Porting and Testsystem PITBULL User's Manual

Support:
beratung@dkrz.de

2015-05-26

Contents

Chapter 1

Cluster Information

1.1 Introduction

PITBULL is a small testsystem which allows users of DKRZ's current supercomputer BLIZZARD to port and optimize their applications for the new Haswell CPU architecture, which will also be used for the first phase of the next supercomputer. This cluster's main purpose is to prepare users and DKRZ admins for a smooth transition to the next installation called MISTRAL that will be available in summer 2015.

1.2 Cluster Nodes

The PITBULL testsystem contains: 1 login, 18 compute, 1 pre/postprocess and 2 admin nodes (see Table 1.1).

type (nodes)	**hostname**	**CPU**	**cores(logical cpus/HT)**	**memory**	**description**
login (1)	btlogin1.dkrz.de (btc1)	2x Intel Xeon E5-2680 v3 (Haswell) @ 2.5GHz	24 (48)	128 GB	login nodes
compute (18)	btc[2-19]	2x Intel Xeon E5-2680 v3 (Haswell) @ 2.5GHz	24 (48)	128 GB	compute nodes
pre/post (1)	btg0	2x Intel Xeon E5-2680 v2 (Ivy-Bridge) @ 2.8GHz	20 (-)	128 GB	pre/postproc. nodes with Nvidia GPU
admin (2)	-	2x Intel Xeon E5-2620 @ 2.0GHz	12 (-)	32 GB	admin nodes

Table 1.1: PITBULL node configuration

The Operating System on the PITBULL cluster is Red Hat Enterprise Linux release 6.4 (Santiago). Regarding the network, we use FDR Infiniband with a non-blocking Fat Tree topology.

1.3 Data Management - Filesystems

On PITBULL we provide the Lustre parallel filesystem. We provide HOME, WORK and SCRATCH partitions, which have different purposes as indicated in the following table:

filesystem	**mount point**	**description**
HOME	/home/ → /lustre/pf	Home filesystem - without backup
WORK	/lustre/work	Project work filesystem - without backup
SCRATCH	/lustre/scratch	User scratch filesystem - without backup, regularly purged
DATA	/pool/data	Input data pool - without backup
sw	/sw → /lustre/sw	Software repository available via module commands

Table 1.2: PITBULL filesystem configuration

The GPFS filesystems on BLIZZARD are not available on PITBULL. The new system MISTRAL will use the same filesystems as PITBULL, but no data will be transferred from PITBULL to MISTRAL. Instead, DKRZ will do an automatic migration of all user data from BLIZZARD GPFS to Lustre for $HOME and $WORK directories as soon as the MISTRAL system is starting production. Data that is needed on PITBULL should be copied manually from GPFS to Lustre.

1.4 Access to the Cluster

Users can have access to the login nodes of the system only through SSH connections. Currently there is no access possible from outside the DKRZ subnet, i.e.. user need to connect to BLIZZARD or WIZARD first. For example, to connect to the system, users must execute from their workstation the following command:

```
bash$ ssh username@blizzard.dkrz.de
bash$ ssh username@btlogin1.dkrz.de
```

Chapter 2

Software Environment

2.1 Modules

Most of the installed software on PITBULL is organized through modules. Loading a module adapts your environment variables to give you access to a specific set of software and its dependencies. The modules are not organized hierarchically but have internal consistency checks for dependencies and can uniquely be identified by naming convention `<modname>/<modversion>`. Optionally, the version of the compiler that was used to build the software is also encoded in the name (for example all modules built with the same Intel compiler version are labelled with e.g. `*-intel15`)

2.1.1 Modules Available

Table 2.1 provides a quick reference to some module categories. The list of available modules will steadily grow to cover the (general) software needs of DKRZ users.

type	modules available
compiler	**intel**: Intel compilers with frontends for C, C++ and Fortran **gcc**: Gnu compiler suite **nag**: NAG compiler
MPI	**intelmpi**: Intel MPI **bullxmpi**: Bullx-MPI with/without mellanox libraries **mvapich2**: MVAPICH2 (an MPI-3 implementation) **openmpi**: Open MPI
tools	**allinea-forge**: Allinea DDT debugger and MAP profiler **cdo**: command line Operators to manipulate and analyse Climate and NWP model Data **ncl**: NCAR Command Language **ncview**: visual browser for netCDF format files **python**: Python

Table 2.1: PITBULL module overview

2.1.2 Using the Module Command

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

command	description
module avail	Shows the available modules
module show <modname>/<version>	Shows what environment variables (paths) will be modified when loading the module
module add <modname>/<version>	Loads a specific module. Default version is loaded if the version is not given
module list	Lists what modules are currently loaded
module rm <modname>/<version>	Unloads a module
module purge	Unloads all modules
module switch <modname>/<version1> <modname>/<version2>	Replaces one module with another

Table 2.2: module command overview

2.1.3 Additional Software

Software which is not directly available via modules, like libraries, is located in the directory

`/sw/rhel6-x64/`

The main build characteristics are reflected by the directory names, e.g..

`/sw/rhel6-x64/netcdf/netcdf_fortran-4.4.2-intel14/`

is the 4.4.2 version of FORTRAN NetCDF library built with the Intel compiler version 14.

REMARK for NetCDF usage:

There is no module to set NetCDF paths for the user. If you need to specify such paths in Makefiles or similar, please use the nc-config and nf-config tool to get the needed compiler flags and libraries, e.g.

```
bash$ /sw/rhel6-x64/netcdf/netcdf_c-4.3.2-intel14/bin/nc-config --cflags
-I/sw/rhel6-x64/netcdf/netcdf_c-4.3.2-intel14/include \
-I/sw/rhel6-x64/sys/libaec-0.3.2-intel14/include \
-I/sw/rhel6-x64/hdf5/hdf5-1.8.14-threadsafe-intel14/include \
-I/sw/rhel6-x64/hdf4/hdf-4.2.10-intel14/include

bash$ /sw/rhel6-x64/netcdf/netcdf_c-4.3.2-intel14/bin/nc-config --libs
-L/sw/rhel6-x64/netcdf/netcdf_c-4.3.2-intel14/lib -lnetcdf

bash$ /sw/rhel6-x64/netcdf/netcdf_fortran-4.4.2-intel14/bin/nf-config --fflags
-I/sw/rhel6-x64/netcdf/netcdf_fortran-4.4.2-intel14/include

bash$ /sw/rhel6-x64/netcdf/netcdf_fortran-4.4.2-intel14/bin/nf-config --flibs
-L/sw/rhel6-x64/netcdf/netcdf_fortran-4.4.2-intel14/lib -lnetcdff \
-L/sw/rhel6-x64/netcdf/netcdf_c-4.3.2-intel14/lib \
-Wl,-rpath,/sw/rhel6-x64/netcdf/netcdf_c-4.3.2-intel14/lib -lnetcdf \
-L/sw/rhel6-x64/hdf5/hdf5-1.8.14-threadsafe-intel14/lib \
-Wl,-rpath,/sw/rhel6-x64/hdf5/hdf5-1.8.14-threadsafe-intel14/lib -lhdf5 -lhdf5_hl \
-L/sw/rhel6-x64/sys/libaec-0.3.2-intel14/lib \
-Wl,-rpath,/sw/rhel6-x64/sys/libaec-0.3.2-intel14/lib -lsz -lz -lcurl -lnetcdf
```

2.2 Compiler and MPI

On PITBULL we have installed the Intel compilers with some wrappers (depending on the loaded MPI module), in order to compile parallel programs using MPI. In addition the GCC is installed for convenience, although we recommend using the Intel compiler.

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, such that we recommend the use of the wrappers instead of the compilers themselves.

language	**compiler**	**Intel MPI Wrapper**	**Bullx MPI Wrapper**
Fortran 90/95/2003	ifort	mpiifort	mpif90
Fortran 77	ifort	mpiifort	mpif77
C++	icpc	mpiicpc	mpic++
C	icc	mpiicc	mpicc

Table 2.3: MPI compiler wrapper overview for Intel compiler

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

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
−sox	Stores useful information like compiler version, options used etc. in the executable
−ipo	Inter-procedural optimization
−xAVX or −xCORE-AVX2	Indicates the processor for which code is created
−help	Gives a long list of quite a big amount of options

Table 2.4: Intel compiler options

2.2.1 Compilation Examples

Compile an MPI program in Fortran using Intel compiler and Intel MPI

```
bash$ module add intel intelmpi
bash$ mpiifort −O2 −xCORE−AVX2 −o mpi_prog program.f90
```

Compile a hybrid MPI/OpenMP program using the Intel compiler and Bullx MPI:

```
bash$ module add intel bullxmpi
bash$ mpif90 −openmp −O2 −xCORE−AVX2 −o mpi_omp_prog program.f90
```

2.2.2 Recommendations

Intel Compiler

Using the compiler option `-xCORE-AVX2` resp. `-xHost` causes the Intel compiler to use full AVX2 support/vectorization (with FMA instructions) which might results in binaries that do not produce MPI decomposition independent results. Switching to `-xAVX` should solve this issue but result in up to 15% slower runtime.

MPI

The Bullx-MPI showed at least for the benchmarks of the HLRE-3 procurement a worse performance compared to Intel-MPI. Nevertheless, this picture might change for the MISTRAL cluster, since the network configuration will be different.

Chapter 3

Batch System - SLURM

3.1 SLURM Overview

SLURM is the Batch System (Workload Manager) used on PITBULL 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/user commands and daemons which are shown in Table 3.1 and Figure 3.1.

daemon	**description**
control daemon (**slurmctld**)	responsible for monitoring of available resources and scheduling of batch jobs, it is running on admin nodes as HA resource
database daemon (**slurmdbd**)	accessing and managing the MySQL database which stores all the information about users, jobs and accounting data
slurm daemon (**slurmd**)	functionality of the batch system and resource management, it is running on each compute node
step daemon (**slurmstepd**)	a job step manager spawned by slurmd to guide the user processes

Table 3.1: Overview on SLURM components

SLURM manages the compute and pre/postprocessing nodes as its main resource of the cluster. Several nodes are grouped together into partitions, which might overlap, i.e. one node might be contained in several partitions. Compared to LoadLeveler on BLIZZARD, partitions are the equivalent of classes, hence partitions are the main concept for users to start jobs on the PITBULL cluster.

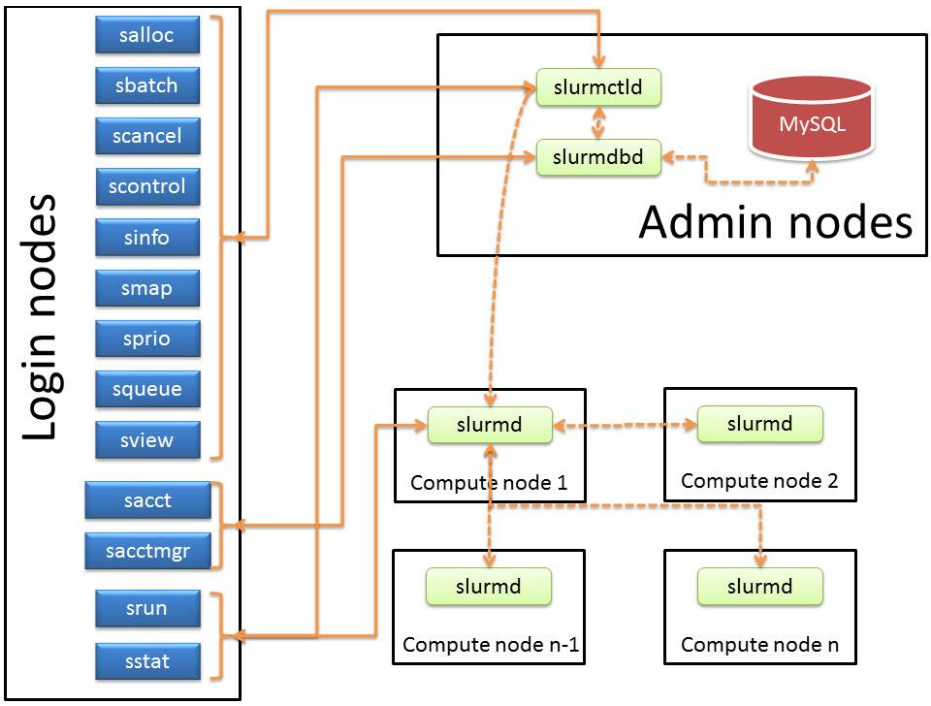

Figure 3.1: SLURM daemons and their interaction

3.2 SLURM 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. Jobs are the allocations of resources by the users in order to execute tasks on the cluster for a specified period of time. Furthermore, the concept of jobsteps is used by SLURM to describe a set of different tasks within the job. One can imagine jobsteps as smaller allocations or jobs within the job, which can be executed sequentially or in parallel during the main job allocation. The following table shows the partitions on PITBULL, the configured limits and default values:

partition	limit	value
compute (default)	• max wall-clock time for each job • min/max number of nodes per job • max number running/submitted jobs per user • node usage	30 minutes 1 / 18 nodes 3 / unlimited jobs exclusive
nightly	• max wall-clock time for each job • min/max number of nodes per job • max number running/submitted jobs per user • node usage • jobs start at 08:00pm from Monday to Friday (plus weekend all day)	8 hours 1 / 18 nodes 3 / unlimited jobs exclusive
shared	• max wall-clock time for each job • min/max number of nodes per job • max number running/submitted jobs per user • max memory per CPU • node usage	8 hours 1 / 1 node 3 / unlimited jobs 2.5 GByte shared
gpu	• max wall-clock time for each job • min/max number of nodes per job • max number running/submitted jobs per user • nodes available	8 hours 1 / 1 node 3 / unlimited jobs btg0

Table 3.2: Overview on SLURM partitions for PITBULL

3.3 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 behaviour. 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.

3.4 Accounting

The main policies concerning the batch model and accounting that are applied on PITBULL:

- Job scheduling according to priorities. The jobs with the highest priorities will be scheduled next.
- Backfilling scheduling algorithm. The scheduler checks the queue and may schedule jobs with lower priorities that can fit in the gap created by freeing resources for the next highest priority jobs.
- For each project a SLURM account is created where the users belong to. Each user might use the contingent from several projects that he belongs to.
- Users can submit jobs even when granted shares are already used - this result in a low priority, but the job might start when the system is empty.

Chapter 4

SLURM Usage

This chapter serves as an overview of user commands provided by SLURM and how users should use the SLURM batch system in order to run jobs on PITBULL. For a comparison to LoadLeveler commands see `http://slurm.schedmd.com/rosetta.pdf` or read the more detailed description of each command's manpage. A concise cheat sheet for SLURM can be downloaded here: `http://slurm.schedmd.com/pdfs/summary.pdf`

4.1 SLURM 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. For examples how to use these command, see Chapter 4.5.

sinfo show information about all partitions and nodes managed by SLURM as well as about general system state. It has a wide variety of filtering, sorting, and formatting options.

squeue 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. The most relevant job states are running (R), pending (PD), completing (CG), completed (CD) and cancelled (CA). The TIME field shows the actual job execution time. The NODELIST (REASON) field indicates on which nodes the job is running or the reason why the job is pending. Typical reasons for pending jobs are waiting for resources to become available (Resources) and queuing behind a job with higher priority (Priority).

sbatch submit a batch script. The script will be executed on the first node of the allocation. 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 parallel applications.

scancel 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.

salloc 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 this shell you should use srun to interactively start a parallel applications. The allocation is released when the user exits the shell.

srun initiate parallel job steps within a job or start an interactive job.

scontrol (primarily used by the administrators) provides some functionality for the users to manage jobs or get some information about the system configuration such as nodes, partitions, jobs, and configurations.

smap graphically shows the state of the partitions and nodes using a curses interface.

sprio query job priorities.

sshare retrieve fair-share information for each account the user belongs to.

sstat query status information related to CPU, task, node, RSS and virtual memory about a running job.

sview graphical user interface to get state information for jobs, partitions, and nodes.

sacct retrieve accounting information about jobs and job steps. For completed jobs sacct queries the accounting database.

sacctmgr (primarily used by the administrators) query information about accounts and other accounting information.

4.2 Batch Jobs

Users submit batch applications (usually shell 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 required allocation options that can be defined in a job script:

#SBATCH option	default value	description
−−nodes=<number>	1	Number of nodes for the allocation
−−ntasks=<number>	1	Number of tasks (MPI processes). Can be omitted if −−nodes and −−ntasks-per-node are given
−−ntasks-per-node=<num>	1	Number of tasks per node. If keyword omitted the default value is used, but there are still 48 CPUs available per node for current allocation (if not shared)
−−cpus-per-task=<num>	1	Number of threads (logical CPUs) per task. Used for OpenMP or hybrid jobs
−−output=<path>	slurm-<jobID>.out	Path to the file for the standard output
−−error=<path>	slurm-<jobID>.out	Path to the file for the standard error
−−time=<walltime>	partition dependent	Requested walltime limit for the job
−−partition=<name>	compute	Partition to run the job
−−mail-user=<email>	username	Email address for notifications
−−mail-type=<mode>	NONE	Event types for email notifications
−−job-name=<jobname>	jobscript's name	Job name
−−account=<projectaccount>	none	Project that should be charged

Table 4.1: SLURM sbatch options

Multiple srun calls can be placed in a single batch script. Options such as `--nodes`, `--ntasks` and `--ntasks-per-node` are taken from the sbatch arguments but can be overwritten for each srun invocation.

Remind the difference between options for selection, allocation and distribution in SLURM. Selection and allocation works with **sbatch**, but task distribution and binding should directly be specified with **srun** (within an sbatch-script). The following steps give an overview, for details see the further documentation below.

1. Resource Selection, e.g.
   - `#SBATCH --nodes=2`
   - `#SBATCH --sockets-per-node=2`
   - `#SBATCH --cores-per-socket=12`
2. Resource Allocation, e.g.
   - `#SBATCH --ntasks=12`
   - `#SBATCH --ntasks-per-node=6`
   - `#SBATCH --ntasks-per-socket=3`
   - `#SBATCH --cpus-per-task=8`
3. Start the application relying on the sbatch options only. Task binding and distribution with srun, e.g.

```
srun --cpu_bind=cores --distribution=block:cyclic <my_binary>
```

4. Start the application using only parts of the allocated resources, one needs to give again all relevant allocation options to srun (like `--ntasks` or `--ntasks-per-node`), e.g.

```
srun --ntasks=2 --ntasks-per-node=1 --cpu_bind=cores \
--distribution=block:cyclic <my_binary>
```

The job script is submitted using:

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

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

CAUTION

On the PITBULL system the setting of `-A` resp. `--account` is necessary to submit a job, otherwise submission will be rejected. You can query the accounts for which job submission is allowed using the command:

```
bash$ sacctmgr list assoc format=account,qos,MaxJobs user=$USER
```

4.2.1 Hyper-Threading (HT)

Similar to the IBM Power6 used in BLIZZARD, the Haswell processors deployed in PIT-BULL offer the possibility of Simultaneous Multithreading (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 cpus or Hardware Threads (HWT). Therefore, a maximum of 48 processes can be executed on each compute node without overbooking.

Each compute node on PITBULL consists of two Intel Xeon E5-2680 v3 processors, located on socket zero and one, with 12 physical cores each. These cores are numbered 0 to 23 and the hardware threads are numbered 24 to 47. Figure 4.1 depicts a node schematically and illustrates the naming convention.

On PITBULL we enabled HT on each compute node and SLURM always uses the option `--threads-per-core=2` implicitly, such that the user is urged to bind the tasks/threads in an appropriate way. In Section 4.2.3 there are examples (commands and job scripts) on how to use HT or not.

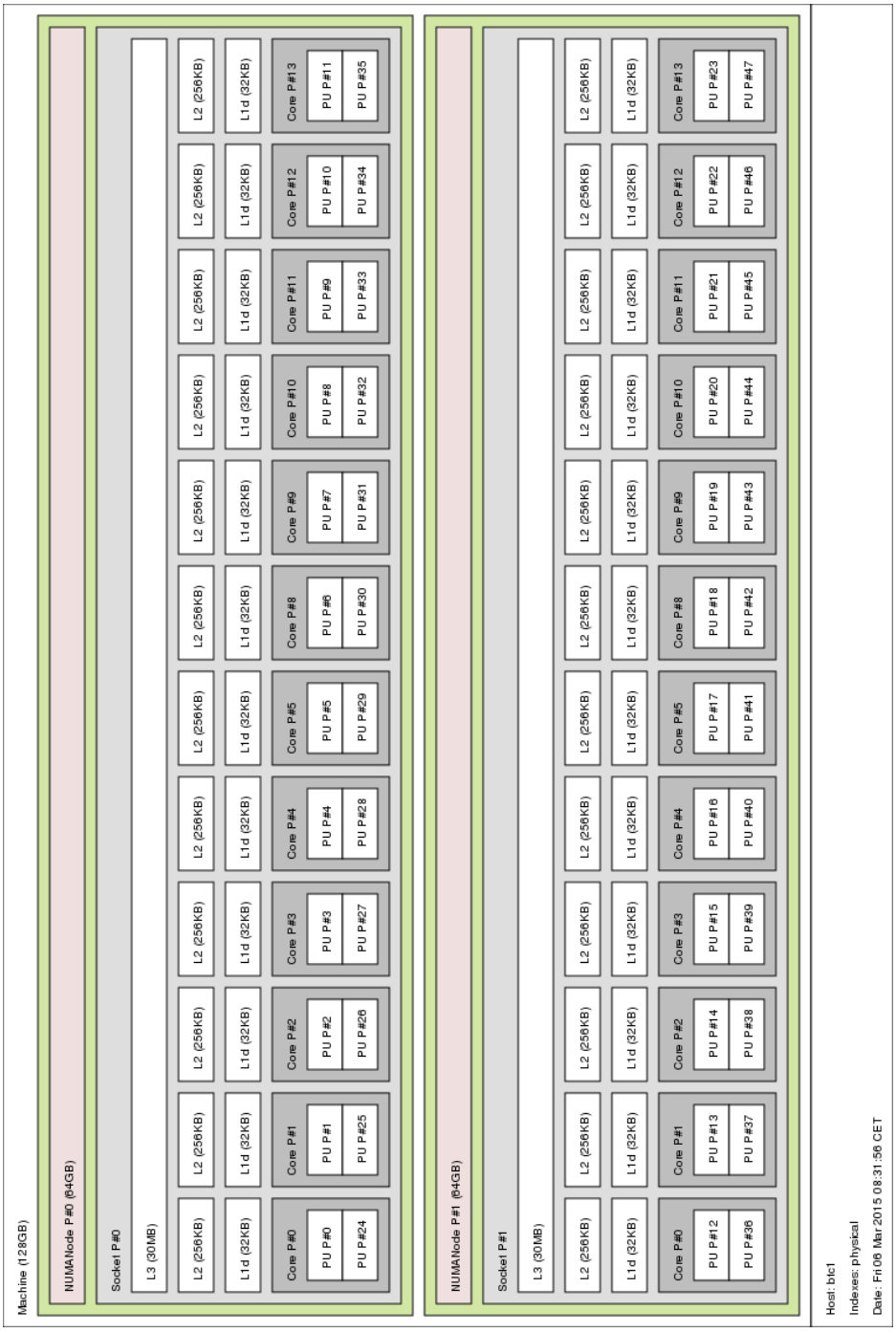

Figure 4.1: Schematic illustration of btc compute nodes

4.2.2 Frequency Scaling

The Intel Haswell processor allows for CPU frequency scaling which in general enables the operating system to scale the CPU frequency up or down in order to save power. CPU frequencies can be scaled automatically depending on the system load or manually by userspace programs. This is done via power schemes for the CPU - so called governors. Only one may be active at a time. The default governor is "ondemand" which allows the operating system to scale down the CPU frequency on the compute nodes to 1.2GHz if they are in idle state. The user can set the governor to "userspace" in order to allow for different CPU frequencies. Therefore the batch job needs to define the desired behaviour via the environmental variable `SLURM_CPU_FREQ_REQ` or via the srun option `--cpu-freq`. Possible values are

- `export SLURM_CPU_FREQ_REQ=2500000` to set a fixed frequency of 2.5GHz; other allowed frequencies are 1.2, 1.3, ..., 2.5 GHz
- `export SLURM_CPU_FREQ_REQ=ondemand` to enable automatically frequency scaling depending on the workload

By default srun configures all CPUs to run at fixed 2.5GHz in order to get similar wallclock runtime between different jobs if no options (or the binaries) are changed.

4.2.3 Job Script Examples

Serial job

```
#!/bin/bash

#SBATCH --job-name=my_job # Specify job name
#SBATCH --partition=shared # Specify partition name
#SBATCH --ntasks=1        # Specify max. number of tasks
                          # to be invoked
#SBATCH --mem=<MB> # Specify  real  memory required
#SBATCH --time=00:30:00 # Set a limit  on the  total  run time
#SBATCH --mail-type=FAIL # Notify user by email in case of
                          # job  failure
#SBATCH --mail-user=you@email # Set your e-mail address
#SBATCH --account=x12345 # Charge resources on this
                          # project account

# execute serial  programs, e.g.
cdo <operator> <ifile> <ofile>
```

CAUTION

The shared partition has a limit of 2560MB memory per CPU, if your serial job needs more memory you have to increase the number of `--ntasks` although you might not use all these CPUs.

OpenMP job

```
#!/bin/bash

#SBATCH --job-name=my_job # Specify job name
#SBATCH --partition=shared # Specify partition name
#SBATCH --ntasks=1        # Specify max. number of tasks
                          # to be invoked
#SBATCH --cpus-per-task=8 # Specify number of CPUs per task
#SBATCH --time=00:30:00 # Set a limit  on the  total  run time
#SBATCH --account=x12345 # Charge resources on this
                          # project account

# bind your OpenMP threads
export OMP_NUM_THREADS=8
export KMP_AFFINITY=granularity=thread,compact,1
export KMP_STACKSIZE=64M
```

```
# execute OpenMP programs, e.g.
cdo -P 8 <operator> <ifile> <ofile>
```

CAUTION

All nodes in the system are able to use HyperThreading, i.e. the option `--threads-per-core=2` is set by default. Hence, one needs to specify the value of `--cpus-per-task` as multiple of HyperThreads. Whether HT is used or not is defined via the envVar `KMP_AFFINITY`, see 4.3 for details.

MPI job

```
#!/bin/bash

#SBATCH --job-name=my_job # Specify job name
#SBATCH --partition=compute # Specify partition name
#SBATCH --nodes=4       # Specify number of nodes
#SBATCH --ntasks-per-node=24 # Specify number of tasks on each node
#SBATCH --cpus-per-task=2 # use 2 CPUs per task in order
                          # to not use HyperThreading
#SBATCH --time=00:30:00 # Set a limit on the total run time
#SBATCH --account=x12345 # Charge resources on this
                          # project account

# Run MPI parallel program using Intel MPI
module load intelmpi

export I_MPI_PMI_LIBRARY=/usr/lib64/libpmi.so

srun -l --cpu_bind=verbose,cores ./myprog
```

```
#!/bin/bash

#SBATCH --job-name=my_job # Specify job name
#SBATCH --partition=compute # Specify partition name
#SBATCH --nodes=4       # Specify number of nodes
#SBATCH --ntasks-per-node=48 # Specify number of tasks on each node
#SBATCH --time=00:30:00 # Set a limit on the total run time
#SBATCH --account=x12345 # Charge resources on this
                          # project account

# Run MPI parallel program using Intel MPI
module load intelmpi

export I_MPI_PMI_LIBRARY=/usr/lib64/libpmi.so

srun -l --cpu_bind=verbose,threads ./myprog
```

Instead of specifying the choice to use HyperThreads or not explicitly via `--cpus-per-task` and `--cpu_bind`, one might also use the srun option `--hint=[no]multithread`. The following example allocates one full node and uses 24 tasks without HyperThreads for the

first program run and then 48 tasks using HyperThreads for the second run. Such a procedure might be used in order to see whether an application takes benefits of the use of HyperThreads or not.

```
#!/bin/bash

#SBATCH --job-name=my_job # Specify job name
#SBATCH --partition=compute # Specify partition name
#SBATCH --nodes=1        # Specify number of nodes
#SBATCH --time=00:30:00 # Set a limit on the total run time
#SBATCH --account=x12345 # Charge resources on this
                         # project account

# Run MPI parallel program using Intel MPI
module load intelmpi

export I_MPI_PMI_LIBRARY=/usr/lib64/libpmi.so

# first  check how myprog performs without HyperThreads
srun -l --cpu_bind=verbose --hint=nomultithread --ntasks=24 ./myprog

# second check how myprog performs with HyperThreads
srun -l --cpu_bind=verbose --hint=multithread --ntasks=48 ./myprog
```

Hybrid MPI/OpenMP job

The first hybrid MPI/OpenMP job example will allocate 4 compute nodes for 1 hour. The job will have 24 MPI tasks in total, 6 tasks per node and 4 OpenMP threads per task. On each node 24 cores will be used (no HyperThreads are used).

```
#!/bin/bash

#SBATCH --job-name=my_job # Specify job name
#SBATCH --partition=compute # Specify partition name
#SBATCH --nodes=4        # Specify number of nodes
#SBATCH --ntasks-per-node=6 # Specify number of task on each node
#SBATCH --cpus-per-task=8 # Allocate that many CPUs for HT
#SBATCH --time=00:30:00 # Set a limit on the total run time
#SBATCH --account=x12345 # Charge resources on this
                         # project account

# bind your OpenMP threads
export OMP_NUM_THREADS=4
export KMP_AFFINITY=granularity=core,compact,1
export KMP_STACKSIZE=64M

# Run MPI/OpenMP parallel program
module load intelmpi

export I_MPI_PMI_LIBRARY=/usr/lib64/libpmi.so

srun -l --cpu_bind=verbose,cores ./myprog
```

The second hybrid MPI/OpenMP job example will run on 2 compute nodes having 6 tasks per node and starting 8 threads per node using HyperThreading.

```
#!/bin/bash

#SBATCH --job-name=my_job # Specify job name
#SBATCH --partition=compute # Specify partition name
#SBATCH --nodes=2       # Specify number of nodes
#SBATCH --ntasks-per-node=6 # Specify number of tasks on each node
#SBATCH --cpus-per-task=8 # Allocate that many CPUs for HT
#SBATCH --time=00:30:00 # Set a limit on the total run time
#SBATCH --account=x12345 # Charge resources on this
                         # project account

# bind your OpenMP threads
export OMP_NUM_THREADS=8
export KMP_AFFINITY=granularity=thread,compact,1
export KMP_STACKSIZE=64M

# Run MPI/OpenMP parallel program
module load intelmpi

export I_MPI_PMI_LIBRARY=/usr/lib64/libpmi.so

srun -l --cpu_bind=verbose,threads ./myprog
```

4.2.4 Job Steps

Job steps can be thought of as small allocations or jobs inside the current job/allocation. Each call of srun creates a job-step which implies that one job/allocation given via sbatch can have one or several job steps executed in parallel or sequentially. Instead of submitting many single-node jobs, the user might also use job steps inside a single job having multiple nodes allocated. A job using job steps will be accounted for all the nodes of the allocation regardless if all nodes are used for job steps or not.

The following example uses job steps to execute MPI programs in different job steps sequentially after each other and also parallel to each other inside the same job allocation. In total 4 nodes are allocated: the first 2 job steps run on all nodes after each other, while the job steps 3 and 4 run in parallel each using only 2 nodes.

```
#!/bin/bash

#SBATCH --nodes=4
#SBATCH --time=00:30:00
#SBATCH --account=x12345

# run 2 job steps after each other
srun -N4 --ntasks-per-node=24 --time=00:10:00 ./mpi_prog1
srun -N4 --ntasks-per-node=24 --time=00:20:00 ./mpi_prog2

# run 2 job steps in parallel
srun -N1 -n24 ./mpi_prog3 &
srun -N3 --ntasks-per-node=24 ./mpi_prog4 &
```

4.2.5 Dependency Chains

SLURM supports dependency chains which are collections of batch jobs with defined dependencies. Job dependencies can be defined using the `--dependency` argument of sbatch.

```
#!/bin/bash

#SBATCH --dependency=<type>
```

The available dependency types for job chains are

- **after:<jobID>** job starts when job with <jobID> begun execution
- **afterany:<jobID>** job starts when job with <jobID> terminates
- **afterok:<jobID>** job starts when job with <jobID> terminates successfully
- **afternotok:<jobID>** job starts when job with <jobID> terminates with failure
- **singleton** jobs starts when any previously job with the same job name and user terminates

4.2.6 Job Arrays

SLURM supports job arrays which is a mechanism for submitting and managing collections of similar jobs quickly and easily. Job arrays are only supported for the sbatch command and are defined using the option `--array=<indices>`. All jobs use the same initial options (e.g. number of nodes, time limit, etc.), however since each part of the job array has access to the `SLURM_ARRAY_TASK_ID` environment variable individual setting for each job is possible. For example the following job submission

```
bash$ sbatch --array=1-3 -N1 slurm_job_script.sh
```

will generate a job array containing three jobs. Assuming that the jobID reported by sbatch is 42, then the parts of the array will have the following environment variables set:

```
# array index 1
SLURM_JOBID=42
SLURM_ARRAY_JOB_ID=42
SLURM_ARRAY_TASK_ID=1

# array index 2
SLURM_JOBID=43
SLURM_ARRAY_JOB_ID=42
SLURM_ARRAY_TASK_ID=2

# array index 3
SLURM_JOBID=44
SLURM_ARRAY_JOB_ID=42
SLURM_ARRAY_TASK_ID=3
```

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`.

The following example creates a job array of 42 jobs with indices 0-41. Each job will run on a separate node with 24 tasks per node. Depending on the queuing situation, some jobs may be running and some may be waiting in the queue. Each part of the job array will execute the same binary but with different input files.

```
#!/bin/bash

#SBATCH --nodes=1
#SBATCH --output=prog-%A_%a.out
#SBATCH --error=prog-%A_%a.err
#SBATCH --time=00:30:00
#SBATCH --array=0-41
#SBATCH --account=x12345

srun --ntasks-per-node=24 ./prog input_${SLURM_ARRAY_TASK_ID}.txt
```

4.2.7 MPMD

SLURM supports the MPMD (Multiple Program Multiple Data) execution model that can be used for MPI applications, where multiple executables can have one common `MPI_COMM_WORLD` communicator. In order to use MPMD the user has to set the srun option `--multi-prog <filename>`. This option expects a configuration text file as an argument, in contrast to the SPMD (Single Program Multiple Data) case where srun has to be given the executable.

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

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

In the first field a comma separated list of ranks for the MPI tasks that will be spawned is defined. 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

Listing 4.1: Jobscript frame for the coupled MPI-ESM model using 8 nodes

```
#!/bin/bash

#SBATCH --nodes=8
#SBATCH --ntasks-per-node=24
#SBATCH --cpus-per-task=2
#SBATCH --time=00:30:00
#SBATCH --exclusive
#SBATCH --account=x12345

# Atmosphere
ECHAM_NPROCA=6
ECHAM_NPROCB=16

# Ocean
MPIOM_NPROCX=12
```

```
MPIOM_NPROCY=8

# Paths to executables
ECHAM_EXECUTABLE=../bin/echam6
MPIOM_EXECUTABLE=../bin/mpiom.x

# Derived values useful for running
(( ECHAM_NCPU = ECHAM_NPROCA * ECHAM_NPROCB ))
(( MPIOM_NCPU = MPIOM_NPROCX * MPIOM_NPROCY ))
(( NCPU = ECHAM_NCPU + MPIOM_NCPU ))
(( MPIOM_LAST_CPU = MPIOM_NCPU - 1 ))
(( ECHAM_LAST_CPU = NCPU - 1 ))

# create MPMD configuration file
cat > mpmd.conf <<EOF
0-${MPIOM_LAST_CPU}       $MPIOM_EXECUTABLE
${MPIOM_NCPU}-${ECHAM_LAST_CPU} $ECHAM_EXECUTABLE
EOF

# Run MPMP parallel program using Intel MPI
module load intelmpi

export I_MPI_PMI_LIBRARY=/usr/lib64/libpmi.so

srun -l --cpu_bind=verbose,cores --multi-prog mpmd.conf
```

4.3 Process and Thread Binding

4.3.1 OpenMP jobs

Thread binding is done via Intel runtime library using the `KMP_AFFINITY` environment variable. The syntax is

```
KMP_AFFINITY=[<modifier>,...]<type>[,<permute>][,<offset>]
```

with

- modifier
  - verbose: giving detailed output on how binding was done
  - granularity=core: reserve full physical cores (i.e. two logical CPUs) to run threads on
  - granularity=thread/fine: reserve logical CPUs / HyperThreads to run threads
- type
  - compact: places the threads as close to each other as possible
  - scatter: distributes the threads as evenly as possible across the entire allocation
- permute: controls which levels are most significant when sorting the machine topology map, i.e.. 0=CPUs (default), 1=cores, 2=sockets/LLC
- offset: indicates the starting position for thread assignment

For details please take a look at the Intel manuals or contact DKRZ Beratung. In most cases use

```
export KMP_AFFINITY=granularity=core,compact,1
```

if you do **not** want to use HyperThreads and

```
export KMP_AFFINITY=granularity=thread,compact,1
```

if you intend to use HyperThreads. You might also try scatter instead of compact placement to take benefit from bigger L3 cache.

4.3.2 MPI jobs

Process/task binding can be done via srun options `--cpu_bind` and `--distribution`. The syntax is

```
--cpu_bind=[{quiet,verbose},]type
--distribution=<block|cyclic|arbitrary|plane=<options>[:block|cyclic]>
```

with

- type:
  - cores: bind to physical cores
  - threads: bind to logical CPUs / HyperThreads
- first distribution method (before the ":") controls the distribution of resources across nodes
- second (optional) distribution method (after the ":") controls the distribution of resources across sockets within a node

For details please take a look at the manpage of srun or contact DKRZ Beratung. In most cases use

```
srun --cpu_bind=verbose,cores --distribution=block:cyclic ./myapp
```

if you do not want to use HyperThreads and

```
srun --cpu_bind=verbose,threads --distribution=block:cyclic ./myapp
```

if you intend to use HyperThreads. You might also benefit from different task distributions than block:cyclic.

4.3.3 Hybrid MPI/OpenMP jobs

In this case you need to combine the two binding methods mentioned above. Keep in mind that we are using `--threads-per-core=2` throughout the cluster. Hence you need to specify the amount of CPUs per process/task on the basis of HyperThreads even if you do not intend to use HyperThreads! The following table gives an overview on how to achieve correct binding using a full node

	MPI intranode distribution of tasks =	
	srun –distribution=block:block	**srun –distribution=block:cyclic**

no OpenMP, no HT	`#SBATCH --tasks-per-node=24`
	`#SBATCH --cpus-per-task=2`
	`srun --cpu_bind=cores`
	
	task0:cpu{0,24}, task1:cpu{1,25}, ...
		`#SBATCH --tasks-per-node=24`
		`#SBATCH --cpus-per-task=2`
		`srun --cpu_bind=cores`
		
		task0:cpu{0,24}, task1:cpu{12,36}, ...

no OpenMP, HT	`#SBATCH --tasks-per-node=48`
	`srun --cpu_bind=threads`
	
	task0:cpu0, task1:cpu24, task2:cpu1, ...
		`#SBATCH --tasks-per-node=48`
		`srun --cpu_bind=threads`
		
		task0:cpu0, task1:cpu12, task2:cpu1, ...

4 OpenMP threads, no HT	`#SBATCH --tasks-per-node=6`
	`#SBATCH --cpus-per-task=8`
	`export OMP_NUM_THREADS=4`
	`export KMP_AFFINITY=\`
	`granularity=core,\`
	`compact,1`
	`srun --cpu_bind=cores`
	
	task0:cpu{0,1,2,3,24,25,26,27},
	task1:cpu{4,5,6,7,28,29,30,31},
	...
	task0-thread0:cpu{0,24},
	task0-thread1:cpu{1,25},...
		`#SBATCH --tasks-per-node=6`
		`#SBATCH --cpus-per-task=8`
		`export OMP_NUM_THREADS=4`
		`export KMP_AFFINITY=\`
		`granularity=core,\`
		`compact,1`
		`srun --cpu_bind=cores`
		
		task0:cpu{0,1,2,3,24,25,26,27},
		task1:cpu{12,13,14,15,36,37,38,39},
		...
		task0-thread0:cpu{0,24},
		task0-thread1:cpu{1,25},...

4 OpenMP threads, HT	`#SBATCH --tasks-per-node=12`
	`#SBATCH --cpus-per-task=4`
	`export OMP_NUM_THREADS=4`
	`export KMP_AFFINITY=\`
	`granularity=tread,\`
	`compact,1`
	`srun --cpu_bind=threads`
	
	task0:cpu{0,1,24,25},
	task1:cpu{2,3,26,27},
	...
	task0-thread0:cpu0,
	task0-thread1:cpu1,
	task0-thread2:cpu24,...
		`#SBATCH --tasks-per-node=12`
		`#SBATCH --cpus-per-task=4`
		`export OMP_NUM_THREADS=4`
		`export KMP_AFFINITY=\`
		`granularity=thread,\`
		`compact,1`
		`srun --cpu_bind=threads`
		
		task0:cpu{0,1,24,25},
		task1:cpu{12,13,36,37},
		...
		task0-thread0:cpu0,
		task0-thread1:cpu1,
		task0-thread2:cpu24,...

4.4 Interactive Jobs

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

```
bash$ salloc --nodes=2 --time=00:30:00 --account=x12345
```

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 to spawn interactively their applications. For example:

```
bash$ srun --ntasks=4 --ntasks-per-node=2 --cpus-per-task=4 ./my_code
```

The interactive session is terminated by exiting the shell. In order to run commands directly on the allocated compute nodes, the user has to use ssh to connect to the desired nodes. For example:

```
bash$ salloc --nodes=2 --time=00:30:00 --account=x12345
salloc: Granted job allocation 13258
bash$ squeue -j 13258
   JOBID PARTITION NAME USER ST TIME NODES NODELIST(REASON)
   13258 compute bash x123456 R 0:11 2 btc[2-3]
bash$ hostname # we are still on the login node
btc0
bash$ ssh btc3
user@btc3's password:
user@btc3:~$ hostname
btc3
user@btc3:~$ exit
logout
Connection to btc3 closed.
bash$ exit
salloc: Relinquishing job allocation 13258
salloc: Job allocation 13258 has been revoked.
```

4.5 SLURM Command Examples

4.5.1 Job Control

Hold a job:

```
bash$ scontrol hold 4711
bash$ squeue
 JOBID PARTITION NAME USER ST TIME NODES NODELIST(REASON)
  4711 nightly tst_job b123456 PD 0:00 1 (JobHeldUser)
```

Release a job:

```
bash$ scontrol release 4711
bash$ squeue
 JOBID PARTITION NAME USER ST TIME NODES NODELIST(REASON)
  4711 nightly tst_job b123456 R 0:01 1 btc[7-11]
```

Cancel a job:

```
bash$ scancel 4711
```

4.5.2 Query Commands

Check the Queue:

```
bash$ squeue
JOBID PARTITION NAME USER ST TIME NODES NODELIST(REASON)
13194    compute MR_2_01P k203059 PD 0:00 13 (PartitionTimeLimit)
13263    compute LR0014.r k208024 R 4:03    16 btc[2-17]
```

Check the Queue for one user:

```
bash$ squeue -u k208024
JOBID PARTITION NAME USER ST TIME NODES NODELIST(REASON)
13263    compute LR0014.r k208024 R 4:03    16 btc[2-17]
```

Check partitions and nodes:

```
bash$ sinfo
PARTITION AVAIL TIMELIMIT NODES STATE NODELIST
compute*     up      30:00    18   idle  btc[2-19]
nightly    down   8:00:00    18   idle  btc[2-19]
shared       up   8:00:00     2   idle  btc[18-19]
gpu          up   8:00:00     1   idle  btg0
```

Check one partition:

```
bash$ scontrol show partition nightly
PartitionName=nightly
   AllocNodes=ALL AllowGroups=ALL Default=NO
   DefaultTime=NONE DisableRootJobs=NO GraceTime=0 Hidden=NO
   MaxNodes=UNLIMITED MaxTime=08:00:00 MinNodes=1 MaxCPUsPerNode=UNLIMITED
   Nodes=btc[2-19]
   Priority=1 RootOnly=NO ReqResv=NO Shared=EXCLUSIVE PreemptMode=OFF
   State=DOWN TotalCPUs=864 TotalNodes=18 SelectTypeParameters=N/A
   DefMemPerNode=UNLIMITED MaxMemPerNode=UNLIMITED
```

Check one node:

```
bash$ scontrol show node btg0
NodeName=btg0 Arch=x86_64 CoresPerSocket=10
   CPUAlloc=0 CPUErr=0 CPUTot=20 CPULoad=0.00 Features=(null)
   Gres=gpu:2
   NodeAddr=btg0 NodeHostName=btg0
   OS=Linux RealMemory=128000 AllocMem=0 Sockets=2 Boards=1
   State=IDLE ThreadsPerCore=1 TmpDisk=0 Weight=1
   BootTime=2015-03-02T18:04:57 SlurmdStartTime=2015-03-02T18:05:28
   CurrentWatts=149 LowestJoules=150 ConsumedJoules=37532742
   ExtSensorsJoules=n/s ExtSensorsWatts=0 ExtSensorsTemp=n/s
```

Check the shares:

```
bash$ sshare
Account    User Raw Shares Norm Shares Raw Usage Effectv Usage FairShare
-------------------------------------------------------------------------
x12345  b123456     parent     0.020501         216      0.000000  0.999984
y67890  b123456     parent     0.005390    46128362      0.099114  0.000003
```

Check the priorities:

```
bash$ sprio
  JOBID  PRIORITY      AGE FAIRSHARE PARTITION  QOS
  13194     20991      991         0     10000    10000
```

4.5.3 Accounting Commands

Check user association:

```
bash$ sacctmgr show assoc where user=b123456
   Cluster     Account       User
----------------------------------
      bullp      x12345    b123456
      bullp      y67890    b123456
      bullp      z24680    b123456
```

Check old jobs history:

```
bash$ sacct -X -u b123456
 JobID   JobName Partition Account AllocCPUS    State ExitCode
------------------------------------------------------------
13219     test.sh   compute x12345         96  COMPLETED 0:0
13235    wrf_job    compute x12345         32     FAILED  174:0
13258       bash    compute x12345         96  COMPLETED 0:0
```

Check old jobs with different format and specified time frame:

```
bash$ sacct -X -u b123456 --format="jobid,nnodes,nodelist,state,exit"
          -S 2015-01-01 -E 2015-31-01T23:59:59
        JobID   NNodes        NodeList       State ExitCode
-------------------------------------------------------------
13220                2        btc[2-3]  COMPLETED 0:0
13221                1            btc2     FAILED  174:0
13227                1            btc2     FAILED  174:0
13235                1            btc4     FAILED  174:0
```