Running Jobs on LUMI

Maciej Szpindler
LUMI User Support Team
Cyfronet

21 September 2023

Running Jobs

- Slurm intro
- Slurm partitions
- Interactive and batch jobs
- Job arrays
- Binding tasks to resources
- Running containers
- *Hands-on excercises*

Slurm intro

- **Slurm** is an open source cluster management and job scheduling system which provides:
  - exclusive and/or non-exclusive allocation of resources (compute nodes)
  - infrastructure for starting, executing, and monitoring jobs
  - fair share queue of pending jobs

Slurm version

Note Slurm version on LUMI is **22.05.8** (as of September 2023). Default documentation on the web is for versions 23.02. Please use specific version:

https://slurm.schedmd.com/archive/slurm-22.05.8/

Slurm partition

- Slurm partitions are groups of nodes with similar resources or associated limits
- Logical concept to manage access to LUMI HW partitions (GPU, CPU nodes)
- List of available partitions

```
sinfo –s
```

- Partition details

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

- *Different context of partition: hardware (name), access (Slurm directive), target architecture (environment module)*

Available partitions

Partition name	Max walltime	Max jobs	Max resources/job	HW partition
Slurm partitions allocatable by node (exclusively)				
standard-g	2 days	210 (200 running)	1024 nodes	LUMI-G
standard	2 days	120 (100 running)	512 nodes	LUMI-C
Slurm partitions allocatable by resources (shared)				
small-g	3 days	210 (200 running)	4 nodes	LUMI-G
small	3 days	220 (200 running)	4 nodes	LUMI-C
dev-g	3hours	2 (1 running)	16 nodes	LUMI-G
debug	30 minutes	2 (1 running)	4 nodes	LUMI-C
largemem	1 day	30 (20 running)	1 nodes	LUMI-D

Project account

- Running jobs requires a project account
  - It is created when you are granted project allocation
  - Your project account ID is required to submit a job
  - Account ID has a name **`project_xxxxxxxxx`** (9 digits)
- You can use the **`lumi-allocations`** command to list your projects
  - Alternative is to use the `groups` command to see account IDs
  - Your allocation portal should also show your project's account IDs

```
Data updated: 2023-09-19 11:31:15
Project              |                   CPU (used/allocated)|              GPU (used/allocated)|          Storage (used/allocated)
--------------------------------------------------------------------------------------------------------------------------------
project_465000688    |          0/10000000   (0.0%) core/hours|         0/1000   (0.0%) gpu/hours|            0/10   (0.0%) TB/hours
```

Allocation budget

- Do not use **`sreport`**
  - It will show you usage with different metrics than actual billing units
  - It creates unnecessary load on the Slurm controller

Interactive jobs

- Using `salloc`
  - creates pool of resources reserved for your interactive execution (tasks)
  - the command will start a new shell **on the login node**
  - you can start parallel execution on the allocated nodes with **`srun`**
  - to obtain a shell on the first allocated compute node you can use **`srun --pty`**
  - the allocation can be terminated by exiting the shell with **`exit`**

```
~> salloc --nodes=2 --account=<project_id> --partition=<partition_name> --time=15
  salloc: Granted job allocation 123456
  salloc: Waiting for resource configuration

~> srun --ntasks=32 --cpus-per-task=8 ./mpi_openmp_application
~> exit
```

Interactive jobs

- Using `srun` directly (single job step)
  - You can execute single parallel task with **`srun`** command
  - To start a shell on the first allocated node in a specific job/allocation use
    `srun --interactive --pty --jobid=<jobid> $SHELL`
  - The **`-w nid00XXXX`** option selects a specific compute node
    `srun --interactive --pty --jobid=<jobid> -w nid002217 …`
  - Use `--overlap` option to share resources already used by your other job step (task)

```
~> srun --interactive --pty --jobid=<jobid> top
```

Job launcher

- `srun` is the only parallel launcher on LUMI
  - there is no mpirun nor mpiexec commands
  - returns the highest exit code of all tasks or the highest signal

Batch jobs

- Batch jobs are submitted with **`sbatch job.sh`** command
  - File job.sh is your job script
  - Job script is regular shell script with **`#SBATCH`** directives and execute command(s)
  - You can use Slurm options with directives , from the command line or via environmental variables
- sbatch exits immediately after the script is successfully transferred to the Slurm controller and assigned a Slurm job ID
- Slurm runs a single copy of the batch script on the *first compute node* in the set of allocated nodes
- Both standard output and error are directed to a file **`slurm-<job_id>.out`** by default

Batch job script

- Remember to include the sheebang in the first line of your job script
  - **`#!/bin/bash`** is recommended
  - Skipping the sheebang line or using fancy interpreters may result in module failures
- Directive line is **`#SBATCH`** followed by sbatch option and value
- Command line options overrides any environment variables and environment variables overrides any options set in a batch job script
- You can enable e-mail notifications in the job script
- You can define dependencies between batch jobs

How to list my jobs

- **`squeue`** command shows all current jobs (running and pending)
  - **`--me`** option alias shows only jobs owned by you

LUMI

Sbatch options

- **`--time`** Set a limit on the total run time of the job allocation
- **`--account`** Charge resources used by this job to specified project
- **`--partition`** Request a specific partition for the resource allocation
- **`--job-name`** Specify a name for the job allocation
- **`--mail-user`** Used to specify the email that should receive notification
- **`--mail-type`** When to send an email: `BEGIN`, `END`, `FAIL`, `ALL`

- **`--nodes`** Number of nodes to be allocated
- **`--ntasks`** Maximum number of tasks (MPI ranks)
- **`--ntasks-per-node`** Number of tasks per node
- **`--cpus-per-task`** Number of cores per tasks
- **`--cpus-per-gpu`** `Number` of CPUs per allocated GPU
- **`--gpus`** Total number of GPUs to be allocated for the job
- **`--gpus-per-node`** Number of GPUs per node
- **`--gpus-per-task`** Number of GPUs per task
- **`--mem`** Set the memory per node
- **`--mem-per-cpu`** Memory per allocated CPU cores
- **`--mem-per-gpu`** Memory per allocated GPU

Other Slurm options

- **`--exclusive`** the job is allocated all CPUs and GRES on all nodes in the allocation, but is only allocated as much memory as it requested
- **`--mem=0`** requests all the memory on a node (be careful, "all" but also "any")
- **`--export`** propagates environment variables from the submission environment to the launched application, **`ALL`** by default
- **`--time`** accepts time formats "minutes", "minutes:seconds", "hours:minutes:seconds", "days-hours", "days-hours:minutes" and "days-hours:minutes:seconds"
- **`--reservation=<`*`reservation_names`*`>`** allocates resources for the job from the named reservation
- **`--dependency=<type:job_id[:job_id]>`** defines the condition that the job with ID `job_id` must fulfil before the job which depends on it can start; type includes after, afterany, afterok, afternotok

Job vs step allocations

- Job allocation is the set of resources created with **sbatch** or **salloc**
- Step allocation is the **srun** (or other serial) command executed within an existing job allocation
- Step allocations usually inherit job options but can overwrite them (with possibly slightly different meaning), e.g.
  - **`--exclusive`** option (prevents other jobs from using the same node/dedicates separate cpus to multiple job steps in a single job)
  - **`--cpus-per-task`** option (set steps' specific cpu binding)

Example

```
~> salloc --partition=small-g --exclusive --nodes=1 --tasks=1 --gpus-per-node=1 --cpus-per-task=6 --time=5
~> srun gpu_check -l
MPI 000 - OMP 000 - HWT 001 (CCD0) - Node nid005066 - RT_GPU_ID 0 - GPU_ID 0 - Bus_ID d1(GCD4/CCD0)
MPI 000 - OMP 001 - HWT 002 (CCD0) - Node nid005066 - RT_GPU_ID 0 - GPU_ID 0 - Bus_ID d1(GCD4/CCD0)
MPI 000 - OMP 002 - HWT 003 (CCD0) - Node nid005066 - RT_GPU_ID 0 - GPU_ID 0 - Bus_ID d1(GCD4/CCD0)
MPI 000 - OMP 003 - HWT 004 (CCD0) - Node nid005066 - RT_GPU_ID 0 - GPU_ID 0 - Bus_ID d1(GCD4/CCD0)
MPI 000 - OMP 004 - HWT 005 (CCD0) - Node nid005066 - RT_GPU_ID 0 - GPU_ID 0 - Bus_ID d1(GCD4/CCD0)
MPI 000 - OMP 005 - HWT 006 (CCD0) - Node nid005066 - RT_GPU_ID 0 - GPU_ID 0 - Bus_ID d1(GCD4/CCD0)
```

Here your task runs on a single CCD and closest GPU but a job has entire job exclusively allocated

```
~> srun --partition=small-g --exclusive --nodes=1 --tasks=1 --gpus-per-node=1 --cpus-per-task=6 --time=5 \
   gpu_check -l
```

Single step allocation may result with the same binding but if you are lucky; it depends on other jobs already sharing the node!

LUMI

Automatic requeuing

- LUMI is using Slurm **automatic requeuing** of jobs upon **node** failure
  - your job is automatically resubmitted if any of it's nodes allocated fails
  - identical job ID is used and the previous output truncated

- Disable automatic requeuing with **`--no-requeue`** option
- Avoid your output file being truncated with **`--open-mode=append`** option
- Use the value of the **`SLURM_RESTART_COUNT`** variable
  - The value of this variable is `0` for first time the job is run
  - If the job has been restarted then the value is incremented

Generic Job script

```
#!/bin/bash -l
#SBATCH --job-name=examplejob          # Job name
#SBATCH --output=examplejob.o%j        # Name of stdout output file
#SBATCH --error=examplejob.e%j         # Name of stderr error file
#SBATCH --partition=standard           # Partition name, mandatory for job to be scheduled
#SBATCH --nodes=10                     # Total number of nodes
#SBATCH --ntasks=640                   # Total number of mpi tasks
#SBATCH --mem=100G                     # Allocate 100 gigabytes memory per node
#SBATCH --time=1-12:00:00              # Run time (d-hh:mm:ss)
#SBATCH --mail-type=all                # Send email at begin and end of job
#SBATCH --account=project_<id>         # Project for billing, mandatory for job to be scheduled
#SBATCH --mail-user=username@domain.com
# Any other commands must follow the #SBATCH directives

# Launch MPI code srun
srun ./your_application # Use srun instead of mpirun or mpiexec
```

Examples

```
~> sbatch --account=dummy-invalid-project job.sh
sbatch: error: Batch job submission failed: Invalid account or
account/partition combination specified
```

Missing or wrong project id

```
~> sbatch job.sh
sbatch: error: Batch job submission failed: No partition
  specified or system default partition
```

Missing or wrong project id

```
#SBATCH --ntasks-per-node=8
#SBATCH --cpus-per-task=8
~> sbatch job.sh
sbatch: error: Batch job submission failed: Requested node
  configuration is not available
```

Not all 64 cores of a node are available to users' jobs

Job scripts and modules

- You need to load modules to set specific environment for your job
  - Otherwise it is propagated from your current shell
  - Remember the sheebang line
- With the **LUMI** software stack
  - Use partition modules **`partition/C/G/L`** to set target architecture for the application or the library
  - Do not confuse with Slurm **`--partition`** selection (access) option
- With CrayEnv (native) software environment
  - Use **`craype-*`** module family to set target architecture
- Hardcoding modules in the `bashrc` profile may cause troubles

Job arrays

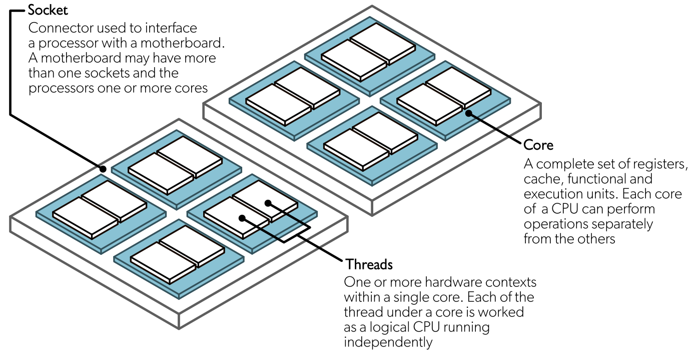
- Slurm job array submits a given number of independent jobs
- Use **`--array`** option to define the number of array tasks
- The **`SLURM_ARRAY_TASK_ID`** environment variable identifies each array task uniquely
- Job arrays use IDs of the form **`<jobid>_<arrayindex>`**
    - pending array task are shown as one entry with it's IDs combined
    - running ones are shown as individual jobs
- Job array is subject to limits same as single job (see partition's limits)

```
#!/bin/bash
#SBATCH --array=1-16
#SBATCH --output=array_%A_%a.out
#SBATCH --error=array_%A_%a.err
#SBATCH --time=01:00:00
#SBATCH --ntasks=1
#SBATCH --mem=4G

# Print the task index. echo "My SLURM_ARRAY_TASK_ID: " $SLURM_ARRAY_TASK_ID
srun ./myapp --input input_data_${SLURM_ARRAY_TASK_ID}.inp
```

Concept of socket, core and threads

- Compute nodes use Non-Uniform Memory Access (NUMA) design
  - 2 threads per core, 64 cores, 4 NUMA domains per socket
  - 1 socket for LUMI-G nodes
  - 2 sockets for LUMI-C nodes
- Memory in the local NUMA node can be accessed faster
- Slum can bind process or thread to a specific core
  - For improved memory access performance
  - It works only for exclusive node access,
  - Default for the `standard` and `standard-g` partitions


Socket
Connector used to interface a processor with a motherboard. A motherboard may have more than one sockets and the processors one or more cores
Core
A complete set of registers, cache, functional and execution units. Each core of a CPU can perform operations separately from the others
Threads
One or more hardware contexts within a single core. Each of the thread under a core is worked as a logical CPU running independently

Slurm tasks

- Task is a single application process executing as a part of job step
  - Single step can execute multiple tasks (e.g. MPI application parallel processes)
  - Each task can spawn its threads (with a hybrid MPI-OpenMP approach)
- For a job (step) with multiple tasks Slurm can
  - Distribute tasks across nodes, sockets, cores and multi-threads
  - Bind tasks to physical CPU cores or NUMA domains
  - Bind tasks to GPUs
  - Allocate multiple CPU cores for task's thread execution

LUMI

Tasks distribution

- Slurm can use different policies to distribute tasks (MPI ranks, logical processes)
  - **`--distribution=<dist>`** option
  - **`<dist>`** can be subdivided in multiple levels for nodes, sockets and cores
  - Requires exclusive access
- Node level
  - **block (default)** distributes tasks to a node such that consecutive tasks share a node
  - **cyclic** consecutive tasks are distributed over consecutive nodes (in a round-robin fashion)
- Socket level
  - **`block`** consecutive tasks are distributed on the same socket
  - **`cyclic (default)`** tasks are distributed in a round-robin fashion across sockets
- Core level
  - inherits from second distribution method
- Combine distribution levels with semicolon, for instance **--distribution=block:block**

Multi-threading

- Hyperthreads
  - **`--hint=nomultithread`** (default) Slurm option disables use of hyperthreads
  - Hardware threads are visible as cores 64-127 (LUMI-G) 128-255 (LUMI-C)
- Software multi-threading
  - OpenMP runtime controls thread affinity and pinning
  - Display binding with **`OMP_DISPLAY_AFFINITY=TRUE`** environmental variable
  - **`OMP_PLACES`** defines where to pin threads on, values threads, cores, sockets
  - **`OMP_PROC_BIND`** defines how threads are mapped to the places
    - **`spread`** distributes (spread) the threads as evenly as possible
    - **`close`** binds threads close to the master thread
    - **`master`** binds threads to the same place as the master thread
    - **`false`** allows threads to be moved between places and disables thread affinity

Binding tasks to resources

- Requires exclusive access
- CPU binding (srun only) **`--cpu-bind=<bind>`**
  - **`threads`** tasks are pinned to the logical threads
  - **`cores`** tasks are pinned to the cores
  - **`sockets`** tasks are pinned to the sockets
  - **`map_cpu:<list>`** custom bindings of tasks with <list> a comma-separated list of CPUIDs
  - **`mask_cpu:<list>`** custom bindings of tasks with <list> a comma-separated hexadecimal values of mask for cores
- GPU binding with **`--gpu-bind=<bind>`**
  - **closest** binds each task to the closest GPU (can bind to multiple GPUs)
  - **map_gpu:<list>** custom bindings of tasks with <list> a comma-separated list of GPUIDs
  - **mask_gpu:<list>** custom bindings of tasks with <list> a comma-separated hexadecimal values of mask for GPUs
- Memory binding is also possible

Combining tasks and threads

- For a hybrid MPI+OpenMP jobs use **`--cpus-per-task`** option
  - Allocates multiple cores per task (MPI rank)
  - It still requires **`OMP_NUM_THREADS`** for explicit control
  - ***NOTE***: Beginning with 22.05, srun is not inheriting the --cpus-per-task value requested by salloc or sbatch. On LUMI the behavior is patched but in some cases still needs to be requested again with the call to srun or set with the **`SRUN_CPUS_PER_TASK`** environment variable if desired for the task(s)

Multi GPU runs

- Automatic GPU assignment
  - **`--gpus-per-node, --gpu-bind=closest`**
  - May assign multiple GPUs to a single task
- Explicit GPU mapping
  - **`--gpu-bind=map_gpu:<gpu_id_for_task_0>, <gpu_id_for_task_1>,...`**
  - Using ROCm environment variable **`ROCR_VISIBLE_DEVICES`**
  - Custom **`select_gpu`** wrapper script
  - Caution with **`--gpus-per-task`** option, it can brake direct GPU communication
- GPU-aware MPI
  - Turn on with **`MPICH_GPU_SUPPORT_ENABLED=1`** MPI variable
  - Allows to use device pointers (buffers)
  - Map tasks to network interfaces with **`MPICH_OFI_NIC_POLICY=GPU`**

Example

See
https://docs.csc.fi/apps/gromacs/#example-batch-script-for-lumi-full-gpu-node

```
#!/bin/bash
#SBATCH --partition=standard-g
#SBATCH --account=<project>
#SBATCH --time=00:15:00
#SBATCH --nodes=1
#SBATCH --gpus-per-node=8
#SBATCH --ntasks-per-node=8
module load GROMACS/2023.2-cpeGNU-22.12-hipSYCL-GPU
```

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

```
cat << EOF > select_gpu
#!/bin/bash
export ROCR_VISIBLE_DEVICES=\$SLURM_LOCALID
exec \$*
EOF
chmod +x ./select_gpu
```

Exclusive access allows custom CPU/thread binding

```
export OMP_NUM_THREADS=7
CPU_BIND="mask_cpu:fe000000000000,fe00000000000000"
CPU_BIND="${CPU_BIND},fe0000,fe000000"
CPU_BIND="${CPU_BIND},fe,fe00"
CPU_BIND="${CPU_BIND},fe00000000,fe0000000000"
```

Enables GPU acceleration within application code

```
srun --cpu-bind=$CPU_BIND ./select_gpu gmx_mpi \
   mdrun -nb gpu -bonded gpu -pme gpu -update gpu \
   <gromacs_opts>
```

Enables GPU awarness with MPI

This gives you proper task / GPU assignment

Note: the module from this example does not provide PME GPU decomposition. For multi-node runs you should consider a version with HeFFTe library enabled.

Simultaneous job steps

- Slurm allows simultaneous job steps in a single allocation (job)
- Example from the srun manual

```
#!/bin/bash
srun -n4 prog1 &
srun -n3 prog2 &
srun -n1 prog3 &
srun -n1 prog4 &
wait
```

- Please be cautious when using such job construction
- We have seen serious issues with such jobs (should be fixed now)
- Simultaneous steps may help to fully utilize standard partition nodes with jobs that not scale
- You may consider combining **`--exclusive`** and **`--exact`** srun options to distribute step tasks within the node

Low-noise mode

- LUMI-G nodes have the *low-noise* mode activated
  - There are now eight cores reserved (#0, 8, 16, 24, 32, 40, 48 and 56)
  - Only 56 cores are available to the jobs
  - Jobs requesting 64 cores/node will never run
  - This eliminates OS jitter and allows symmetric NUMA task bindings
- Default core bindings may be sub-optimal
  - Thread team belonging to one task (MPI rank) may spread on multiple NUMA domains
  - Symmetric distribution requires 7 cores per GPU
  - Use custom binding with CPU masks
  - Works only with exclusive allocation (mind `small-g` and `dev-g` partitions)

Understanding bitmasks

- Slurm uses hexadecimal masks to select which CPU cores tasks should bind to
  - Bits ordered right to left
  - First bit masks core #0
  - Each task need it's mask
- Single mask for 7 cores out of 8 (disabling core #0)
  - Core numbers: `76543210`
  - Binary mask: `11111110`
  - Hexadecimal value: `0xfe`
- Slurm expression
  - Allocation (salloc/sbatch)

```
--nodes=1 --ntasks-per-node=1 --partition=small-g --exclusive
--nodes=1 --ntasks-per-node=1 --partition=standard-g
```

  - Execution (srun)

```
--cpu-bind=mask_cpu:0xfe bash -c 'taskset -cp $$'
```

More bitmasks

- More tasks to allocate full node symmetrically with 7 tasks per each CCD
  - First CCD:
    - Binary mask: **11111110** (8 bits, zero at first), hexadecimal value: **0xfe** (2 digits)
  - Second CCD:
    - **1111111000000000** (16 bits, zeros at first 9 bits), hexadecimal value: **0xfe00** (4 digits)
  - Third CCD:
    - **111111100000000000000000** (24 bits), hexadecimal value: **0xfe0000** (6 digits)
  - ...
- Complete masks
  - sbatch/salloc: **--ntasks-per-node=8 --exclusive**
  - srun:

```
--cpu-bind=mask_cpu:0xfe,0xfe00,\                            #cores  1-7,   9-15
                    0xfe0000,0xfe000000,\                    #cores 17-23, 25-31
                    0xfe00000000,0xfe0000000000,\            #cores 33-39, 41-47
                    0xfe000000000000,0xfe00000000000000      #cores 49-55, 57-63
```

Inspecting binding with MPI

- Use **`MPICH_CPUMASK_DISPLAY=1`** variable to print actual bitmask for MPI ranks

```
[PE_0]: cpumask set to 7 cpus on nid007916, cpumask = 00000000000000000 00000000000000000000000000000 00000000000000000000000000000 0000000000000000000000000000 0000000000000000011111110
[PE_1]: cpumask set to 7 cpus on nid007916, cpumask = 00000000000000000 00000000000000000000000000000 00000000000000000000000000000 0000000000000000000000000000 00000000011111110000 00000
[PE_2]: cpumask set to 7 cpus on nid007916, cpumask = 00000000000000000 00000000000000000000000000000 00000000000000000000000000000 0000000000000000000000000000 01111111000000000000 00000
[PE_3]: cpumask set to 7 cpus on nid007916, cpumask = 00000000000000000 00000000000000000000000000000 00000000000000000000000000000 0000000000000000000001111111 00000000000000000000 00000
[PE_4]: cpumask set to 7 cpus on nid007916, cpumask = 00000000000000000 00000000000000000000000000000 00000000000000000000000000000 0000000000000111111100000000 00000000000000000000 00000
[PE_5]: cpumask set to 7 cpus on nid007916, cpumask = 00000000000000000 00000000000000000000000000000 00000000000000000000000000000 0000011111110000000000000000 00000000000000000000 00000
[PE_6]: cpumask set to 7 cpus on nid007916, cpumask = 00000000000000000 00000000000000000000000000000 00000000000000000000000000111 1111000000000000000000000000 00000000000000000000 00000
[PE_7]: cpumask set to 7 cpus on nid007916, cpumask = 00000000000000000 00000000000000000000000000000 00000000000000000011111110000 0000000000000000000000000000 00000000000000000000 00000
```

Another binding diagnose

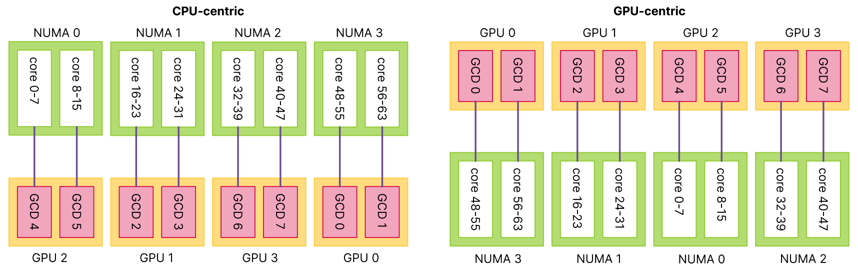
- **`hybrid_check`** tool from the **`lumi-CPEtools`** module (LUMI Software Stack)

```
~> srun --cpus-per-task=7 --hint=nomultithread hybrid_check -r


++ hybrid_check: MPI rank   0/8   OpenMP thread   0/7   on cpu   1/128  of nid007916 mask 1-7
++ hybrid_check: MPI rank   0/8   OpenMP thread   1/7   on cpu   2/128  of nid007916 mask 1-7
++ hybrid_check: MPI rank   0/8   OpenMP thread   2/7   on cpu   3/128  of nid007916 mask 1-7
++ hybrid_check: MPI rank   0/8   OpenMP thread   3/7   on cpu   4/128  of nid007916 mask 1-7
++ hybrid_check: MPI rank   0/8   OpenMP thread   4/7   on cpu   5/128  of nid007916 mask 1-7
++ hybrid_check: MPI rank   0/8   OpenMP thread   5/7   on cpu   6/128  of nid007916 mask 1-7
++ hybrid_check: MPI rank   0/8   OpenMP thread   6/7   on cpu   7/128  of nid007916 mask 1-7
++ hybrid_check: MPI rank   1/8   OpenMP thread   0/7   on cpu   9/128  of nid007916 mask 9-15
++ hybrid_check: MPI rank   1/8   OpenMP thread   1/7   on cpu  10/128  of nid007916 mask 9-15
++ hybrid_check: MPI rank   1/8   OpenMP thread   2/7   on cpu  11/128  of nid007916 mask 9-15
++ hybrid_check: MPI rank   1/8   OpenMP thread   3/7   on cpu  12/128  of nid007916 mask 9-15
++ hybrid_check: MPI rank   1/8   OpenMP thread   4/7   on cpu  13/128  of nid007916 mask 9-15
++ hybrid_check: MPI rank   1/8   OpenMP thread   5/7   on cpu  14/128  of nid007916 mask 9-15
++ hybrid_check: MPI rank   1/8   OpenMP thread   6/7   on cpu  15/128  of nid007916 mask 9-15
```

Adding GPUs to equation

- Note no direct correspondence between the NUMA region order and GPU numbering
  - See **`rocm-smi`** for topology output
- CPU-centric approach (1 task/GPU)
  - Use masks from previous slide
  - Use **`select_gpu`** wrapper
  - Or try **`--gpu-bind=map_gpu:<map>`**
- GPU-centric approach
  - Reorder task cpu masking



Inspecting GPU and CPU bindings

- **`lumi-CPEtools`** module provides
  - **`hybrid_check`** tool program showing masks for CPU binding
  - **`gpu_check`** combines CPU binding with GPU assigment
  - See hands-on examples to experiment with these tools
- Taskset simple linux utility for checking cpu masks

Complete script (CPU centric binding)

```
#!/bin/bash -l
#SBATCH --partition=standard-g        # Partition (queue) name
#SBATCH --nodes=1                     # Total number of nodes
#SBATCH --ntasks-per-node=8           # 8 MPI ranks per node
#SBATCH --gpus-per-node=8             # Allocate one gpu / MPI rank
#SBATCH --time=5                      # Run time (d-hh:mm:ss)
#SBATCH --account=<project_account>   # Project for billing

CPU_BIND="mask_cpu:0xfe,0xfe00,"
CPU_BIND="${CPU_BIND}0xfe0000,0xfe000000,"
CPU_BIND="${CPU_BIND}0xfe00000000,0xfe0000000000,"
CPU_BIND="${CPU_BIND}0xfe000000000000,0xfe00000000000000"

GPU_BIND="map_gpu:4,5,2,3,6,7,0,1"
```

```
export OMP_NUM_THREADS=7
export OMP_PROC_BIND=close
export OMP_PLACES=cores

export MPICH_GPU_SUPPORT_ENABLED=1

srun --cpu-bind=${CPU_BIND} --gpu-bind=${GPU_BIND} \
./hello_jobstep/hello_jobstep
```

This will expose single GPU to one task

Complete script (GPU centric binding)

```
#!/bin/bash -l
#SBATCH --partition=standard-g        # Partition (queue) name
#SBATCH --nodes=1                     # Total number of nodes
#SBATCH --ntasks-per-node=8           # 8 MPI ranks per node
#SBATCH --gpus-per-node=8             # Allocate one gpu / MPI rank
#SBATCH --time=5                      # Run time (d-hh:mm:ss)
#SBATCH --account=<project_account>   # Project for billing

CPU_BIND="mask_cpu:0xfe000000000000,0xfe00000000000000,"
CPU_BIND="${CPU_BIND}0xfe0000,0xfe000000,"
CPU_BIND="${CPU_BIND}0xfe,0xfe00,"
CPU_BIND="${CPU_BIND}0xfe00000000,0xfe0000000000"
```

```
export OMP_NUM_THREADS=7
export OMP_PROC_BIND=close
export OMP_PLACES=cores

export MPICH_GPU_SUPPORT_ENABLED=1

srun --cpu-bind=${CPU_BIND} ./hello_jobstep/hello_jobstep
```

This will expose all GPUs to every task

Complete script (using wrapper)

```
#!/bin/bash -l
#SBATCH --partition=standard-g        # Partition (queue) name
#SBATCH --nodes=1                     # Total number of nodes
#SBATCH --ntasks-per-node=8           # 8 MPI ranks per node
#SBATCH --gpus-per-node=8             # Allocate one gpu / MPI rank
#SBATCH --time=5                      # Run time (d-hh:mm:ss)
#SBATCH --account=<project_account>   # Project for billing

cat << EOF > select_gpu
#!/bin/bash
export ROCR_VISIBLE_DEVICES=\$SLURM_LOCALID
exec \$*
EOF
chmod +x ./select_gpu
```

```
CPU_BIND="mask_cpu:0xfe000000000000,0xfe00000000000000,"
CPU_BIND="${CPU_BIND}0xfe0000,0xfe000000,"
CPU_BIND="${CPU_BIND}0xfe,0xfe00,"
CPU_BIND="${CPU_BIND}0xfe00000000,0xfe0000000000"

export OMP_NUM_THREADS=7
export OMP_PROC_BIND=close
export OMP_PLACES=cores

export MPICH_GPU_SUPPORT_ENABLED=1

srun --cpu-bind=${CPU_BIND} ./select_gpu \
 ./hello_jobstep/hello_jobstep
```

This will expose again single GPUs to each task

Container jobs

- LUMI provides the **`singularity`** runtime included in the HPE Cray OS
  - No modules need to be loaded
  - No custom versions are supported
- No container build service is provided on LUMI currently
  - Bring your own container policy
  - Docker container can be run without manual conversion
  - Use native .sif file or Docker repository/registry
- You can run containers with `srun` directly

```
srun --partition=<partition> --account=<account_id> singularity exec ubuntu_21.04.sif \
cat /etc/os-release
```

Running container from the registry

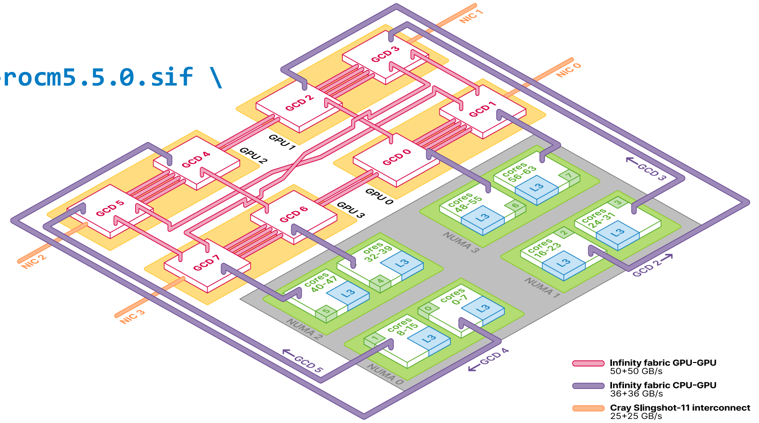
- Pulling container from the DockerHub

```
singularity pull docker://rocm/tensorflow-build:latest-focal-python3.8-rocm5.5.0
```

- Running the container in the interactive mode

```
srun --pty \
  --ntasks=1 --gpus=8 --partition=dev-g \
  --account=<account_id> --time=10 \
  singularity exec tensorflow-build_latest-focal-python3.8-rocm5.5.0.sif \
  rocm-smi --showtopo
```

```
GPU[0]          : (Topology) Numa Node: 3
GPU[1]          : (Topology) Numa Node: 3
GPU[2]          : (Topology) Numa Node: 1
GPU[3]          : (Topology) Numa Node: 1
GPU[4]          : (Topology) Numa Node: 0
GPU[5]          : (Topology) Numa Node: 0
GPU[6]          : (Topology) Numa Node: 2
GPU[7]          : (Topology) Numa Node: 2
```



Binding file systems in the container

- LUMI filesystem (`/scratch` or `/project`) are not accessible from within the container
- They need to be explicitly bound by passing the `-B/--bind` command line option to the `singularity` command
- Simply binding `/scratch` or `/project` will not work
    - These paths are symlinks on LUMI, you must bind full paths to make them available in the container

Running containers in parallel

- For MPI containers, the image must use MPICH ABI-compatible MPI version
- Using the host MPI
  - Install singularity bindings from the LUMI Software Stack
    ```
    module load LUMI partition/<lumi-partition> EasyBuild-user
    eb singularity-bindings-system-cpeGNU-<toolchain-version>.eb -r
    ```
  - Run the container with the specific environment
    ```
    module load singularity-bindings
    srun --partition=<partition> --account=<account> --nodes=2 singularity run <mpi_container>.sif
    ```
- Using the container MPI
  - Run with Slurm generic PMI mode
    ```
    srun --partition=<partition> --account=<account> --nodes=2 \
      --mpi=pmi2 \
      singularity run <mpi_container>.sif
    ```

- No support for OpenMPI at this stage, although second approach may work for specific builds

Known issues

- **`--gpu-bind=closest`** still does not work as expected. On **`standard-g`**, it will not give you the proper GPUs (apart from other problems with Slurm doing the binding). On **`small-g`**, it will not enforce an allocation with the proper CPU cores for the GPUs in your allocation.
- The Slurm GPU binding is still incompatible with shared memory communication between GPUs in different tasks, as is used by, e.g., GPU-aware Cray MPICH intra-node communication. So the trick of avoiding Slurm doing the binding and do a manual binding instead via the **`select_gpu`** script used in the LUMI documentation, is still needed.
- **`MPICH_CPUMASK_DISPLAY=1`** is not showing actual thread binding with all OMP runtimes