JÜLICH
FORSCHUNGSZENTRUM

Batch Usage in JSC
Introduction to Slurm

May 2015 | Chrysovalantis Paschoulas | c.paschoulas@fz-juelich.de

Batch System Concepts (1)

- A **cluster** consists of a set of tightly connected "identical" computers that are presented as a single system and work together to solve computation-intensive problems. The nodes are connected through high speed local network and have access to shared resources like shared file-systems, etc.
- **Batch Processing** is the execution of programs or jobs without user's intervention.
- A **job** is the execution of user-defined work-flows by the batch system.
- **Resource Manager** is the software responsible for managing the resources of a cluster, like tasks, nodes, CPUs, memory, network, etc. It manages also the execution of the jobs. It makes sure that jobs are not overlapping on the resources and handles also their I/O. Usually it is controlled by the scheduler.

Batch System Concepts (2)

- **Scheduler/Workload-Manager** is the software that controlls user's jobs on a cluster. It receives jobs from users and controls the resource manager to make sure that the jobs are completed successfully. Handles the job submissions and put jobs into queues. It offers many features like:
  - user commands for managing the jobs (start, stop, hold)
  - interfaces for defining work-flows and job dependencies
  - interfaces for job monitoring and profiling (accounting)
  - partitions and queues to control jobs according to policies and limits
  - scheduling mechanisms, like backfilling according to priorities

- **Batch System** is the combination of a scheduler and a resource manager. It combines all the features of these two parts in an efficient way. Slurm for example offers both functionalities: scheduling and resource management.

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.
- For each project a Linux group is created where the users belong to. Each user has available contingent from one project only*.
- 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 per job = Number-of-nodes x Walltime
- Contingent/CPU-Quota states for the projects: 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 that will be 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, will replace *slurmd* on the compute nodes. *psslurm* is under development by ParTec and JSC in the context of our collaboration.
- Slurm's configuration in JSC:
  - 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 as back-end storage
  - User and job limits configured by QoS and Partitions.
  - No preemption configured. Running jobs cannot be preempted.
  - Prologue and Epilogue, with pshealthcheck from Parastation.

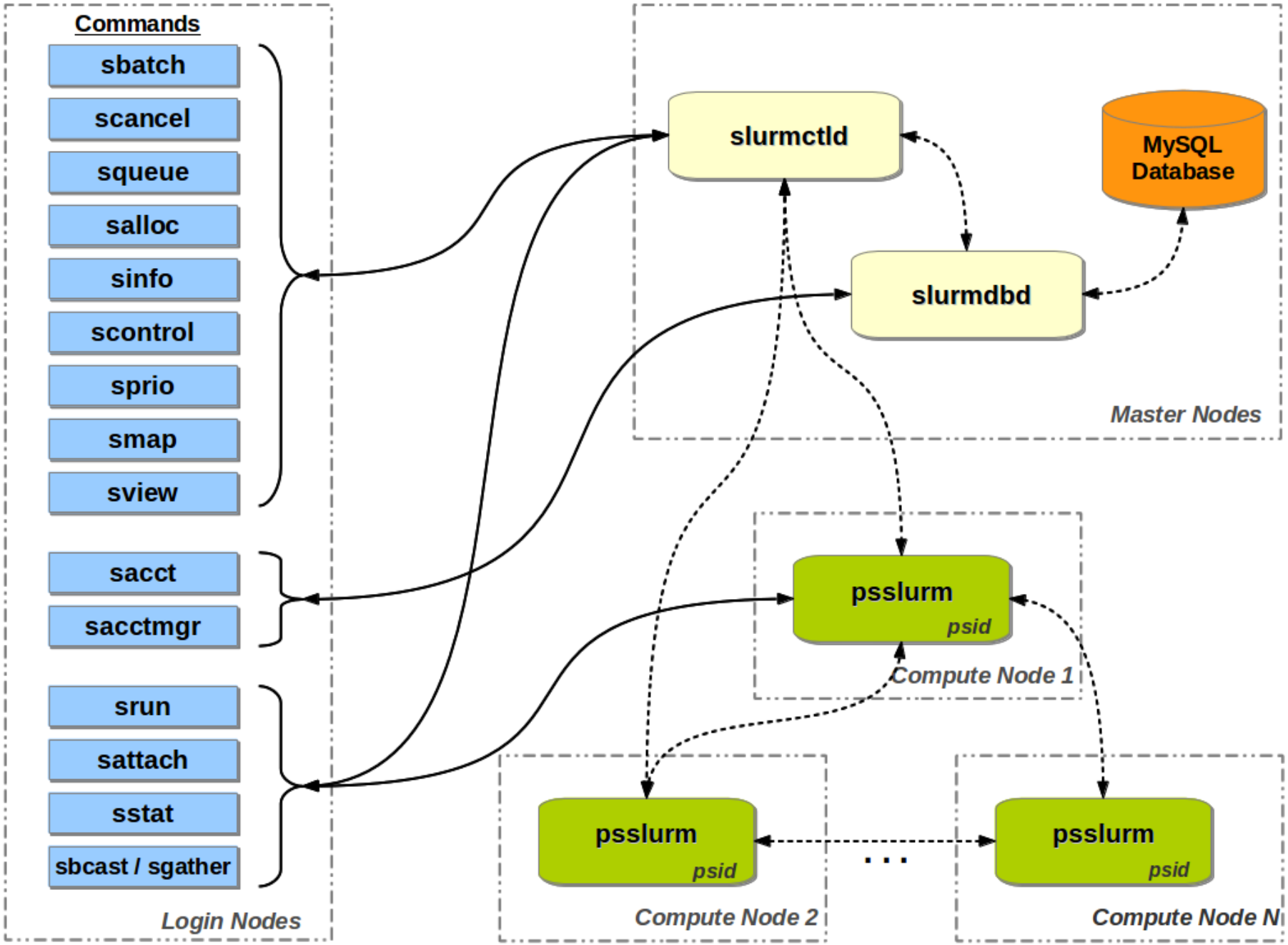
Slurm Introduction (2)

- 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
  - max. submitted/queued jobs per user
- Other limits are enforced also by the Quality-of-Services (QoS), according to the contingent of user's group, e.g. the maximum wall-time limit.
- 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 group's contingent user jobs are given certain QoS:
  - ***normal***: group has contingent, high job priorities.
  - ***lowcont***: this months contingent was used.

    penalty -> lower job priorities
  - ***nocont***: all contingent of the 3 months time-frame was used.

    penalty -> lowest job priorities and lower max. wall-time limit
  - ***nolimits***: special QoS for the admins (testing or maintenance)
  - ***suspended***: the group's project has ended; user cannot submit jobs

Slurm Architecture



JUROPATEST

- **JUROPATEST** is a test system which allows users of JSC's current general purpose supercomputer JUROPA to port and optimize their applications for the new Haswell CPU architecture.
- Partitions:
  - ***batch***: default partition, includes all compute nodes

    Limits: max. 6h, default 30mins; max. 16 nodes, default 1 node
  - ***large***: all compute nodes, intended for large jobs

    Limits: max. 1h; default 30mins; max. 62 nodes; default 17 nodes
  - ***maint***: all compute nodes, used by the admins
- For both batch and large partitions:
  - → max. 4 running jobs
  - → max. 20 pending/queued jobs

System Usage – Modules

- The installed software of the clusters is organized through a hierarchy of modules. Loading a module adapts your environment variables to give you access to a specific set of software and its dependencies.
- Preparing the module environment includes two steps:
  1. Load one of the available tool-chains. There are 3 levels of tool-chains: a) *Compilers*, b) *Compilers+MPI* and c) *FullToolchains.*
  2. Then load other application modules, which where built with currently loaded tool-chain.
- Useful commands:

List available toolchains	`$ module avail`
Load a toolchain	`$ module load intel--para/2014.11`
List all loaded modules	`$ module list`
List available modules	`$ module avail`
Check a package	`$ module spider Boost/1.56.0`
Load a module	`$ module load Boost/1.56.0`
Unload a module	`$ module unload Boost/1.56.0`
Unload all loaded modules	`$ module purge`

System Usage – Compilation

- On our clusters in JSC, we offer some wrappers to the users, in order to compile and execute parallel jobs using MPI. The current wrappers are:

  mpicc, mpicxx, mpif77, mpif90

- Some useful compiler options that are commonly used:

```
-g          Enables debugging.
-openmp     Enables the use of OpenMP.
-O[0-3]     Sets the optimization level.
-L          A path can be given to the linker for searching libraries.
-D          Defines a macro.
-U          Undefines a macro.
-I          Adds further directories to the include file search path.
```

- Compile an MPI program in C++:

```
mpicxx --O2 --o mpi_prog program.cpp
```

- Compile a hybrid MPI/OpenMP program in C:

```
mpicc --openmp --o mpi_prog program.c
```

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> [command args]]
```

- List of the most important submission/allocation options:

```
-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.
```

Slurm – Job Submission Examples

- Submit a job requesting 2 nodes for 1 hour, with 28 tasks per node (implied value of ntasks: 56):

```
sbatch -N2 --ntasks-per-node=28 --time=1:00:00 jobscript
```

- Submit a job array of 4 jobs with 1 node per job, with the default wall-time:

```
sbatch --array=0-3 -N1 jobscript
```

- Submit a job-script in the large partition requesting 62 nodes for 2 hours:

```
sbatch -N62 -p large -t 2:00:00 jobscript
```

- Submit a job requesting all available mail notifications to the specified email address:

```
sbatch -N2 --mail-user=email@address.com --mail-type=ALL jobscript
```

- Specify a job name and the standard output/error files:

```
sbatch -N1 -J myjob -o MyJob-%j.out -e MyJob-%j.err jobscript
```

- Allocate 4 nodes for 1 hour:

```
salloc -N4 --time=60
```

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...]
```

- Some useful 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.
- 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

sleep 5

hostname
```

Job-Script – Parallel Job

- 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 2 nodes with 1 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=60

srun --ntasks-per-node=1 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

export OMP_NUM_THREADS=56

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

Job-Script – MPI Job

- In the following example, an MPI application will start 112 tasks on 4 nodes running 28 tasks per node requesting a wall-time limit of 15 minutes in batch partition. Each MPI task will run on a separate core of the CPU.

```
#!/bin/bash

#SBATCH --nodes=4
#SBATCH --ntasks=112
#SBATCH --output=mpi-out.%j
#SBATCH --error=mpi-err.%j
#SBATCH --time=00:15:00
#SBATCH --partition=batch

srun -N4 --ntasks-per-node=28 ./mpi-prog
```

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 35 MPI tasks in total, 7 tasks per node and 4 OpenMP threads per task. On each node 28 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

export OMP_NUM_THREADS=4

srun -N 5 --ntasks-per-node=7 --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 28 OpenMP threads per task (56 CPUs per node). ** Job was executed on JUROPATEST with 28 Logical Cores per Socket.*

```
#!/bin/bash

#SBATCH --ntasks=6
#SBATCH --ntasks-per-node=2
#SBATCH --cpus-per-task=28
#SBATCH --output=mpi-out.%j
#SBATCH --error=mpi-err.%j
#SBATCH --time=00:20:00
#SBATCH --partition=batch

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 2 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
#SBATCH --time=06:00:00
#SBATCH --partition=batch

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
```

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: `after`, `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 format for specifying an array-job is: `<base job id>_<array index>`
- 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

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

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 -c 7 ./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 today“* at login.
- Get recent status updates by subscribing to the system high-messages:

  http://juelich.de/jsc/CompServ/services/high_msg.html
- JUROPA online documentation:

  http://www.fz-juelich.de/ias/jsc/juropa
- JUROPATEST online documentation:

  http://www.fz-juelich.de/ias/jsc/juropatest
- JUROPATEST - Slurm User Manual (*pdf*):

  http://fz-juelich.de/ias/jsc/juropatest-batchpdf
- JURECA online documentation:

  http://www.fz-juelich.de/ias/jsc/jureca
- User support at FZJ:
  - Email: sc@fz-juelich.de
  - Phone: +49 2461 61-2828

JÜLICH
FORSCHUNGSZENTRUM

Questions?