUs and GPUs. To make sure the code runs optimally, always use `lumi-affinity.sh` script and `srun --cpu-bind=${CPU_BIND} ./select_gpu` invocation to optimally pin CPU and GPU tasks to the devices.

```
#!/bin/bash
#SBATCH --partition=small-g            # partition to use
#SBATCH --account=project_465000934    # project for billing
#SBATCH --reservation=gromacs_thursday # reservation for 25 January
#SBATCH --exclusive                    # new! to reserve the whole
#SBATCH --time=00:10:00                # maximum execution time of
#SBATCH --nodes=1                      # we run on 1 node
#SBATCH --gpus-per-node=..             # fill in number of  GPU dev
#SBATCH --ntasks-per-node=...          # fill in number of MPI rank

module use /appl/local/csc/modulefiles
module load gromacs/2023.3-gpu
source ${GMXBIN}/lumi-affinity.sh      # new! script to configure L

export OMP_NUM_THREADS=7

export MPICH_GPU_SUPPORT_ENABLED=1
export GMX_ENABLE_DIRECT_GPU_COMM=1
export GMX_FORCE_GPU_AWARE_MPI=1

srun --cpu-bind=${CPU_BIND} ./select_gpu \
     gmx_mpi mdrun -npme 1 -nb gpu -pme gpu -bonded gpu -update gpu
                   -g ex3.3_${SLURM_NTASKS}x${OMP_NUM_THREADS}_jID$
                   -nsteps -1 -maxh 0.017 -resethway -notunepme
```

⚠ The script above uses `--exclusive` flag, reserving the whole node. This is necessary to be able to set CPU and GPU affinities and it makes performance measurements from short runs more predictable. However, exclusive reservations should not be used for long runs unless you are using all eight GPUs.

- Look at the absolute performance in the log files.
- How does absolute performance scale with increasing number of GPUs?
- *Bonus*: In Exercise 3.2, the performance gains were limited because there is not enough PME work to offset the overhead of extra communication. Try running on two GPUs (`--gpus-per-node=2`) with three ranks (`--ntasks-per-node=3`) so that one GPU does PP work, and the other does PP+PME. How does the performance compare to Exercise 3.2?
- *Bonus*: The frequency of domain decomposition and neighbor search (`nstlist`) is a free parameter and can impact performance. Observe the default in the log files; try other values (e.g. double and triple) and observe how the performance changes.

▼ Help with the solution

Sample log files for the exercise session.:

- ex 3.1 (https://github.com/Lumi-supercomputer/gromacs-on-lumi-workshop/tree/main/Exercise-3.1/STMV)
- ex 3.2 (https://github.com/Lumi-supercomputer/gromacs-on-lumi-workshop/tree/main/Exercise-3.2/STMV)
- ex 3.3 (https://github.com/Lumi-supercomputer/gromacs-on-lumi-workshop/tree/main/Exercise-3.3/STMV)

4. Ensemble parallelization across multiple GPUs

🎯 **Learning goals**

- Understand how to set up and run ensemble simulations with `-multidir`.
- Understand the tradeoffs between simulation and aggregate ensemble throughput and how it relates to hardware utilization efficiency.
- *Advanced*: Explore the impact of task mapping on aggregate throughput.

The `mdrun` simulation engine provides the *multi-simulation* feature to run various types of ensemble simulations. For details of how to set up an ensemble simulation see the GROMACS user guide section on multi-simulations (https://manual.gromacs.org/current/user-guide/mdrun-features.html#running-multi-simulations).

In this exercise we will learn how to run ensemble simulations using `mdrun` *multi-simulation* feature. We will explore how to optimize ensemble performance and efficiency, and learn about tradeoffs between simulation throughput, aggregate throughput and how these relate to hardware utilization efficiency of heterogeneous hardware and GPU accelerators.

The example system uses a strongly coupled ensemble setup based on multi-walker AWH. The AWH setup is flexible in terms of ensemble size and can employ up to 32 members.

To run multi-simulations the mdrun option `-multidir` can be used. Note that the `-multidir` feature requires one input directory per ensemble member which should contain simulation input (tpr) file and where outputs will be written. The directory structure for `-multidir` runs is provided in the input tarball.

Exercise 4.1: Ensemble runs with `-multidir`

When the simulation system is relatively small, it may not be able to fully saturate modern HPC GPUs. In such cases, we can achieve better hardware utilization by assigning multiple ensemble members to a GPU. By doing so we provide more (independent) work to each GPU, which can significantly improve *aggregate simulation throughput* and allows making *more efficient use of GPU hardware*.

To explore this, we will use a fixed amount of hardware resources and vary the ensemble size. Starting with the GPU-resident setup from Exercise 2.2 we will run multi-GPU ensemble runs on all eight GPUs of a single LUMI-G node.

```
#!/bin/bash
#SBATCH --partition=small-g            # partition to use
#SBATCH --account=project_465000934    # project for billing
#SBATCH --reservation=gromacs_thursday # reservation for 25 January
#SBATCH --exclusive                    # new! to reserve the whole
#SBATCH --time=00:10:00                # maximum execution time of
#SBATCH --nodes=1                      # we run on 1 node
#SBATCH --gpus-per-node=8              # the number of GPU devices
#SBATCH --ntasks-per-node=..           # fill in the number of MPI

module use /appl/local/csc/modulefiles
module load gromacs/2023.3-gpu
source ${GMXBIN}/lumi-affinity.sh   #

export OMP_NUM_THREADS=...          # fill in the number of threads
num_multi=...                       # change ensemble size

# Change "??" in -multidir flag below to the ensemble size
srun --cpu-bind=${CPU_BIND} ./select_gpu \
    gmx_mpi mdrun -multidir sim_{01..??} \
    -nb gpu -pme gpu -bonded gpu -update gpu \
    -g ex4.1_${SLURM_NNODES}N_multi${num_multi}_jID${SLURM_JOB_ID}
    -nsteps -1 -maxh 0.017 -resethway -notunepme
```

- As a baseline, launch one simulation per GPU, hence an 8-way ensemble on one LUMI-G node.
- Next, submit jobs with multiple (2,3 and 4) simulations per GPU on one LUMI-G node.

- How does the performance (ns/day) of *each ensemble member* simulation change as you increase the number of simulations per GPU?
- How does the aggregate performance per node change as you increase the number of simulations per GPU?
- *Bonus*: log in to the compute node and observe the GPU utilization (using the `rocm-smi` tool) during the ensemble runs with lowest/highest aggregate performance.
  - Try: `srun --interactive --pty --jobid=<jobid> $SHELL`, and once on the compute node, `rocm-smi -u`
  - Alternatively, run `rocm-smi` directly: `srun --interactive --pty --jobid=<jobid> rocm-smi -u`
  - replace `<jobid>` with the actual Slurm job ID of your job

Exercise 4.2: Trading efficiency for higher simulation throughput

In some cases, we want to increase the (non-aggregate) performance of the ensemble, e.g. if we need to sample longer. In such cases, we might have to trade performance for efficiency.

In this exercise, we will combine what we have learnt in the exercises 3.x and 4.1. We will assign an increasing amount of GPU resources to each ensemble member to improve simulation performance. To do that, we will use a fixed 16-way ensemble and run it varying the amount of compute resources.

```bash
#!/bin/bash
#SBATCH --partition=small-g            # partition to use
#SBATCH --account=project_465000934    # project for billing
#SBATCH --reservation=gromacs_thursday # reservation for 25 January
#SBATCH --exclusive                    # new! to reserve the whole
#SBATCH --time=00:10:00                # maximum execution time of
#SBATCH --nodes=...                    # fill in the number of node
#SBATCH --gpus-per-node=8              # the number of GPU devices
#SBATCH --ntasks-per-node=..           # fill in the number of MPI

module use /appl/local/csc/modulefiles
module load gromacs/2023.3-gpu
source ${GMXBIN}/lumi-affinity.sh

export OMP_NUM_THREADS=...           # fill in the number of OpenMP

export MPICH_GPU_SUPPORT_ENABLED=1
export GMX_ENABLE_DIRECT_GPU_COMM=1
export GMX_FORCE_GPU_AWARE_MPI=1

num_multi=16                         # ensemble size

# set -npme to 0 or 1
srun --cpu-bind=${CPU_BIND} ./select_gpu \
    gmx_mpi mdrun -multidir sim_{01..16} \
    -npme ... \
    -nb gpu -pme gpu -bonded gpu -update gpu \
    -g ex4.2_${SLURM_NNODES}N_multi${num_multi}_jID${SLURM_JOB_ID}
    -nsteps -1 -maxh 0.017 -resethway -notunepme
```

- As a baseline, we will use the single-node case, that is 16-way ensemble with 2 simulations per GPU (or a half-GPU per ensemble member) on a single LUMI-G node. Run this setup or you can reuse this result from the previous exercise.
- Next, submit jobs with increasing the amount of GPU resources assigned to each ensemble member, by increasing the total number of nodes used from 1 to 2, and 4.

- *Note* that as mentioned earlier, this is a small simulation system and therefore its scaling is limited, hence it is not useful to try to use more than 2-4 GPUs/simulation.

- How does the performance (ns/day) of *each ensemble member* simulation change as you increase the number of nodes/GPUs used?
- How does the aggregate performance per node change as you increase the number of nodes/GPUs used?
- *Bonus*: observe the GPU utilization using `rocm-smi` during the 1- and 4-node ensemble runs. On LUMI you can do so by launching `watch rocm-smi` in a job that "overlaps" with an existing job running as described in the LUMI docs (https://docs.lumi-supercomputer.eu/runjobs/scheduled-jobs/interactive/#using-srun-to-check-running-jobs) or in the Exercise 4.1.

Advanced Exercise 4.3: Exploring task mapping for ensemble runs

In previous exercises, we used the fully GPU-offloaded GPU-resident mode, hence the CPU was left mostly idle except for the small amount of AWH and bonded computation which is not offloadable. As we explored in Exercise 2.2, exploiting the heterogeneous nature of the GROMACS engine, we can shift more work to the CPU cores that are otherwise left mostly idle.

Starting with the setup from the previous exercise, explore mapping some lighter-weight compute tasks to the CPU, e.g. `-bonded` or `-update`.

- As a baseline, we will use the fully GPU-offloaded runs from the previous exercise.
- Submit jobs with `-bonded cpu` and/or `-update cpu`
- *Bonus*: repeat the same for a larger simulation system, like STMV.

- Compare the log file when different tasks are GPU-offloaded. How do *per-simulation* and *per-node* aggregate performance change?
- Bonus: Look at the STMV log files. Do a STMV and aquaporin show the same behavior?

▼ help with the results
Sample log files for the exercise session.:

- ex 4.1 (https://github.com/Lumi-supercomputer/gromacs-on-lumi-workshop/tree/main/Exercise-4.1)
- ex 4.2 (https://github.com/Lumi-supercomputer/gromacs-on-lumi-workshop/tree/main/Exercise-4.2)