JÜLICH
FORSCHUNGSZENTRUM

Batch Usage on JSC

Introduction to Slurm

May 2020 | Chrysovalantis Paschoulas <c.paschoulas@fz-juelich.de> | HPS group @ JSC

Batch System Concepts

- **Resource Manager** is the software responsible for managing the resources of a cluster, usually controlled by a scheduler.
  - It manages resources like tasks, nodes, CPUs, memory, network, etc.
  - It handles the execution of the jobs on the compute nodes.
  - It makes sure that jobs are not overlapping on the resources.
- **Scheduler** is the software that controls user's jobs on a cluster according to policies. It receives and handles jobs from the users and controls the resource manager. It offers many features like:
  - Scheduling mechanisms (backfill, fifo, etc)
  - Partitions, queues and QoS to control jobs according to policies/limits
  - Interfaces for defining work-flows (jobscripts) or job dependencies and commands for managing the jobs (submit, cancel, etc)
- **Batch-System/Workload-Manager** is the combination of a scheduler and a resource manager. It combines all the features of these two parts in an efficient way.

JSC Batch Model

- Job **scheduling according to priorities**. The jobs with the highest priorities will be scheduled next.
- **Backfilling scheduling algorithm**. The scheduler checks the queue and may schedule jobs with lower priorities that can fit in the gap created by freeing resources for the next highest priority jobs.
- **No node-sharing**. The smallest allocation for jobs is one compute node. Running jobs do not disturb each other.
- CPU-Quota modes: **monthly** and **fixed**. The projects are charged on a monthly base or get a fixed amount until it is completely used.
- Accounted CPU-Quotas/job = Number-of-nodes x Walltime x cores/node (corehours)
- Contingent/CPU-Quota states for the projects (for monthly mode): normal, low-contingent, no-contingent.
- Contingent priorities: ***normal*** > ***lowcont*** > ***nocont***. Users without contingent get some penalties for the their jobs, but they are still allowed to submit and run jobs.

Slurm Introduction (1)

- **Slurm** is the chosen Batch System (Workload Manager) that is used on JURECA. Slurm is an open-source project developed by SchedMD. For our clusters *psslurm*, which is a plug-in of *psid* daemon and part of the Parastation Cluster tools, is replacing *slurmd* on the compute nodes. *psslurm* is under development by ParTec and JSC in the context of our collaboration.

- Slurm's configuration on our clusters:
  - High-availability for the main daemons slurmctld and slurmdbd.
  - Backfilling scheduling algorithm.
  - No node-sharing.
  - Job scheduling according to priorities.
  - Accounting mechanism: slurmdbd with MySQL/MariaDB database.
  - User and job limits configured by QoS and Partitions.
  - No preemption configured. Running jobs cannot be preempted.
  - Prologue and Epilogue, with pshealthcheck from Parastation.
  - Generic resources (GRES) for different types of resources on the nodes.

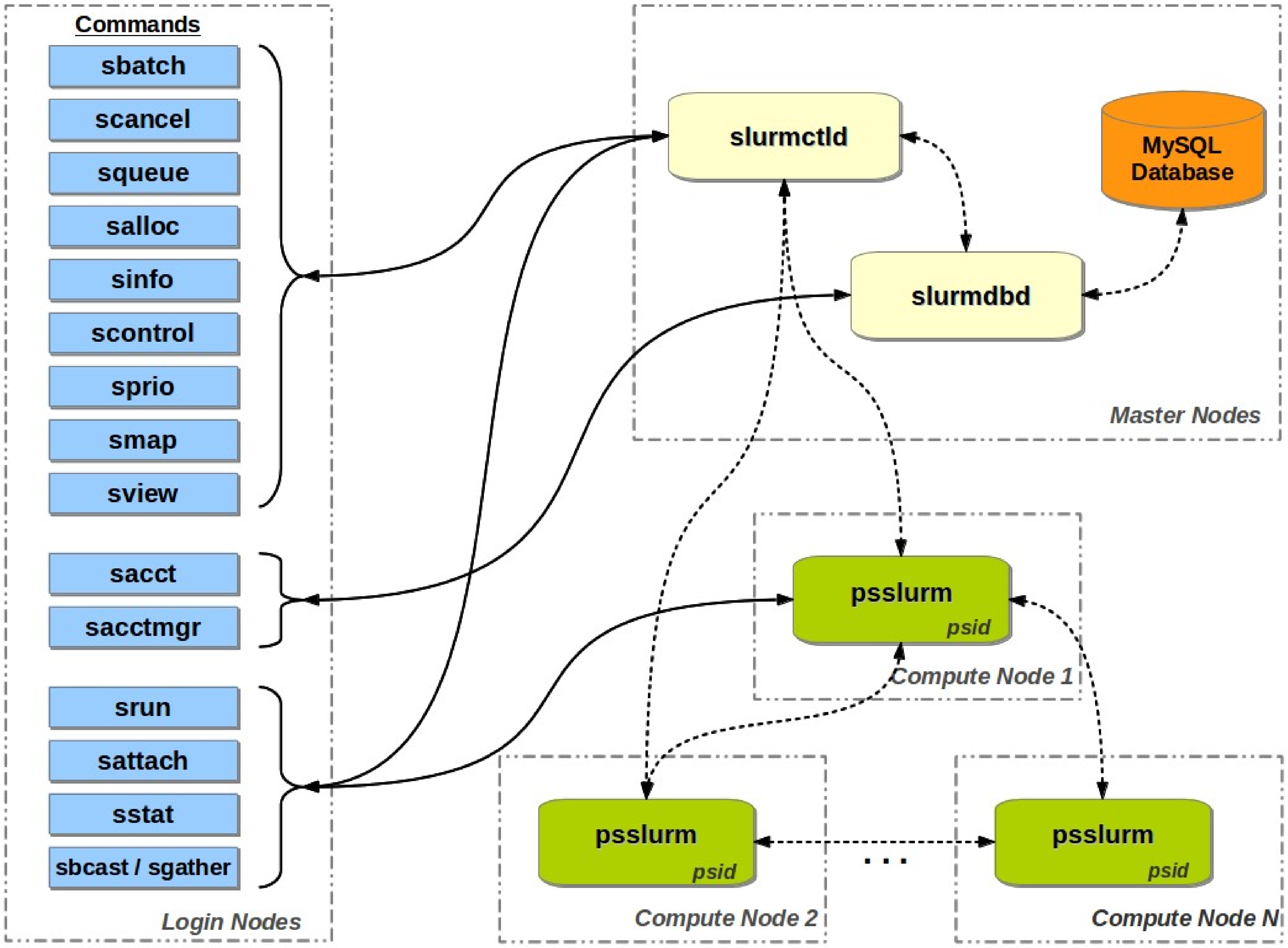
Slurm Introduction (2)

- Slurm groups the compute nodes into Partitions (similar to queues from Moab). Some limits and policies can be configured for each Partition:
    - allowed users, groups or accounts
    - max. nodes and max. wall-time limit per job
- Other limits are enforced also by the Quality-of-Services (QoS), according to the contingent of user's group, e.g. max. wall-time limit, max number or queued or running jobs per user, etc...
- Default limits/settings are used when not given by the users, like: number of nodes, number of tasks per node, wall-time limit, etc.
- According to project's contingent status jobs are given certain QoS:
    - ***normal***: group has contingent, high job priorities.
    - ***lowcont***: this months contingent was used.

      penalty -> lower job priorities, max. wall-time limit and max. running jobs
    - ***nocont***: all contingent of the 3 months time-frame was used.

      penalty -> lowest job priorities, lower max. wall-time limit and max. jobs
    - ***suspended***: the group's project has ended; user cannot submit jobs

Slurm Architecture



Generic Resources

- Slurm provides the functionality to define generic resources (GRES) for each node type. These generic resources can be used during job submissions in order to allocate nodes with specific resources or features. Users can request GRES with the “`--gres`” submission option.

- On our clusters we have set 2 types of GRES:
  - **Memory**: e.g. `mem96, mem128, mem256, mem512, mem1024, etc..`
  - **GPUs**: `gpu:[0-N]`

- We use GRES for 2 reasons:
  - Users can specify the resources they request during submission.
  - We keep track of those resources for accounting and statistics: GRES info are stored in Slurm database.

Slurm – User Commands (1)

- **`salloc`** is used to request interactive jobs/allocations.
- **`sattach`** is used to attach standard input, output, and error plus signal capabilities to a currently running job or job step.
- **`sbatch`** is used to submit a batch script (which can be a bash, Perl or Python script).
- **`scancel`** is used to cancel a pending or running job or job step.
- **`sbcast`** is used to transfer a file to all nodes allocated for a job.
- **`sgather`** is used to transfer a file from all allocated nodes to the currently active job. This command can be used only inside a job script.
- **`scontrol`** provides also some functionality for the users to manage jobs or query and get some information about the system configuration.
- **`sinfo`** is used to retrieve information about the partitions, reservations and node states.
- **`smap`** graphically shows the state of the partitions and nodes using a curses interface. We recommend **`llview`** as an alternative which is supported on all JSC machines.

Slurm – User Commands (2)

- **`sprio`** can be used to query job priorities.
- **`squeue`** allows to query the list of pending and running jobs.
- **`srun`** is used to initiate *job-steps* mainly within a job or start an interactive jobs. A job can contain multiple job steps executing sequentially or in parallel on independent or shared nodes within the job's node allocation.
- **`sshare`** is used to retrieve fair-share information for each user.
- **`sstat`** allows to query status information about a running job.
- **`sview`** is a graphical user interface to get state information for jobs, partitions, and nodes.
- **`sacct`** is used to retrieve accounting information about jobs and job steps in Slurm's database.
- **`sacctmgr`** allows also the users to query some information about their accounts and other accounting information in Slurm's database.

** For more detailed info please check the online documentation and the man pages.*

Slurm – Job Submission

- There are 2 commands for job allocation: **sbatch** is used for batch jobs and **salloc** is used to allocate resource for interactive jobs. The format of these commands:

  sbatch [options] <jobscript> [args...] // salloc [options] [<command> [args...]]

- List of the most important submission/allocation options:

`-A|--account`	`Charge CPU-Quota to specified account (budget ID).`
`-c|--cpus-per-task`	`Number of logical CPUs (hardware threads) per task.`
`-e|--error`	`Path to the job's standard error.`
`-i|--input`	`Connect the jobscript's standard input directly to a file.`
`-J|--job-name`	`Set the name of the job.`
`--mail-user`	`Define the mail address for notifications.`
`--mail-type`	`When to send mail notifications. Options: BEGIN,END,FAIL,ALL`
`-N|--nodes`	`Number of compute nodes used by the job.`
`-n|--ntasks`	`Number of tasks (MPI processes).`
`--ntasks-per-node`	`Number of tasks per compute node.`
`-o|--output`	`Path to the job's standard output.`
`-p|--partition`	`Partition to be used from the job.`
`-t|--time`	`Maximum wall-clock time of the job.`
`--gres`	`Request nodes with specific Generic Resources.`
`--disable-turbomode`	`Disable CPU turbo mode.`
`--globres`	`Custom global resources, like filesystem licenses, etc...`

Slurm - Submission filter and GRES

- Slurm is using a submission filter with the following functionality*:
  - Deny jobs requesting multiple partitions, we allow only one.
  - Disable the --requeue options. We do not allow users to requeue their jobs.
  - Deny submission if budget account was not defined (with `--account` or `-A`).
  - By default add the memXXX GRESs when missing, users can always specify the memXXX GRES if they want.
  - When a job is submitted in the partitions with GPUs then the submission is denied if no gpu GRES was specified.
  - Deny jobs with wrong memXXX GRESs, e.g. job submitted to mem512 partition with GRES mem128.

- Examples:
  - Submit a job in the *gpus* partition requesting 4 GPUs per node:

    ```
    sbatch -N 2 -p gpus -A <budgetID> --gres=gpu:4 <job-script>
    ```

  - Submit a job in the *mem512* partition:

    ```
    sbatch -N 4 -p mem512 -A <budgetID> --gres=mem512 <job-script>
    ```

Slurm – Spawning Command

- With **`srun`** the users can spawn any kind of application, process or task inside a job allocation. *srun* should be used either:
  1. Inside a job script submitted by sbatch (starts a job-step).
  2. After calling salloc (execute programs interactively).
- Command format:

```
srun [options...] <executable> [args...]
```

- `srun` accepts almost all allocation options of `sbatch` and `salloc`. There are however some other unique options:

```
--forward-x            Enable X11 forwarding only for interactive jobs.
--pty                  Execute a task in pseudo terminal mode.
--multiprog <file>     Run different programs with different arguments
                       for each task specified in a text file.
```

- Note: In order to spawn the MPI applications, the users should always use *srun* and not `mpiexec`.

Job-Script – Serial Job

- Instead of passing options to *sbatch* from the command-line, it is better to specify these options using the "`#SBATCH`" directives inside the job scripts which must be positioned in the very beginning of the job-script!
- Here is a simple example where some system commands are executed inside the job script. This job will have the name "TestJob". One compute node will be allocated for 30 minutes. Output will be written in the defined files. The job will run in the default partition batch.

```
#!/bin/bash

#SBATCH -J TestJob
#SBATCH -N 1
#SBATCH -o TestJob-%j.out
#SBATCH -e TestJob-%j.err
#SBATCH --time=30
#SBATCH -A <budgetID>

sleep 5

hostname
```

Job-Script – Tasks in Parallel

- Here is a simple example of a job script where we allocate 4 compute nodes for 1 hour. Inside the job script, with the srun command we request to execute on 4 nodes with 2 process per node the system command *hostname*, requesting a walltime of 10 minutes. In order to start a parallel job, users have to use the *srun* command that will spawn processes on the allocated compute nodes of the job.

```
#!/bin/bash

#SBATCH -J TestJob
#SBATCH -N 4
#SBATCH -o TestJob-%j.out
#SBATCH -e TestJob-%j.err
#SBATCH --time=10
#SBATCH -A <budgetID>

srun --ntasks-per-node=2 hostname
```

Job-Script – OpenMP Job

- In this example the job will execute an OpenMP application named “omp-prog”. The allocation is for 1 node and by default, since there is no node-sharing, all CPUs of the node are available for the application. The output filenames are also defined and a walltime of 2 hours is requested. Note: It is important to define and export the variable OMP_NUM_THREADS that will be used by the executable.

```
#!/bin/bash

#SBATCH -J TestOMP
#SBATCH -N 1
#SBATCH -o TestOMP-%j.out
#SBATCH -e TestOMP-%j.err
#SBATCH --time=02:00:00
#SBATCH -A <budgetID>

export OMP_NUM_THREADS=48

/home/user/test/omp-prog
```

Job-Script – MPI Job

- In the following example, an MPI application will start 204 tasks on 3 nodes running 68 tasks per node requesting a wall-time limit of 45 minutes in booster partition. Each MPI task will run on a separate core of the CPU. Users can change the modules also inside the jobscript.

```
#!/bin/bash

#SBATCH --nodes=3
#SBATCH --ntasks=204
#SBATCH --output=mpi-out.%j
#SBATCH --error=mpi-err.%j
#SBATCH --time=00:45:00
#SBATCH --partition=booster
#SBATCH -A <budgetID>

module purge
module load Architecture/KNL
module load Intel ParaStationMPI

srun ./mpi-prog   # implied --ntasks-per-node=68
```

Job-Script – Hybrid Job

- In this example, a hybrid MPI/OpenMP job is presented. This job will allocate 5 compute nodes for 2 hours. The job will have 30 MPI tasks in total, 6 tasks per node and 4 OpenMP threads per task. On each node 24 cores will be used (no SMT enabled). **Note**: It is important to define the environment variable “`OMP_NUM_THREADS`” and this must match with the value of the option “`--cpus-per-task/-c`”.

```
#!/bin/bash

#SBATCH -J TestJob
#SBATCH -N 5
#SBATCH -o TestJob-%j.out
#SBATCH -e TestJob-%j.err
#SBATCH --time= 02:00:00
#SBATCH --partition=large
#SBATCH -A <budgetID>

export OMP_NUM_THREADS=4

srun -N 5 --ntasks-per-node=6 --cpus-per-task=4 ./hybrid-prog
```

Job-Script – Hybrid Job with SMT

- The CPUs on our clusters support Simultaneous Multi-Threading(SMT). SMT is enabled by default for Slurm. In order to use SMT, the users must either allocate more than half of the Logical Cores on each Socket or by setting some specific CPU-Binding(Affinity) options.
- This example presents a hybrid application which will execute “hybrid-prog” on 3 nodes using 2 MPI tasks per node and 24 OpenMP threads per task (48 CPUs per node).

```
#!/bin/bash

#SBATCH --nodes=3
#SBATCH --ntasks-per-node=2
#SBATCH --cpus-per-task=24
#SBATCH --output=mpi-out.%j
#SBATCH --error=mpi-err.%j
#SBATCH --time=00:20:00
#SBATCH --partition=batch
#SBATCH -A <budgetID>

export OMP_NUM_THREADS=${SLURM_CPUS_PER_TASK}
srun ./hybrid-prog
```

Job-Script – Multiple Job-Steps

- Slurm introduces the concept of **job-steps**. A job-step can be viewed as a smaller job or allocation inside the current allocation. Job-steps can be started only with the *srun* command.
- The following example shows the usage of job-steps. With *sbatch* we allocate 32 compute nodes for 6 hours. Then we spawn 3 job-steps. The first step will run on 16 compute nodes for 50 minutes, the second step on 2 nodes for 10 minutes and the third step will use all 32 allocated nodes for 5 hours.

```
#!/bin/bash
#SBATCH -N 32 -t 06:00:00 -p batch -A <budgetID>

srun -N 16 -n 32 -t 00:50:00 ./mpi-prog1
srun -N 2 -n 4 -t 00:10:00 ./mpi-prog2
srun -N 32 --ntasks-per-node=2 -t 05:00:00 ./mpi-prog3
```

- To run multiple job-steps in parallel put the srun commands in background and call wait:

```
#!/bin/bash
#SBATCH -N 10 -t 05:00:00 -p batch -A <budgetID>

srun -N 2 --ntasks-per-node=24 ./mpi-prog1 &
srun -N 8 --ntasks-per-node=2 ./mpi-prog2 &
wait
```

Job Dependencies & Job-Chains

- Slurm supports dependency chains which are collections of batch jobs with defined dependencies. Job dependencies can be defined using the "`--dependency`" or "`-d`" option of *sbatch*. The format is:

  ```
  sbatch -d <type>:<jobID> <jobscript>
  ```

  Available dependency types: `afterany, afternotok, afterok, ...`

- Below is an example of a bash script for starting a chain of jobs. The script submits a chain of "`$NO_OF_JOBS`". Each job will start only after successful completion of its predecessor.

```
#!/bin/bash

NO_OF_JOBS=<no-of-jobs>
JOB_SCRIPT=<jobscript-name>

JOBID=$(sbatch ${JOB_SCRIPT} 2>&1 | awk '{print $(NF)}')

I=0
while [ ${I} -le ${NO_OF_JOBS} ]; do
  JOBID=$(sbatch -d afterok:${JOBID} ${JOB_SCRIPT} 2>&1 | awk '{print
$(NF)}')
  let I=${I}+1
done
```

Job Arrays

- Slurm supports job-arrays which can be defined using the option "`-a`" or "`--array`" of *sbatch* command. To address a job-array, Slurm provides a base array ID and an array index for each job. The syntax for specifying an array-job is: `--array=<range of indices>`
- Slurm exports also 2 env. variables that can be used in the job scripts:

```
SLURM_ARRAY_JOB_ID  : base array job ID
SLURM_ARRAY_TASK_ID : array index
```

- Some additional options are available to specify the std-in/-out/-err file names in the job scripts: "%A" will be replaced by `SLURM_ARRAY_JOB_ID` and "%a" will be replaced by `SLURM_ARRAY_TASK_ID`.

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --output=prog-%A_%a.out
#SBATCH --error=prog-%A_%a.err
#SBATCH --time=02:00:00
#SBATCH --array=1-20
#SBATCH -A <budgetID>

srun -N 1 --ntasks-per-node=1  ./prog input_${SLURM_ARRAY_TASK_ID}.txt
```

Heterogeneous Jobs - Job Packs (1)

- Submission commands:
  - Multiple independent job specifications identified in command line using the ":" separator, e.g.:

    ```
    salloc -N 2 -p gpus : -N 16 -p batch
    ```
  - The job specifications are sent to slurmctld daemon as a list in a single RPC.
  - The entire request is validated and accepted or rejected.
- Job scripts:
  - Use "`#SBATCH packjob`" between the other "`#SBATCH`" options in jobscript to separate job packs and their groups of resources.
  - With srun and the ":" format you can spawn jobsteps using heterogeneous resources. With the srun's option "`--pack-group`" you can define which pack-group (=job in the pack) of resources will be used for the jobsteps, an example of a jobscript:

```
#!/bin/bash
#SBATCH -N 32 --ntasks-per-node=8 -p batch
#SBATCH packjob
#SBATCH -N 1 --ntasks-per-node=1 -p batch
#SBATCH packjob
#SBATCH -N 16 –gres=gpu:4 –ntasks-per-node=4 -p gpus

srun --pack-group=1 exec1 : --pack-group=0 exec1 : --pack-group=2 exec2
```

Heterogeneous Jobs - Job Packs (2)

- Interactive jobs:

```
$ salloc -N2 -p batch : -N2 --gres=gpu:4 -p gpus -t 01:00:00

$ srun --pack-group=1 -N2 –gres=gpu:4 -t 00:05:00 cuda_exec
$ srun ...
```

- In queue:

```
$ squeue
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
           55271+0    batch      bash paschoul PD       0:00      2 (BeginTime)
           55271+1    booster    bash paschoul PD       0:00      2 (BeginTime)
```

- Manage/address heterogeneous jobs using the “+”:

```
$ scontrol show job 55271      → Show all jobs in the pack.
$ scontrol show job 55271+0    → Show first job in pack.
$ scontrol show job 55271+1    → Show second job in pack.
```

- Accounting:

```
$ sacct -j 55271
       JobID    JobName  Partition    Account  AllocCPUS      State ExitCode
------------ ---------- ---------- ---------- ---------- ---------- --------
55271+0            bash    cluster        zam         64    RUNNING      0:0
55271+0.0      hostname                   zam          2  COMPLETED      0:0
55271+0.1      hostname                   zam          2  COMPLETED      0:0
...
```

Interactive Jobs

- Interactive sessions can be allocated using the *salloc* command. The following command will allocate 2 nodes for 30 minutes:

```
$ salloc --nodes=2 -t 00:30:00
```

- After a successful allocation, *salloc* will start a shell on the login node where the submission happened. After the allocation the users can execute *srun* in order to spawn interactively their applications on the compute nodes. For example:

```
$ srun -N 4 --ntasks-per-node=2 -t 00:10:00 ./mpi-prog
```

- The interactive session is terminated by exiting the shell. It is possible to obtain a remote shell on the compute nodes, after *salloc*, by running *srun* with the pseudo-terminal "`--pty`" option and a shell as argument:

```
$ srun --cpu_bind=none -N 2 --pty /bin/bash
```

- It is also possible to start an interactive job and get a remote shell on the compute nodes with *srun* (*not recommended without salloc*):

```
$ srun --cpu_bind=none -N 1 -n 1 -t 01:00:00 --pty /bin/bash -i
```

Further Information

- Updated status of the systems:
  - See „*Message of the day*“ at login.
- Get recent status updates by subscribing to the system high-messages:

  https://www.fz-juelich.de/ias/jsc/EN/News/SystemMessages/systemmessages_node.html
- Check the online documentation of each system.
- User support at FZJ:
  - Email: sc@fz-juelich.de
  - Phone: +49 2461 61-2828

Questions?