JÜLICH
FORSCHUNGSZENTRUM

~ JURECA ~

User's Manual
for the Batch System - Slurm

[Slurm integrated with Parastation]

Author	Chrysovalantis Paschoulas
Support	sc@fz-juelich.de
Contributors	Dorian Krause, Philipp Thörnig, Eric Gregory, Theodoros Stylianos Kondylis
Document version	2.3.0 (2017-Dec-11)

Table of Contents

1 Cluster Information

1.1 Introduction

After more than five years of successful operation the JUROPA general-purpose supercomputer has been shutoff on the 24th of June 2015. The successor JURECA (Juelich Research on Exascale Cluster Architectures) is projected to reach a peak performance of about 1.8 PFLOPS per second once fully installed. In order to minimize the service interruption for users the system is installed in two phases. The first phase was consisted by 260 compute nodes and since 2nd of November 2015 the second phase is available and in production including in total 1884 compute nodes.The JURECA system is based on Intel Xeon E5-2680 v3 Haswell CPUs with 12 cores per CPU and utilizes the scalable V-class server architecture of T-Platforms. Compute nodes are dual-socket systems, with 24 cores per node. Different sizes of DDR4 memory will be offered in the full system. The normal (thin) nodes are equipped with 128 GiB memory. For applications with higher memory demands two other types of nodes with 256 GiB per node and 512 GiB per node are available. Accelerated applications can take advantage of the compute nodes equipped with NVIDIA K80 GPUs. Several login nodes are available. Additionally, visualization nodes with large main memory and latest generation NVIDIA K40 GPUs for pre-/post-processing are available. The JURECA compute nodes are interconnected with Mellanox EDR 100 Gbps technology organized in a fully non-blocking fat tree.

Starting from November 2017, JURECA cluster was extended with the addition of a Booster partition. The Booster part of JURECA is based on Intel Xeon Phi 7250F CPUs (KNLs). Each node is equipped with one KNL with 68 cores, 272 hardware threads in total and 96GB RAM.

The WORK and HOME filesystems are mounted from JUST storage cluster offering site-wide access to user data. JURECA also features major advances in the software stack. The system is launched with the latest CentOS 7 Linux enterprise distribution, a Parastation MPI implementation with MPI-3 support and a hierarchical module environment for the simplified usage of the software offerings by JSC. The batch system on JURECA is the open-source Slurm workload manager together with the Parastation resource management, which has been a core element of the JUROPA software stack.

1.2 Cluster Nodes

Type (Node Num.)	Hostname	CPU	Cores(SMT)	RAM	Resources
Thin Compute (1605)	jrc[0036-0455,0491-0940,1138-1382,1395-1884]	2x Intel Xeon E5-2680 v3 (Haswell) @ 2.5GHz	24 (48)	128 GB DDR4	mem128
Fat type-1 (128)	jrc[1010-1137]	2x Intel Xeon E5-2680 v3 (Haswell) @ 2.5GHz	24 (48)	256 GB DDR4	mem256
Fat type-2 (64)	jrc[0946-1009]	2x Intel Xeon E5-2680 v3 (Haswell) @ 2.5GHz	24 (48)	512 GB DDR4	mem512
GPUs (75)	jrc[0001-0035,0456-0490,0941-0945]	2x Intel Xeon E5-2680 v3 (Haswell) @ 2.5GHz	24 (48)	128 GB DDR4	mem128, gpu:4 (2x Nvidia K80)
Visualization type-1 (8)	jrc[1385-1392]	2x Intel Xeon E5-2680 v3 (Haswell) @ 2.5GHz	24 (48)	512 GB DDR4	mem512, gpu:2 (2x Nvidia K40)
Visualization type-2 (2)	jrc[1393-1394]	2x Intel Xeon E5-2680 v3 (Haswell) @ 2.5GHz	24 (48)	1 TB DDR4	mem1024, gpu:2 (2x Nvidia K40)
Booster (1640)	jrc[5001-6640]	Intel Xeon Phi 7250F (KNL) @ 1.40GHz	68 (272)	96 GB DDR4	mem96

The Intel Xeon Phis (KNLs) can be configured in various NUMA modes where according to the mode different number of NUMA domains are exposed to the OS. The available modes are: All2All, Hemisphere, Quadrant, SNC2 and SNC4. SNC2 gives two NUMA domains, SNC4 four and Quadrant depends on the MCDRAM mode. The nodes with the KNLs are equipped with 96 GB of DDR4 but in the SoC there is also availabile 16GB of MCDRAM which can be configured either as a direct-mapped cache (Cache Mode), addressable memory (Memory/Flat Mode) or mixed mode (Hybrid Mode, for us 50%-50%). The Quadrant NUMA mode provides one NUMA domain with Cache Memory mode and two NUMA domains with the other Memory modes.

JURECA's login nodes:

Type (Node Num.)	External Hostname	Internal Hostname	CPU	Cores(SMT)	Resources
Normal Login (12)	jureca.fz-juelich.de jureca[01-12].fz-juelich.de	jrl[01-12]	2x Intel Xeon E5-2680 v3 (Haswell) @ 2.5GHz	24 (48)	256 GB DDR4
Visualization Login (2)	jurecavis.fz-juelich.de jurecavis[01-02].zam.kfa-juelich.de	jrc[1383-1384]	2x Intel Xeon E5-2680 v3 (Haswell) @ 2.5GHz	24 (48)	512 GB DDR4 2x Nvidia K40

The external hostnames “jureca.fz-juelich.de” and “jurecavis.fz-juelich.de” are aliases for redirecting to the login nodes in a round-robin fashion.

1.3 Data Management - Filesystems

On JURECA we provide GPFS shared filesystems. We provide home, scratch and archive file-systems, which have different purposes. The home filesystems are supposed to be used for user's data storage with the safety of backups (TSM backup), the scratch filesystem should be used as a fast storage for the data produced by the jobs (no backup and purged regularly) and the archive ones are to be used for long-term data archiving. Here is a small matrix with all filesystems available to the users:

Filesystem	Mount Point	Description
GPFS $WORK	`/work`	Scratch filesystem – without backup

GPFS $HOME	`/homea`
	`/homeb`
	`/homec`
		Home filesystems – with TSM backup

GPFS $ARCH	`/arch` `/arch2`	Archiving filesystems – with TSM backup. Available only on the login nodes.
GPFS $DATA	`/data`	Special filesystem used only by certain groups – with TSM backup
User local binaries (GPFS)	`/usr/local`	Software repository available via module commands

The GPFS filesystems on JURECA are mounted from JUST storage cluster. JUQUEEN and JUDGE users should be aware that they will work in the same $HOME and $WORK directories as on these production machines. Please note that JSC has already done an automatic migration of all user data from Lustre to GPFS for $HOME and $WORK directories. The old home can be found under “~/juropa/”.

1.4 Access to the Cluster

Users can have access to the login nodes of the system only through SSH connections. There are 12 login nodes in total. There is configured a round-robin shared hostname between the login nodes:

jureca.fz-juelich.de. The users can still connect to specific login nodes by using the individual hostname of each node: *jureca[01-12].fz-juelich.de* For example, to connect to the system, users must execute from their workstation the following command:

```
$ ssh username@jureca.fz-juelich.de
```

or to a specific login node (the second one for example):

```
$ ssh username@jureca02.fz-juelich.de
```

In a similar manner user can access also the visualization login nodes: *jurecavis.fz-juelich.de*

It is not possible to login by supplying username/password credentials. Instead, password-free login based on SSH key exchange is required. The public/private ssh key pair has to be generated on the workstation you are using for accessing JURECA. On Linux or UNIX-based systems, the key pair can be generated by executing:

```
$ ssh-keygen -t [dsa|rsa]
```

It is required to protect the SSH key with a non-trivial pass phrase to fulfill the FZJ security policy. The generated public ssh key contained in the file "`id_dsa.pub`" or "`id_rsa.pub`" on user's workstation must be uploaded through the web interface from Dispatch when initially applying for a user account on JURECA system. This SSH key afterwards will be automatically stored in the file "`$HOME/.ssh/authorized_keys`" on the cluster.

1.5 Shell Environment

The default shell for all users on JURECA is BASH (/bin/bash). After a successful login, user's shell environment is defined in files "`$HOME/.bash_profile`" and "`$HOME/.bashrc`". Since the GPFS filesystems are shared between different clusters in JSC, that means the users' home directories are also shared on all system where the users have access to. This makes it more difficult for the users to create the correct or desired shell environment for each system. In order to solve this issue, a file has been created on all systems which contains a string with the system's name. The file is:

```
/etc/FZJ/systemname
```

This file is available on all login and compute nodes. The users can read this file and depending on the system they are logged-in they can set the desired environment. On JURECA the string that is stored in that file is "`jureca`".

1.6 Modules

The installed software on JURECA 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. The hierarchical organization of the modules ensures that you get a consistent set of dependencies, for example all built with the same compiler version or all relying on the same implementation of MPI. The module hierarchy is built upon toolchains. Loading a toolchain module gives access to all other packages that were built on top of that toolchain. The lowest level contains just the compilers (like Intel compilers `icc, ifort`). After loading a compiler the users have access to the MPI implementations (like ParaStationMPI). And after loading both compiler and MPI modules then it is possible to access all other modules that were built on top of them, like libraries and tools. An application is only accessible to the user when its module is loaded. You can load an application module only when the toolchain modules containing its dependencies are loaded first. Various tools are available as first level toolchains without the need to load any compilers, like TotalView, Vtune, etc. The software also is

organized in stages and users can load the development or older stages to have access to software versions that are not available in current and default stage.

After the extension of JURECA with the Booster partition, the modules were also extended accordingly to support the new architecture. The users now can choose the architecture they wish to use before the load any modules. The default architecture is as expected Haswell which comes from the Cluster partition.

Modules and Toolchains hierarchy

Here is a quick reference of the software that is provided by each toolchain level:

Type	Modules available
Compilers	**GCC:** Gnu compilers with frontends for C, C++, Objective-C, Fortran, Java & Ada **Intel**: Intel C and C++ compilers **PGI**: PGI compiler
MPI	**ParaStationMPI**: for Intel and GCC compilers **IntelMPI**: for Intel compiler **MVAPICH2**: for Intel, GCC and PGI compilers
Math libs	**OLF, MKL**
Other Tools	**CUDA, Inspector, JUBE, Java, TotalView, Vtune, Vampir, ...**

Using the module command

Users should load, unload and query modules though the module command. Several useful module commands are:

Command	Description
`module avail`	Shows the available toolchains and what modules are compatible to load right now according to the currently loaded toolchain.
`module load <modname>/<modversion>`	Loads a specific module. Default version if it is not given.
`module list`	Lists what modules are currently loaded.
`module unload <modname>/<modversion>`	Unloads a module.
`module purge`	Unloads all modules
`module spider <modname>`	Finds the location of a module within the module hierarchy.

As we said above, in order to load a desired application module it is necessary first to load the correct toolchains (dependencies). Therefore, preparing the module environment includes the following steps:

1. [Optional] Choose SW architecture: Architecture/Haswell (default) or Architecture/KNL.
2. Load one of the available compilers, e.g. Intel.
3. [Optional] Load an MPI runtime implementation.
4. Finally load the desired application modules, which where built with the loaded toolchains.

Following we will give some examples of the module command:

List the available toolchains:

```
$ module avail
------------------------ /usr/local/software/jureca/UI/Compilers ------------------------
   GCC/5.4.0                     Intel/2017.0.098-GCC-5.4.0 (D)
   Intel/2016.4.258-GCC-5.4.0    PGI/16.9-GCC-5.4.0
-------------------------- /usr/local/software/jureca/UI/Tools --------------------------
   Advisor/2017_update1                  TotalView/2016T.07.11-beta
…
--------------------------- /usr/local/software/jureca/Devel ----------------------------
   Developers/InstallSoftware       (D)    Stages/Devel (S,D)    Stages/2016b (S) ...
```

Load a SW architecture, e.g. KNL:

```
$ module load Architecture/KNL
```

Load a compiler (without version the default is used):

```
$ module load Intel
```

List all loaded modules (only compiler):

```
$ module list
Currently Loaded Modules:
  1) GCCcore/.5.4.0   3) icc/.2017.0.098-GCC-5.4.0     5) Intel/2017.0.098-GCC-5.4.0
  2) binutils/.2.27   4) ifort/.2017.0.098-GCC-5.4.0
```

Load an MPI impl. (built with Intel compiler):

```
$ module load ParaStationMPI
```

List again all loaded modules:

```
$ module list
Currently Loaded Modules:
  1) GCCcore/.5.4.0                5) Intel/2017.0.098-GCC-5.4.0
  2) binutils/.2.27                6) pscom/.Default
  3) icc/.2017.0.098-GCC-5.4.0     7) ParaStationMPI/5.1.5-1
  4) ifort/.2017.0.098-GCC-5.4.0
```

List all application modules available for currently loaded toolchains:

```
$ module avail
/usr/local/software/jureca/Stages/2016b/modules/all/MPI/intel/2017.0.098-GCC-5.4.0/psmpi/5.1.5-1
   ABINIT/8.0.8b                     ParMETIS/4.0.3
   AMBER/14-update13                 ParaView/5.1.2-OSMesa
   ARPACK-NG/3.4.0                   ParaView/5.1.2                    (D)
   ASE/3.11.0-Python-2.7.12          QuantumESPRESSO/6.0
   Boost/1.61.0-Python-2.7.12        R/3.3.1
…
```

Get information about a package:

```
$ module spider Boost # or module spider  Boost/1.61.0-Python-2.7.12
```

Load an application module:

```
$ module load  Boost/1.61.0-Python-2.7.12
```

Unload all currently loaded modules:

```
$ module purge
```

Load all desired application modules + dependencies at once:

```
$ module load Intel ParaStationMPI <app1> <app2> ...
```

Accessing Old Software

Software on JURECA is organized in stages. By default only the most recent stage with up-to-date software is available. To access older (or in development) versions of software installations, you must manually extend your module path using the command:

```
$ module use /usr/local/software/jureca/<Other-Stage>
```

1.7 Compilers

On JURECA we offer some wrappers to the users, in order to compile and execute parallel jobs using MPI. Different wrappers are provided depending on the MPI version that is used. Users can choose the compiler's version using the module command (see the modules section).

The following table shows the names of the MPI wrapper procedures for the Intel compilers as well as the names of compilers themselves. The wrappers build up the MPI environment for your compilation task, so please always use the wrappers instead of the compilers:

Programming Language	Compiler	Parastation MPI Wrapper	Intel MPI Wrapper
Fortran 90	`ifort`	`mpif90`	`mpiifort`
Fortran 77	`ifort`	`mpif77`	`mpiifort`
C++	`icpc`	`mpicxx`	`mpicpc`
C	`icc`	`mpicc`	`mpiicc`

In the following table we present some useful compiler options that are commonly used:

Option	Description
`-openmp`	Enables the parallelizer to generate multi-threaded code based on the OpenMP directives.
`-g`	Creates debugging information in the object files. This is necessary if you want to debug your program.
`-O[0-3]`	Sets the optimization level.
`-L`	A path can be given in which the linker searches for libraries
`-D`	Defines a macro.
`-U`	Undefines a macro.
`-I`	Allows to add further directories to the include file search path.
`-H`	Gives the include file order. This options is very useful if you want to find out which directories are used and in which order they are applied.
`-sox`	Stores useful information like compiler version, options used etc. in the executable.
`-ipo`	Inter-procedural optimization.
`-axCORE-AVX2`	Indicates the processor for which code is created.
`-help`	Gives a long list of quite a big amount of options.

Compilation Examples

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
```

1.8 Batch model & Accounting

Following, we present the main policies concerning the batch model and accounting that are applied on JURECA:

- Job scheduling according to priorities. The jobs with the highest priorities will be scheduled next.
- Back-filling 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.
- Contingent/CPU-Quota states for the projects: normal, low-contingent, no-contingent.
- Contingent priorities: normal > lowcont > nocont. Users without contingent get a penalty to the priorities of their jobs, but they are still allowed to submit and run jobs.

2 Batch System – Slurm

2.1 Slurm Overview

Slurm is the Batch System (Workload Manager) of JURECA cluster. Slurm (Simple Linux Utility for Resource Management) is a free open-source resource manager and scheduler. It is a modern, extensible batch system that is widely deployed around the world on clusters of various sizes. A Slurm installation consists of several programs and daemons.

The Slurm control daemon (**slurmctld**) is the central brain of the batch system, responsible for monitoring the available resources and scheduling batch jobs. The slurmctld runs on an administrative node with a special setup to ensure availability of the services in case of hardware failures. Most user programs such as *srun*, *sbatch*, *salloc* and *scontrol* interact with the slurmctld. For the purpose of job accounting slurmctld communicates with Slurm database daemon (**slurmdbd**).

Slurm stores all the information about users, jobs and accounting data in its own database. The functionality of accessing and managing these data is implemented in **slurmdbd**. In our case, slurmdbd is configured to use a MySQL database as the back-end storage. To interact with slurmdbd and get information from the accounting database, Slurm provides commands like *sacct* and *sacctmgr*.

In contrast to the Moab/Torque combination where Moab provides scheduling and Torque performs resource management (like batch job start or node health monitoring) Slurm combines the functionality of the batch system and resource management. For this purpose Slurm provides the **slurmd** daemon which runs on the compute nodes and interacts with slurmctld. For the executing of user processes, **slurmstepd** instances are spawned by slurmd to shepherd the user processes. On JURECA cluster no slurmd/slurmstepd daemons are running on the compute nodes. Instead the process management is performed by **psid** the management daemon from the Parastation Cluster Suite which has a proven track record on the JUROPA system. Similar to the architecture of the JUROPA resource management system, where a psid plugin called **psmom** replaces the Torque daemon on the compute nodes, a plugin of psid called **psslurm** replaces slurmd on the compute nodes of JURECA. Therefore only one daemon is required on the compute nodes for the resource management which minimizes jitter (which can affect large-scale applications). For the end-users, there is no real difference visible because of this integration between Slurm and Parastation. Currently, **psslurm** is under active development by ParTec and JSC in the context of the JUROPA collaboration.

The Batch System manages the compute **nodes**, which are the main resource entity of the cluster. Slurm groups the compute nodes into **partitions**. These partitions are the equivalent of queues in Moab. It is possible for different partitions to overlap, which means that the compute nodes can belong to multiple partitions. Also partitions can be configured with certain limits for the **jobs** that will be executed. Jobs are the allocations of resources by the users in order to execute tasks on the cluster for a specified period of time. Slurm introduces also the concept of **job-steps**, which are sets of (possibly parallel) tasks within the jobs. One can imagine job-steps as smaller allocations or jobs within the job, which can be executed sequentially or in parallel during the main job allocation.

In **Figure 1** we present the architecture of the daemons and their interactions with the user commands of Slurm.

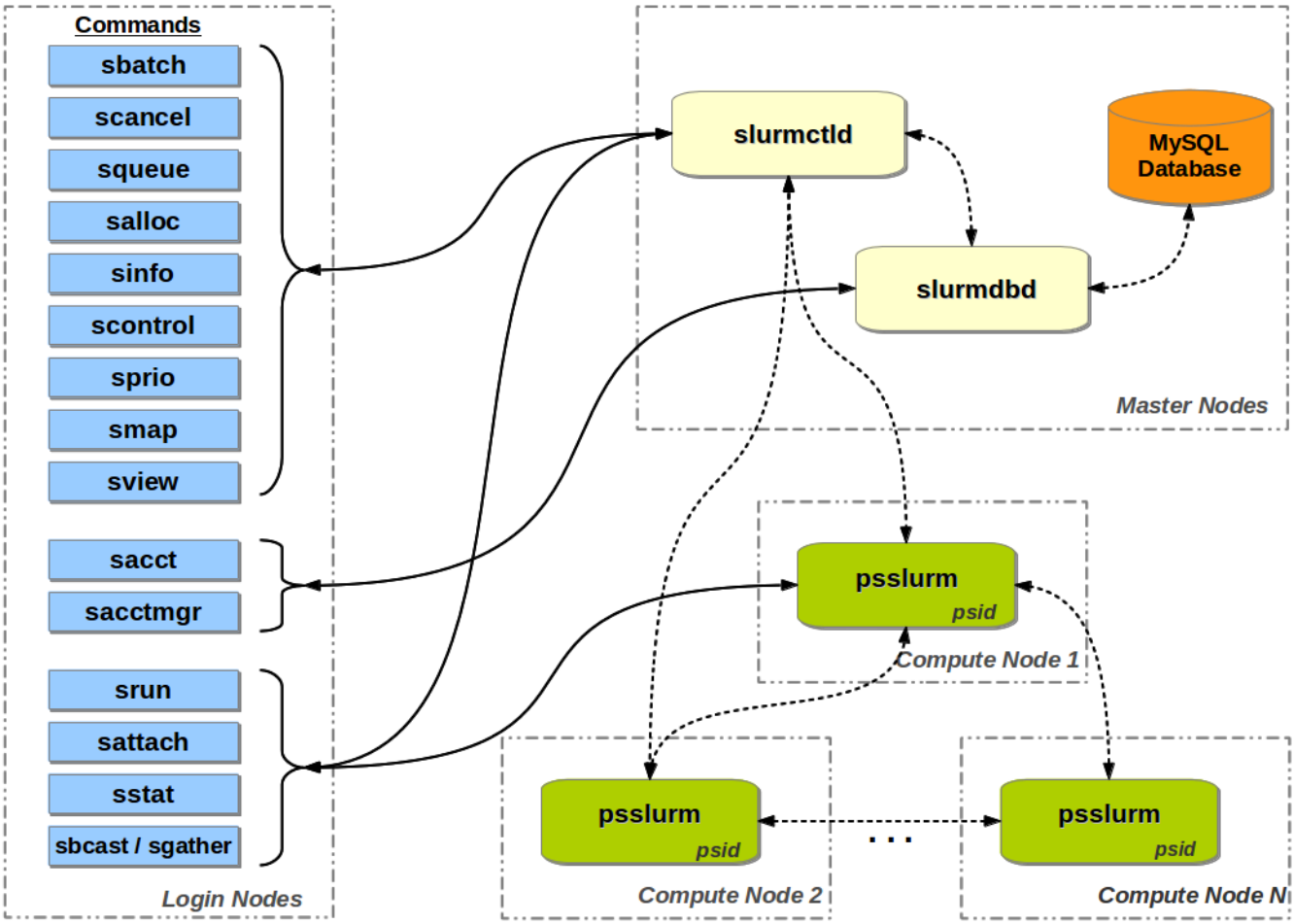

Figure 1.

2.2 Slurm Configuration

- High-Availability for the main controllers **slurmctld** and **slurmdbd**.
- Back-filling scheduling algorithm.
- No node-sharing.
- Job scheduling according to priorities.
- Accounting mechanism: **slurmdbd** with MySQL database as back-end storage.
- User and job limits enforced by QoS (Quality of Service) and some hard-limits configured in the partition settings. There is a QoS for each contingent state: normal, lowcont, nocont and suspended. Users without contingent are set to a different QoS and get a penalty for their job priorities.
- No preemption configured. Running jobs cannot be preempted.
- Prologue and Epilogue, with pshealthcheck from Parastation. The prologue checks the status of the nodes at job start and Epilogue cleans up the nodes after job completion.
- Same limits/configurations for batch and interactive jobs (no difference between batch and interactive jobs for Slurm, different behavior than Moab).

2.3 Partitions

In Slurm multiple nodes can be grouped into partitions which are sets of nodes with associated limits (for wall-clock time, job size, etc.). These limits are hard-limits for the jobs and can not be overruled by the specified limits in QoS. Partitions may overlap and nodes may belong to more than one partition, making **partitions serve as general purpose queues**, similar to Moab's *queues*. The following table shows the current partitions on JURECA with their configured maximum limits and default values:

Partition	Max. walltime per job (according to contingent)	Default walltime per job	Min. nodes per job	Max. nodes per job	Default nodes per job	Max. total used resources per user
devel *(interactive jobs) (thin nodes)*	*normal:* 2 hours *low-/nocont:* 2 hours	30 mins	1	8	1	10 nodes + QoS dependent
batch *(default partition) (batch jobs) (thin + fat type-1)*	*normal:* 24 hours *low-/nocont:* 6 hours	1 hour	1	256	1	QoS dependent
mem256 *(mem. bounded jobs) (fat type-1)*	*normal:* 24 hours *low-/nocont:* 6 hours	1 hour	1	128	1	QoS dependent
mem512 *(mem. bounded jobs) (fat type-2)*	*normal:* 24 hours *low-/nocont:* 6 hours	1 hour	1	32	1	QoS dependent
mem1024 *(mem. bounded jobs) (fat with gpus) (2x Nvidia K40)*	*normal:* 24 hours *low-/nocont:* 6 hours	1 hour	1	2	1	1 running job, 1 submitted job + QoS dependent
develgpus *(interactive jobs) (2x Nvidia K40)*	*normal:* 2 hours *low-/nocont:* 2 hours	30 mins	1	2	1	QoS dependent
gpus *(gpu accel. jobs) (2x Nvidia K80)*	*normal:* 24 hours *low-/nocont:* 6 hours	1 hour	1	32	1	QoS dependent
vis *(gpu accel. jobs) (2x Nvidia K40)*	*normal:* 24 hours *low-/nocont:* 6 hours	1 hour	1	4	1	QoS dependent
develbooster *(devel./compilation)*	*normal:* 6 hours *low-/nocont:* 6 hours	1 hour	1	8	1	10 nodes + QoSdependent
booster *(booster nodes/KNL)*	*normal:* 24 hours *low-/nocont:* 6 hours	1 hour	1	64	1	QoS dependent
modetestbooster *(various KNL conf.)*	*normal:* 24 hours *low-/nocont:* 6 hours	1 hour	1	32	1	64 nodes + QoSdependent
maint maintbooster maintglobal *(used only by admins)*	-	-	-	-	-	-

*Other partitions: *large*, *largegpus*, *largevis and largebooster.*

The *devel*, *develgpus* and *develbooster* partitions are intended for small (usually) interactive jobs focused on development and application optimizations. The *develbooster* should be used also in general for the compilation of users' appliacations targeting the KNL architecture.

The *batch* partition is intended for the normal production jobs which is the default partition and includes the thin and the fat type-1 compute nodes. The *mem256, mem512 and mem1024* partitions are intended for memory bounded jobs and include the fat type nodes.

The *gpus* partition includes the nodes that are equipped with 2 Nvidia K80 GPUs (Note: 4x Nvidia devices/GPUs available on each node, because each K80 card has 2 GPUs inside). The *vis* partition includes both visualization node types with 512 GB and 1024 GB memory and they are equipped with 2 Nvidia K40 GPUs.

The *booster* partition contains 1520 KNL nodes which are confiugured with Quadrant NUMA Mode with Hydrid50 Memory mode. The 24 KNL nodes in *develbooster* partition are configured in the same way but they are meant to be used for software development, small and short tests and also compilation of applications meant for the KNL architecture. The purpose of *modetestbooster* partition is for testing different KNL configurations and includes 96 KNL nodes which are divided into 3 different groups with 32 nodes each.

The *large, largegpus, largevis and largebooster* partitions include the same nodes as *batch, gpus, vis and booster* accordingly and the only difference is that they don't limit the max. number of nodes per job, so they are intended to be used for large jobs.

The *maint** partitions are not for normal usage, instead they are supposed to be used only by admins usually during off-line maintenance.

2.4 Slurm's Accounting Database

Slurm manages its own data in two different ways. First, there is a run-time engine in memory, backed-up with state files that is managed by slurmctld and second, there is the MySQL database that is managed by slurmdbd. Slurm stores all the important information in its MySQL database, like: cluster information, events, accounts, users, associations, QoS and job history. An association is the combination of cluster, account, user and partition. Associations are stored in a tree-like hierarchical structure starting with the root node with the accounts as its children and users as children of the accounts. In each association it is possible to specify fair-share, job limits and QoS.

To interact with slurmdbd and get accounting information from the database Slurm provides the commands *sacct* (and *sacctmgr* for admins and operators or for users to query only information).

2.5 Job Limits – QoS

As we describe above, the limits of the partitions are the hard-limits that put an upper limit for the jobs. However, the actual job limits are enforced by the limits specified in both partitions and Quality-of-Services, which means that first the QoS limits are checked/enforced, but these limits can never go over the partition limits.

One QoS is configured for each possible contingent status: *normal*, *lowcont*, *nocont*. These QoS play the most important role to define the job priorities. By defining those QoS the available range of priorities is separated into three sub-ranges, one for each contingent mode. Also one more QoS is defined with the name *suspended* which will be given to all associations that belong to users/projects that have ended and/or are not allowed to submit jobs anymore. Following we present the list with the configured Quality-of-Services:

Name	Priority	Flags	Max. Walltime per Job	Max. Allocated Nodes per User	Max. Running Jobs per User	Max. Submitted Jobs per User
normal	150,000	DenyOnLimit	24 hours	3764	128	4096
lowcont	100,000	DenyOnLimit	6 hours	1884	64	4096
nocont	50,000	DenyOnLimit	6 hours	1884	64	4096
suspended	0	DenyOnLimit	-	0	0	0

Each association in Slurm's database belongs to one user only. In each association there are two entries regarding the QoS. One entry with the list of available QoS and another entry with the Default-QoS (used when QoS is not specified with options). In every association only one available QoS is defined (same as default) for each user depending on the contingent status. This is implemented in JSC's accounting mechanism and **the users are not allowed to change their QoS**. The limits are enforced to the users by setting the correct QoS for their association according to their contingent. Job limits are enforced by that QoS in combination with the partition limits. If the users request allocations over the limits then the submission will fail (flag DenyOnLimit).

2.6 Generic Resources - GRES

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. Slurm can be configured also to deny allocations which don't specify any GRES for certain nodes or partitions and this feature is used for the some of JURECA's partitions like *gpus* and *mem512*. The GRES configuration can be used also to extract important accounting information about the types of resources that the users are requesting/allocating.

The following table includes all configured generic resources on JURECA:

GRES Name	Description
mem96	96 GB memory on node
mem128	128 GB memory on node
mem256	256 GB memory on node
mem512	512 GB memory on node
mem1024	1024 GB memory on node
gpu	Node equipped with GPUs

The following table show the GRES that are configured for each node type:

Node Type	List of GRES
booster	mem96
thin	mem128
fat type-1	mem256
fat type-2	mem512
gpu	mem128, gpu:4
vis type-1	mem512, gpu:2
vis type-2	mem1024, gpu:2

As it is shown on the previous table, Slurm allows to define multiple resources for each type of nodes. The mem* GRES is not consumable and it has one count, but for the gpu GRES it is configured a number which defines how many GPUs are available for each node type.

The following table shows the partitions and the list of GRES for the nodes that are included in them:

Partitions	**List of GRES**
devel	mem128
batch *large*	mem128, mem256
mem256	mem256
mem512	mem512
mem1024	mem1024
develgpus *gpus* *largegpus*	mem128, gpu:4
vis *largevis*	mem512, mem1024, gpu:2
booster *develbooster* *modetestbooster* *largebooster*	mem96
maint	mem128, mem256, mem512, mem1024, gpu:[2,4]
maintbooster	mem96
maintglobal	mem96, mem128, mem256, mem512, mem1024, gpu:[2,4]

During job submissions, Slurm will deny any submission when a user requests a GRES that is not configured for the desired partition or set of nodes.

Job Submission Filter

During job submission, a submission filter is configured to take certain actions depending on the partition and the resources that are allocated. Here is the list of the configured rules for this filter:

- Deny jobs requesting multiple partitions, only one is allowed.
- Disable the --requeue options. We do not allow users to requeue their jobs.
- Deny job submission on *modetestbooster* partition when Feature is missing.
- By default add the mem* GRES when missing, users can always specify the mem* 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 mem* GRES, e.g. job submitted to *mem512* partition with GRES *mem128*.

The GRES can be defined during submission with option "`--gres=<list of gres>`" of the commands *sbatch* & *salloc*. In chapter 3.2 examples will be given on how to request GRES during submissions.

2.7 Priorities

Slurm schedules the jobs according to their priorities, which means that the jobs with the highest priorities will be executed next. With the back-filling algorithm though, jobs (usually small) with lower priorities can be schedule next if they can fit and run on the available resources before the next high-priority job is scheduled to start. Slurm has a very simple and well defined priority mechanism that allows us to define exactly the batch model we want. Following, we present how Slurm calculates the priorities for each job:

```
Job_priority = (PriorityWeightAge) * (age_factor) +
               (PriorityWeightFairshare) * (fair-share_factor) +
               (PriorityWeightJobSize) * (job_size_factor) +
               (PriorityWeightPartition) * (partition_factor) +
               (PriorityWeightQOS) * (QoS_factor)
```

Slurm uses five factors to calculate the job priorities: Age, Fairshare, Job-Size, Partition and QoS. The possible range of values for the factors is between 0.0 (min) and 1.0 (max). For each factor we have defined a weight that is used in the job-priority equation. Following is the list of weights we have configured:

Weight	**Value**
WeightQOS	200,000
WeightAge	32,500
WeightJobSize	14,500
WeightFairshare	3,000
WeightPartition	0

It is clear now that QoS plays an important role for the calculation of the priorities. With the different QoS that have been defined, it is possible to create different priority ranges according to the contingent of the users. Below follows a table with the priority ranges for each contingent mode:

Contingent Status	**Priority Ranges**
normal	150,001 – 200,000
lowcont	100,001 – 150,000
nocont	50,00 – 100,000
suspended	0

For each contingent state the available range for priorities is 50k and is calculated from three factors: a) job age, b) job size and c) fair-share. In current setup, the partition factor is not used which means no difference in the priorities between different partitions.

2.8 Job Environment

On the compute nodes the whole shell environment is passed to the jobs during submission. With some options of the allocation commands, users can change this default behavior. The users can load modules and prepare the desired environment before job submission, and then this environment will be passed to the jobs that will be submitted. Of course, a good practice is to include module commands inside the job-scripts, in order to have full control of the environment of the jobs.

2.9 SMT

Similar to the Intel Nehalem processors in JUROPA, the Haswell processors in JURECA offer the possibility of Simultaneous Multi-Threading (SMT) in the form of the Intel Hyper-Threading (HT) Technology. With HT enabled each (physical) processor core can execute two threads or tasks simultaneously. The operating system thus lists a total of 48 logical cores or Hardware Threads (HWT). Therefore a maximum of 48 processes can be executed on each compute node without overbooking.

Each compute node on JURECA consists of two CPUs, located on socket zero and one, with 12 physical cores. These cores are numbered 0 to 23 and the hardware threads are named 0 to 47 in a round-robin fashion. **Figure 2** depicts a node schematically and illustrates the naming convention.

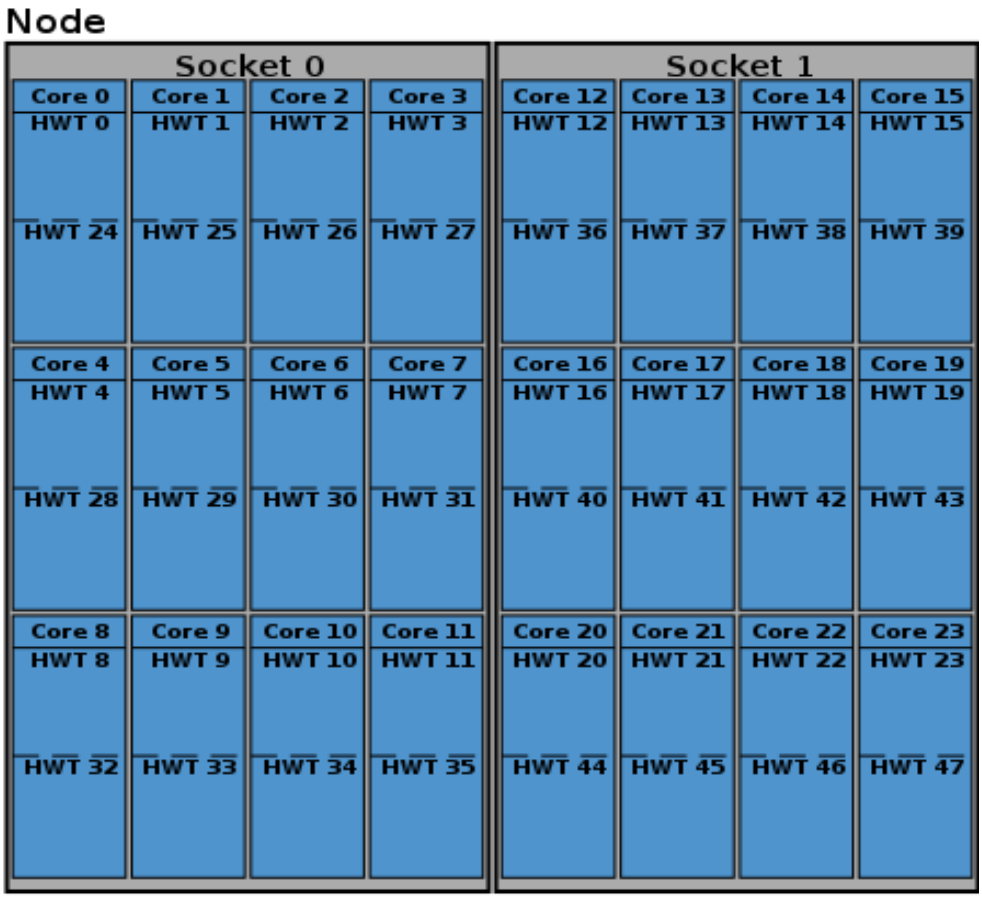

Figure 2.

Using SMT on JURECA

The Slurm batch system on JURECA does not differentiate between physical cores and hardware threads. In the Slurm terminology each hardware thread is a CPU. For this reason each compute node reports a total of 48 CPUs in the *scontrol* show node output. Therefore whether or not threads share a physical core depends on the total number of tasks per node (`--ntasks-per-node` and `--cpus-per-task`) and the process pinning.

The use of the last 24 hardware threads can be disabled with the option `--hint=nomultithread` of *srun* command. This option leads to overbooking of the same logical cores as soon as more than 24 threads are executed. For most application, this option is not beneficial and the default value should be used (`--hint=multithread`).

In chapter "*4.1 Job script examples*", there are some examples about SMT.

How to profit from SMT

Processes which are running on the same physical core will share several of the resources available to that particular core. Therefore, applications will profit most from SMT if processes running on the same core which are complementary in their usage of resources (e.g., complementary computation and memory-access phases). On other hand, processes with similar resource usage may compete for bandwidth or functional units and hamper each other. We recommend to test whether your code profits from SMT or not.

In order to test whether your application benefits from SMT one should compare the timings of two runs on the same number of physical cores (i.e., number of nodes specified with --nodes should be the same for both jobs): One job without SMT (*t1*) and one job with SMT (*t2*). If *t2* is lower than *t1* your application benefits from SMT. In practice, *t1/t2* will be less than 1.5 (e.g., a runtime improvement of maximal 50% will be achieved through SMT). However, applications may show a smaller benefit or even slow down when using SMT.

Please note that the process binding may have a significant impact on the measured run times t1 and t2.

2.10 Processor Affinity

Each JURECA compute node features 24 physical and 48 logical cores. The Linux operating system on each node has been designed to balance the computational load dynamically by migrating processes between cores where necessary. For many high performance computing applications, however, dynamic load balancing is not beneficial since the load can be predicated *a priori* and process migration may lead to performance loss on the JURECA compute nodes which fall in the category of Non-Uniform Memory Access (NUMA) architectures. To avoid process migration, processes can be pinned (or bound) to a logical core through the resource management system. A pinned process (or thread) is bound to a specific set of cores (which may be a single or multiple logical cores) and will only run on the cores in this set.

Slurm allows users to modify the process binding by means of the `--cpu_bind` option of *srun*. While the available options of *srun* are standard across all Slurm installations, the implementation of process affinity is done in plugins and thus may differ between installations. On JURECA a custom pinning implementation is used. In contrast to other options, the processor affinity options need to be directly passed to *srun* and must not be given to *sbatch* or *salloc*. In particular, the option cannot be specified in the header of a batch script.

Note: The option `--cpu_bind=cores` is not supported on JURECA and will be rejected.

Default processor affinity

Since the majority of applications benefit from strict pinning that prevents migration - unless explicitly prevented - all tasks in a job step are pinned to a set of cores which heuristically determines the optimal core set based on the job step specification. In job steps with `--cpus-per-task=1` (the default) each task is pinned to a single logical core as shown in **Figure 3**. In job steps with a `--cpus-per-task` count larger than one (e.g., threaded applications), each task/process will be assigned to a set of cores with cardinality matching the value of `--cpus-per-task`, see **Figure 4**.

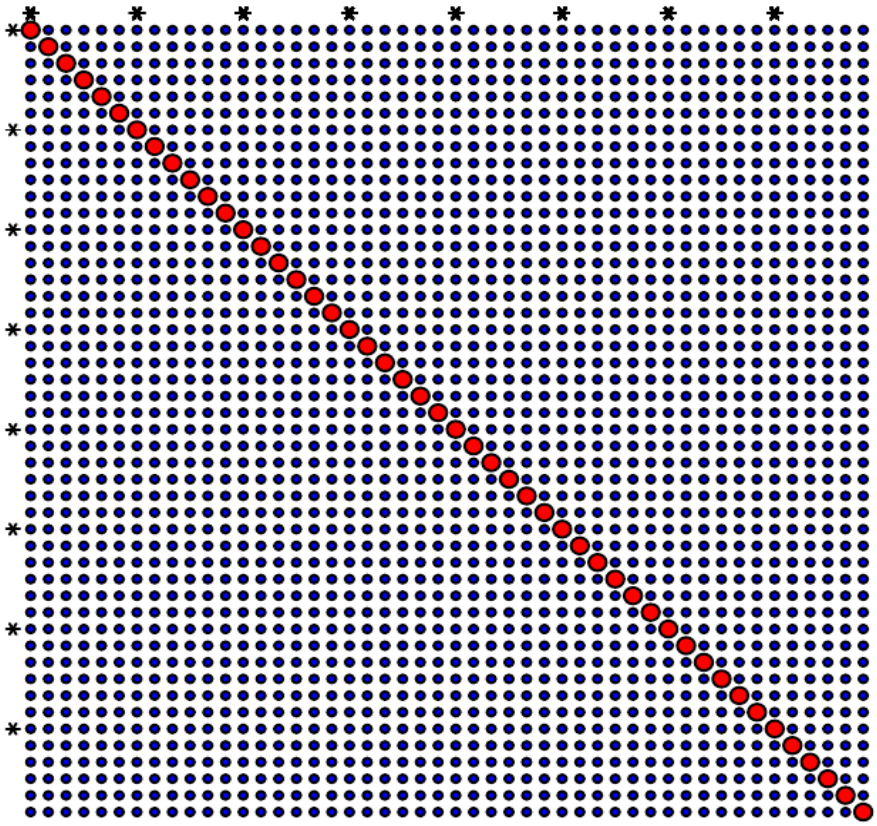

Figure 3.

In **Figure 3** we have the visualization of the processor affinity of a 48 tasks job-step on a single JURECA node. Each column corresponds to a logical core and each row to a task/process. A red dot indicates that the task can be scheduled on the corresponding core. For the purpose of presentation, stars are used to highlight cores/tasks 0, 6, 12, 18, ...42.

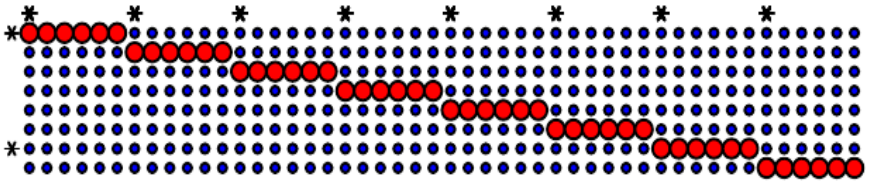

Figure 4.

In **Figure 4** we have the visualization of the processor affinity of a 8 tasks job-step with the option `--cpus-per-task=6` (a hybrid MPI/OpenMP job with 8 MPI processes and `OMP_NUM_THREADS=6`). Pinning of the individual threads spawned by each task is not in the hand of the resource management system but managed by the runtime (e.g., the OpenMP runtime library).

Note: It is important to specify the correct `--cpus-per-task` count to ensure an optimal pinning for hybrid applications.

The processor affinity masks generated with the options `--cpu_bind=rank` and `--cpu_bind=threads` coincide with the default binding scheme.

Note: The distribution of processes across sockets can be affect with the option `-m` of *srun* command. Please see the man page of *srun* for more information.

Binding to sockets

With the option `--cpu_bind=sockets` processes can be bound to sockets, see **Figure 5**.

Figure 5.

In **Figure 5** we have the visualization of the processor affinity for a two tasks job-step with the option `--cpu_bind=sockets`. This option can be further combined with `--hint=nomultithread` to restrict task zero to cores 0 to 11 and task two to cores 12 to 23.

On JURECA, locality domains coincide with sockets so that options `--cpu_bind=ldoms` and `--cpu_bind=sockets` give the same results.

Manual pinning

For advanced use cases it can be desirable to manually specify the binding masks or core sets for each task. This is possible using the options `--cpu_bind=map_cpu` and `–cpu_bind=mask_cpu`.

For example, the following command spawns two tasks pinned to core 1 and 5, respectively:

```
srun -n 2 --cpu_bind=map_cpu:1,5
```

The next command spawns two tasks pinned to cores 0 and 1 (`0x3 = 3 = 20 + 21`) and cores 2 and 3 (`0xC = 11 = 22 + 23`), respectively:

```
srun -n 2 --cpu_bind=mask_cpu:0x3,0xC
```

Disabling pinning

Processor binding can be disabled using the argument --cpu_bind=none to srun. In this case, each thread may execute on any of the 48 logical cores and the scheduling of the processes is up to the operating system. On JURECA the options --cpu_bind=none and --cpu_bind=boards achieve the same result.

3 Slurm User Commands

In this section we will give first a list of all commands with a short description and then later we will describe with more details the functionality of each command, giving also some examples.

3.1 List of Commands

Slurm offers a variety of user commands for all the necessary actions concerning the jobs. With these commands the users have a rich interface to allocate resources, query job status, control jobs, manage accounting information and to simplify their work with some utility commands.

Here is the list of all Slurm's user commands:

`salloc` is used to request interactive jobs/allocations. When the job is started a shell (or other program specified on the command line) is started on the submission host (login node). From the shell *srun* can be used to interactively spawn parallel applications. The allocation is released when the user exits the shell.

`sattach` is used to attach standard input, output, and error plus signal capabilities to a currently running job or job step. One can attach to and detach from jobs multiple times.

`sbatch` is used to submit a batch script (which can be a bash, Perl or Python script). The script will be executed on the first node in the allocation chosen by the scheduler. The working directory coincides with the working directory of the *sbatch* directory. Within the script one or multiple *srun* commands can be used to create job steps and execute (MPI) parallel applications. **Note**: *mpiexec* is not supported on JURECA. *srun* is the only supported method to spawn MPI applications. In the future the *mpirun* command from Intel MPI may be supported.

`scancel` is used to cancel a pending or running job or job step. It can also be used to send an arbitrary signal to all processes associated with a running job or job step.

`sbcast` is used to transfer a file to all nodes allocated for a job. This command can be used only inside a job script.

`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` is primarily used by the administrators to view or modify Slurm configuration, like partitions, nodes, reservations, jobs, etc. However it 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. It has a wide variety of filtering, sorting, and formatting options.

`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.

`sprio` can be used to query job priorities.

`squeue` allows to query the list of pending and running jobs. By default it reports the list of pending jobs sorted by priority and the list of running jobs sorted separately according to the job priority.

`srun` is used to initiate job steps mainly within a job or start an interactive job. *srun* has a wide variety of options to specify resource requirements. 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. For older jobs sacct queries the accounting database.

`sacctmgr` is primarily used by the administrators to view or modify accounting information in Slurm's database. However, it allows also the users to query some information about their accounts and other accounting information.

Note: Man pages exist for all daemons, commands, and API functions. The command option "--help" also provides a brief summary of the available options.

3.2 Allocation Commands

sbatch & salloc

The commands *sbatch* and *salloc* can be used to allocate resources. *sbatch* is used for batch jobs. The arguments for the *sbatch* command is the allocation options followed by the jobscript. *sbatch* gets the allocation options either from the command line or from the job script (using #SBATCH directives). *salloc* is used to allocate resources for interactive jobs.

Command format:

```
sbatch [options] jobscript [args...]

salloc [options] [<command> [command args]]
```

Here we present some useful options only for *sbatch* command:

Option	Description
`-a <indexes>` `--array=<indexes>`	Submit a job array (set of jobs). Each job can be identified by its index number.
`--export=<env variables | ALL | NONE>`	Specify which environment variables will be passed to the job. Default is ALL.
`--ignore-pbs`	Ignore any "#PBS" options in the job script.
`--wrap=<command string>`	Wraps a command in a simple "sh" shell script.
`-d <dependency_list>` `--dependency=<dependency_list>`	Delay the start of the job until the specified dependencies have been satisfied.

These three commands (*sbatch*, *salloc* and *srun*) share many allocation options. The most useful and commonly used allocation options are explained in following table:

Option	Description
`--begin=<time>`	Delay and schedule job after the specified time.
`--cores-per-socket=<cores>`	Allocate nodes with at least the specified number of cores per socket.
`-c <ncpus>` `--cpus-per-task=<ncpus>`	Number of logical CPUs (hardware threads) per task. This option is only relevant for hybrid/OpenMP jobs.
`-D <directory>`	Set the working directory of the job.
`-e <filename pattern>` `--error=<filename pattern>`	Path to the job's standard error. Slurm supports format strings containing replacement symbols such as `%j` (job ID).
`--gres=<list of gres>`	Comma separated list of GRES.
`-H` `--hold`	Job will be submitted in a held state (zero priority). Can be released with "scontrol release <job_id>".
`-i <filename pattern>` `--input=<filename pattern>`	Connect the jobscript's standard input directly to the specified file.
`-J <jobname>` `--job-name=<jobname>`	Set the name of the job.
`--mail-user`	Define the mail address to receive mail notification.
`--mail-type`	Define when to send a mail notifications. Valid options: BEGIN, END, FAIL, REQUEUE or ALL.
`-N <minnodes[-maxnodes]>` `--nodes=<minnodes[-maxnodes]>`	Number of compute nodes used by the job. Can be omitted if --ntasks and --ntasks-per-node is given.
`-n <number>` `--ntasks=<number>`	Number of tasks (MPI processes). Can be omitted if --nodes and --ntasks-per-node is given.
`--ntasks-per-core=<ntasks>`	Number of tasks that will run on each CPU.
`--ntasks-per-node=<ntasks>`	Number of tasks per compute node.
`-o <filename pattern>` `--output=<filename pattern>`	Path to the job's standard output. Slurm supports format strings containing replacement symbols such as %j (job ID).
`-p <partition_names>` `--partition=<partition_names>`	Partition to be used. The argument can be either devel, *batch*, etc on JURECA. If omitted, *batch* is the default.
`--reservation=<name>`	Allocate resources from the specified reservation.
`-t <time>` `--time=<time>`	Maximal wall-clock time of the job.
`--tasks-per-node=<n>`	Same as --ntasks-per-node.

Note: *srun* can also be used to start interactive jobs but we suggest to use *salloc*. *srun* should be used only to start job steps and spawn the processes (like MPI tasks) inside an allocation.

Implied allocation options

Depending on the combination of the allocation options that are used during submission, some other allocation options can be omitted because they are implied and the system calculates them automatically or the default values are used. Following there is table with these combinations:

Used options	Implied options (can be omitted)
`--nodes & --ntasks`	`--ntasks-per-node`
`--nodes & --ntasks-per-node`	`--ntasks`
`--nodes`	`--ntasks` (default is 1 task per node)
`--ntasks`	`--nodes & --ntasks-per-node`

Examples:

Submit a job requesting 2 nodes for 1 hour, with 24 tasks per node (implied value of ntasks: 48):

```
sbatch -N2 --ntasks-per-node=24 --time=1:00:00 jobscript
```

Submit a job-script allocating 4 nodes with 16 tasks in total (implied: 4 tasks per node) for 30 minutes:

```
sbatch -N4 -n16 -t 30 jobscript
```

Submit a job array of 4 jobs with 1 node per job, with the default walltime:

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

Submit a job-script in the batch partition requesting 64 nodes for 2 hours:

```
sbatch -N64 -p batch -t 2:00:00 jobscript
```

Submit a job without a job-script but wrapping a shell command:

```
sbatch -N4 -n4 --wrap="srun hostname"
```

Submit a job requesting the execution to start after the specified date:

```
sbatch --begin=2015-01-11T12:00:00 -N2 --time 2:00:00 jobscript
```

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

```
sbatch -N2 --mail-user=myemail@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
```

Start an interactive job and allocate 4 nodes for 1 hour:

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

Start an interactive job with *srun* and allocate 1 node for 10 minutes in devel partition:

```
srun -N1 -p devel -t 10 --pty -u /bin/bash -i
```

Generic Resources – GRES

As we described in chapter 2.6, generic resources has been configured for each node type. In order to request nodes with specific GRES resources the option "`--gres`" must be used during submissions. The following tables shows the combinations of GRES types that are available for each partition:

Partition	List of GRES
devel *batch* *large*	**MAY:** `--gres=mem128`
mem256	**MAY:** `--gres=mem256`
mem512	**MAY:** `--gres=mem512`
mem1024	**MAY:** `--gres=mem1024`
develgpus *gpus* *largegpus*	**MAY:** `--gres=mem128` **MUST:** `--gres=gpu:[1-4]`
vis *largevis*	**MAY:** `--gres=mem512 OR --gres=mem1024` **MUST:** `--gres=gpu:[1-2]`
maint	-

Examples:

Submit a job requesting 2 nodes in *devel* partition (by default GRES mem128 will be added):

```
sbatch -N2 -p devel jobscript
```

Submit a job requesting 32 nodes in *batch* partition (by default GRES mem128 will be added):

```
sbatch -N32 -p batch jobscript
```

Submit a job requesting 8 nodes in *batch* partition with 256 GB memory:

```
sbatch -N8 -p batch --gres=mem256 jobscript
```

Submit a job requesting 4 nodes in *mem512* partition (will be denied if no --gres=mem512 is given):

```
sbatch -N4 -p mem512 --gres=mem512 jobscript
```

Submit a job requesting 8 nodes and **2 GPUs per node** in *gpus* partition (must give --gres=gpu:X):

```
sbatch -N8 -p gpus –gres=gpu:2 cuda-jobscript
```

Submit a job requesting 32 nodes and **4 GPUs per node** in *gpus* partition:

```
sbatch -N32 -p gpus –gres=gpu:4 jobscript
```

Submit a job requesting the 2 fat visualization nodes with **2 GPUs per node** in *vis* partition:

```
sbatch -N32 -p vis --gres=mem1024,gpu:2 jobscript
```

3.3 Spawning commands

srun

With *srun* the users can spawn any kind of application, process or task inside a job allocation. It can be a shell command, any single-/multi-threaded executable in binary or script format, MPI application or hybrid application with MPI and OpenMP. When no allocation options are defined with *srun* command the options from *sbatch* or *salloc* are inherited.

srun should be used either,

1. Inside a job script submitted by *sbatch*.
2. Or after calling *salloc*.

Note: To start an application with Parastation MPI, the users should use only *srun* and not *mpiexec*. For Intel MPI, *mpirun* is not supported yet but it will be later.

Command format:

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

The allocation options of *srun* for the job-steps are (almost) the same as for sbatch and salloc (please see the table above with allocation options). There are also some useful options only for srun:

Option	Description
`--forward-x`	Enable X11 forwarding only for interactive jobs.
`--multi-prog <filename>`	Run different programs with different arguments for each task specified in a text file.
`--pty`	Execute the first task in pseudo terminal mode.
`-r <num>` `--relative=<num>`	Execute a jobstep inside allocation with relative index of a node.
`--exclusive`	Allocate distinct cores for each task.

Examples:

Spawn 48 tasks on 4 nodes (12 tasks per node) for 30 minutes:

```
srun -N4 -n48 -t 30 executable
```

Spawn 12 tasks on 2 nodes (6 tasks per node), specifying in a file the executables for each task:

```
srun -n12 -N2 --multi-prog ./tasks.conf
---
./tasks.conf:
  0-5  hostname
  6-11  ./executable2
---
```

Inside a job-script, execute 6 tasks on 1 node without sharing cores with other job-steps:

```
srun --exclusive -n6 -N1 mpi-prog
```

3.4 Query Commands

squeue

With *squeue*, we can see the current status information of the queued and running jobs.

Command format:

```
squeue [OPTIONS...]
```

Some of the most useful *squeue* options are:

Option	Description
`-A <account_list>` `--account=<account_list>`	List jobs for the specified accounts.
`-a` `--all`	Show information about jobs and job-steps for all partitions.
`-r` `--array`	Optimized display for job arrays.
`-h` `--noheader`	Do not print the header of the output.
`-i <seconds>` `--iterate=<seconds>`	Repeatedly print information at the specified interval.
`-l` `--long`	Report more information.
`-o <output_format>` `--format=<output_format>`	Specify the information that will be printed (columns). Please read the man pages for more information.
`-p <part_list>` `--partition=<part_list>`	List jobs only from the specified partitions.
`-R <reservation_name>` `--reservation <reservation_name>`	List jobs only for the specified reservation.
`-S <sort_list>` `--sort=<sort_list>`	Specify the order of the listed jobs.
`--start`	Print the expected start time for each job in the queue.
`-t <state_list>` `--states=<state_list>`	List jobs only with the specified state (failed, pending, running, etc).
`-u <user_list>` `--user=<user_list>`	Print the jobs of the specified user.

Examples:

Repeatedly print queue status every 4 seconds:

```
squeue -i 4
```

Show jobs in the devel partition:

```
squeue -p devel
```

Show jobs that belong to a specific user:

```
squeue -u user01
```

Print queue status with a custom format, showing only job ID, partition, user and job state:

```
squeue --format="%.18i %.9P %.8u %.2t"
```

Normally, the jobs will pass through several states during their life-cycle. Typical job states from submission until completion are: PENDING (PD), RUNNING (R), COMPLETING (CG) and COMPLETED (CD). However there are plenty of possible job states for Slurm. The following table describes the most common states:

State Code	State Name	Description
CA	CANCELLED	Job was explicitly cancelled by the user or an administrator. The job may or may not have been initiated.
CD	COMPLETED	Job has terminated all processes on all nodes.
CF	CONFIGURING	Job has been allocated resources, but is waiting for them to become ready for use.
CG	COMPLETING	Job is in the process of completing. Some processes on some nodes may still be active. Usually Slurm is running job's epilogue during this state.
F	FAILED	Job terminated with non-zero exit code or other failure condition.
NF	NODE_FAIL	Job terminated due to failure of one or more allocated nodes.
PD	PENDING	Job is awaiting resource allocation.
R	RUNNING	Job currently has an allocation. **Note**: Slurm is always running the prologue at the beginning of each job before the actual execution of user's application.
TO	TIMEOUT	Job terminated upon reaching its walltime limit.

sview

With *sview*, we get a graphical overview of the cluster. It shows information about system configuration, partitions, nodes, jobs, reservations. Some actions also are possible through the GUI. No options are available for *sview.* Users can just call the command and they will get the graphical window.

sinfo

With *sinfo*, we can get information and check the current state of partitions, nodes and reservations. This command is useful for checking the availability of the nodes.

Command format:

```
sinfo [OPTIONS...]
```

Some of the most useful *sinfo* options are:

Option	Description
-a --all	Show information about all partitions.
-d --dead	Show information only for the non-responding (dead) nodes.
-i <seconds> --iterate=<seconds>	Repeatedly print information at the specified interval.
-l --long	Report more information.
-n <nodes> --nodes=<nodes>	Show information only about the specified nodes.
-N --Node	Show information in a node-oriented format.

`-o <output_format>` `--format=<output_format>`	Specify the information that will be printed (columns). Please read the man pages for more information.
`-p <partition>` `--partition=<partition>`	Show information in a node-oriented format.
`-r` `--responding`	Show information only for the responding nodes.
`-R` `--list-reasons`	List the reasons why nodes are not in a healthy state.
`-s` `--summarize`	List partitions without many details for the nodes.
`-t <states>` `--states=<states>`	List nodes only with the specified state (e.g. allocated, down, drain, idle, maint, etc).
`-T` `--reservation`	Show information about the reservations.

Examples:

Show information about nodes in idle state:

```
sinfo -t idle
```

Show information about partitions and nodes in a summarized way:

```
sinfo -s
```

List all reservations:

```
sinfo -T
```

Show information for partition devel:

```
sinfo -p devel
```

Depending on the options, the *srun* command will print the states of the partitions and the nodes. The partitions may be in state UP, DOWN or INACTIVE. The UP state means that a partition will accept new submissions and the jobs will be scheduled. The DOWN state allows submissions to a partition but the jobs will not be scheduled. The INACTIVE state means that not submissions are allowed.

The nodes also can be in various states. Node state code may be shortened according to the size of the printed field. A node can have also a combination of states, like IDLE+MAINT. The following table shows the most common node states:

Shortened State	State Name	Description
alloc	ALLOCATED	The node has been allocated.
comp	COMPLETING	The job associated with this node is in the state of COMPLETING.
down	DOWN	The node is unavailable for use.
drain	DRAINING & DRAINED	While in DRAINING state any running job on the node will be allowed to run until completion. After that and in DRAIN state the node will be unavailable for use.
idle	IDLE	The node is not allocated to any jobs and is available for use.
maint	MAINT	The node is currently in a reservation with a flag of "maintenance".
resv	RESERVED	The node is in an advanced reservation and not generally available.

smap

With *smap*, we can get a graphical overview of the cluster. It shows information about the nodes and the jobs that are running on them.

Command format:

```
smap [OPTIONS...]
```

Some of the most useful *smap* options are:

Option	Description
`-c` `--commandline`	Send output to the command-line, without using curses.
`-D <option>` `--display=<option>`	Define the display mode of smap. Please read the man pages for more information.
`-h` `--noheader`	Do not print the header of the output.
`-H` `--show_hidden`	Show information about hidden partitions and their jobs.
`-i <seconds>` `--iterate=<seconds>`	Repeatedly print information at the specified interval.
`-n <node_list>` `--nodes <node_list>`	Show information only for the specified nodes.

sprio

With *sprio*, we can check the priorities of all pending jobs in the queue.

Command format:

```
sprio [OPTIONS...]
```

Some of the most useful *sprio* options are:

Option	Description
`-h` `--noheader`	Do not print the header of the output.
`-j <job_id_list>` `--jobs=<job_id_list>`	Show information only about the requested jobs.
`-l` `--long`	Report more information.
`-n` `--norm`	Print the the normalized priority factors of the jobs.
`-o <output_format>` `--format=<output_format>`	Specify the information that will be printed (columns). Please read the man pages for more information.
`-u <user_list>` `--user=<user_list>`	Show information about the jobs of the specified users.
`-w` `--weights`	Print the configured weights for each factor.

Examples:

Show information about priorities of all queued jobs in a long format:

```
sprio -l
```

Show priority information for job 777:

```
sprio -j 777
```

Show the priorities of all jobs that belong to the specified user:

```
sprio -u user1
```

Show priority information in a custom format, printing only job ID, priority and user:

```
sprio -o "%.7i %.10Y  %.8u"
```

scontrol

This command is primarily used by the administrators to manage Slurm's configuration. However it provides also some functionality for the users to manage jobs or query and get some information about the system configuration. Here we present the way to query and get various information with *scontrol*:

Command format:

```
scontrol [OPTIONS...] [COMMAND...]
```

Some of the most useful *scontrol* query commands are:

Command	Description
`show hostlist <host_list>`	Return a compressed regular expression for the given comma separated host list.
`show hostlistsorted <host_list>`	Return a compressed and sorted regular expression for the given comma separated host list.
`show hostnames <host_regex>`	Expand the given regular expression to a full list of hosts.
`show job [<job_id>]`	Show information about all jobs or about the specified job.
`show node [<node_name>]`	Show information about all nodes or about the specified node.
`show partition [<partition_name>]`	Show information about all partitions or about the specified one.
`show reservation [<reservation_name>]`	Show information about all reservations or about the specified one.
`show step [<step_id>]`	Show information about all jobsteps or about the specified one.

Examples:

Expand and print a list of hostnames for the specified range:

```
scontrol show hostname jrc[0106-0115]
```

Show information about the job 777:

```
scontrol show job 777
```

Show information about the node jrc0117:

```
scontrol show node jrc0117
```

Show information about the partition batch:

```
scontrol show partition batch
```

sshare

With *sshare*, we can retrieve fairshare information and check the current value of the fairshare factor that is used to calculate the priorities of the jobs.

Command format:

```
sshare [OPTIONS...]
```

Some of the most useful options of *sshare* are:

Option	Description
`-A <account_list>` `--accounts=<account_list>`	Show information for the specified accounts. By default users belong only to one account.
`-h` `--noheader`	Do not display the header in the beginning of the output.
`-l` `--long`	Show more information.
`-p` `--parsable`	Print information in a parsable way. Delimit output with “|”, with a “|” in the end.
`-P` `--parsable2`	Print information in a parsable way. Delimit output with “|”, without a “|” in the end.

Examples:

Print information about the user's shares in a long format:

```
sshare -l
```

Print information about the user's shares in a parsable way:

```
sshare -P
```

Print information about the user's shares without the initial header in the output:

```
sshare -n
```

3.5 Job Control Commands

scancel

With *scancel*, we can signal or cancel jobs, job arrays or job steps.

Command format:

```
scancel [OPTIONS...] [job_id[_array_id][.step_id]...]
```

Some of the most useful options of the *scancel* command are:

Option	Description
`-A <account>` `--account=<account>`	Restrict the operation only to the jobs under the specified account.
`-b` `--batch`	Send a signal to the batch job shell and its child processes.
`-i` `--interactive`	Enables interactive mode. User must confirm for each operation.
`-n <job_name>` `--name=<job_name>`	Cancel a job with the specified name.
`-p <partition_name>` `--partition=<partition_name>`	Restrict the operation only to the jobs that are running in the specified partition.
`-R <reservation_name>` `--reservation=<reservation_name>`	Restrict the operation only to the jobs that are running using the specified reservation.
`-s <signal_name>` `--signal=<signal_name>`	Send a signal to the specified job(s).
`-t <job_state_name>` `--state=<job_state_name>`	Restrict the operation only to the jobs that have the specified state. Please check the man page.
`-u <user_name>` `--user=<user_name>`	Cancel job(s) only from the specified user. If no job ID is given then cancel all jobs of this user.

Examples:

Cancel jobs with ID 777 and 778:

```
scancel 777 778
```

Cancel jobs with the specified names:

```
scancel -n testjob1 testjob2
```

Cancel all jobs in queue (pending, running, etc.) from user1:

```
scancel -u user1
```

Cancel all jobs in partition devel that belong to user1:

```
scancel -p devel -u user1
```

Cancel all jobs from user1 that are in pending state:

```
scancel -t PENDING -u user1
```

scontrol

The *scontrol* command can be also used to manage and do some actions on the jobs:

Command	Description
`hold <job_list>`	Prevent a pending job from being started.
`release <job_list>`	Release a previously held job, so it can start.
`notify <job_id> <message>`	Send message to the standard error (stderr) of a job.

Examples:

Put jobs 777 and 778 in hold:

```
scontrol hold 777 778
```

Release job 777 from hold:

```
scontrol release 777
```

3.6 Job Utility Commands

sattach

With *sattach*, we can attach to a running job-step and get or manage the IO streams of the tasks in that job-step. By default (without options) it attaches to the standard output/error streams.

Command format:

```
sattach [options] <jobid.stepid>
```

Some of the most useful options of *sattach* are:

Option	Description

`--input-filter[=]<task number>`
`--output-filter[=]<task number>`
`--error-filter[=]<task number>`
	Transfer the standard input or print the standard output/error only from the specified task.

`-l` `--label`	Add the task number in the beginning of each line of standard output/error.
`--layout`	Print the task layout information of the job-step without attaching to its I/O streams.
`--pty`	Run task number zero in pseudo terminal.

Examples:

Attach to the output of job 777 and job-step 1:

```
sattach 777.1
```

Attach to the output of job 777 and job-step 2, adding the task ID in the beginning of each line:

```
sattach -l 777.2
```

sstat

With sstat, we can get various status information about running job-steps, for example minimum, maximum and average values for metrics like CPU time, Virtual Memory (VM) usage, Resident Set Size (RSS), Disk I/O, Tasks number, etc.

Command format:

```
sstat [OPTIONS...]
```

Some of the most useful options of *sstat* are:

Option	Description
`-a` `--allsteps`	Show information about all steps for the specified job.
`-e` `--helpformat`	Show the list of fields that can be specified with the "--format" option.
`-i` `--pidformat`	Show information about the pids for each jobstep.
`-j <job(.step)>` `--jobs <job(.step)>`	Show information for the specified jobs or jobsteps.
`-n` `--noheader`	Do not display the header in the beginning of the output.

`-o <field_list>`
`--format=<field_list>`
`--fields=<field_list>`
	Specify the comma separated list of fields that will be displayed in the output. Available fields can be found with "-e" option or in the man pages.

`-p`
`--parsable`
`-P`
`--parsable2`
	Print information in a parsable way. Output will be delimited with "|".

Examples:

Display default status information for job 777:

```
sstat -j 777
```

Display the defined metrics for job 777 in parsable format:

```
sstat -P --format=JobID,AveCPU,AvePages,AveRSS,AveVMSize -j 777
```

3.7 Job Accounting Commands

sacct

With *sacct*, we can get accounting information and data for the jobs and jobsteps that are stored in Slurm's accounting database. Slurm stores the history of all jobs in the database but each user has permissions to check only his/her own jobs.

Command format:

```
sacct [OPTIONS...]
```

Some of the most useful options of *sacct* are:

Option	Description
`-b` `--brief`	Show a brief listing, with the fields: jobid, status and exitcode.
`-e` `--helpformat`	Show the list of fields that can be specified with the "--format" option.
`-E <end_time>` `--endtime=<end_time>`	List jobs with any state (or with specified states using option "--state") before the given date. Please check the man pages for the available time formats.
`-j <job(.step)>` `--jobs=<job(.step)>`	Show information only for the specified jobs/job-steps.
`-l` `--long`	Show full report with all available fields for each reported job/job-step.
`-n` `--noheader`	Do not display the header in the beginning of the output.
`-N <node_list>` `--nodelist=<node_list>`	Show information only for jobs that ran on the specified nodes.
`--name=<jobname_list>`	Show information about jobs with the specified names.
`-o <field_list>` `--format=<field_list>`	Specify the list of fields that will be displayed in the output. Available fields can be found with "-e" option or in the man pages.
`-r <partition_name>` `--partition=<partition_name>`	Show information only for jobs that ran in the specified partitions. Default is all partitions.
`-s <state_list>` `--state=<state_list>`	Filter and show information only about jobs with the specified states, like completed, cancelled, failed, etc. Please check the man pages for the full list of states.
`-S <start_time>` `--starttime=<start_time>`	List jobs with any state (or with specified states using option "--state") after the given date. The default value is 00:00:00 of current date. Check man page for date formats.
`-X` `--allocations`	Show information only for jobs and not for job-steps.

Examples:

Show job information in long format for default period (starting from 00:00 today until now):

```
sacct -l
```

Show job only information (without jobsteps) starting from the defined date until now:

```
sacct -S 2014-10-01T07:33:00 -X
```

Show job and jobstep information printing only the specified fields:

```
sacct -S 2014-10-01 --format=jobid,elapsed,nnodes,state
```

sacctmgr

The *sacctmgr* command is mainly used by the administrators to view or modify accounting information and data in the accounting database. This command provides also an interface with limited permissions to the users for some querying actions.

Command format:

```
sacctmgr [OPTIONS...] [COMMAND...]
```

Some of the most useful commands for *sacctmgr* are:

Command	Description
`show/list* cluster`	Show cluster information.
`show association [where user=<name>]`	List all visible associations or the ones for the specified user.
`show event [where node=<node_name>]`	List all events for all or for the specified nodes.
`show qos [where name=<qos_name>]`	List all or the specified QoS.
`show user`	Show some user information, like privileges, etc.

***** *"show" and "list" commands are the same for sacctmgr*.

Examples:

Show cluster information:

```
sacctmgr show cluster
```

Show the association of user1:

```
sacctmgr show association where user=user1
```

Print all QoSs:

```
sacctmgr show qos
```

Show the privileges of my user:

```
sacctmgr show user
```

3.8 Custom commands from JSC

llview

llview is a cluster monitoring tool implemented in JSC that shows a graphical overview of the cluster. The nodes are grouped and presented per rack, and different coloring is used per job for each allocation on the nodes. The GUI shows the list of all current jobs in the queue, and gives also information about the utilization of the cluster.

Below in **Figure 6** there is a screenshot of *llview*:

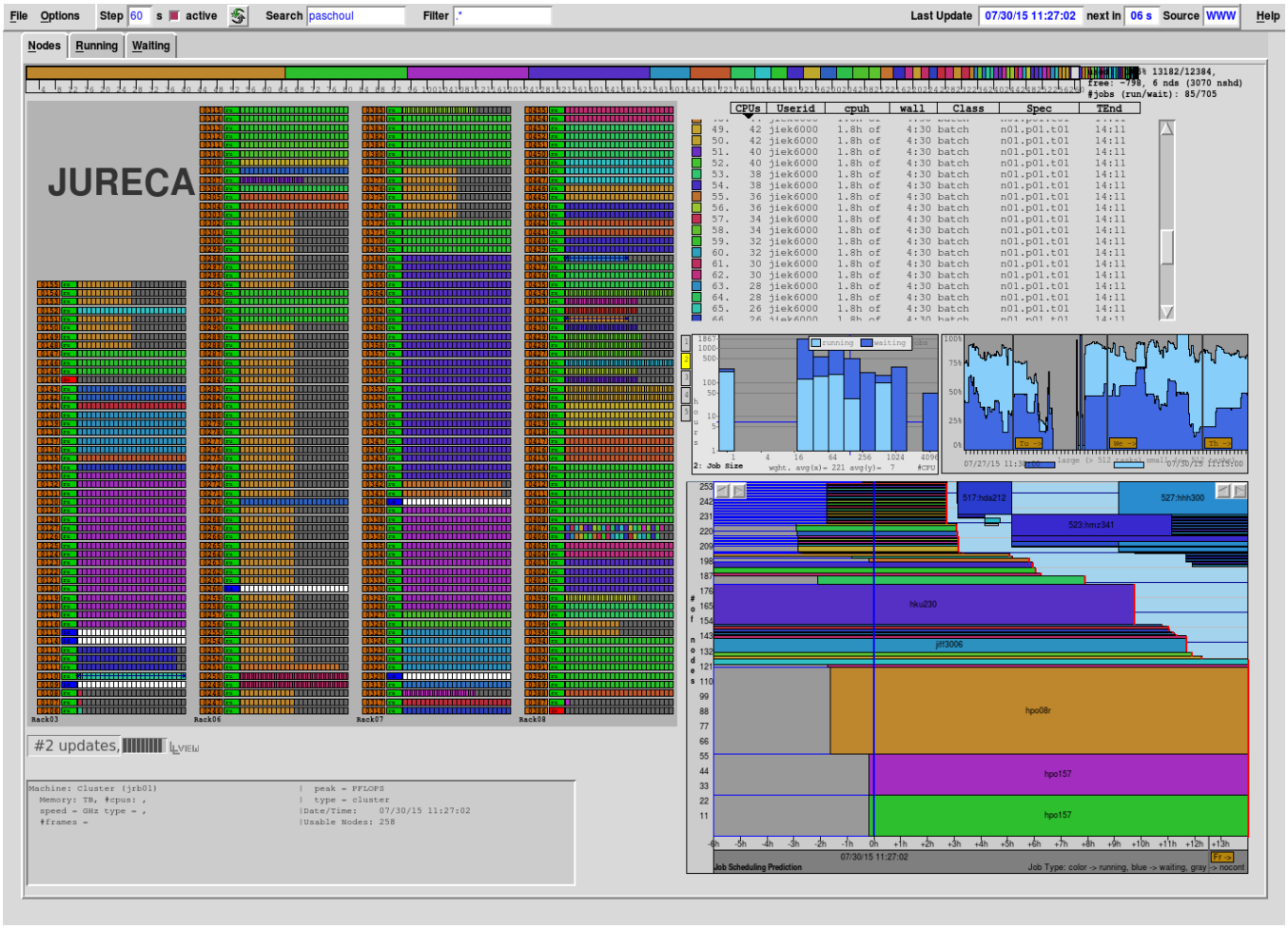

Figure 6.

q_cpuquota

Job accounting is done via a central database in JSC and the information about JURECA jobs will be completed once per day around midnight, based on information obtained from Slurm Accounting Database. Users get information about their current quota-test status or the usage of single jobs by using the command *q_cpuquota*.

Command format:

```
q_cpuquota [OPTIONS...]
```

Some useful options of the *q_cpuquota* commnd are:

Option	Description
`-?`	Print usage information.
`-h <cluster>`	Show information for the specified system (e.g. JURECA).
`-j <jobID>`	Show accounting information for the specified job.
`-t <time>`	Show information about jobs in the specified time period.
`-d <number>`	Show information about jobs of the last specified days.

4 Batch Jobs

Users submit batch applications (usually bash scripts) using the *sbatch* command. In the job scripts, in order to define the *sbatch* parameters #SBATCH directives must be used. The script is executed on the first compute node in the allocation. To execute parallel MPI tasks users call *srun* within their script. With *srun* users can also create job-steps. A job step can allocate the whole or a subset of the already allocated resources from *sbatch*. With these commands Slurm offers a mechanism to allocate resources for a certain walltime and then run many parallel jobs in that frame. The following table describes the most common or necessary allocation options that can be defined in a job script:

Option	Default value	Description
`#SBATCH --nodes=<number>` `#SBATCH -N <number>`	1	Number of nodes for the allocation.
`#SBATCH --ntasks=<number>` `#SBATCH -n <number>`	1	Number of tasks (MPI processes). Can be omitted if --nodes and –ntasks-per-node are given.
`#SBATCH --ntasks-per-node=<num>` `#SBATCH --tasks-per-node=<num>`	1	Number of tasks per node. If keyword omitted the default value is used, but there are still available maximum 56 CPUs per node for current allocation.
`#SBATCH --cpus-per-task=<num>` `#SBATCH -c <num>`	1	Number of threads/VCores per task. Used only for OpenMP or hybrid jobs.
`#SBATCH --output=<path>` `#SBATCH -o <path>`	slurm-<jobID>.out	Path to the file for the standard output.
`#SBATCH --error=<path>` `#SBATCH -e <path>`	slurm-<jobID>.out	Path to the file for the standard error.
`#SBATCH --time=<walltime>` `#SBATCH -t<walltime>`	Depends on the partition	Requested walltime limit for the job.
`#SBATCH --partition=<name>` `#SBATCH -p <name>`	batch	Partition to run the job. Currently available: batch and devel partitions.
`#SBATCH --mail-user=<email>`	username	Email address for notifications.
`#SBATCH --mail-type=<mode>`	NONE	Event types for email notifications.
`#SBATCH --job-name=<jobname>` `#SBATCH -J <jobname>`	jobscript's name	Job name.
`#SBATCH --gres=<list>`	mem128	Generic resources.
`#SBATCH -C, --constraint=<list>`	NONE	Request nodes with specific Features.

Multiple *srun* calls can be placed in a single batch script. Options such as --nodes, --ntasks and --ntasks-per-node are by default taken from the *sbatch* arguments but can be overwritten for each srun invocation. If --nasks-per-node is omitted or set to a value higher than 24 then SMT (simultaneous multi-threading) will be enabled. Each compute node has 24 physical cores and 48 logical cores.

As we described before, the job script is submitted using:

```
sbatch [OPTIONS] <jobscript>
```

On success, *sbatch* writes the job ID to standard out.

Note: In case some allocation options are defined in both command-line and inside the job-script, then the options that were given as arguments in the command-line will be used and the options in the job-script will be ignored.

4.1 Job script examples

Serial job

Example 1: 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
```

Parallel job

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. Options given to *srun* will override the allocation option from *sbatch*. In case of no *srun* options the defined options (with #SBATCH) or the defaults will be used.

Example 2: 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 in a time-frame of 10 minutes.

```
#!/bin/bash
#SBATCH -J TestJob
#SBATCH -N 4
#SBATCH -o TestJob-%j.out
#SBATCH -e TestJob-%j.err
#SBATCH --time=60

srun -N2 --ntasks-per-node=1 -t 10 hostname
```

OpenMP job

Example 3: In this example the job will execute an OMP 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=48

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

MPI job

Example 4: In the following example, an MPI application will start 96 tasks on 4 nodes running 24 tasks per node (no SMT) requesting a walltime 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=96
#SBATCH --output=mpi-out.%j
#SBATCH --error=mpi-err.%j
#SBATCH --time=00:15:00
#SBATCH --partition=batch

srun -N4 --ntasks-per-node=24 ./mpi-prog
```

MPI jobs with SMT

On each node there are 28 real cores available and, with SMT enabled, 48 virtual cores. In order to enable SMT the users just have to request from Slurm to allocate more than 24 CPUs on each compute node. Following there are some examples where SMT is enabled:

Example 5: In this example we have an MPI application starting 1536 tasks in total on 32 nodes using 48 logical CPUs (hardware threads) per node (SMT enabled) requesting a time period of 20 minutes. The *batch* partition is used.

```
#!/bin/bash -x

#SBATCH --nodes=32
#SBATCH --ntasks=1536
#SBATCH --ntasks-per-node=48 # can be omitted #
#SBATCH --output=mpi-out.%j
#SBATCH --error=mpi-err.%j
#SBATCH --time=00:20:00
#SBATCH --partition=batch

srun ./mpi-prog
```

Example 6: In this example, the job script will start the program "`mpi-prog`" on 4 nodes using 48 MPI tasks per node, where two MPI tasks will be executed on each physical core.

```
#!/bin/bash

#SBATCH --nodes=4
#SBATCH --ntasks=192 # can be omitted #
#SBATCH --ntasks-per-node=48
#SBATCH --output=mpi-out.%j
#SBATCH --error=mpi-err.%j
#SBATCH --time=00:15:00
#SBATCH --partition=batch

srun ./mpi-prog
```

Hybrid Jobs

Example 7: 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 that was given to the option "`--cpus-per-task`".

```
#!/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=batch

export OMP_NUM_THREADS=4

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

Example 8: In this example, there is a hybrid application which will start 2 tasks per node on 4 allocated nodes and starting 12 threads per node (no SMT). In order to set the environment variable "`OMP_NUM_THREADS`", Slurm's variable "`SLURM_CPUS_PER_TASK`" is used which is defined by the option "`--cpus-per-task`".

```
#!/bin/bash
#SBATCH -N 4
#SBATCH -n 8  # can be omitted #
#SBATCH --ntasks-per-node=2
#SBATCH --cpus-per-task=6
#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
```

Hybrid jobs with SMT

Example 9: This example shows a hybrid application that will start 4 tasks per node on 3 allocated nodes and starting 12 threads per task, using in total 48 cores per node (SMT enabled).

```
#!/bin/bash
#SBATCH --nodes=3
#SBATCH --ntasks=12
#SBATCH --ntasks-per-node=4
#SBATCH --cpus-per-task=12
#SBATCH --output=mpi-out.%j
#SBATCH --error=mpi-err.%j
#SBATCH --time=00:20:00

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

Example 10: 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=6
#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

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

Intel MPI jobs

In order to run Intel MPI jobs user can use *srun*. The *mpirun* command is currently not supported. That means for now the users can not export and use the environment variables from Intel MPI, because *srun* does not work with them. Users will be informed when *mpirun* will be supported.

4.2 Job steps

In a previous chapter we described job-steps as small allocations or jobs inside the current job. Each call of *srun* will create a new job-step. It is up to the users to decide how they will create job-steps. It is possible to have one job-step after another using all the allocated nodes each time, or to have many job-steps running in parallel. Instead of submitting many single-node jobs, known as farming, it is suggested to the users to do farming using job-steps inside a single job. In this case, since all CPUs are available to the job, the only bounding factor is the memory per task (and the walltime). The users will be accounted for all the nodes of the allocation regardless if all nodes are used for job-steps or not.

Example 11: In the following example it is presented how to execute MPI programs in different job-steps sequentially inside a job allocation. In total 4 nodes are allocated for 2 hours. In this job 3 job-steps will be created. The first job-step will run on 4 nodes having 1 MPI task per node for 20 minutes. After that the second job-step will be executed on 3 nodes with 24 MPI tasks per node for 1 hour. And in the end the last job-step will run on 4 nodes with 48 MPI tasks per node using all virtual cores on each node (SMT) and it will finish when the MPI application will be completed or will be canceled by the scheduler if it will reach the walltime limit.

```
#!/bin/bash

#SBATCH --nodes=4
#SBATCH --output=mpi-out.%j
#SBATCH --error=mpi-err.%j
#SBATCH --time=02:00:00

srun -N4 --ntasks-per-node=1  --time=00:20:00 ./mpi-prog1
srun -N3 --ntasks-per-node=24 --time=01:00:00 ./mpi-prog2
srun -N4 --ntasks-per-node=48 ./mpi-prog3
```

Example 12: In the following example we show a job script where two different job-steps are initiated within one job. In total 24 cores are allocated on two nodes. Each job step uses 24 cores on each compute node. With the option "`--exclusive`" we ensure that distinct CPUs (Virtual Cores) are allocated for each job-step. Here the job-steps will be executed in parallel (in order to put the processes on the background "&" is needed in the end of each command line and then the "`wait`" shell command after the *srun* commands will ensure that the job will wait until all job-steps are completed):

```
#!/bin/bash

#SBATCH --nodes=2
#SBATCH --output=mpi-out.%j
```

```
#SBATCH --error=mpi-err.%j
#SBATCH --time=00:20:00

srun -N1 --exclusive -n 24 ./mpi-prog1 &
srun -N1 --exclusive -n 24 ./mpi-prog2 &

wait
```

4.3 Dependency Chains

Slurm supports dependency chains which are collections of batch jobs with defined dependencies, similar to job chains of Moab on JUROPA. Job dependencies can be defined using the --dependency argument of *sbatch*. The format is:

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

The available dependency types for job-chains are: *after*, *afterany*, *afternotok* and *afterok*. For more information please check the man page of *sbatch*.

Example 13: Below is an example of a job-script for the handling of job chains. The script submits a chain of "`$NO_OF_JOBS`". A job will only start after successful completion of its predecessor. Please note that a job which exceeds its time-limit is not marked successful.

```
#!/bin/bash -x
# submit a chain of jobs with dependency

# number of jobs to submit
NO_OF_JOBS=<no of jobs>

# define jobscript
JOB_SCRIPT=<jobscript>

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

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

4.4 Job Arrays

Slurm supports job-arrays and offers a mechanism to easily manage these collections of jobs. Job arrays are only supported for the *sbatch* command and, as we described previously, they can be defined using the options "--array" or "-a". To address a job-array, Slurm provides a base array ID and an array index unique for each job. The format for specifying an array job is first the base array jobID followed by "_" and then the array index:

```
 <base job id>_<array index>
```

Slurm exports two environment variables that can be used in the job script to identify each array-job:

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

Some additional options are available to specify the *stdin*, *stdout*, and *stderr* file names: option "`%A`" will be replaced by the value of `SLURM_ARRAY_JOB_ID` and option "`%a`" will be replaced by the value of `SLURM_ARRAY_TASK_ID`.

Also each job in an array has its own normal unique job ID. This ID is exported in the environment variable

```
SLURM_JOBID
```

Example 14: In the following example, the job-script will create a job array of 4 jobs with indices 0-3. Each job will run on 1 node for 1 hour and will execute different script (`script_[0-3].sh`).

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --output=prog-%A_%a.out
#SBATCH --error=prog-%A_%a.err
#SBATCH --time=01:00:00
#SBATCH --array=0-3

./script_${SLURM_ARRAY_TASK_ID}.sh
```

Example 15: In the following job-script a job array of 20 jobs will be submitted with indices 1-20. Each job will run on a separate node with 2 hours walltime limit. Some may be running and some may be waiting in the queue. For this job array all jobs will execute the same binary "`prog`" with different input files (`input_[1-20].txt`):

```
#!/bin/bash -x
#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 --partition=batch

srun -N1 --ntasks-per-node=1  ./prog input_${SLURM_ARRAY_TASK_ID}.txt
```

4.5 MPMD

Slurm supports the MPMD model (Multiple Program Multiple Data Execution Model) that can be used for MPI applications, where multiple executables can have one common `MPI_COMM_WORLD` communicator. For this purpose Slurm provides the option "`--multi-prog`" for the *srun* command only. This option expects a configuration text file as an argument and the format is:

```
srun [OPTIONS..] --multi-prog <text-file>
```

Each line of the configuration file can have two or three possible fields separated by space and the format is like this:

```
<list of task ranks> <executable> [<possible arguments>]
```

In the first field is defined a comma separated list of ranks for the MPI tasks that will be spawned. Possible values are integer numbers or ranges of numbers. The second field is the path/name of the executable. And the third field is optional and defines the arguments of the program.

Example 16: In this example there is a simple configuration file with name "*multi.conf*". This file defines three MPI programs. For the first executable `mpi-prog1` only one instance will be executed with rank 0 and one integer argument. For the second program `mpi-prog2` Slurm will create two tasks with ranks 4 and 6 and each one will have the path of a file as argument. For the third program `mpi-prog3` five MPI tasks will be executed with ranks 1, 2, 3, 5 and 7 without any arguments.

```
0       ./mpi-prog1 0
4,6     ./mpi-prog2 ./tmp.txt
1-3,5,7 ./mpi-prog3
```

Following is the job-script that will start this MPMD job. The job-script allocates 4 nodes for 1 hour. The command *srun* will start this MPMD application, where all 4 nodes will be used with 2 MPI tasks per node (8 tasks in total). It can be submitted with *sbatch*:

```
#!/bin/bash
#SBATCH --nodes=4
#SBATCH --time=01:00:00

srun -N4 --ntasks-per-node=2 --multi-prog ./multi.conf
```

The "`--multi-prog`" option can be used of course for any kind of binary and its usage is not restricted to MPI jobs only, but it is the only way to apply the MPMD model.

4.6 GPU Computing

JURECA features a number of accelerated compute nodes equipped with GPUs in the develgpus and gpus partitions. In order to access them the partition argument -p develgpus (or --partition develgpus) or -p gpus (or --partition gpus), respectively, must be provided to sbatch or salloc. The GPU compute nodes feature two NVIDIA Tesla K80 GPUs with a dual-GPU design. Each Tesla K80 is visible as two separate GPUs on the host for a total of four available GPUs per compute node. In addition to the partition parameter, the number of requested GPUs must be specified using the --gres=gpu:X argument with X in the range one to four. The applications can utilize those GPUs by using supported libraries like CUDA, OpenCL or OpenACC.

Example 17: In this example there is an MPI application which starts 96 tasks on 4 nodes using 24 CPUs per node and 4 GPUs per node. The program must be able to coordinate the access to the four GPU devices on each of the four nodes.

```
#!/bin/bash -x
#SBATCH --nodes=4
#SBATCH --ntasks=96
#SBATCH --ntasks-per-node=24
#SBATCH --output=gpu-out.%j
#SBATCH --error=gpu-err.%j
#SBATCH --time=00:15:00
#SBATCH --partition=gpus

#SBATCH --gres=gpu:4

srun ./gpu-prog
```

Example 18: In this example there four independent instances (job steps) of a GPU program running on one node using one CPU thread and one GPU device each. The program is pinned to CPU core 0, 6, 12 and 18, respectively.

```
#!/bin/bash -x
#SBATCH --nodes=1
#SBATCH --output=gpu-out.%j
#SBATCH --error=gpu-err.%j
#SBATCH --time=00:20:00
#SBATCH --partition=gpus
#SBATCH --gres=gpu:4

srun --exclusive -n 1 --gres=gpu:1 --cpu_bind=map_cpu:0 ./gpu-prog &
srun --exclusive -n 1 --gres=gpu:1 --cpu_bind=map_cpu:6 ./gpu-prog &
srun --exclusive -n 1 --gres=gpu:1 --cpu_bind=map_cpu:12 ./gpu-prog &
srun --exclusive -n 1 --gres=gpu:1 --cpu_bind=map_cpu:18 ./gpu-prog &

wait
```

4.7 Booster/KNL nodes with different configuration

The booster and develbooster partitions can be used normally e.g. as the batch partition, users just have to define the partition name during submission. Both booster and develbooster partitions are configured in Quadrant NUMA mode with Hybrid50 Memory mode, which means 2 NUMA domains are exposed and the total available RAM is increased by 8GB (~104GB total). A little bit more complex is the modetestbooster partition where we have mixed KNL configurations. The purpose of this partition is to test different KNL configurations (their configuration may change in the future). It is divided into 3 different groups of 32 nodes each. Currently the KNL configurations are: a) SNC4 + Flat, b) SNC4 + Cache and c) Quadrant + Cache.

In Slurm those groups have been configured to have different "Features", and the current list of Features is: a) snc4flat, b) snc4cache and c) quadcache. In order to use certain configuration users must apply also the constraint submission option:

```
sbatch/salloc -C, --constraint=<feature>
```

For example to allocate KNLs with snc4flat:

```
sbatch <options> -p modetestbooster -C snc4flat jobscript
```

Users can check the configured Features of Booster partitions with command:

```
sinfo --partition=modetestbooster,booster,develbooster -o "%P %f %D %N"
```

Also users can check the idle nodes in modetestbooster partition grouped according to their config:

```
sinfo --partition=modetestbooster -o "%P %t %f %D %N" --state=idle
```

One important thing to remember is that submissions in modetestbooster partition are **denied** when users do not specify any Feature.

5 Interactive Jobs

5.1 Interactive Session

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

```
salloc --nodes=2 --time=00:30:00
```

Once an allocation has been made, the *salloc* command will start a bash on the login node where the submission was done. After a successful allocation the users can execute *srun* from that shell and they can spawn interactively their applications. For example:

```
srun --ntasks=4 --ntasks-per-node=2 --cpus-per-task=7 ./hybrid-prog
```

The interactive session is terminated by exiting the shell. In order to obtain a shell on the first allocated compute nodes (like command "`msub -I`" from Moab), the users can start a remote shell from within the current session and connect it to a pseudo terminal (pty) using the *srun* command with a shell as an argument. For example:

```
srun --cpu_bind=none --nodes=2 --pty /bin/bash
```

After gaining access to the remote shell it is possible to run *srun* again from that remote shell in order to execute interactively applications without any delays (no scheduling delays since the allocation has already been granted). Below follows a transcript of an exemplary interactive session:

```
$ salloc --nodes=2 --time=00:01:00
salloc: Pending job allocation 4749
salloc: job 4749 queued and waiting for resources
salloc: job 4749 has been allocated resources
salloc: Granted job allocation 4749

$ hostname
jrl03

$ srun --ntasks 2 --ntasks-per-node=2 hostname
jrc0161
jrc0162

$ srun --cpu_bind=none --nodes=1 --ntasks=1 --pty /bin/bash -i

$ hostname
jrc0161

$ logout

$ hostname
jrl03

$ exit
exit
salloc: Relinquishing job allocation 4749
salloc: Job allocation 4749 has been revoked.
```

Note: When the users want to start a remote shell on the compute nodes, they should always give the option "`--cpu_bind=none`" to the *srun* command in order to disable the default pinning. If this option is

not given then the default CPU binding settings will pin the processes in an unexpected way, e.g. sometimes restricting the processes on one core only. Here is an example how it should be used:

```
$ srun --cpu_bind=none --nodes=1 --ntasks=1 --pty /bin/bash -i
```

5.2 X Forwarding

The X11 forwarding support has been implemented with the "`--forward-x`" option of the *srun* command. It is similar to the option "`msub -X`" from Moab. X11 forwarding is required for users who want to use applications or tools which provide a GUI.

Here is an example that shows how to use this feature:

```
$ salloc --nodes=1 --time=00:01:00
...

$ srun --cpu_bind=none --nodes=1 --ntasks=1 --forward-x --pty /bin/bash -i

$ ./GUI-App
```

Note: User accounts will be charged per allocation whether the compute nodes are used or not. Batch submission is the preferred way to execute jobs.

6 From Moab/Torque to Slurm

On JUROPA we were using the combination of Moab and Torque for the Batch System. Moab works as the scheduler and Torque is the resource manager. However, on JURECA we use Slurm as scheduler and resource manager. In this chapter we will compare and give some information about these two solutions and we will try to help the users have an easier migration from Moab/Torque to Slurm.

6.1 Differences between the Systems

Here we will compare and declare some differences between Moab and Slurm:

	Moab	Slurm
Resource Management	Not supported. Needs an external Resource Manager (like Torque).	A flexible and capable resource manager (in our case psslurm on the nodes).
Nodes	It is possible to set nodes for batch and interactive jobs only, or both.	No difference between batch and interactive jobs for Slurm.
Queues	Partitions separate node into groups. Queues are used for job submission on one partition only.	Slurm defines only partitions. For Slurm the partitions are used as queues. Partitions can overlap and we can specify limits.
Priorities	Complex priorities mechanism.	Easy to configure, maintain and manage. The desired batch model from JSC can be easily applied.
Limits/Policy	Good support for limits and policies configuration.	Highly configurable: define limits and policies per partition/account/user. Enforce limits with QoS.
Job scripts	Define job-script options with #MSUB.	Define job-script options with #SBATCH.

In the following table you can see some of the differences between Torque and Slurm:

	Torque	Slurm
Scheduling	Integrates only a simple FIFO scheduler, needs external scheduler.	Slurm is a capable scheduler with support for backfilling algorithm.
Output files	With the default options, stores output locally on the nodes. Upon completion files are gathered at destination.	Standard output and error files are created in the final destination immediately.
Working directory	Must explicitly change to current working directory.	Jobs start to run in the directory where they were submitted from.
Job Steps	Not supported by Torque.	Flexible allocations within jobs.
Task Distribution	Possible to specify different number of tasks per set of nodes, e.g.: “-l nodes=1:ppn=2+nodes=4:ppn=8”	Possible to specify only the same number of tasks on all nodes with the allocation options.
Environment	If users want to export the whole shell environment, they must use the option “-V”.	The environment defined in user's shell during submission will be automatically exported to the job.

6.2 User Commands Comparison

The following table presents commands with similar functionality from Slurm, Moab and Torque:

User Commands	Slurm	Moab	Torque
Job Submission	`sbatch`	`msub`	`qsub`
Job deletion	`scancel`	`canceljob`	`qdel`
Job status	`squeue` `scontrol show job`	`checkjob`	`qstat`
Job hold	`scontrol hold`	`mjobctl -h`	`qhold`
Job release	`scontrol release`	`mjobctl -u`	`qrls`
Queue list	`squeue`	`showq`	`qstat -Q`
Cluster status	`sinfo`	`---`	`qstat -a`
Node list	`scontrol show nodes`	`---`	`pbsnodes -l`
GUI	`sview`	`---`	`xpbsmon`

The table below compares the allocation options of *msub* and *sbatch*:

Allocation option	Moab/Torque (msub)	Slurm (sbatch)
Number of nodes	`-l nodes=<number>`	`--nodes=<number>` `-N <number>`
Number of total tasks	None	`--ntasks=<number>` `-n <number>`
Number of tasks/cpus per node	`-l ppn=<number>`	`--ntasks-per-node=<num>` `--tasks-per-node=<num>`
Number of threads per task	`-v tpt=<number>`	`--cpus-per-task=<num>` `-c <num>`
File for the standard output	`-o <path>`	`--output=<path>` `-o <path>`
File for the standard error	`-e <path>`	`--error=<path>` `-e <path>`
Walltime limit	`-l walltime=<time>`	`--time=<walltime>` `-t <walltime>`
Partition/Queue selection	`-q <queue>`	`--partition=<queue>` `-p <queue>`
Email for notifications	`-M <email>`	`--mail-user=<email>`
Event types for notifications	`-m <mode>`	`--mail-type=<mode>`
Job name	`-N <jobname>`	`--job-name=<jobname>` `-J <jobname>`
Interactive jobs	`-I`	None (use salloc or srun)
Job dependencies	`-W depend=<mode>:<jobID>`	`--dependency=<dependency_list>` `-d <dependency_list>`

7 Examples

7.1 Template job-scripts

Template MPI job-script:

```
#!/bin/bash
#SBATCH -J <jobname>
#SBATCH -N <number>
#SBATCH -n <number> # can be omitted
#SBATCH --ntasks-per-node=<number>
#SBATCH -o <jobname>-%j.out
#SBATCH -e <jobname>-%j.err
#SBATCH --mail-type=<BEGIN, END, FAIL, or ALL>
#SBATCH --mail-user=<email>
#SBATCH --partition=<batch | devel>
#SBATCH --time=<time>

# run MPI application below (with srun)
```

Template Hybrid job-script:

```
#!/bin/bash
#SBATCH -J <jobname>
#SBATCH -N <number>
#SBATCH -n <number> # can be omitted
#SBATCH --ntasks-per-node=<number>
#SBATCH --cpus-per-task=<number>
#SBATCH -o <jobname>-%j.out
#SBATCH -e <jobname>-%j.err
#SBATCH --mail-type=<BEGIN, END, FAIL, or ALL>
#SBATCH --mail-user=<email>
#SBATCH --partition=<batch | devel>
#SBATCH --time=<time>

export OMP_NUM_THREADS=${SLURM_CPUS_PER_TASK}

# run Hybrid application below (with srun)
```

7.2 Modules

Check loaded modules:

```
$ module list
No modules loaded
```

Check available Toolchains:

```
$ module avail
…
```

Load a compiler and an MPI runtime and check loaded modules:

```
$ module load Intel ParaStationMPI
$ module list
```

Check available packages:

```
$ module avail
…
```

Load a module:

```
$ module load Boost/1.61.0-Python-2.7.12
$ module list
…
```

Purge all modules:

```
$ module purge
$ module list
No modules loaded
```

Check a package:

```
$ module spider Boost
…
```

Check a specific version of a package:

```
$ module spider Boost/1.61.0-Python-2.7.12
…
```

7.3 Compilation

MPI program example (file *mpi.c*):

```
#include <stdio.h>
#include <mpi.h>

int main ( int argc, char** argv )
{

        int rank, size;
        char processor_name [MPI_MAX_PROCESSOR_NAME];
        int name_len;

        // Initialize the MPI environment.
        MPI_Init( &argc, &argv );

        // Get the number of processes.
        MPI_Comm_size ( MPI_COMM_WORLD, &size);

        // Get the rank of the process.
        MPI_Comm_rank ( MPI_COMM_WORLD, &rank );

        // Get the name of the processor.
        MPI_Get_processor_name ( processor_name, &name_len );

        // Print out.
          printf( "Hello world from processor %s, rank %d out of %d processors.\n", processor_name,
rank, size);

        // Finalize the MPI environment.
        MPI_Finalize();

        return 10;
}
```

Hybrib program example (file *hybrid.c*):

```
#include <stdio.h>
```

```
#include <mpi.h>
#include "mpi.h"

#define _NUM_THREADS 16

int main ( int argc, char** argv )
{

        int rank, size, count, total;
        char processor_name [MPI_MAX_PROCESSOR_NAME];
        int name_len;

//      omp_set_num_threads(_NUM_THREADS);

        // Initialize the MPI environment.
        MPI_Init( &argc, &argv );

        // Get the number of processes.
        MPI_Comm_size ( MPI_COMM_WORLD, &size);

        // Get the rank of the process.
        MPI_Comm_rank ( MPI_COMM_WORLD, &rank );

        // Get the name of the processor.
        MPI_Get_processor_name ( processor_name, &name_len );

        count = 0;

        #pragma omp parallel reduction(+:count)
        {
                count = count + omp_get_num_threads();
                total = omp_get_num_threads();

        }

        // Print out.
         printf( "Hello world from processor %s, rank %d out of %d processors. OpenMP threads: %d\n",
processor_name, rank, size, total);

        // Finalize the MPI environment.
        MPI_Finalize();

        return 0;
}
```

Compile the MPI program:

```
$ mpicc -o mpi-prog mpi.c
```

Compile the Hybrid program:

```
$ mpicc -openmp -o hybrid-prog hybrid.c
```

7.4 Job submission

Job-script for an MPI job (file *mpiscript.sh*):

```
#!/bin/bash
#SBATCH -J mpitest
#SBATCH -N 4
```

```
#SBATCH --ntasks-per-node=24
#SBATCH -o mpitest-%j.out
#SBATCH -e mpitest-%j.err
#SBATCH --partition=batch
#SBATCH --time=00:30:00

# run MPI application below (with srun)
srun -N 4 --ntasks-per-node=24 ./mpi-prog
```

Submit the MPI job-script:

```
$ sbatch ./mpiscript.sh
```

Job-script for a Hybrid job (file *hybridtest.sh*):

```
#!/bin/bash
#SBATCH -J hybridtest
#SBATCH -N 4
#SBATCH --ntasks-per-node=24
#SBATCH --cpus-per-task=2
#SBATCH -o hybridtest-%j.out
#SBATCH -e hybridtest-%j.err
#SBATCH --mail-type=END
#SBATCH --mail-user=c.paschoulas@fz-juelich.de
#SBATCH --time=00:30:00

export OMP_NUM_THREADS=${SLURM_CPUS_PER_TASK}

# run Hybrid application below (with srun)
srun -N 4 --ntasks-per-node=24 -c ${SLURM_CPUS_PER_TASK} ./hybrid-prog
```

Submit the Hybrid job-script

```
$ sbatch ./hybridscript.sh
```

7.5 Job Control

Hold a job:

```
$ scontrol hold 14900
$ squeue
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
...
             14900     batch hybridte paschoul PD       0:00      4 (JobHeldUser)
...
```

Release a job:

```
$ scontrol release 14900
$ squeue
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
...
             14900     batch hybridte paschoul  R       0:01      4 jrc[120-123]
...
```

Cancel a job:

```
$ scancel 14905
```

7.6 Query Commands

Check the Queue

```
$ squeue
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
             44210     batch Simulati jics4002 PD       0:00      1 (Dependency)
             44211     batch Simulati jics4002 PD       0:00      1 (Dependency)
```

```
             44213     batch Simulati jics4002 PD       0:00      1 (Dependency)
             44214     batch Simulati jics4002 PD       0:00      1 (Dependency)
             44215     batch Simulati jics4002 PD       0:00      1 (Dependency)
             44216     batch Simulati jics4002 PD       0:00      1 (Dependency)
             44217     batch Simulati jics4002 PD       0:00      1 (Dependency)
             44241     batch equil2_1      cao  R      29:17      8 jrc[0387-0388,0406-0407,0450-0453]
             44283     batch expl-g01 paj15340  R    1:59:57      3 jrc[0251,0341-0342]
             43141     batch scr.N50M esmi2000  R   19:46:04      4 jrc[0369-0372]
             43140     batch scr.N50M esmi2000  R   19:52:01      4 jrc[0439-0440,0443-0444]
             43856     batch Vito-ANA   hgr221  R   19:32:11      2 jrc[0454-0455]
             43847     batch   F3T_CR   hku230  R   16:02:55     30 jrc[0343-0368,0400-0403]
             44342     batch run-scri jias5002  R      51:07      3 jrc[0415-0417]
             44230     batch submit3a jiek6000  R      51:07      1 jrc0305
             44231     batch submit3b jiek6000  R      51:07      1 jrc0418
             44238     batch submit5a jiek6000  R      51:07      1 jrc0441
             44239     batch submit5b jiek6000  R      51:07      1 jrc0442
             44242     batch submit6a jiek6000  R      49:08      1 jrc0304
             40618     batch  bridge1 jiff3006  R   14:06:56     10 jrc[0136-0140,0321-0325]
             43800     batch      job   hgr240  R   14:45:36      2 jrc[0134,0316]
             43799     batch      job   hgr240  R   14:47:35      2 jrc[0141,0317]
             43796     batch      job   hgr240  R   15:01:28      1 jrc0319
             41497     batch   job.sh jics6402  R   14:06:56      2 jrc[0249-0250]
             43932     batch Simulati   hgr283  R   11:26:17      4 jrc[0152,0447-0449]
```

Check the Queue for one user:

```
$ squeue -u paschoul
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
             14910     batch  mpitest paschoul PD       0:00      4 (QOSResourceLimit)
             14911     batch  mpitest paschoul PD       0:00      4 (QOSResourceLimit)
             14912     batch hybridte paschoul  R       0:02      4 jrc[0120-0123]
             14908     batch  mpitest paschoul  R       0:02      4 jrc[095-098]
```

Check partitions and nodes:

```
$ sinfo
PARTITION AVAIL  TIMELIMIT  NODES  STATE NODELIST
batch*       up    2:00:00      5  drain j3c[090,115-116,119,124]
batch*       up    2:00:00     65   idle j3c[061-089,091-114,117-118,120-123,125-130]
large      down    1:00:00      5  drain j3c[090,115-116,119,124]
large      down    1:00:00     65   idle j3c[061-089,091-114,117-118,120-123,125-130]
```

Check off-line nodes:

```
$ sinfo -R
REASON               USER      TIMESTAMP           NODELIST
#401 - NodeHardware  root      2015-07-29T21:18:54 jrc0320
#403 - NodeHardware  root      2015-07-30T07:58:53 jrc0340
#402 - NodeHardware  root      2015-07-30T04:12:14 jrc0144
Golden client        root      2015-07-20T16:59:45 jrc0386
#401 - NodeHardware  root      2015-07-29T21:18:54 jrc0320
#403 - NodeHardware  root      2015-07-30T07:58:53 jrc0340
#402 - NodeHardware  root      2015-07-30T04:12:14 jrc0144
Golden client        root      2015-07-20T16:59:45 jrc0386
#401 - NodeHardware  root      2015-07-29T21:18:54 jrc0320
```

Check reservations:

```
$ sinfo -T
RESV_NAME     STATE           START_TIME             END_TIME     DURATION  NODELIST
test         ACTIVE  2014-11-14T15:24:47  2015-10-01T00:00:00  320-07:35:13  jrc0128
```

Check one partition:

```
$ scontrol show partition batch
PartitionName=batch
```

```
   AllowGroups=ALL AllowAccounts=ALL AllowQos=ALL
   AllocNodes=ALL Default=YES
   DefaultTime=01:00:00 DisableRootJobs=NO GraceTime=0 Hidden=NO
   MaxNodes=64 MaxTime=1-00:00:00 MinNodes=1 LLN=NO MaxCPUsPerNode=48
   Nodes=jrc[0116-0155,0246-0455]
   Priority=1 RootOnly=NO ReqResv=NO Shared=NO PreemptMode=OFF
   State=UP TotalCPUs=12000 TotalNodes=250 SelectTypeParameters=N/A
   DefMemPerNode=UNLIMITED MaxMemPerNode=UNLIMITED
```

Check one node:

```
$ scontrol show node jrc0130
NodeName=jrc0130 Arch=x86_64 CoresPerSocket=12
   CPUAlloc=48 CPUErr=0 CPUTot=48 CPULoad=24.03 Features=normal
   Gres=mem128:no_consume:1
   NodeAddr=jrc0130 NodeHostName=jrc0130 Version=psslurm-41-p14.03
   OS=Linux RealMemory=128952 AllocMem=0 Sockets=2 Boards=1
   State=ALLOCATED ThreadsPerCore=2 TmpDisk=0 Weight=1
   BootTime=2015-07-27T11:34:29 SlurmdStartTime=2015-07-27T11:34:53
   CurrentWatts=0 LowestJoules=0 ConsumedJoules=0
   ExtSensorsJoules=n/s ExtSensorsWatts=0 ExtSensorsTemp=n/s
```

Check the shares:

```
$ sshare
             Account       User Raw Shares Norm Shares   Raw Usage Effectv Usage  FairShare
-------------------- ---------- ---------- ----------- ----------- ------------- ----------
root                                          1.000000  7347830935      1.000000   0.500000
 root                    paschoul          1    0.000002           0      0.000000   1.000000
 deep                                   3000    0.004902           0      0.000000   1.000000
 eau00                                  3000    0.004902           0      0.000000   1.000000
 ecy00                                  3000    0.004902   460420812      0.062667   0.000142
 esmi17                                 3000    0.004902           0      0.000000   1.000000
 esmi19                                 3000    0.004902        1494      0.000000   0.999971
 esmi20                                 3000    0.004902    65827880      0.008957   0.281814
 grs200                                 3000    0.004902     1257977      0.000171   0.976079
 grs300                                 3000    0.004902         478      0.000000   0.999991
 grs400                                 3000    0.004902    45140069      0.006144   0.419461
 hac29                                  3000    0.004902          87      0.000000   0.999998
 hbn15                                  3000    0.004902           0      0.000000   1.000000
 hbn23                                  3000    0.004902           0      0.000000   1.000000
 hbn29                                  3000    0.004902    79553390      0.010826   0.216354
 hbn30                                  3000    0.004902    19171256      0.002609   0.691441
 hbn31                                  3000    0.004902   117928315      0.016051   0.103343
 hbn32                                  3000    0.004902           0      0.000000   1.000000
 hbn33                                  3000    0.004902           0      0.000000   1.000000
 zam                                    3000    0.004902    66646822      0.009071   0.277284
  zam                    paschoul       3000    0.000037       46031      0.000075   0.247122
 zdv590                                 3000    0.004902           0      0.000000   1.000000
```

Check the priorities:

```
$ sprio
          JOBID   PRIORITY        AGE  FAIRSHARE    JOBSIZE        QOS
            203      46776      32500          0      14277          0
           1771      46776      32500          0      14277          0
           6659      34303      32500         15       2788          0
           6660      34303      32500         15       2788          0
           6767      36084      32500         15       3569          0
           8435       1016        794          0        223          0
           8597       5208       4985          0        223          0
           8633        932        710          0        223          0
```

```
          8797      32555      32500          0         56          0
          8801      32555      32500          0         56          0
          8805      32555      32500          0         56          0
          8886       1304       1082          0        223          0
          8996      14752      11183          0       3569          0
         35800      14400      10831          0       3569          0
         36848       4878       4821          2         56          0
         36858       1615       1558          2         56          0
         39621       1609       1552          2         56          0
         39672        983        926          2         56          0
         39714       1749       1692          2         56          0
         39726       1770       1713          2         56          0
         40608     101329        771          0        558     100000
         44116        744        688          0         56          0
         44122        744        688          0         56          0
         44257     100662        606          0         56     100000
         44258     100661        606          0         56     100000
         44259     100661        606          0         56     100000
         44260     100661        606          0         56     100000
         44261     100661        606          0         56     100000
         44262     100661        606          0         56     100000
         44263     100661        606          0         56     100000
         44264     100661        606          0         56     100000
         44265     100661        606          0         56     100000
         44266     100661        606          0         56     100000
         44267     100661        606          0         56     100000
```

7.7 Accounting Commands

Check user association:

```
$ sacctmgr show assoc where user=paschoul
   Cluster     Account        User  Partition      Share GrpJobs GrpNodes  GrpCPUs  GrpMem GrpSubmit
GrpWall  GrpCPUMins MaxJobs MaxNodes  MaxCPUs MaxSubmit     MaxWall  MaxCPUMins                  QOS
Def QOS GrpCPURunMins
---------- ---------- ---------- ---------- --------- ------- -------- -------- ------- ---------
-----------    -----------    -------    --------    --------    ---------    -----------    -----------
-------------------- --------- -------------
          jureca                    zam       paschoul                                          3000
normal    normal
           jureca                   root        paschoul               maint                       1
nolimits  nolimits
```

Check all QoSs:

```
$ sacctmgr show qos
       Name   Priority  GraceTime     Preempt PreemptMode                                    Flags
UsageThres  UsageFactor    GrpCPUs    GrpCPUMins  GrpCPURunMins  GrpJobs    GrpMem  GrpNodes  GrpSubmit
GrpWall  MaxCPUs  MaxCPUMins MaxNodes     MaxWall MaxCPUsPU MaxJobsPU MaxNodesPU MaxSubmitPU
----------  ----------  ----------  ----------  -----------  ----------------------------------------
----------  -----------  --------  -----------  -------------  -------  -------  --------  ---------
----------- -------- ----------- -------- ----------- --------- --------- ---------- -----------
    normal      100000   00:00:00                 cluster                                DenyOnLimit
1.000000
280  1-00:00:00                280        280        4096
   lowcont           0   00:00:00                 cluster                                DenyOnLimit
1.000000
280    06:00:00                  24        280        4096
    nocont           0   00:00:00                 cluster                                DenyOnLimit
1.000000
280    06:00:00                  24        280        4096
 suspended           0   00:00:00                 cluster                                 DenyOnLimit
1.000000
0    00:00:00                   0          0           0
  nolimits      100000   00:00:00                 cluster                                DenyOnLimit
1.000000
```

Check one QoS:

```
$ sacctmgr show qos where name=normal
      Name   Priority  GraceTime     Preempt PreemptMode                                    Flags
UsageThres  UsageFactor    GrpCPUs    GrpCPUMins  GrpCPURunMins  GrpJobs    GrpMem  GrpNodes  GrpSubmit
GrpWall  MaxCPUs  MaxCPUMins MaxNodes     MaxWall MaxCPUsPU MaxJobsPU MaxNodesPU MaxSubmitPU
----------  ----------  ----------  ----------  -----------  ----------------------------------------
----------  -----------  --------  -----------  -------------  -------  -------  --------  ---------
----------- -------- ----------- -------- ----------- --------- --------- ---------- -----------
    normal      100000   00:00:00                 cluster                                DenyOnLimit
1.000000
280  1-00:00:00                280        280        4096
```

Check old jobs history:

```
$ sacct -X -u hgu147
       JobID    JobName  Partition    Account  AllocCPUS      State ExitCode
------------ ---------- ---------- ---------- ---------- ---------- --------
42861            T34pH4      batch      hgu14         48    TIMEOUT      0:1
42866            T28pH8      batch      hgu14         48    TIMEOUT      0:1
42872            T26pH8      batch      hgu14         48    RUNNING      0:0
42873            T26pH5      batch      hgu14         48    RUNNING      0:0
42874            T26pH4      batch      hgu14         48    RUNNING      0:0
42875            T24pH8      batch      hgu14         48    RUNNING      0:0
42876            T24pH5      batch      hgu14         48    RUNNING      0:0
42877            T22pH8      batch      hgu14         48    RUNNING      0:0
42878            T22pH5      batch      hgu14         48    RUNNING      0:0
```

Check old jobs with different format and specified time frame:

```
$ sacct -X -u kraused --format="jobid,user,nnodes,nodelist,state,exit" -S 2014-11-15T00:00:00 -E 2015-
11-17T18:00:00
       JobID      User   NNodes        NodeList      State ExitCode
------------ --------- -------- --------------- ---------- --------
2              kraused        1         jrc0106  COMPLETED      0:0
3              kraused        8  jrc[0106-0113]     FAILED      2:0
10             kraused        8  jrc[0108-0115]  COMPLETED      0:0
297            kraused       70 jrc[0282-0306,+ CANCELLED+      0:0
298            kraused       70 jrc[0106-0125,+     FAILED    127:0
299            kraused        2   None assigned CANCELLED+      0:0
300            kraused        2   None assigned CANCELLED+      0:0
301            kraused        2   None assigned CANCELLED+      0:0
302            kraused        2   None assigned CANCELLED+      0:0
```

```
303            kraused        2   None assigned CANCELLED+      0:0
304            kraused        2   None assigned CANCELLED+      0:0
305            kraused        2   None assigned CANCELLED+      0:0
306            kraused        2   None assigned CANCELLED+      0:0
307            kraused        2   None assigned CANCELLED+      0:0
308            kraused        2   None assigned CANCELLED+      0:0
309            kraused        2   None assigned CANCELLED+      0:0
310            kraused        2   None assigned CANCELLED+      0:0
311            kraused        2   None assigned CANCELLED+      0:0
312            kraused        2   None assigned CANCELLED+      0:0
313            kraused        2   None assigned CANCELLED+      0:0
314            kraused        2   None assigned CANCELLED+      0:0
315            kraused        2   None assigned CANCELLED+      0:0
316            kraused       70 jrc[0282-0306,+ CANCELLED+      0:0
317            kraused       70 jrc[0106-0125,+ CANCELLED+      0:0
318            kraused        2   None assigned CANCELLED+      0:0
319            kraused        2   None assigned CANCELLED+      0:0
320            kraused        2   None assigned CANCELLED+      0:0
321            kraused        2   None assigned CANCELLED+      0:0
36016          kraused        1         jrc0454     FAILED    127:0
36017          kraused        1         jrc0455     FAILED    127:0
36018          kraused        1         jrc0307  COMPLETED      0:0
36019          kraused        1         jrc0411  COMPLETED      0:0
36020          kraused        1         jrc0412  COMPLETED      0:0
36021          kraused        1         jrc0413  COMPLETED      0:0
36022          kraused        1         jrc0414  COMPLETED      0:0
36023          kraused        1         jrc0415  COMPLETED      0:0
36024          kraused        1         jrc0429  COMPLETED      0:0
36025          kraused        1         jrc0430  COMPLETED      0:0
36026          kraused        1         jrc0431  COMPLETED      0:0
36027          kraused        1         jrc0432  COMPLETED      0:0
36028          kraused        1         jrc0433  COMPLETED      0:0
36029          kraused        1         jrc0298  COMPLETED      0:0
36030          kraused        1         jrc0299  COMPLETED      0:0
36031          kraused        1         jrc0300  COMPLETED      0:0
36032          kraused        1         jrc0301  COMPLETED      0:0
```

8 Changelog

Version 2.3.0

- Chapter 1.1: Added info about the Booster Partition.
- Chapter 1.2: Added Booster Nodes in the table.
- Chapter 1.6: Added info for Booster/Haswell modules architectures.
- Chapter 2.3: Added Booster partitions in the table.
- Chapter 4.7: Created new chapter about Features and job submissions on Booster partitions.

Version 2.2.0

- Chapter 1.2: Updated the correct number of visualization nodes in the table of nodes.
- Chapter 2.3: Updated/reformatted partitions table and added new info/partitions.
- Chapter 2.5: Updated the QoS table.
- Chapter 2.6: Updated the GRES tables and the rules of the job submission filter.
- Chapter 3.2: Updated the GRES table.

Version 2.1.0

- Updated tables with JURECA nodes in section 1.2, fixed some out-dated values and added new entry for visualization logins.
- Updated chapter 1.6 with the new way to use the modules on JURECA.
- Updated the table with the partitions in chapter 2.3.
- Updated the table with the QoS limits and priorities in chapter 2.5.
- Corrected the tables about Generic Resources in chapter 2.6.
- Updated the priority values of the tables in chapter 2.7.
- Created chapter 4.6 about GPU computing.

Version 2.0.1

- Fixed some typos and the borders of a few tables.

Version 2.0.0

- Extended documentation for Phase 2 (complete system). All nodes types and the new partitions are documented in this version. Also new sections were added about the General Resourses (GRES) of Slurm.

Version 1.0.0

- First version of this document.

~~~

NOTE: This document was created using LibreOffice Writer and then it was exported to the PDF format. There is a known issue where the users cannot copy from this document some commands and then paste them on their terminals. Currently we don't know if there is a fix for this issue, but we will investigate it.
~~~