

# LUMI

A white wolf is the central focus, standing in a futuristic, blue-toned digital environment. The background is filled with vertical data streams, glowing particles, and a grid-like structure, suggesting a high-tech or virtual world. The overall aesthetic is clean and modern.

## Slurm on LUMI

**Tuomas Lunttila**

LUMI User Support Team (LUST), CSC

April 2026

Slides updated from previous version,  
authored by Kurt Lust (LUST, UAntwerp)

# What is Slurm?

- Slurm is a resource manager for supercomputers
  - It manages nodes, CPU cores, GPUs, ...,
  - starts jobs and cleans up after jobs,
  - and can be used to start applications in a job
- and Slurm is a job scheduler:
  - It assigns jobs to resources,
  - based on policies sets by sysadmins to ensure a good use of the machine and a fair distribution of resources among projects
- Most popular job scheduler and resource manager at the moment
  - Though it is starting to show its age and has trouble dealing with the deep hierarchy of resources on modern supercomputers
  - So using Slurm will **not always be as straightforward as we would like it...**
- And there are some tricks needed on LUMI...

# Slurm concepts: Physical resources

- Node: The hardware that runs a single operating system image
- Socket: On LUMI a physical socket
- Core: Physical core in a system
- Thread: Hardware thread (virtual core). Sometimes called SMT = Simultaneous MultiThreading, or hyperthreads
- CPU: A “consumable resource” and can be different things on different systems. On LUMI: CPU=core.
- We already see a big problem with Slurm: Three levels in the hierarchy of an AMD Milan processor (socket, NUMA domain and L3 cache region/chiplet) covered by only one concept.
- GPU: An accelerator, on LUMI one GCD of an MI250X

# Slurm concepts: Logical resources

- Partition: A job queue with limits and access control
- Job: A resource allocation request  
Each job gets a unique job ID
- Job step: A set of (possibly parallel) tasks within a job
  - This is where all the work is done
  - The job script itself runs in a special job step called the batch job step
  - An MPI application typically runs in its own job step
- Task: Executes in a job step and corresponds to a Linux process:
  - A shared memory program is a single task
  - MPI application: Each rank (=MPI process) is a task
    - Pure MPI: Each task uses a single CPU (also single core for us)
    - Hybrid MPI/OpenMP: Each task uses multiple CPUs
  - A single task can not use more CPUs than available in a single node

# Slurm is first and foremost a batch scheduler

- A cluster is a large and expensive machine
  - So the cluster has to be used as efficiently as possible
  - Which implies that we **cannot loose time waiting for input** as in an interactive program
- And **few programs can use the whole capacity** (also depends on the problem to solve)
  - So the cluster is a shared resource, each simultaneous user gets a fraction of the machine depending on **their requirements**
- Moreover there are a **lot of users**, so one sometimes has to **wait a little**.
- Hence **batch jobs** (script with **resource specifications**) submitted to a **queueing system with a scheduler** to select the next job in a fair way based on available resources and scheduling policies.
- Though there are some facilities for interactive jobs

# A Slurm batch script

L U M I

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

```
#SBATCH --jobname=name_of_job  
#SBATCH --partition=small  
#SBATCH --ntasks=2 --cpus-per-task=4  
#SBATCH --mem-per-cpu=1g  
#SBATCH --time=01:00:00
```

```
#SBATCH --account=project_465000000
```

```
#SBATCH --output=stdout.file  
#SBATCH --error=stderr.file
```

```
module load LUMI/24.03 partition/C  
module load lumi-CPEtools/1.2-cpeCray-24.03
```

```
srun hybrid_check
```

This is a bash script (but could be, e.g., Perl which we do not encourage)

Specify the resources for the job (and some other instructions for the resource manager), but look as comments to Bash

Build a suitable environment for your job. The script runs where you launched the batch job!

The command(s) that you want to execute in your job.

# Slurm partitions

- Slurm partitions are (possibly overlapping) groups of nodes with similar resources or associated limits
- Each partition targets a particular job profile and can have its own policies to support that profile
- Two types of partitions on LUMI
  - Exclusive node use by a single job
    - Ensures a clean environment for large parallel jobs
    - Possible to map tasks on available resources for optimal performance
  - Allocatable by resources (CPU and GPU)
    - Nodes are shared by multiple users and multiple jobs
    - The distribution of cores is not always continuous nor is a proper mapping of cores onto GPU ensured
    - Fragmentation of resources is a real problem!
- Default settings for certain Slurm parameters can differ per partition
  - Use common sense when requesting resources

# Slurm partitions (2)

Partition name	Max walltime	Max jobs	Max resources/job	HW partition
<b>Slurm partitions allocatable by node (exclusively)</b>				
standard-g	2 days	210 (200 running)	1024 nodes	LUMI-G
standard	2 days	120 (100 running)	512 nodes	LUMI-C
<b>Slurm partitions allocatable by resources (shared)</b>				
small-g	3 days	210 (200 running)	4 nodes	LUMI-G
dev-g	3 hours	2 (2 running)	32 nodes	LUMI-G
small	3 days	220 (200 running)	4 nodes	LUMI-C
debug	30 minutes	2 (2 running)	4 nodes	LUMI-C
largemem	1 day	30 (20 running)	1 node	LUMI-D CPU
lumid	4 hours	2 (1 running)	1 node	LUMI-D viz GPU

# Slurm partitions: Useful commands

- List of available partitions

```
sinfo -s
```

- Partition details

```
scontrol show partition <partition-name>
```

- Some of the limits on jobs in the queue:

```
sacctmgr show assoc where account=<my project number> user=$USER
```

**sinfo -s**

**sinfo -o "%11P %.5a %.10l %.20F %N"**

Allocated/Idle/Other/Total

PARTITION		AVAIL	TIMELIMIT	NODES(A/I/O/T)	NODELIST
debug		up	30:00	1/7/0/8	nid[002500-002501,...]
interactive		up	8:00:00	3/1/0/4	nid[002502,002507...]
q_fiqci	up	15:00	0/1/0/1	nid002598	
q_industry	up	15:00	0/1/0/1	nid002598	
q_nordiq	up	15:00	0/1/0/1	nid002503	
small	up	3-00:00:00	244/45/17/306	nid[002280-002499,...]	
standard	up	2-00:00:00	1613/0/115/1728	nid[001000-002279,...]	
dev-g	up	3:00:00	24/22/2/48	nid[005002-005025,...]	
small-g	up	3-00:00:00	191/2/5/198	nid[005026-005123,...]	
standard-g	up	2-00:00:00	2359/30/339/2728	nid[005124-007851]	
largemem	up	1-00:00:00	0/5/1/6	nid[000101-000106]	
lumid	up	4:00:00	1/6/1/8	nid[000016-000023]	

Restricted access

# Accounting of jobs

- The use of resources by a job is billed to projects, not users
  - As users can have multiple projects, you have to specify the project account (project\_46YXXXXXX) with every command that creates an allocation
- Billing is based on what others cannot use because of your job
  - Taking into account a proportional use of cores, memory and GPUs (actually GCDs)
  - E.g., ask for one core but half the memory of a CPU node and you will be billed for half a node, even when using the small partition
  - Ask for one core or one GPU in the standard or standard-g partition and you will be billed for the whole node
  - Billing based on the number of cores, the amount of memory and the number of GPUs.
- Slurm accounting features do not produce the correct numbers
  - Check the state of your allocation with [lumi-workspaces](#) or [lumi-allocations](#)

# Queueing and fairness

- The Slurm partition setup of LUMI prioritises to some extent larger jobs
  - And most nodes are reserved for jobs that use them exclusively (standard partitions)
- Your job is queued until resources are available for the requested time window
  - Each job has a unique job ID which is a number
  - Each job also has an evolving priority (depending on size, how much you have run recently, how long the job has been in the queue)
  - So you're not served in a first come, first served way!
- Factors that decide on your job's queue priority: [sprio](#)
  - **Fairshare** is a mechanism that favours users/projects that haven't been running a lot in the past few days
- **Backfill**: If a small job fits into the gap left when collecting resources for a bigger job, that job may be started even though it is not the highest priority job

# Managing Slurm jobs

- [squeue](#) to examine the job queue
  - Flag `--me` shows your jobs only
  - Flag `--start` shows the current estimate for the start time of your job
- [scancel <jobID>](#) will cancel the job with the given jobID.
- [sstat -j <job>](#) will show information about the **running** job with given jobID.
  - Real-time information gathered from the resource manager component of Slurm
  - See at the end of this presentation
- [sacct -j <jobID>](#) will show information about any job, also when finished
  - Information from the Slurm accounting database, so with some delay
  - See at the end of this presentation

# Creating a Slurm job

Slurm has three commands to create jobs and start job steps in a job:

- **salloc** only creates a job allocation but no job step.
  - Shell on the node on which the command is issued, not on the compute resources
  - Leave the shell (control-D or exit) to end the allocation
  - Good for interactive work
- **srun** creates a job step in a job allocation
  - When run outside a job allocation it will also create a job allocation
  - Be careful when using it to also create a job allocation as some options work differently as for the commands meant to start a job allocation
- **sbatch** creates a job allocation and then the batch job step to run the job script
  - Resources for the batch job step are not always what you expect
  - Use **srun** to create further job steps

# Passing options to srun, salloc and sbatch

- Lowest priority (for [sbatch](#)): Using [#SBATCH](#) lines at the start of the bash script
  - These lines should not be interrupted by commands in the script
- Pass through environment variables:
  - [SBATCH\\_\\*](#) for [sbatch](#), [SALLOC\\_\\*](#) for [salloc](#), [SLURM\\_\\*](#) and [SRUN\\_\\*](#) for [srun](#)
  - Will overwrite values on [#SBATCH](#) lines
  - See the manual pages of [sbatch](#), [salloc](#) and [srun](#)
  - Risk: You forget that they exist...
- Highest priority: Flags and options given on the command line
  - Specify [before](#) the job script ([sbatch](#)) or before the command to execute ([srun](#)) as otherwise they will be considered options of the batch script or command respectively
  - Override options in [#SBATCH](#) lines and environment variables
- Several options given to [sbatch](#) and [salloc](#) are also forwarded to [srun](#) commands in the job script (via [SLURM\\_\\*](#) environment variables) and may conflict with options specified on the command line!

# Specifying options

- Slurm commands have way more options than we can discuss and if and how they work may depend on the specific configuration of Slurm
- Slurm commands can exist in two variants:
  - Long variant with double dash: `--long-option=<value>` or `--long-option <value>`
  - Single-letter variant with single dash: `-S <value>` or `-S<value>`
- Not all combinations are valid, and use common sense
  - Overspecifying resources may not be a good idea
  - Underspecifying resources isn't a good idea either as some defaults may be used
  - Some combinations for resources just don't make sense
    - In the following slides we'll try to structure this a bit

# Some common options to all partitions

- For some resources a different strategy should be used for “allocatable by node” and “allocatable by resource” partitions and this will be discussed later.
- Common to both:
  - Specify the account: `--account=project_46YXXXXXX` or `-A project_46YXXXXXX`  
Job will not run without!
  - Specify the partition: `--partition=<partition>` or `-p <partition>`
  - Maximum wall time for the job: `--time=<timespec>` or `-t <timespec>`  
`<timespec>`: minutes, minutes:seconds, hours:minutes.seconds and more
  - Name of the job: `--job-name=<name>` or `-J <name>`
    - More readable output of `squeue`
    - Can be used to name output files also
  - Specifying a reservation: `--reservation=<names>`: Use resources from the given reservation(s)
    - For trainings or the “hero runs”
  - Mail options exist but do not work on LUMI with no plans to make them available

# Redirecting output

- Using `--output / -o` and `--error / -e`
  - No `--output` and no `--error` : stdout and stderr redirected to `slurm-<jobid>.out`
  - Use `--output` but no `--error` : stdout and stderr redirected to the given file
  - No `--output` but `--error` specified: stdout redirected to `slurm-<jobid>.out`, stderr to the file given with `--error`
  - Both `--output` and `--error` : stdout redirected to the file pointed to by `--output` and stderr redirected to the file pointed to by `--error`.
- It is possible to insert codes in the file name that will be replaced with the corresponding Slurm information when the job starts.
  - Examples are “%x” for the job name or “%j” for job id.
  - See the [manual page of sbatch](#), [section “filename pattern”](#).

# Requesting resources: CPUs and GPUs

Two strategies:

- **“Per-node allocations”**: Request suitable nodes (number and partition) with `sbatch` or `salloc`, can postpone specifying the job step structure (tasks etc.) until the job step is run
  - Logical ways of allocating resources on “allocatable-by-node” partitions
  - On “allocatable per resource” partitions: Use `--exclusive` with `sbatch` and `salloc`
  - Ultimate flexibility in the job as you can run multiple job steps with a different structure in the same job
  - Binding (next session) fully supported
- **“Per-core allocations”**: Specify the job step structure and optionally limit the choice of slurm by also specifying the number of nodes
  - Works on “allocatable-by-resource” and “allocatable-by-node” partitions
  - Without knowing the job step structure Slurm cannot create a correct allocation
  - Restrictions when you want to also use a different job step structure in the same job
  - Severe restrictions on the bindings that can be done afterwards

# Per-node allocations

- Specify the partition: `--partition=<partition>` or `-p <partition>`
- Specify the number of nodes: `--nodes=<number>` or `-N <number>`
- In a allocatable-by-resource partition, specify `--exclusive`
- We now have all cores and all GPUs in the requested nodes available, and
  - All memory on the allocatable-by-node partitions
  - But not on the allocatable-by-resource partitions: 112GB on small and 64 GB on small-g
  - Request all available memory in a node: `--mem=0`
  - Safer: Request the maximum reasonable amount for the node: the memory capacity minus 32 GB.
    - `--mem=224G` for the regular compute nodes of LUMI-G (small and standard partitions)
    - `--mem=480G` for LUMI-G (small-g, standard-g) and the 512 GB nodes of LUMI-C (small only)
    - `--mem=992G` for the 1 TB nodes of LUMI-C (small only)
- ... And you're done, but if you insist ...

# Per-node allocations: CPUs

- You will get all the CPUs on the allocated nodes so no need to explicitly request them as a resource
- But if you insist:
  - No option to request “CPUs per node” in one go, instead
  - `--sockets-per-node=<nr_sockets> --cores-per-socket=<nr_cores>`
    - LUMI-C: `--sockets-per-node=2 --cores-per-socket=64`
    - LUMI-G: `--sockets-per-node=1 --cores-per-socket=56`  
due to the low-noise mode
  - Shortcut via `--extra-node-info=<nr_sockets>[:nr_cores>]` or `-B =<nr_sockets>[:nr_cores>]`
    - LUMI-C: `--extra-node-info=2:64`
    - LUMI-G: `--extra-node-info=1:56`
- Note `--threads-per-core=2` does not work
  - See later for how to enable/disable hyperthreading

# Per-node allocations: GPUs

- You will get all the GPUs on the allocated nodes so no need to explicitly request them as a resource
- But if you insist, 3 options:
  - Most logical: `--gpus-per-node=8` or `--gpus-per-node=mi250:8`
    - Lower than 8 also possible, but it does not make sense as you are billed for them anyway
  - Total number of GPUs in the job using `--gpus=<number>` or `-G <number>`
    - Be careful when you adapt the number of nodes (or don't request all GPUs)!
  - Equivalent to `--gpus-per-node: --gres=gpu:8`, or `--gres=gpu:mi250:8`
    - GRES = Generic consumable Resource
    - Multiple types possible so need to specify the type `gpu`
    - As these are forwarded to `srun`, it will save you from specifying them there
- Options to specify number of CPUs and number and type of GPUs in a per-node allocation are more meant for clusters with heterogeneous partitions

# Per-node allocations: Starting a job step

- Job script starts in the batch job step
  - Can run serial and shared-memory multithread programs
  - But not always the environment that you expect: All hardware threads on the first node of the allocation.
- From `salloc`: Shell on the login nodes but no shell in any of the allocated resources, so a new job step needed to run on the compute nodes
- Command to start a new job step: `srun`
  - Typical case: Creates a number of equal-sized tasks, so needs
    - Number of tasks
    - CPUs per task (with or without hardware threading)
    - In some cases: GPUs accessible to each task
  - Multiple ways of doing this, we propose one possible scheme:

# Per-node allocations: Starting a job step (2)

- Specifying the number of tasks
  - Either total number: `--ntasks=<number>`, or `-n <number>`
    - Risk: Forget to adapt when adapting number of nodes
    - Advantage: In some cases the number of tasks will not be a multiple of the number of nodes, and not all nodes can have the same number of tasks
  - Or per node: `--tasks-per-node=<number>`: Logical in a per node allocation **but broken...**
    - Only works as argument for `sbatch`, or remove `SLURM_NTASKS` and `SLURM_NPROCS`.
- Specify the number of CPUs (cores on LUMI) for each task:  
`--cpus-per-task=<number>` or `-c <number>` to **bind** CPUs to tasks
- GPUs: In theory... (much more about this in the next lecture)
  - GPUs exclusive for a task: `--gpus-per-task=<number>` to **bind** GPUs to each task
  - GPUS shared with multiple tasks: `--ntasks-per-gpu=<number>`
- Options can be specified via `#SBATCH` but some issues with `--cpus-per-task`

# Turning hardware threading on or off

- Hardware threads are enabled by default at the system level
- In Slurm on LUMI they are **disabled** by default in regular job steps
  - Turning them on: `#SBATCH --hint=multithread`
  - Turning them off: `#SBATCH --hint=nomulthread`  
Not needed as this is the default
  - Effect corresponds to allocating by thread in `srun`, first using both hardware threads of the first available core, etc.
    - So `--cpus-per-task=4`, with `srun` will give 4 hardware threads on 2 cores
- When used with `srun`, `srun` will not complain but the outcome is wrong
  - See the notes if you are interested, too detailed for this lecture

# Example hardware threading on (1)

```
#!/usr/bin/bash
#SBATCH --job-name=slurm-HWT-standard-multithread
#SBATCH --partition=standard
#SBATCH --nodes=1
#SBATCH --hint=multithread
#SBATCH --time=2:00
#SBATCH --output=%x-%j.txt
#SBATCH --account=project_46YXXXXXX

module load LUMI/24.03 partition/C lumi-CPEtools/1.1-cpeGNU-24.03

echo -e "Job script:\n$(cat $0)\n"

set -x
srun -n 1 -c 4 omp_check -r
set +x
sleep 2
echo -e "\nsacct for the job:\n$(sacct -j $SLURM_JOB_ID)\n"
```

# Example hardware threading on (2)

```
+ srun -n 1 -c 4 omp_check -r
```

Running 4 threads in a single process

```
++ omp_check: OpenMP thread 0/4 on cpu 0/256 of nid001847 mask 0-1, 128-129
++ omp_check: OpenMP thread 1/4 on cpu 1/256 of nid001847 mask 0-1, 128-129
++ omp_check: OpenMP thread 2/4 on cpu 128/256 of nid001847 mask 0-1, 128-129
++ omp_check: OpenMP thread 3/4 on cpu 129/256 of nid001847 mask 0-1, 128-129
```

```
+ set +x
```

sacct for the job:

JobID	JobName	Partition	Account	AllocCPUS	State	ExitCode
4238728	slurm-HWT+	standard	project_4+	256	RUNNING	0:0
4238728.bat	batch		project_4+	256	RUNNING	0:0
4238728.0	omp_check		project_4+	4	COMPLETED	0:0

# Per-core allocations: When to use?

- If your job is too small to fill a complete node and you want to consume less billing units
- But
  - Less control over allocation of cores and GPUs
  - So performance penalty for codes that depend on proper mapping of tasks on L3 cache regions, NUMA nodes and/or socket or on having the closest GPU to each task
  - And this penalty is also unpredictable
  - Unpredictable performance penalty from users sharing the node
- Slurm
  - Has problems with the GPU topology on LUMI
  - Does not support the hierarchy in the compute nodes of LUMI: One can only request nodes or cores, not L3 cache regions, NUMA nodes or sockets
- Resources can be spread over more than the minimal number of nodes
  - And Slurm needs to know how they will be used at job allocation time

# Per-core allocations: Resource request (1)

- Slurm must know the intended use: More flexibility to allocate 4 4-thread MPI ranks than 1 16-core shared memory run
  - Specify job step structure in `#SBATCH` lines
- Specify first the number of tasks
  - `--ntasks=<number>`, or `-n <number>` recommended
  - Possible to restrict the number of nodes: `--nodes=<number>` or `--nodes=<min>-<max>`
  - Can use `--ntasks-per-node` with `--nodes`, but it does not make much sense.
    - If you can fill up nodes you can better use a per node allocation
    - If you cannot fill up a node it doesn't make much sense to spread over more nodes than the minimal number and let the remaining cores fill up with tasks from other users

# Per-core allocations: Resource request (2)

- Specify the number of CPUs (cores) needed for each task
  - Use `--cpus-per-task=<number>`, or `-c <number>`
  - Note: `--cpus-per-task` no longer propagated to `srun`, but patch on LUMI (that fails in some cases)
- LUMI-G: Specify the number of GPUs required
  - Easy way:
    - Use `--gpus-per-task=<number>` to bind one or more GPUs to a single task
    - Use `--ntasks-per-gpu=<number>` if GPUs are shared by multiple tasks
  - This will at least ensure that cores and GPUs are spread across nodes in the right way
  - More in the next lecture

# Per-core allocations: Resource request (3)

- Memory
  - `--mem-per-task` does not exist unfortunately, instead in function of other resources:
  - `--mem-per-cpu=<number>`: Memory per allocated core (use k, m, g)
  - `--mem-per-gpu=<number>`: System memory per allocated GPU, but **not GPU memory!**
  - `--mem=<number>` (memory per node) only makes sense on a single node

# Warning: Allocation per socket?

- Allocations per socket with `--sockets-per-node`, and `--ntasks-per-socket`?
- No!
  - `--sockets-per-node` specifies a property of the node that you want and is not used to ask for just a single socket
    - And irrelevant on LUMI
  - `--ntasks-per-socket` does not work the same as `--ntasks-per-node`:
    - `--ntasks-per-node` : exact number
    - `--ntasks-per-socket` : maximum number
    - Sometimes every word in the manual page matters!

# Per-core allocation: Different job steps (1)

- `srun` can use a different job step structure
  - If there are no more tasks than requested via `#SBATCH` and no more resources per task than requested via `#SBATCH`
  - Or if an entire number of tasks in the new structure fits in the tasks request by `#SBATCH` and the total number of tasks is not more than the original number of tasks multiplied with that entire number
  - Some other cases may work or may not work depending on the actual allocation
  - With GPUs things become very complicated to avoid binding issues if Slurm's way of binding does not work for you

# Per-core allocation: Different job steps (2)

- Example: Create an allocation suitable for 4 MPI processes with 4 cores each:  
`#SBATCH --ntasks=4`  
`#SBATCH --cpus-per-task=4`
- Multithreaded program started without `srun`
  - Would run in the batch step, so all allocated cores in the first node of the allocation
    - `--hint=nomultithread` not enforced
    - So could be running on 4 C / 8 HWT, 8 C / 16 HWT, 12 C / 24 HWT or 16 C / 32 HWT
- `srun --ntasks=1 --cpus-per-task=3`: No problems
- `srun --ntasks=2 --cpus-per-task=4`: No problems
- `srun --ntasks=4 --cpus-per-task=1`: No problems
- `srun --ntasks=16 --cpus-per-task=1`: Works as 4 of these tasks fit in every original task and the total number is not higher than  $4^*4$ .
- `srun --ntasks=1 --cpus-per-task=16`
  - Will produce a warning because the task is bigger than the original task
  - But will work if you're lucky and all cores are on one node

# The job environment

- Job steps started with `sbatch`, `salloc` or `srun` inherit the environment
  - Natural behaviour for `salloc`, as it starts a shell on the calling node
  - Very useful for `srun`, and it would be a pain otherwise
  - May be surprising to some for `sbatch` as the environment for the login nodes may not be the best for the compute nodes
- Change the job(step) environment with `--export`:
  - `--export=NONE`: Do not pass the environment, but Slurm will attempt to create a new user environment even if no login shell is used in the batch script.
  - `--export=ALL,PAR1=VALUE1`: Pass the variables from the environment and add the variable `PAR1` with value `VALUE1`.
    - **Fragile!** A comma in the value `VALUE1` will have unexpected results.
    - Usefull with `srun` in a heterogeneous job to pass different values of an environment variable to different parts of the heterogeneous jobs, e.g., a different value for `OMP_NUM_THREADS`

# Passing arguments to a batch script

- If the same batch script is used for multiple computations, it may be useful to be pass arguments to that script.
- Arguments on the `sbatch` command line after the script are passed to the script and can be accessed as regular bash script arguments through `$1` etc.
  - So no need to use `--export` for that even though it is possible
- Note: `$0` is the full path to the batch script but will actually refer to a buffered copy!

# Passing arguments: Example

```
#!/usr/bin/bash
#SBATCH --partition=small
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=1
#SBATCH --time=5:00
#SBATCH --account=project_46YXXXXXX
```

```
echo "Batch script parameter 0: $0"
echo "Batch script parameter 1: $1"
echo "Environment variable PAR1: $PAR1"
```

with

```
$ sbatch --export=ALL,PAR1="Hello" slurm-small-parameters.slurm 'Wow, this works!'
```

will produce

```
Batch script parameter 0: /var/spool/slurmd/job4278998/slurm_script
Batch script parameter 1: Wow, this works!
Environment variable PAR1: Hello
```

# Automatic requeueing

- LUMI uses the Slurm automatic requeueing of jobs upon node failure
  - Your job is automatically resubmitted if any of its allocated nodes fail
  - Identical job ID is used and the previous output truncated
- If this bothers you:
  - Disable automatic requeueing with `--no-requeue` option
  - Avoid your output file being truncated with `--open-mode=append` option
  - Detect restart in the job script: Use the value of `SLURM_RESTART_COUNT`
    - Starts at 0 for the first run and is incremented at every restart

# Job dependencies

- LUMI does not allow extreme run times, but you can use job dependencies to schedule multiple jobs that should execute one after another
- E.g., `sbatch --dependency=afterok:<jobID> jobdepend.slurm` will only start the job defined by `jobdepend.slurm` after the job with job ID has finished successfully
- Many other possibilities, including
  - Start another job only after a list of jobs have ended
  - Start another job only after a job has failed
  - And many more, check the [sbatch manual page](#) and [look for --dependency](#).
- How to get the jobID? `--parsable` option of `sbatch`  
`first=$(sbatch --parsable jobfirst.slurm)`  
`sbatch --dependency=afterok:$first jobdepend.slurm`

# Interactive jobs with salloc

- Using `salloc`
  - creates a pool of resources reserved for interactive execution
  - and starts a new shell on the node where you called `salloc` (usually a login node)
    - As such it does not take resources away from other job steps that you will run
- Execute any sequential, shared memory, distributed memory or hybrid code on the allocated compute nodes using `srun`
- Obtaining an interactive shell on the first allocated node: `srun --pty $SHELL`
  - But will take away resources from other job steps!
- Terminate the allocation by exiting the shell with `exit` or ctrl-D
- Good for, e.g., development of batch script

```
salloc --account-project_46YXXXXXX --partition=standard --nodes=2 --time=15
```

# Interactive jobs with srun

- Good to get an interactive shell with one or more cores to work directly in the shell, e.g., for compilation
  - But not ideal to spawn further job steps with srun as the interactive shell already fills a task slot
  - You'd rarely need a whole node for that kind of work so small and small-g may be your partitions of choice
- To start:

```
srun --account=project_462YXXXXXX --partition=<partition> \  
    --ntasks=1 --cpus-per-task=<number> --time=<time> --pty $SHELL  
srun -A project_462YXXXXXX -p <partition> -n 1 -c <number> -t <time> --pty $SHELL
```
- To end: Exit the shell with `exit` or ctrl-D

# Inspecting a running job

- A variant of the `srun` scenario on the previous slide
- But now
  - Do not need a new allocation, as that already exists, so need to specify that allocation
  - Usually the job will be using all its resources, so need to overlap resources with those of already running job steps
- To start an interactive shell on the first allocated node of a specific job/allocation:  
`srun --jobid=<jobid> --overlap --pty $SHELL`
- To start an interactive shell on another node of the job:  
`srun --jobid=<jobid> --nodelist=nid00XXXX --overlap --pty $SHELL`  
`srun --jobid=<jobid> -w nid00XXXX --overlap --pty $SHELL`
- Instead of a shell you could also directly run a command, e.g., `top`
- Note: You can see which nodes are allocated to a job via `queue`, `sstat` or `salloc`.

# Job arrays

- Mechanism to submit a large number of related jobs with the same batch script at once

job\_array.slurm

```
#!/bin/bash
#SBATCH --partition=small
#SBATCH --ntasks=1 --cpus-per-task=1 --mem-per-cpu=1G
#SBATCH --time 15:00

INPUT_FILE="input_${SLURM_ARRAY_TASK_ID}.dat"
OUTPUT_FILE="output_${SLURM_ARRAY_TASK_ID}.dat"

./test_set -input ${INPUT_FILE} -output ${OUTPUT_FILE}
```

← for every run, there is a separate input file and an associated output file

```
$ sbatch --array 1-50 job_array.slurm
```

- Will generate 50 jobs, run with `SLURM_ARRAY_TASK_ID` going from 1 to 50

# Heterogeneous jobs

- Run one or more executables in multiple configurations within one `MPI_COMM_WORLD`
  - For example, a climate model may consist of separate atmosphere and ocean models as well as IO servers
- Two ways
  - Create groups in `#SBATCH` lines, separated by `#SBATCH hetjob` lines, and recall these groups with `srun`
  - Just reserve the nodes and do the rest with `srun`, separating parts with a colon
- Slurm support is not very good
  - Treated as multiple jobs which can give problems with scheduling
  - Only in `srun`: Still separate job steps that like to run on separate nodes
- Example: 2 components in the heterogeneous job:
  - Part 1: Application A on 1 node with 32 tasks with 4 OpenMP threads each
  - Part 2: Application B on 2 nodes with 4 tasks per node with 32 OpenMP threads each
  - Parts can be on different partitions

# Heterogeneous jobs: Example with #SBATCH

L U M I

```
#!/bin/bash
#SBATCH --account=project_46YXXXXXX
#SBATCH --partition=standard
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=32
#SBATCH --cpus-per-task=4
#SBATCH hetjob
#SBATCH --partition=standard
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=4
#SBATCH --cpus-per-task=32

unset OMP_NUM_THREADS

srun --het-group=0 --cpus-per-task=4 --export=ALL,OMP_NUM_THREADS=4 ./app_A : \
    --het-group=1 --cpus-per-task=32 --export=ALL,OMP_NUM_THREADS=32 ./app_B
```

- This is an example where the patch for the modified behaviour of `srun` with respect to `-cpus-per-task` does not work.

# Heterogeneous jobs: Example with srun

```
#!/bin/bash
#SBATCH --account=project_46YXXXXXX
#SBATCH --partition=standard
#SBATCH --nodes=3
```

```
unset OMP_NUM_THREADS
```

```
srun --ntasks=32 --cpus-per-task=4 --export=ALL,OMP_NUM_THREADS=4 ./app_A : \
    --ntasks=8 --cpus-per-task=32 --export=ALL,OMP_NUM_THREADS=32 ./app_B
```

```
srun --nodes=1 --ntasks-per-node=32 --cpus-per-task=4 ./app_A : \
    --nodes=2 --ntasks-per-node=4 --cpus-per-task=32 ./app_B
```

# Slurm multi-prog

- Another way to run multiple binaries in one `MPI_COMM_WORLD`
- `--multi-prog mpmc.conf`
  - The config file lists which binary each task will run:  

```
# config file example
0-3          ./a.out
4,6          ./b.out
5            ./c.out
```
- A single Slurm job, so avoids scheduling problems the heterogeneous jobs can have
  - Cannot use multiple partitions
- Specifying configurations for the components (number of threads, bindings etc.) can be tricky
  - Usually easiest to get exclusive nodes and do bindings manually outside Slurm

# Simultaneous job steps

- It is possible to have multiple simultaneous job steps in a single allocation

```
#!/usr/bin/bash  
srun -n4 -c16 exe1 &  
sleep 2  
srun -n8 -c8 exe2 &  
wait
```

- Remarks

- Can be useful if you want to do proper binding but cannot fill a node with a single run
- `sleep` command: To avoid errors from `srun` about not being ready to start a step
- The `wait` command is important as otherwise the job would be killed instantly!
- Tricky with binding (next session) and/or GPUs, may need `--overlap`
- `--exact` may be useful to guarantee the exact resources are available to each job step?
- Makes most sense on exclusive nodes; known problems on LUMI in other cases

# Job monitoring commands

## Real-time information about running jobs: sstat (1)

- Show (a lot of) real-time information about a particular job or job step:  
`sstat -j 1234567`  
`sstat -j 1234567.0`
- It is possible to specify a subset of fields to display using the `-o`, or `--format` option.
- Example for an MPI job: Get an idea of the load balancing:

```
$ sstat -a -j 1234567 -o JobID,MinCPU,AveCPU
```

JobID	MinCPU	AveCPU
-----	-----	-----
1234567.bat+	00:00:00	00:00:00
1234567.1	00:23:44	00:26:02

- Shows for each job step the minimum and average amount of consumed CPU time.
- Step 1 in this case is an MPI job with a slight load imbalance
- As step 0 isn't running anymore, we don't get to see it

# Job monitoring commands

## Real-time information about running jobs: sstat (2)

- Check resident memory:

```
$ sstat -a -j 1234567 -o JobID,MaxRSS,MaxRSSTask,MaxRSSNode
```

```
JobID      MaxRSS MaxRSSTask MaxRSSNode
```

```
-----
```

```
1234567.bat+ 25500K      0 nid001522
```

```
1234567.1    153556K     0 nid001522
```

# Job monitoring commands

## Information about (terminated) jobs: sacct (1)

- `sacct` shows information kept in the job accounting database.
  - So for running jobs the information may enter only with a delay
  - The command to check resource use of a finished application

- Default output for a job:

```
$ sacct -j 1234567
```

```
JobID      JobName Partition  Account AllocCPUS  State ExitCode
-----
1234567    healthy_u+ standard project_4+  512 COMPLETED  0:0
1234567.bat+ batch      project_4+  256 COMPLETED  0:0
1234567.0  gmx_mpi_d project_4+   2 COMPLETED  0:0
1234567.1  gmx_mpi_d project_4+  512 COMPLETED  0:0
```

- Select what you want to see:
  - `--brief`: Very little output, just the state and exit code of each step
  - `--long`: A lot of information, even more than `sstat`
  - `-o` or `--format`: Specify the columns you want to see

# Job monitoring commands

## Information about (terminated) jobs: sacct (2)

- Example: Get resource use of a job:

```
$ sacct -j 1234567 --format JobID%-13,AllocCPUS,MinCPU%15,AveCPU%15,MaxRSS,AveRSS  
--units=M
```

JobID	AllocCPUS	MinCPU	AveCPU	MaxRSS	AveRSS
-----					
1234567	512				
1234567.batch	256	00:00:00	00:00:00	25.88M	25.88M
1234567.0	2	00:00:00	00:00:00	5.05M	5.05M
1234567.1	512	01:20:02	01:26:19	173.08M	135.27M

- This was a two node MPI job with very little memory use per task
  - `--units=M` to get output in megabytes rather than kilobytes
  - `%15` in some field names: Use a 15 character wide field rather than the standard width
  - `%-13`: Similar, but left justified
- List of all fields: `sacct --helpformat` or `sacct -e`

# Job monitoring commands

## Information about (terminated) jobs: sacct (3)

- Selecting jobs to show information about:
  - By default: All jobs that have run since midnight
  - `--jobs` or `-j` : give information about a specific job or jobs (when specifying multiple jobids separated by a comma)
  - `--starttime=<time>` or `-S <time>` : Jobs that have been running since the indicated start time, format: HH:MM[:SS] [AM|PM], MMDD[YY] or MM/DD[YY] or MM.DD[.YY], MM/DD[YY]-HH:MM[:SS] and YYYY-MM-DD[THH:MM[:SS]] ([] denotes an optional part)
  - `--endtime=<time>` or `-E <time>` : Jobs that have been running before the indicated end time.
  - Max query length limited to 30 days (sacct is hard on Slurm controller)
- There are way more features to filter jobs, but some of them are mostly useful for system administrators
- More information: [sacct manual page](#)

# Job monitoring commands

## Generating reports from Slurm accounting data: sreport

- `sreport` is a Slurm tool to extract reports from the Slurm accounting database
  - But much of that information is of little use on LUMI as the billing is not done by Slurm but by an external script that gets its data from the Slurm accounting database
  - And hence the correct billing information is not available in the Slurm accounting database nor can it be easily derived from summary reports.
  - E.g., the amount of core hours reported is for all partitions and hence irrelevant to compute CPU billing units consumed

**Questions?**

