DKRZ
DEUTSCHES
KLIMARECHENZENTRUM

# MISTRAL User's Manual
# Phase2 Version

Support:
beratung@dkrz.de

2016-08-01

# Contents

# Chapter 1

# Cluster Information

## 1.1 Introduction

MISTRAL, the High Performance Computing system for Earth system research (HLRE-3), is DKRZ's first petascale supercomputer. The HPC system has a peak performance of 3.14 PetaFLOPS and consists of approx. 3,000 compute nodes, 100,000 compute cores, 240 Terabytes of memory, and 54 Petabytes of disk. For access to MISTRAL you need to be a member in at least one active HLRE project, have a valid user account, and accept DKRZ's "Guidelines for the use of information-processing systems of the Deutsches Klimarechenzentrum GmbH (DKRZ)".

## 1.2 Cluster Nodes

The MISTRAL Phase 1 system was brought into operation in July 2015 and consists of approx. 1500 nodes. The compute nodes are housed in bullx B700 DLC (Direct Liquid Cooling) blade systems with two nodes forming one blade. Each node has two sockets, equipped with an Intel Xeon E5-2680 v3 12-core processor (Haswell) sharing 30 MiB L3 cache each. The processor clock-rate is 2.5 GHz. The MISTRAL phase 2 system is operational since July 2016 and adds another 1,434 nodes. The phase 2 nodes differ from those of phase 1 in the CPU type only. The new nodes use 2 Intel Xeon CPU E5-2695 v4 (aka Broadwell) CPUs running at 2.1 GHz, and each socket has 18 cores and 45MiB L3 cache. Thus, 24 physical cores per node are available on phase 1 and 36 on phase 2 respectively. Due to active Hyper-Threading, the operating system recognizes two logical CPUs per physical core. The aggregated main memory is 240 TB. The parallel file system Lustre provides 54 PB of usable disk space. The peak performance of the system is 3.14 PFLOPS/s.

Different kinds of nodes are available to users: 6 login nodes, 5 nodes for interactive data processing and analysis, approx. 3000 compute nodes for running scientific models, 32 fat memory nodes for pre- and postprocessing of data, and 12 nodes for running advanced visualization or GPGPU applications. See Table 1.1 for a listing of the specifics of different node types.

| type (nodes) | hostname | CPU | GPUs | memory |
|---|---|---|---|---|
| login (6)<br>*mistral.dkrz.de* | mlogin[100-105] | 2x12 core Intel Haswell @ 2.5GHz | none | 256 GB |
| interactive prepost (5)<br>*mistralpp.dkrz.de* | m[11550-11554] | 2x12 core Intel Haswell @ 2.5GHz | none | 256 GB |
| compute (1404) | m[10000-11367, 11404-11421, 11560-11577] | 2x12 core Intel Haswell @ 2.5GHz | none | 64 GB |
| compute (110) | m[11368-11403, 11422, 11431, 11440-11511] | 2x12 core Intel Haswell @ 2.5GHz | none | 128 GB |
| prepost (32) | m[11512-11543] | 2x12 core Intel Haswell @ 2.5GHz | none | 256 GB |
| visual/gpgpu (12) | mg[100-111] | 2x12 core Intel Haswell @ 2.5GHz | Nvidia Tesla K80 2x GK110BGL | 256 GB |
| compute2 (1116) | m[20000-21115] | 2x18 core Intel Broadwell @ 2.1GHz | none | 64 GB |
| compute2 (270) | m[21116-21385] | 2x18 core Intel Broadwell @ 2.1GHz | none | 128 GB |
| compute2 (48) | m[21386-21433] | 2x18 core Intel Broadwell @ 2.1GHz | none | 256 GB |

Table 1.1: MISTRAL node configuration

The Operating System on the Mistral cluster is Red Hat Enterprise Linux release 6.4 (Santiago). The batch system and workload manager is SLURM. All compute, pre-/postprocessing, and visualization nodes are integrated in one FDR InfinBand (IB) fabric with three Mellanox SX6536 director switches and fat tree topology with a blocking factor of 1:2:2. The measured bandwidth between two arbitrary compute nodes is 5.9 GByte/s with a latency of 2.7 $\mu$s. A scheme of the Infiniband topology is given in Picture 1.1, illustrating the blocking factors depending on which nodes are used for a specific job.

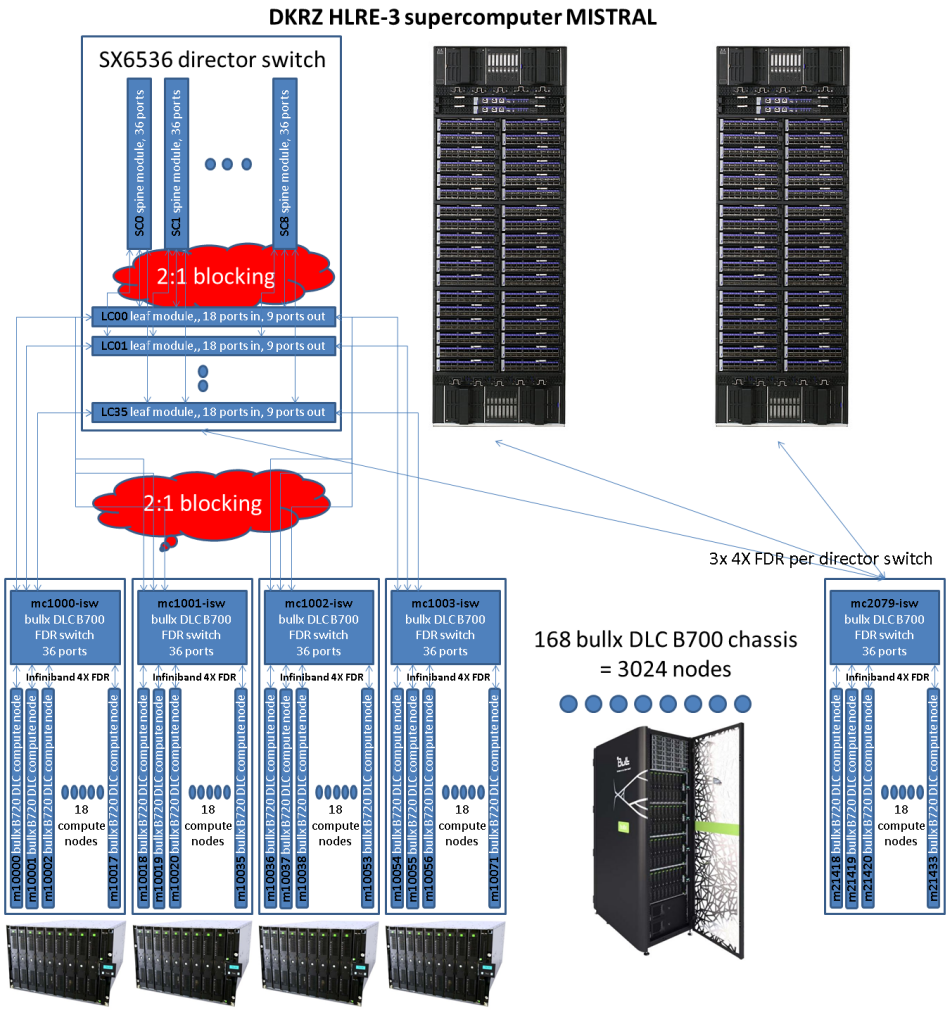


Figure 1.1: DKRZ mistral IB topology

## 1.3 Data Management - Filesystems

On MISTRAL, we provide the Lustre parallel filesystem version 2.5. Users have access to three different storage spaces: HOME, WORK, and SCRATCH. Each storage area has a specific purpose as described below.

HOME is the file system where users' sessions start upon login to MISTRAL. It is backed up and should be used to store shell setup files, source codes, scripts, and important files.

WORK is a project space available through the allocations process and shared between all users of a project. It provides disk space for large amounts of data, but it is not backed up. It can be used e.g. for writing raw model output and processing of data that is accessible to all project members.

SCRATCH is provided for temporary storage and processing of large data. To prevent the file system from overflowing, old data is automatically deleted. The granted retention period is 14 days.

The Lustre file system is available on all nodes (login and compute), so you can use them during interactive sessions and in batch jobs. The table below provides further details on the different storage spaces.

| File System | HOME | WORK | SCRATCH |
|---|---|---|---|
| path | /pf/[a,b,g,k,m,u]/<userid> | /work/<project> | /scratch/[a,b,g,k,m,u]/<userid> |
| envVar | $HOME | | |
| description | • Assigned to the user account<br>• Storage of personal settings files, source codes and scripts | • Assigned to project account<br>• Interim storage of output from running applications and frequently accessed data | • Assigned to user account<br>• Temporary storage and processing of large data sets |
| quota | 24 GB | according to annual project allocation | 15 TB |
| backup | yes, please contact DKRZ user's consultancy to restore files deleted by mistake | **no** | **no** |
| automatic data deletion | no | no | **yes** |
| data life time | until user account deletion | 1 month after project expiration | 14 days since the last file access |

Table 1.2: MISTRAL file system configuration

## 1.4 Access to the Cluster

The High Performance Computing system MISTRAL can be only accessed via Secure Shell (SSH) network protocol. For the file transfer between different hosts, SSH provides SCP and SFTP.

### 1.4.1 Login

You can log into MISTRAL with the following ssh command, replacing <userid> with your username:

```
bash$ ssh <userid>@mistral.dkrz.de
```

After having logged into MISTRAL, you will find yourself on one of the six login nodes mlogin[100-105]. The login nodes serve as the front ends to the compute nodes of the HPC cluster. They are intended for the editing and compilation of source code files, as well as for submitting, monitoring and cancelling of batch jobs. They can also be used for none time- and memory-intensive serial processing tasks. The routine data analysis and visualization, however, have to be performed on the interactive pre-/post-processing system **mistralpp.dkrz.de** or on prepost/visualization nodes. For interactive testing and debugging of parallel programs, you can use SLURM `salloc` command to allocate the required number of nodes.

### 1.4.2 Password

All DKRZ systems are managed by the LDAP protocol. The password can be changed through the DKRZ online services. A user defined password must contain at least eight non blank characters and must be a combination of upper and lower-case letters, numbers and special characters. In case you do not remember your password please contact DKRZ user's consultancy. Members of MPI and UniHH/CEN should contact CIS/CEN-IT.

### 1.4.3 Login Shell

The default login shell for new DKRZ users is bash. You can change your login shell to tcsh or ksh using the DKRZ online services. The settings that you would like to use every time you log in can be put into special shell setup files. A login bash shell looks for `.bash_profile`, `.bash_login` or `.profile` in your home directory and executes commands from the first file found. A non-login bash shell or bash subshell reads `.bashrc` file. Tcsh always reads and executes `.cshrc` file. If tcsh is invoked as the login shell, the file `.login` is sourced additionally. The typical tasks and settings that can be put in the shell setup files are for example:

- Creation of a custom prompt
- Modification of search path for external commands and programs
- Definition of environment variables needed by programs or scripts
- Definition of aliases
- Execution of commands (e.g. '`module load <modname>/<version>`')

# Chapter 2

# Software Environment

## 2.1 Modules

To cover the software needs of the DKRZ users and to maintain different software versions, DKRZ uses the module environment. 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 they have internal consistency checks for dependencies and can uniquely be identified with the naming convention `<modname>/<modversion>`. Optionally, the version of the compiler that was used to build the software is also encoded in its name (for example all modules built with the same Intel compiler version are labelled with e.g. `*-intel14`).

### 2.1.1 The Available Modules

Table 2.1 provides a quick reference to some module categories. The list of the available modules will steadily grow to cover the (general) software needs of the DKRZ users. Upon building new software, the complete list of the available tools is dynamically updated and it can be found at https://www.dkrz.de/Nutzerportal-en/doku/mistral/softwarelist

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

Table 2.1: MISTRAL module overview

### 2.1.2 Using the Module Command

Users can load, unload and query modules through the module command. The most important module sub-commands are listed in the table below.

| command | description |
|---|---|
| module avail | Shows the list of all the available modules |
| module show <modname>/<version> | Shows environment changes the modulefile <modname>/<version> will cause if loaded |
| module add <modname>/<version> | Loads a specific module. Default version is loaded if the version is not given |
| module list | Lists all modules 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

**Hint**: if only the <modname> (i.e. without <modversion>) is supplied, the lexically highest software version is loaded by default. If you want to make sure that the module version is not changed within job chains, you must explicitely supply the <modversion>.

For all the details of the module command, please refer to the man page or execute '`module --help`'.

To use the module command in a script you can source one of the following files in your script before any invocation of the module command:

```
# in bash or ksh script
bash$ source /sw/rhel6-x64/etc/profile.mistral

# in tcsh or csh script
csh$ source /sw/rhel6-x64/etc/csh.mistral
```

The '`module avail`' command provides up-to-date information on the installed software and their versions. For a comprehensive list of software and tools available on MISTRAL, please refer to the Software List at https://www.dkrz.de/Nutzerportal-en/doku/mistral/softwarelist

## 2.2 Compiler and MPI

On MISTRAL, we provide the Intel, GCC (GNU Compiler Collection), NAG, and PGI compilers and several Message Passing Interface (MPI) implementations: Bullx MPI with and without Mellanox tools, Intel MPI, MVAPICH2, and OpenMPI. No compilers and MPIs are loaded by default.

For most applications, we recommend to use the Intel compilers and Bullx MPI library with Mellanox tools to achieve the optimal performance on MISTRAL. For some applications, running on a small number of nodes might achieve a slightly better performance with the Intel compilers and Intel MPI.

Compilers and appropriate MPI libraries can be selected by loading the corresponding module files, for example:

```
# Use the latest versions of Intel compiler and Bullx MPI with Mellanox MXM + FCA tools
bash$ module load intel mxm fca bullxmpi_mlx

# Use the latest versions of Intel compiler and Intel MPI
bash$ module load intel intelmpi
```

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 wrappers overview for Intel compiler

The table below lists some useful compiler options that are commonly used for the Intel compiler. For further information, please refer to the man pages of the compiler or the comprehensive documentation on the Intel website https://software.intel.com/en-us/intel-software-technical-documentation.

| **option** | **description** |
|---|---|
| −qopenmp | 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<library path> | A path can be given in which the linker searches for libraries |
| −D | Defines a macro |
| −U | Undefines a macro |
| −I<include path> | Allows to add further directories to the include file search path |
| −sox | Stores useful information like the 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 options |

Table 2.4: Intel compiler options

### 2.2.1 Compilation Examples

Compile a hybrid MPI/OpenMP program using the Intel Fortran compiler and Bullx MPI with MXM and FCA:

```
bash$ module add intel mxm fca bullxmpi_mlx
bash$ mpif90 −qopenmp −xCORE−AVX2 −fp−model source −o mpi_omp_prog program.f90
```

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

```
bash$ module add intel intelmpi
bash$ mpiifort -O2 -xCORE-AVX2 -fp-model source -o mpi_prog program.f90
```

### 2.2.2 Recommendations

#### Intel Compiler

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

#### bullxMPI

The bullx-MPI was used throughout for the benchmarks of the HLRE-3 procurement. From BULL/ATOS point of view, a good environment will be to use `bullxMPI_mlx` with MXM, i.e. load the specific environment before compiling

```
bash$ module add intel mxm/3.3.3002 fca/2.5.2393 bullxmpi_mlx/bullxmpi_mlx- 1.2.8.3
bash$ mpif90 -O2 -xCORE-AVX2 -o mpi_prog program.f90
```

One must respect the order of loading the modules: compiler, MXM/FCA and afterwards bullx MPI. If the MXM/FCA environment is not loaded, one will use the bullx MPI without MXM and FCA tools.

In order to use the MXM (Mellanox Messaging) to accelerate the underlying send/receive (or put/get) messages, the following variables have to be set:

```
export OMPI_MCA_pml=cm
export OMPI_MCA_mtl=mxm
export MXM_RDMA_PORTS=mlx5_0:1
```

Alternatively, the default OpenMPI behavior can be specified using:

```
export OMPI_MCA_pml=obl
export OMPI_MCA_mtl=^mxm
```

Furthermore, FCA (Fabric Collectives Accelerations) accelerates the underlying collective operations used by the MPI/PGAS languages. To use FCA, one must specify the following variables:

```
export OMPI_MCA_coll=^ghc # disable BULLs GHC algorithm for collectives
export OMPI_MCA_coll_fca_priority=95
export OMPI_MCA_coll_fca_enable=1
```

You will find the bullxMPI documentation by Atos at
https://www.dkrz.de/Nutzerportal-en/doku/mistral/manuals.

#### bullxMPI and OpenMPI

Unlimited stacksize might have negative influence on performance - better use real needed amount, e.g.

```
ulimit -s 102400 # using bash

limit stacksize 102400 # using csh
```

In batch jobs, you will also have to propagate this setting from the job head node to all other compute nodes when invoking srun, i.e.

```
srun --propagate=STACK [any other options]
```

### IntelMPI

A good starting point for MPI based tuning is the following setting which enforces shared memory for MPI intranode communication and DAPL UD (user datagram) internode communication:

```
export I_MPI_FABRICS=shm:dapl
export I_MPI_FALLBACK=0
export I_MPI_DAPL_UD=enable
export I_MPI_DAPL_UD_PROVIDER=ofa-v2-mlx5_0-1u
```

### Libraries

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.

```
# Get paths to netCDF include files
bash$ /sw/rhel6-x64/netcdf/netcdf_c-4.3.2-gcc48/bin/nc-config --cflags

-I/sw/rhel6-x64/netcdf/netcdf_c-4.3.2-gcc48/include \
-I/sw/rhel6-x64/sys/libaec-0.3.2-gcc48/include \
-I/sw/rhel6-x64/hdf5/hdf5-1.8.14-threadsafe-gcc48/include

# Get options needed to link a C program to netCDF
bash$ /sw/rhel6-x64/netcdf/netcdf_c-4.3.2-gcc48/bin/nc-config --libs

-L/sw/rhel6-x64/netcdf/netcdf_c-4.3.2-gcc48/lib \
-Wl,-rpath,/sw/rhel6-x64/netcdf/netcdf_c-4.3.2-gcc48/lib -lnetcdf


# Get paths to Fortran netCDF include files
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

# Get options needed to link a Fortran program to netCDF
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 \
-Wl,-rpath,/sw/rhel6-x64/netcdf/netcdf_fortran-4.4.2-intel14/lib \
-L/sw/rhel6-x64/netcdf/netcdf_c-4.3.2-gcc48/lib \
-Wl,-rpath,/sw/rhel6-x64/netcdf/netcdf_c-4.3.2-gcc48/lib \
-L/sw/rhel6-x64/hdf5/hdf5-1.8.14-threadsafe-gcc48/lib \
-Wl,-rpath,/sw/rhel6-x64/hdf5/hdf5-1.8.14-threadsafe-gcc48/lib \
-L/sw/rhel6-x64/sys/libaec-0.3.2-gcc48/lib \
-Wl,-rpath,/sw/rhel6-x64/sys/libaec-0.3.2-gcc48/lib \
-lnetcdf -lhdf5_hl -lhdf5 -lsz -lcurl -lz
```

# Chapter 3

# Batch System - SLURM

## 3.1 SLURM Overview

SLURM is the Batch System (Workload Manager) used on MISTRAL 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 available resources and scheduling batch jobs. It is running on admin nodes as an 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, pre-/post-processing and visualisation 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 the main concept for users to start jobs on the MISTRAL cluster.

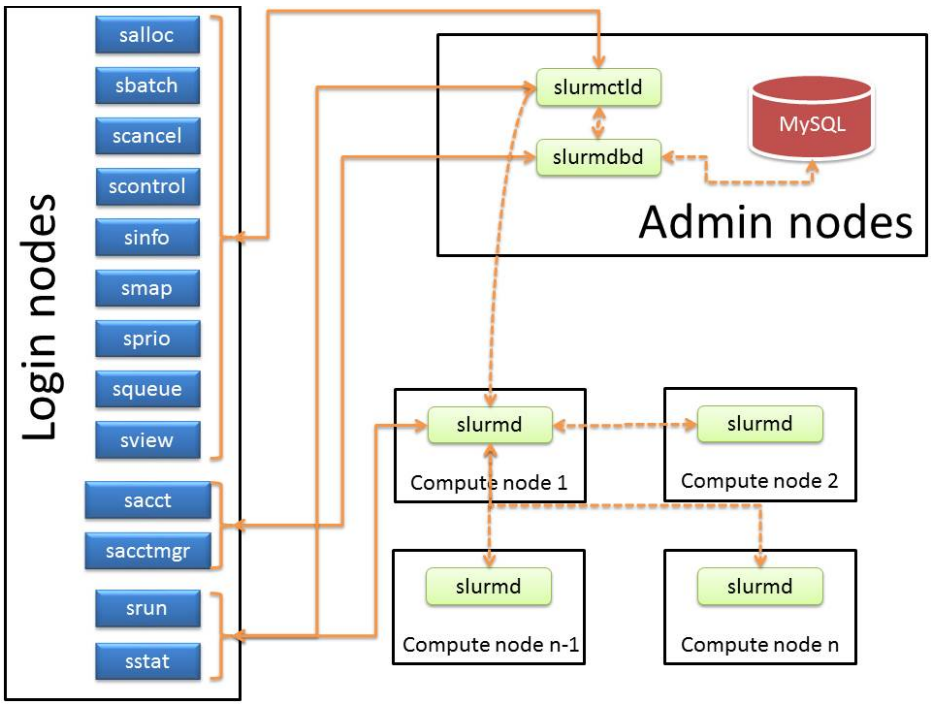


Figure 3.1: SLURM daemons and their interaction

## 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. The defined partitions can overlap, i.e. one node might be contained in several partitions.

**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 SLURM `sinfo` command lists all partitions and nodes managed by SLURM on MISTRAL as well as provides general information about the current status of the nodes:

```
bash$ sinfo
PARTITION AVAIL  TIMELIMIT  NODES  STATE NODELIST
compute*     up    8:00:00     13  down* m[10000,10278,...,11406]
compute*     up    8:00:00     14    mix m[10001,...,10036-10041]
compute*     up    8:00:00     81  alloc m[10042,,...,11332-11345]
compute*     up    8:00:00   1388   idle m[10002-10010,,...,11511]
prepost      up    4:00:00      3 drain* m[11518,11532,11554]
prepost      up    4:00:00     45   idle m[11512,...,11555-11559]
shared       up 7-00:00:00      1  down* m10000
shared       up 7-00:00:00     14    mix m[10001,...,10048]
shared       up 7-00:00:00     17  alloc m[10042,...,11332-11345]
shared       up 7-00:00:00     68   idle m[10002,...,11296-11331]
gpu          up    4:00:00     11   idle mg[100-101,103-111]
miklip       up 2-00:00:00      5   idle m[11419-11422,11431]
compute2     up    8:00:00   1000  alloc m[20000-20827,...,21395]
compute2     up    8:00:00    433   idle m[20828,...,21396-21433]
```

For detailed information about all available partitions and their limits, use the SLURM `scontrol` command as follows:

```
bash$ scontrol show partition
```

The following four partitions are currently defined on MISTRAL:

compute This partition consists of 1535 phase1 compute nodes (equipped with Haswell CPUs) and is intended for running parallel scientific applications. The compute nodes allocated for a job are used exclusively and cannot be shared with other jobs.

compute2 This partition consists of 1434 phase2 compute nodes (equipped with Broadwell CPUs) and is intended for running parallel scientific applications. The compute nodes allocated for a job are used exclusively and cannot be shared with other jobs.

shared This partition is defined on 100 nodes and can be used to run small jobs not requiring a whole node for the execution, so that one compute node can be shared between different jobs. The partition is dedicated for execution of shared memory applications parallelized with OpenMP or pthreads as well as for serial and parallel data processing jobs.

prepost The prepost partition is made up of 32 large-memory nodes. It is dedicated for memory intensive data processing jobs. Furthermore, interactive usage of nodes is permitted on this partition. If over-subscription is explicitly requested by the user using the "–share" option on job submission, resources can be shared with other jobs.

gpu The 12 nodes in this partition are additionally equipped with Nvidia Tesla K80 GPUs and can be used for 3-dimensional data visualization or execution of applications ported to GPUs. The nodes in this partition will replace Halo cluster in the future.

The limits configured for different partitions are listed in the table below.

| **partition** | **compute/compute2** | **prepost** | **shared** | **gpu** |
|---|---|---|---|---|
| *MaxNodes* | 512 | 2 | 1 | 2 |
| *MaxTime* | 8 hours | 12 hours | 7 days | 12 hours |
| *Shared* | exclusive | yes | yes | yes |
| *MaxMemPerCPU* | node limit | 5 GByte | 1.25 GByte | 5 GByte |

Table 3.2: Overview on SLURM partitions for MISTRAL

Beginning with September 1st 2016, all jobs on MISTRAL have to be assigned to a partition - there is no longer a default partition available. Choosing the partition can be done in various ways

- environment variable

  ```
  export SBATCH_PARTITION=<partitionname>
  ```

- batch script option

  ```
  #SBATCH [-p|--partition=]<partitionname>
  ```

- command line option

| sbatch [−p|−−partition=]<partitionname> |
|---|

Note that an environment variable will override any matching option set in a batch script, and command line option will override any matching environment variable.

## 3.3 Job Limits - QoS

As stated above, the partitions have several hard-limits that put an upper limit for the jobs on the wall-clock or other constraints. However, the actual job limits are enforced by the limits specified in both partitions and so called Quality-of-Services (QoS), meaning that using a special QoS the user might weaken the partition limits.

These QoS features play an important role to define the job priorities. By defining some QoS properties, the possible priorities can be modified in order to, e.g., enable earlier starttime of the jobs. In the following, we present the current list with the configured Quality-of-Services. Users are kindly asked to contact us should they have any demand for creating a new QoS feature.

| **QoS** | **express** |
|---|---|
| *description* | higher priority |
| *limits* | 4 nodes, 20 min wallclock |

Table 3.3: Overview on SLURM QoS for MISTRAL

## 3.4 Priorities and Accounting

The main policies concerning the batch model and accounting that are applied on MISTRAL are also defined via SLURM.

- SLURM schedules jobs according to their priorities. The jobs with the highest priorities will be scheduled next.
- Usage of backfilling scheduling algorithm: the SLURM 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 results in a low priority, but the job might start when the system is empty.

SLURM has a simple but well defined priority mechanism that allows to define different weighting models - the so called Multi-factor Job Priority plugin of SLURM. The actual priority for batch jobs on MISTRAL is based on a weighted sum of the following factors:

- age_factor $\in [0, 1]$ with 1 when age is more than PriorityMaxAge (0 day, 12 hours)
- FairShare_factor $\in [0, 1]$ as explained in detail at https://www.dkrz.de/Nutzerportal/dokumentationen/de-mistral/de-running-jobs/de-accounting-and-priorities

- QOS_factor $\in [0, 1]$ normalized according to 'sacctmgr show qos' (e.g. normal = 0, express = 0.1, bench = 1)

with the weights:

- PriorityWeightFairshare=1000
- PriorityWeightQOS=1000
- PriorityWeightAge=100

The final job priority is then calculated as

$$\begin{aligned} Job\_priority = & (PriorityWeightAge) * (age\_factor) + \\ & (PriorityWeightFairshare) * (fairshare\_factor) + \\ & (PriorityWeightQOS) * (QOS\_factor) \end{aligned} \tag{3.1}$$

While the command 'squeue' has format options (%p and %Q) that display a job's composite priority, the command 'sprio' can be used to display a breakdown of the priority components for each job, e.g.

```
bash$ sprio
          JOBID    PRIORITY        AGE  FAIRSHARE  QOS
        1421556        1175        100        975  100
        2015831         274         20        204   50
        2017372         258          0        258    0
        ...
```

## 3.5 Job Environment

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

# 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 MISTRAL. 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 Command Overview

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 jobs status, control jobs, manage accounting information and simplify their work with some utility commands. For the examples of how to use these command, see Chapter 4.6.

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

**squeue** queries 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 a higher priority (Priority).

**sbatch** submits 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** cancels 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** requests interactive jobs/allocations. As soon as a job starts , a shell (or other program specified on the command line) also start on the submission host (login node). From this shell, you should use srun to interactively start a parallel application. The allocation is released when the user exits the shell.

**srun** initiates parallel job steps within a job or start an interactive job.

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

**sprio** queries job priorities.

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

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

**sacct** retrieves accounting information about jobs and job steps. For completed jobs, sacct queries the accounting database.

## 4.2 Allocation Commands

A job allocation, i.e. a request for computing resources, can be created using the SLURM `salloc`, `sbatch` or `srun` command.

The usual way to allocate resources and execute a job on MISTRAL is to write a batch script and submit it to SLURM with the `sbatch` command - see section 4.2.3 for details. Alternatively, an interactive allocation can be used via the `salloc` command or a parallel job can directly be started with the `srun` command.

### 4.2.1 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 shell 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 m[10001-10002]

bash$ hostname # we are still on the login node
mlogin103

bash$ ssh m10001

user@m10001:~$ hostname
```

```
m10001

user@m10001:~$ exit
logout
Connection to m10001 closed.

bash$ exit # we need to exit in order to release the allocation
salloc: Relinquishing job allocation 13258
salloc: Job allocation 13258 has been revoked.
```

### 4.2.2 Spawning Command

With `srun` the users can spawn any kind of application, process or task inside a job allocation or directly start executing a parallel job (and indirectly ask SLURM to create the appropriate 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 defined with `srun` command, the options from `sbatch` or `salloc` will be inherited.

`srun` should preferably be used either

1. inside a job script submitted by `sbatch` - see 4.2.3.
2. or after calling `salloc`.

The allocation options of `srun` for the job-steps are (almost) the same as those for `sbatch` and `salloc` (please see the table in section 4.2.3 for some allocation options).

**Examples:**

Spawn 48 tasks on 2 nodes (24 tasks per node) for 30 minutes:

```
bash$ srun -N 2 -n 48 -t 30 -A xy1234 ./my_small_test_job
```

You will have to specify the account to be used for this job in the same manner as for `salloc` and `sbatch`.

### 4.2.3 Batch Jobs

Users submit batch applications using the `sbatch` command. The batch script is usually a shell script consisting of two parts: resources requests and job steps. Resources requests are, for example, the number of nodes needed to execute the job, the number of tasks, the time duration of the job etc. Job steps are user's tasks that must be executed. The resources requests and other SLURM submission options are prefixed by '`#SBATCH`' and must precede any executable commands in the batch script. For example:

```
#!/bin/bash
#SBATCH --partition=compute
#SBATCH --account=xz0123
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=24
#SBATCH --time=00:30:00

# Begin of section with executable commands
set -e
ls -l
srun ./my_program
```

The script itself is regarded by SLURM as the first job step and is (serially) executed on the first compute node in the job allocation. To execute parallel MPI tasks, users call `srun` within their script. Thereby, a new job step is initiated. It is possible to execute parallel programs in the form of job steps in any configuration within the job allocation. This means that a job step can use all allocated resources or several job steps (created via multiple `srun` calls) can use a subset of allocated resources.

The following table describes the most common or required allocation options that can be defined in a batch script:

| #SBATCH option | default value | description |
|---|---|---|
| `--nodes=<number>`<br>`-N <number>` | 1 | Number of nodes for the allocation |
| `--ntasks=<number>`<br>`-n <number>` | 1 | Number of tasks (MPI processes). Can be omitted if `--nodes` and `--ntasks-per-node` are given |
| `--ntasks-per-node=<number>` | 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=<number>`<br>`-c <number>` | 1 | Number of threads (logical cores) per task. Used mainly for OpenMP or hybrid jobs |
| `--output=<path>/<file pattern>`<br>`-o <path>/<file pattern>` | `slurm-%j.out` | Standard output file |
| `--error=<path>/<file pattern>`<br>`-e <path>/<file pattern>` | `slurm-%j.out` | Standard error file |
| `--time=<walltime>`<br>`-t <walltime>` | partition dep. | Requested walltime limit for the job |
| `--partition=<name>`<br>`-p <name>` | - | Partition to run the job |
| `--mail-user=<email>` | username | Email address for notifications |
| `--mail-type=<mode>` | - | Event types for email notifications. Possible values are NONE, BEGIN, END, FAIL, REQUEUE, ALL, TIMELIMIT |
| `--job-name=<jobname>`<br>`-J <jobname>` | job script's name | Job name |
| `--account=<project>`<br>`-A <project>` | none | Project that should be charged |
| `--requeue`<br>`--no-requeue` | no-requeue | Specifies whether the batch job should be requeued after a node failure. When a job is requeued, the batch script is initiated from its beginning! |

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 inherited from the `sbatch` arguments, but can be

overwritten for each `srun` invocation.

The complete list of parameters can be inquired from the `sbatch` man page:

```
bash$ man sbatch
```

As already mentioned above, the batch script is submitted using the SLURM `sbatch` command:

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

On success, `sbatch` writes the job ID to the standard output. Options provided on a command line supersede the same options defined in the batch script.

Remember 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`
3. Starting 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. Starting 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>
   ```

All environment variables set at the time of submission are propagated to the SLURM jobs. With some options of the allocation commands (like `--export` for sbatch or srun), 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 in job scripts, in order to have full control of the environment of the jobs.

**NOTE**: on the MISTRAL cluster setting either `-A` or `--account` is necessary to submit a job, otherwise the 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
```

Furthermore, you will have to specify the partition on which the job will run by using either the `-p` or `--partition` option to sbatch. Otherwise the submission will be rejected (note: we will enforce this starting at September 1st 2016 - please be prepared and modify your batch scripts accordingly).

## 4.3 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-per-cpu=1280      # Specify  real memory required per CPU
#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 --account=xz0123        # Charge resources on this
                                # project account
#SBATCH --output=my_job.o%j     # File name for standard output
#SBATCH --error=my_job.e%j      # File name for standard error output

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

**Note**: The shared partition has a limit of 1280MB memory per CPU. In case your serial job needs more memory, you have to increase the number of tasks (using option `--ntasks`) although you might not use all these CPUs. Alternatively, you can try to run your job in the partition prepost which has maximal 5120 MB memory per CPU.

### OpenMP job without HyperThreading

```
#!/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=16      # Specify number of CPUs per task
#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 --account=xz0123        # Charge resources on this
                                #project account
#SBATCH --output=my_job.o%j     # File name for standard output
#SBATCH --error=my_job.e%j      # File name for standard error output

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

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

**Note**: You need to specify the value of `--cpus-per-task` as a multiple of HyperThreads (HT). The environment variable `KMP_AFFINITY` needs to be set correspondingly. Whether HT is used or not is defined via the envVar `KMP_AFFINITY`, see 4.5.2 for details.

### OpenMP job with HyperThreading

```
#!/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 --mail-type=FAIL        # Notify user by email in case of
                                # job failure
#SBATCH --account=xz0123        # Charge resources on this
                                #project account
#SBATCH --output=my_job.o%j     # File name for standard output
#SBATCH --error=my_job.e%j      # File name for standard error output

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

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

### MPI job without HyperThreading

The overall setting of the batch script does not vary whether one is using Intel MPI or bullx MPI (or any other MPI implementation). Only specific modules might be used and/or environmental variables should be set in order to fine-tune the used MPI. Especially, the parallel application should always be started using the `srun` command instead of invoking mpirun, mpiexec or others.

In the following examples 288 cores are used to execute a parallel program. The examples differ in whether MISTRAL phase 1 nodes (i.e. using 12 nodes from the partition compute) or MISTRAL phase 2 nodes (only 8 nodes from the partition compute2) are used. Also the settings for programs built with BullxMPI resp. IntelMPI are distinguished.

- phase1 nodes, bullxMPI

```
#!/bin/bash
#SBATCH --job-name=my_job
#SBATCH --partition=compute
#SBATCH --nodes=12
#SBATCH --ntasks-per-node=24
#SBATCH --time=00:30:00
#SBATCH --mail-type=FAIL
#SBATCH --account=xz0123
#SBATCH --output=my_job.o%j
```

```
#SBATCH --error=my_job.e%j

# limit stacksize ... adjust to your programs need
ulimit -s 102400

# Environment settings to run a MPI parallel program
# compiled with BullxMPI and Mellanox libraries
# Load environment
module load intel/version_you_used
module load mxm/3.3.3002
module load fca/2.5.2393
module load bullxmpi_mlx/bullxmpi_mlx-1.2.8.3
# Settings for Open MPI and MXM (MellanoX Messaging)
# library
export OMPI_MCA_pml=cm
export OMPI_MCA_mtl=mxm
export OMPI_MCA_mtl_mxm_np=0
export MXM_RDMA_PORTS=mlx5_0:1
export MXM_LOG_LEVEL=ERROR
# Disable GHC algorithm for collective communication
export OMPI_MCA_coll=^ghc

# Use srun (not mpirun or mpiexec) command to launch
# programs compiled with any MPI library
srun -l --propagate=STACK --cpu_bind=cores \
 --distribution=block:cyclic ./myprog
```

- phase2 nodes, bullxMPI

```
#!/bin/bash
#SBATCH --job-name=my_job
#SBATCH --partition=compute2
#SBATCH --nodes=8
#SBATCH --ntasks-per-node=36
#SBATCH --time=00:30:00
#SBATCH --mail-type=FAIL
#SBATCH --account=xz0123
#SBATCH --output=my_job.o%j
#SBATCH --error=my_job.e%j

# limit stacksize ... adjust to your programs need
ulimit -s 102400

# Environment settings to run a MPI parallel program
# compiled with BullxMPI and Mellanox libraries
# Load environment
module load intel/version_you_used
module load mxm/3.3.3002
module load fca/2.5.2393
module load bullxmpi_mlx/bullxmpi_mlx-1.2.8.3
# Settings for Open MPI and MXM (MellanoX Messaging)
# library
export OMPI_MCA_pml=cm
export OMPI_MCA_mtl=mxm
```

```
export OMPI_MCA_mtl_mxm_np=0
export MXM_RDMA_PORTS=mlx5_0:1
export MXM_LOG_LEVEL=ERROR
# Disable GHC algorithm for collective communication
export OMPI_MCA_coll=^ghc

# Use srun (not mpirun or mpiexec) command to launch
# programs compiled with any MPI library
srun -l --propagate=STACK --cpu_bind=cores \
  --distribution=block:cyclic ./myprog
```

- phase1 nodes, IntelMPI

```
#!/bin/bash
#SBATCH --job-name=my_job
#SBATCH --partition=compute
#SBATCH --nodes=12
#SBATCH --ntasks-per-node=24
#SBATCH --time=00:30:00
#SBATCH --mail-type=FAIL
#SBATCH --account=xz0123
#SBATCH --output=my_job.o%j
#SBATCH --error=my_job.e%j

# limit stacksize ... adjust to your programs need
ulimit -s 102400

# Environment settings to run a MPI parallel program
# compiled with Intel MPI
module load intel/version_you_used
module load intelmpi/version_you_used
export I_MPI_PMI_LIBRARY=/usr/lib64/libpmi.so

# Use srun (not mpirun or mpiexec) command to launch
# programs compiled with any MPI library
srun -l --propagate=STACK --cpu_bind=cores \
  --distribution=block:cyclic ./myprog
```

- phase2 nodes, IntelMPI

```
#!/bin/bash
#SBATCH --job-name=my_job
#SBATCH --partition=compute2
#SBATCH --nodes=8
#SBATCH --ntasks-per-node=36
#SBATCH --time=00:30:00
#SBATCH --mail-type=FAIL
#SBATCH --account=xz0123
#SBATCH --output=my_job.o%j
#SBATCH --error=my_job.e%j

# limit stacksize ... adjust to your programs need
ulimit -s 102400
```

```
# Environment settings to run a MPI parallel program
# compiled with Intel MPI
module load intel/version_you_used
module load intelmpi/version_you_used
export I_MPI_PMI_LIBRARY=/usr/lib64/libpmi.so

# Use srun (not mpirun or mpiexec) command to launch
# programs compiled with any MPI library
srun -l --propagate=STACK --cpu_bind=cores \
  --distribution=block:cyclic ./myprog
```

### MPI job with HyperThreading

The following examples all ask for 144 MPI-tasks. When using Hyper-Threading, two tasks can use one physical CPU leading to a reduced number of nodes needed for a job - at the expense of a possibly slower runtime. Again, the examples differ in the used partition and MPI implementation.

- phase1 nodes, bullxMPI

```
#!/bin/bash
#SBATCH --job-name=my_job
#SBATCH --partition=compute
#SBATCH --nodes=3
#SBATCH --ntasks-per-node=48
#SBATCH --time=00:30:00
#SBATCH --mail-type=FAIL
#SBATCH --account=xz0123
#SBATCH --output=my_job.o%j
#SBATCH --error=my_job.e%j

# limit stacksize ... adjust to your programs need
ulimit -s 102400

# Environment settings to run a MPI parallel program
# compiled with BullxMPI and Mellanox libraries
# Load environment
module load intel/version_you_used
module load mxm/3.3.3002
module load fca/2.5.2393
module load bullxmpi_mlx/bullxmpi_mlx-1.2.8.3
# Settings for Open MPI and MXM (MellanoX Messaging)
# library
export OMPI_MCA_pml=cm
export OMPI_MCA_mtl=mxm
export OMPI_MCA_mtl_mxm_np=0
export MXM_RDMA_PORTS=mlx5_0:1
export MXM_LOG_LEVEL=ERROR
# Disable GHC algorithm for collective communication
export OMPI_MCA_coll=^ghc

# Use srun (not mpirun or mpiexec) command to launch
# programs compiled with any MPI library
srun -l --propagate=STACK --cpu_bind=threads \
```

```
  --distribution=block:cyclic ./myprog
```

- phase2 nodes, bullxMPI

```
#!/bin/bash
#SBATCH --job-name=my_job
#SBATCH --partition=compute2
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=72
#SBATCH --time=00:30:00
#SBATCH --mail-type=FAIL
#SBATCH --account=xz0123
#SBATCH --output=my_job.o%j
#SBATCH --error=my_job.e%j

# limit stacksize ... adjust to your programs need
ulimit -s 102400

# Environment settings to run a MPI parallel program
# compiled with BullxMPI and Mellanox libraries
# Load environment
module load intel/version_you_used
module load mxm/3.3.3002
module load fca/2.5.2393
module load bullxmpi_mlx/bullxmpi_mlx-1.2.8.3
# Settings for Open MPI and MXM (MellanoX Messaging)
# library
export OMPI_MCA_pml=cm
export OMPI_MCA_mtl=mxm
export OMPI_MCA_mtl_mxm_np=0
export MXM_RDMA_PORTS=mlx5_0:1
export MXM_LOG_LEVEL=ERROR
# Disable GHC algorithm for collective communication
export OMPI_MCA_coll=^ghc

# Use srun (not mpirun or mpiexec) command to launch
# programs compiled with any MPI library
srun -l --propagate=STACK --cpu_bind=threads \
  --distribution=block:cyclic ./myprog
```

- phase1 nodes, IntelMPI

```
#!/bin/bash
#SBATCH --job-name=my_job
#SBATCH --partition=compute
#SBATCH --nodes=3
#SBATCH --ntasks-per-node=48
#SBATCH --time=00:30:00
#SBATCH --mail-type=FAIL
#SBATCH --account=xz0123
#SBATCH --output=my_job.o%j
#SBATCH --error=my_job.e%j

# limit stacksize ... adjust to your programs need
```

```
ulimit -s 102400

# Environment settings to run a MPI parallel program
# compiled with Intel MPI
module load intel/version_you_used
module load intelmpi/version_you_used
export I_MPI_PMI_LIBRARY=/usr/lib64/libpmi.so

# Use srun (not mpirun or mpiexec) command to launch
# programs compiled with any MPI library
srun -l --propagate=STACK --cpu_bind=threads \
  --distribution=block:cyclic ./myprog
```

- phase2 nodes, IntelMPI

```
#!/bin/bash
#SBATCH --job-name=my_job
#SBATCH --partition=compute2
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=72
#SBATCH --time=00:30:00
#SBATCH --mail-type=FAIL
#SBATCH --account=xz0123
#SBATCH --output=my_job.o%j
#SBATCH --error=my_job.e%j

# limit stacksize ... adjust to your programs need
ulimit -s 102400

# Environment settings to run a MPI parallel program
# compiled with Intel MPI
module load intel/version_you_used
module load intelmpi/version_you_used
export I_MPI_PMI_LIBRARY=/usr/lib64/libpmi.so

# Use srun (not mpirun or mpiexec) command to launch
# programs compiled with any MPI library
srun -l --propagate=STACK --cpu_bind=threads \
  --distribution=block:cyclic ./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 Haswell node and uses 24 tasks without Hyper-Threading for the first program run and then 48 tasks using Hyper-Threading for the second run. Such a procedure might be used in order to see whether an application benefits from 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 --mail-type=FAIL      # Notify user by email
#SBATCH --account=xz0123      # Charge resources on this
```

```
                                    # project account

# Environment settings to execute a parallel program compiled
# with Intel MPI
module load intelmpi
export I_MPI_PMI_LIBRARY=/usr/lib64/libpmi.so

# First check how myprog performs without Hyper-Threads
srun -l --cpu_bind=verbose --hint=nomultithread --ntasks=24 ./myprog

# Second check how myprog performs with Hyper-Threads
srun -l --cpu_bind=verbose --hint=multithread --ntasks=48 ./myprog
```

### Hybrid MPI/OpenMP job without Hyper-Threading

The following job example will allocate 4 Haswell compute nodes from the compute partition for 1 hour. The job will launch 24 MPI tasks in total, 6 tasks per node and 4 OpenMP threads per task. On each node 24 cores will be used. These settings have to be adapted to 36 physical CPUs if the compute2 partition is to be used. Furthermore, one has to slightly change the loaded modules and environmental variables set when IntelMPI should be used.

```
#!/bin/bash
#SBATCH --job-name=my_job          # job name
#SBATCH --partition=compute        # partition name
#SBATCH --nodes=4                  # number of nodes
#SBATCH --ntasks-per-node=6        # number of (MPI) tasks per node
#SBATCH --time=01:00:00            # Set a limit on the total run time
#SBATCH --mail-type=FAIL           # Notify user by email
#SBATCH --account=xz0123           # Charge resources on project account
#SBATCH --output=my_job.o%j        # File name for standard output
#SBATCH --error=my_job.e%j         # File name for standard error output

# Bind your OpenMP threads
export OMP_NUM_THREADS=4
export KMP_AFFINITY=verbose,granularity=core,compact,1
export KMP_STACKSIZE=64m

# Environment settings to run a MPI/OpenMP parallel program compiled
# with Bullx MPI and Mellanox libraries, load environment
module load intel
module load mxm/3.3.3002
module load fca/2.5.2393
module load bullxmpi_mlx/bullxmpi_mlx-1.2.8.3

# Settings for Open MPI and MXM (MellanoX Messaging) library
export OMPI_MCA_pml=cm
export OMPI_MCA_mtl=mxm
export OMPI_MCA_mtl_mxm_np=0
export MXM_RDMA_PORTS=mlx5_0:1
export MXM_LOG_LEVEL=ERROR

# Disable GHC algorithm for collective communication
```

```
export OMPI_MCA_coll=^ghc

# limit stacksize ... adjust to your programs need
ulimit -s 102400

# Environment settings to run a MPI/OpenMP parallel program compiled
# with Intel MPI, load environment
module load intelmpi
export I_MPI_PMI_LIBRARY=/usr/lib64/libpmi.so

# Use srun (not mpirun or mpiexec) command to launch programs compiled
# with any MPI library
srun -l --propagate=STACK --cpu_bind=cores --cpus-per-task=8 ./myprog
```

### Hybrid MPI/OpenMP job with Hyper-Threading

The following example will run on 2 compute nodes while having 6 MPI tasks per node and starting 8 threads per node using Hyper-Threading.

```
#!/bin/bash
#SBATCH --job-name=my_job        # job name
#SBATCH --partition=compute      # partition name
#SBATCH --nodes=2                # number of nodes
#SBATCH --ntasks-per-node=6      # number of (MPI) tasks on each node
#SBATCH --time=01:00:00          # Set a limit on the total run time
#SBATCH --mail-type=FAIL         # Notify user by email
#SBATCH --account=xz0123         # Charge resources on project account
#SBATCH --output=my_job.o%j      # File name for standard output
#SBATCH --error=my_job.e%j       # File name for standard error output

# Bind your OpenMP threads
export OMP_NUM_THREADS=8
export KMP_AFFINITY=verbose,granularity=thread,compact,1
export KMP_STACKSIZE=64m

# Environment settings to run a MPI/OpenMP parallel program compiled
# with Bullx MPI and Mellanox libraries, load environment
module load intel
module load mxm/3.3.3002
module load fca/2.5.2393
module load bullxmpi_mlx/bullxmpi_mlx-1.2.8.3

# Settings for Open MPI and MXM (MellanoX Messaging) library
export OMPI_MCA_pml=cm
export OMPI_MCA_mtl=mxm
export OMPI_MCA_mtl_mxm_np=0
export MXM_RDMA_PORTS=mlx5_0:1
export MXM_LOG_LEVEL=ERROR

# Disable GHC algorithm for collective communication
export OMPI_MCA_coll=^ghc

# limit stacksize ... adjust to your programs need
```

```
ulimit -s 102400

# Environment settings to run a MPI/OpenMP parallel program compiled
# with Intel MPI, load environment
module load intelmpi
export I_MPI_PMI_LIBRARY=/usr/lib64/libpmi.so

# Use srun (not mpirun or mpiexec) command to launch programs
# compiled with any MPI library
srun -l --propagate=STACK --cpu_bind=cores --cpus-per-task=8 ./myprog
```

## 4.4 Adapting job-scripts for MISTRAL phase2

Since phase1 and phase2 nodes of MISTRAL are equipped with different Intel CPUs, you will have to slightly adapt your existing job scripts in oder to use both partitions. The following table gives an overview on the differences and which partitions are affected.

| phase | partitions | CPU | cores per node | processor frequency |
|---|---|---|---|---|
| 1 | compute, prepost, shared,gpu, miklip | Xeon E5-2680 v3 processor (Haswell - HSW) | 24 | 2.5 GHz |
| 2 | compute2 | Xeon E5-2695 v4 processor (Broadwell - BDW) | 36 | 2.1 GHz |

Table 4.2: Difference of MISTRAL phase1 and phase2 nodes

As the table indicates just two issues arise if batch scripts should be useable for both phases:

- different number of cores per node
- different processor frequency

### Setting the right CPU frequency for each partition

SLURM allows to request that the job step initiated by the srun command shall be run at the requested frequency (if possible) on the CPUs selected for that step on the compute node(s). This can be done via

- srun option `--cpu-freq`
- environmental variable `SLURM_CPU_FREQ_REQ`

If none of these options is set, DKRZ slurm automatically chooses the appropriate frequency for the underlying processor. We therefore recommend to not set the frequency explicitly.

In case that a wrong frequency is defined via envVar (e.g. setting `SLURM_CPU_FREQ_REQ=2500000` for the BDW nodes in compute2 partition) a warning message on stdout is given like

```
[DKRZ−slurm WARNING] CPU−frequency chosen (2500000) not supported on partition
                     compute2 − frequency will be set to nominal instead!
```

If you are using a wrong frequency for the srun option `--cpu-freq`, a warning message on stdout is given, but this time the automatic frequency adaption falls back to the minimal frequency:

```
[DKRZ−slurm WARNING] CPU−frequency chosen (2501000) not supported on partition
                     compute2 − frequency will fall back to minimum instead!
```

## Setting the right number of cores

When allocating nodes using the sbatch or salloc command one has to specify the targeted partition and therefore the type of CPU directly. Nevertheless, jobscripts that were originally written to run on the 24 core Intel Haswell nodes (i.e. in the compute partition) will in general also run in the compute2 partition but do not make use of the full node.

We attempt to catch these cases and issue an info message on stdout like

```
[DKRZ−slurm INFO] it seems that your job is not using all CPUs on BDW nodes
[DKRZ−slurm INFO] tasks_per_node 24, cpus_per_tasks 2
```

The most critical sbatch/srun option in this context is `--ntasks-per-node`. Setting e.g. a value of 24 is appropriate for Haswell nodes but uses only 2/3 of the CPUs on Broadwell nodes. Hence, you should pay major attention to this when adapting your batch scripts for the compute2 partition.

Writing more flexible batch scripts, that are able to run on both kinds of CPU, requires avoidance of sbatch options that prescribe the number of tasks per entity, i.e. you should not use a prescribed number of nodes and one of

- `--ntasks-per-core=<ntasks>`
- `--ntasks-per-socket=<ntasks>`
- `--ntasks-per-node=<ntasks>`
- `--tasks-per-node=<n>`

Instead define the total number of tasks that your MPI parallel program will be using for srun by specifying

```
−n <number> or −−ntasks=<number>
```

in combination with the number of CPUs needed per task, e.g. `--cpus-per-task=2` for a pure MPI parallel program not using HyperThreading.

The following batch script will run an MPI job without HyperThreading either in the compute or compute2 partition - only depending on the `#SBATCH --partition` choice.

```
#!/bin/bash
#SBATCH −−job−name=my_job
#SBATCH −−partition=compute # or compute2 for BDW nodes
#SBATCH −−ntasks=72
#SBATCH −−cpus−per−task=2
#SBATCH −−time=00:30:00
#SBATCH −−mail−type=FAIL
#SBATCH −−account=xz0123
```

```
#SBATCH --output=my_job.o%j
#SBATCH --error=my_job.e%j

srun -l --propagate=STACK --cpu_bind=cores ./myprog
```

When submitted to the compute partition, the job will run on 3 nodes with 24 tasks per node. While in the compute2 partition the same job only takes 2 nodes with 36 tasks per node.

### Writing job scripts eligible to run on several partitions

The `--partition` option also allows for a comma separate list of names. In this case the job will run completely on the partition offering earliest initiation with no regard given to the partition name ordering - i.e. nodes will not be mixed between the partitions! Be aware that the total number of tasks should be a multiple of both, 24 and 36, in order to fully populate all nodes. Otherwise, some nodes might be underpopulated. The example above might therefore be modified to use

```
#!/bin/bash
#SBATCH --job-name=my_job
#SBATCH --partition=compute, compute2
#SBATCH --ntasks=72
...
```

which in general will decrease the waiting time of the job in the submit queue since more nodes are suitable to schedule the job on. **Attention**: compute2 partition (Broadwell nodes) shows a slightly lesser performance due to the lower CPU-frequency compared to compute partition (Haswell nodes). You should take this into account when submitting jobs that are eligible to run on both partitions.

## 4.5 Advanced SLURM Features

### 4.5.1 Hyper-Threading (HT)

Similar to the IBM Power6 used in BLIZZARD, the Haswell and Broadwell processors deployed for MISTRAL 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. We visualize this in the following for the Haswell nodes only - the equivalent for Broadwell nodes is obvious.

Each node on MISTRAL phase1 partition 'compute' consists of two Intel Xeon E5-2680 v3 processors, located on socket zero and one. The first 24 processing units are physical cores labelled from 0 to 23. The second 24 processing units are Hyper threads labelled from 24 to 47. Figure 4.1 depicts a node schematically and illustrates the naming convention.

On MISTRAL, we have HT enabled on each compute node and SLURM always uses the option `--threads-per-core=2` implicitly so that the user is urged to bind the tasks/threads in an appropriate way. In Section 4.3, there are examples (commands and job scripts) on how to use HT or not.

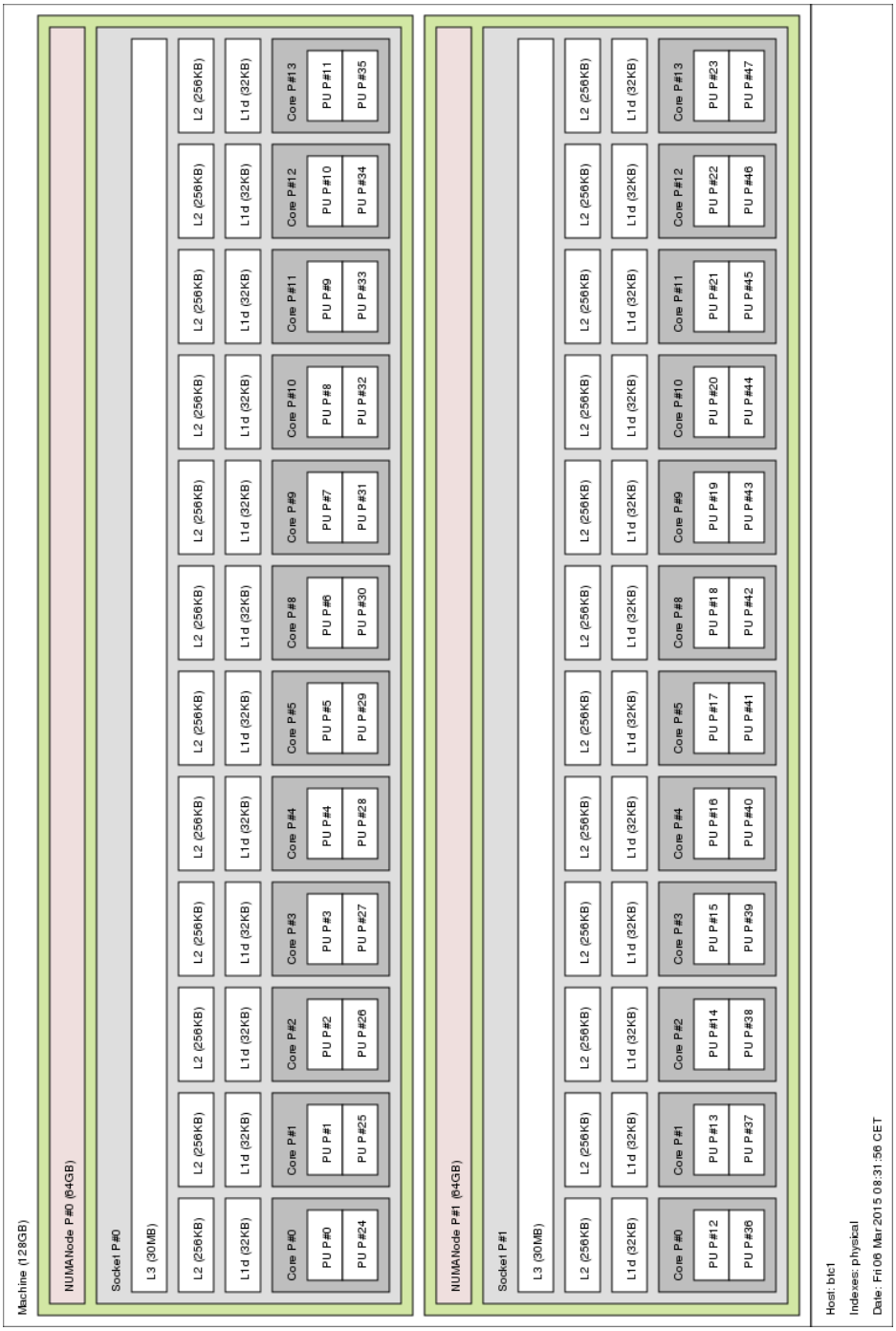


Figure 4.1: Schematic illustration of Haswell compute nodes

### 4.5.2 Process and Thread Binding

#### 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: gives detailed output on how binding was done.

  - granularity=core: reserves the full physical cores (i.e. two logical CPUs) to run threads on.
  - granularity=thread/fine: reserves 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 user's consultancy. In most cases, use

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

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

```
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.

### 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: binds to physical cores
  - threads: binds to logical CPUs / HyperThreads
- the first distribution method (before the ":") controls the distribution of resources across nodes
- the 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 user's consultancy. In most cases, use

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

if you do not want to use HyperThreads and use

```
bash$ 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.

### 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 Haswell node in the compute partition

| | **MPI intranode distribution of tasks =** | |
|---|---|---|
| | **srun –distribution=block:block** | **srun –distribution=block:cyclic** |
| **no OpenMP, no HT** | `#SBATCH --tasks-per-node=24`<br>`srun --cpu_bind=cores`<br><br>task0:cpu{0,24}, task1:cpu{1,25}, ... | `#SBATCH --tasks-per-node=24`<br>`srun --cpu_bind=cores`<br><br>task0:cpu{0,24}, task1:cpu{12,36}, ... |
| **no OpenMP, HT** | `#SBATCH --tasks-per-node=48`<br>`srun --cpu_bind=threads`<br><br>task0:cpu0, task1:cpu24, task2:cpu1, ... | `#SBATCH --tasks-per-node=48`<br>`srun --cpu_bind=threads`<br><br>task0:cpu0, task1:cpu12, task2:cpu1, ... |
| **4 OpenMP threads, no HT** | `#SBATCH --tasks-per-node=6`<br>`export OMP_NUM_THREADS=4`<br>`export KMP_AFFINITY=\`<br>`granularity=core,\`<br>`compact,1`<br>`srun --cpu_bind=cores \`<br>`--cpus-per-task=8`<br><br>task0:cpu{0,1,2,3,24,25,26,27},<br>task1:cpu{4,5,6,7,28,29,30,31},<br>...<br>task0-thread0:cpu{0,24},<br>task0-thread1:cpu{1,25},... | `#SBATCH --tasks-per-node=6`<br>`export OMP_NUM_THREADS=4`<br>`export KMP_AFFINITY=\`<br>`granularity=core,\`<br>`compact,1`<br>`srun --cpu_bind=cores \`<br>`--cpus-per-task=8`<br><br>task0:cpu{0,1,2,3,24,25,26,27},<br>task1:cpu{12,13,14,15,36,37,38,39},<br>...<br>task0-thread0:cpu{0,24},<br>task0-thread1:cpu{1,25},... |
| **4 OpenMP threads, HT** | `#SBATCH --tasks-per-node=12`<br>`export OMP_NUM_THREADS=4`<br>`export KMP_AFFINITY=\`<br>`granularity=tread,\`<br>`compact,1`<br>`srun --cpu_bind=threads \`<br>`--cpus-per-task=4`<br><br>task0:cpu{0,1,24,25},<br>task1:cpu{2,3,26,27},<br>...<br>task0-thread0:cpu0,<br>task0-thread1:cpu1,<br>task0-thread2:cpu24,... | `#SBATCH --tasks-per-node=12`<br>`export OMP_NUM_THREADS=4`<br>`export KMP_AFFINITY=\`<br>`granularity=thread,\`<br>`compact,1`<br>`srun --cpu_bind=threads \`<br>`--cpus-per-task=4`<br><br>task0:cpu{0,1,24,25},<br>task1:cpu{12,13,36,37},<br>...<br>task0-thread0:cpu0,<br>task0-thread1:cpu1,<br>task0-thread2:cpu24,... |

Table 4.3: SLURM binding options for MPI/OpenMP jobs on Haswell nodes

### 4.5.3 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) that 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. The 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 template for the coupled MPI-ESM model using 8 Haswell nodes in the compute partition

```
#!/bin/bash

#SBATCH --nodes=8
#SBATCH --ntasks-per-node=24
#SBATCH --partition=compute
#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
export I_MPI_FABRICS=shm:dapl
export I_MPI_FALLBACK=0
export I_MPI_DAPL_UD=enable

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

### 4.5.4 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,implying 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 if a multiple nodes are allocated. A job using job steps will be accounted for all the nodes of the allocation regardless of the fact 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 --partition=compute
#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 &
```

### 4.5.5 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>** the job starts when another job with <jobID> begun execution.
- **afterany:<jobID>** the job starts when another job with <jobID> terminates.
- **afterok:<jobID>** the job starts when another job with <jobID> terminates successfully.
- **afternotok:<jobID>** the job starts when another job with <jobID> terminates with failure.
- **singleton** the job starts when any previous job with the same job name and user terminates.

### 4.5.6 Job Arrays

SLURM supports job arrays, which are mechanisms 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. the number of nodes, the time limit, etc.), however, an individual setting for each job is possible since each part of a job array has access to the environment variable `SLURM_ARRAY_TASK_ID`. 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 with 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 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 the different input files.

```
#!/bin/bash

#SBATCH --nodes=1
#SBATCH --partition=compute
```

```
#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
```

## 4.6 SLURM Command Examples

### 4.6.1 Query Commands

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

CA CANCELLED : The job was explicitly cancelled by the user or an administrator. The job may or may not have been initiated.

CD COMPLETED : The job has terminated all processes on all nodes.

CF CONFIGURING : The job has allocated its required resources, but is waiting for them to become ready for use.

CG COMPLETING : The job is in the process of completing. Some processes on some nodes may still be active.

F FAILED : The job terminated with a non-zero exit code or other failure conditions.

NF NODE_FAIL : The job terminated due to the failure of one or more allocated nodes.

PD PENDING : The job is awaiting for the resource allocation.

R RUNNING : The job currently has an allocation.

TO TIMEOUT : The job terminated upon reaching its walltime limit.

Some examples of how users can query their jobs status are as follows:

- List all jobs submitted to SLURM

```
bash$ squeue
```

- List all jobs submitted by you

```
bash$ squeue -u $USER
```

- Check available partitions and nodes

```
bash$ sinfo
```

Depending on the options, the `sinfo` 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 can also be in various states. Node state codes may be shortened according to the size of the printed field. The following shows the most common node states:

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 the DRAINING state, any running job on the node will be allowed to run until completion. After that it turn into the DRAINED state, such that 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.

- Query the configuration and limits for one specific partition (here compute)

```
bash$ scontrol show partition compute
```

- Check one node (here m10010):

```
bash$ scontrol show node m10010
```

### 4.6.2 Job Control

The `scontrol` command is primarily used by the administrators to manage SLURM's configuration. However, it provides also some functionality for the users to manage jobs and get some information about the system configuration.

- Show information about the job 4242

```
bash$ scontrol show job 4242
```

- Cancel the job with SLURM JobId 4711

```
bash$ scancel 4711
```

- Cancel all your jobs

```
bash$ scancel -u $USER
```

- Display status information of running job 4242

```
bash$ sstat -j 4242
```

`sstat` provides various status information (e.g. CPU time, Virtual Memory (VM) usage, Resident Set Size (RSS), Disk I/O etc.) for running jobs. The metrics of interest can be specified using option `--format` or `-o` (s. next example).

- Display the selected status information of running job 4242

```
bash$ sstat -o JobID,AveCPU,AvePages,MaxRSS,MaxVMsize -j 4242
# For a list of all available metrics use the option --helpformat or look into sstat man page
bash$ sstat --helpformat
bash$ man sstat
```

- Hold the pending job with SLURM JobId 5711

```
bash$ scontrol hold 5711
bash$ squeue
 JOBID PARTITION     NAME     USER ST  TIME  NODES NODELIST(REASON)
  5711   compute tst_job b123456 PD  0:00      1 (JobHeldUser)
```

- Release the job with SLURM JobId 5711

```
bash$ scontrol release 5711
```

### 4.6.3 Accounting Commands

With `sacct`, one can get the 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.

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

```
bash$ sacct -l
```

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

```
bash$ sacct -S 2015-01-07T00:42:00 -X
```

Show job information with a 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
```

The `sacctmgr` command is mainly used by the administrators to view or modify the accounting information and data in the accounting database. This command provides also an interface with limited permissions to the users for some querying actions. The most useful command is to show all associations a user is allowed to submit jobs:

```
bash$ sacctmgr show assoc where user=<user_id>
```

List all or the specified QoS:

```
bash$ sacctmgr show qos [where name=<qos_name>]
```

# Chapter 5

# Data Processing

A part of the Mistral cluster is reserved for data processing and analysis and can be deployed for tasks like

- time and memory intensive data processing using CDO, NCO, netCDF, afterburner, tar, gzip/bzip, etc.
- data analysis and simple visualization using MATLAB, Mathematica, R, Python, Scilab, NCL, GrADS, FERRET, IDL, GMT, etc.
- archiving and downloading data to/from HPSS tape archive via pftp
- connecting to external servers via sftp, lftp, scp, globus toolkit
- downloading data from CERA/WDCC data base using jblob

and so on.

Only the advanced visualization applications like Avizo Green, Avizo Earth, Paraview, Vapor etc. need to run on Mistral nodes dedicated for 3D visualization, as described in the section Visualization on Mistral (see https://www.dkrz.de/Nutzerportal-en/doku/vis/visualization-on-mistral).

Below, different procedures on how to access hardware resources provided for data processing and analysis are described. In general, the following three ways are possible:

- Use interactive nodes mistralpp.dkrz.de.
- Start an interactive session on a node in the SLURM partition prepost.
- Submit a batch job to the SLURM partition prepost or shared.

## Interactive nodes mistralpp

Five nodes are currently available for interactive data processing and analysis. The nodes can directly be accessed via ssh:

```
bash$ ssh -X <userid>@mistralpp.dkrz.de
```

On the interactive nodes, resources (memory and CPU) are shared among all users logged into the node. This might negatively influence the node performance and extend the run time of applications.

## Interactive use of nodes managed by SLURM

To avoid oversubscribing nodes on mistralpp and obtain dedicated resources for your interactive work, you can make a resource allocation using the SLURM salloc command and log into the allocated node via ssh. The example below illustrates this approach. The name of the allocated node is set by SLURM in the environment variable `SLURM_JOB_NODELIST`.

```
bash$ salloc -p prepost -A xz0123 -n 1 -t 60 -- /bin/bash -c 'ssh -X \
      $SLURM_JOB_NODELIST'
```

Please take care to adapt the settings in the example above (project account (option -A), number of tasks (option -n), wall-clock time (option -t) etc.) to your actual needs.

For hints on how to set the default SLURM account and define a shell alias or function to allocate resources and log into a node in one step, please refer to our Mistral Tips and Tricks website.

## Submitting a batch job

In case your data processing programs do not require an interactive control, you can also submit a regular batch job. Below, there is a batch script example for a job that will use one core on one node in the partition prepost for twenty minutes. Insert your own job name, project account, file names for the standard output and error output, resources requirements, and the program to be executed.

```
#!/bin/bash
#SBATCH -J my_job          # Specify job name
#SBATCH -p prepost         # Use partition prepost
#SBATCH -N 1               # Specify number of nodes
#SBATCH -n 1               # Specify max. number of tasks to be invoked
#SBATCH -t 20              # Set a limit on the total run time
#SBATCH -A xz0123          # Charge resources on this project account
#SBATCH -o my_job.o%j      # File name for standard output
#SBATCH -e my_job.e%j      # File name for standard error output

# Execute a serial program, e.g.
ncl my_script.ncl
```