

Table of Contents

Information about the batch system (SLURM)	2
Overview	2
Remark about environment	2
An introductory example	2
From a shell on a node	2
Running directly from the front ends	2
Batch script	3
Heterogeneous jobs	4
Heterogeneous jobs with MPI communication across modules	5
Available Partitions	5
Information on past jobs and accounting	5
FAQ	5
Is there a cheat sheet for all main Slurm commands?	5
Why's my job not running?	5
How can I check which jobs are running in the machine?	6
How do I do chain jobs with dependencies?	6
How can check the status of partitions and nodes?	6
Can I join stderr and stdout like it was done with -joe in Torque?	6
What is the default binding/pinning behaviour on DEEP?	7
How do I use SMT on the DEEP CPUs?	8

Information about the batch system (SLURM)

Please confer `/etc/slurm/README`.

The documentation of Slurm can be found [?here](#).

Overview

Slurm offers interactive and batch jobs (scripts submitted into the system). The relevant commands are `srun` and `sbatch`. The `srun` command can be used to spawn processes (**please do not use `mpiexec`**), both from the frontend and from within a batch script. You can also get a shell on a node to work locally there (e.g. to compile your application natively for a special platform).

Remark about environment

By default, Slurm passes the environment from your job submission session directly to the execution environment. Please be aware of this when running jobs with `srun` or when submitting scripts with `sbatch`. This behavior can be controlled via the `--export` option. Please refer to the [?Slurm documentation](#) to get more information about this.

In particular, when submitting job scripts, it is recommended to load the necessary modules within the script and submit the script from a clean environment.

An introductory example

Suppose you have an mpi executable named `hello_mpi`. There are three ways to start the binary.

From a shell on a node

First, start a shell on a node. You would like to run your mpi task on 4 machines with 2 tasks per machine:

```
niessen@deepl:src/mpi > srun --partition=sdv -N 4 -n 8 --pty /bin/bash -i
niessen@deeper-sdv04:/direct/homec/zdvex/niessen/src/mpi >
```

The environment is transported to the remote shell, no `.profile`, `.bashrc`, ... are sourced (especially not the modules default from `/etc/profile.d/modules.sh`).

Once you get to the compute node, start your application using `srun`. Note that the number of tasks used is the same as specified in the initial `srun` command above (4 nodes with two tasks each):

```
niessen@deeper-sdv04:/direct/homec/zdvex/niessen/src/mpi > srun ./hello_cluster
srun: cluster configuration lacks support for cpu binding
Hello world from process 6 of 8 on deeper-sdv07
Hello world from process 7 of 8 on deeper-sdv07
Hello world from process 3 of 8 on deeper-sdv05
Hello world from process 4 of 8 on deeper-sdv06
Hello world from process 0 of 8 on deeper-sdv04
Hello world from process 2 of 8 on deeper-sdv05
Hello world from process 5 of 8 on deeper-sdv06
Hello world from process 1 of 8 on deeper-sdv04
```

You can ignore the warning about the cpu binding. ParaStation will pin you processes.

Running directly from the front ends

You can run the application directly from the frontend, bypassing the shell:

```
niessen@deepl:src/mpi > srun --partition=sdv -N 4 -n 8 ./hello_cluster
Hello world from process 4 of 8 on deeper-sdv06
Hello world from process 6 of 8 on deeper-sdv07
Hello world from process 3 of 8 on deeper-sdv05
Hello world from process 0 of 8 on deeper-sdv04
```

```
Hello world from process 2 of 8 on deeper-sdv05
Hello world from process 5 of 8 on deeper-sdv06
Hello world from process 7 of 8 on deeper-sdv07
Hello world from process 1 of 8 on deeper-sdv04
```

In this case, it can be useful to create an allocation which you can use for several runs of your job:

```
niessen@deep1:src/mpi > salloc --partition=sdv -N 4 -n 8
salloc: Granted job allocation 955
niessen@deep1:~/src/mpi>srunch ./hello_cluster
Hello world from process 3 of 8 on deeper-sdv05
Hello world from process 1 of 8 on deeper-sdv04
Hello world from process 7 of 8 on deeper-sdv07
Hello world from process 5 of 8 on deeper-sdv06
Hello world from process 2 of 8 on deeper-sdv05
Hello world from process 0 of 8 on deeper-sdv04
Hello world from process 6 of 8 on deeper-sdv07
Hello world from process 4 of 8 on deeper-sdv06
niessen@deep1:~/src/mpi> # several more runs
...
niessen@deep1:~/src/mpi>exit
exit
salloc: Relinquishing job allocation 955
```

Batch script

Given the following script `hello_cluster.sh`: (it has to be executable):

```
#!/bin/bash

#SBATCH --partition=sdv
#SBATCH -N 4
#SBATCH -n 8
#SBATCH -o /homec/zdvex/niessen/src/mpi/hello_cluster-%j.log
#SBATCH -e /homec/zdvex/niessen/src/mpi/hello_cluster-%j.err
#SBATCH --time=00:10:00

srunch ./hello_cluster
```

This script requests 4 nodes with 8 tasks, specifies the stdout and stderr files, and asks for 10 minutes of walltime. Submit:

```
niessen@deep1:src/mpi > sbatch ./hello_cluster.sh
Submitted batch job 956
```

Check what it's doing:

```
niessen@deep1:src/mpi > squeue
```

JOBID	PARTITION	NAME	USER	ST	TIME	NODES	NODELIST(REASON)
956	sdv	hello_cl	niessen	R	0:00	4	deeper-sdv[04-07]

Check the result:

```
niessen@deep1:src/mpi > cat hello_cluster-956.log
Hello world from process 5 of 8 on deeper-sdv06
Hello world from process 1 of 8 on deeper-sdv04
Hello world from process 7 of 8 on deeper-sdv07
Hello world from process 3 of 8 on deeper-sdv05
Hello world from process 0 of 8 on deeper-sdv04
Hello world from process 2 of 8 on deeper-sdv05
```

```
Hello world from process 4 of 8 on deeper-sdv06
Hello world from process 6 of 8 on deeper-sdv07
```

Heterogeneous jobs

As of version 17.11 of Slurm, heterogeneous jobs are supported. For example, the user can run:

```
srun --partition=dp-cn -N 1 -n 1 hostname : --partition=dp-dam -N 1 -n 1 hostname
dp-cn01
dp-dam01
```

Please notice the `:` separating the definitions for each sub-job of the heterogeneous job. Also, please be aware that it is possible to have more than two sub-jobs in a heterogeneous job.

The user can also request several sets of nodes in a heterogeneous allocation using `salloc`. For example:

```
salloc --partiton=dp-cn -N 2 : --partition=dp-dam -N 4
```

In order to submit a heterogeneous job via `sbatch`, the user needs to set the batch script similar to the following one:

```
#!/bin/bash

#SBATCH --job-name=imb_execute_1
#SBATCH --account=deep
#SBATCH --mail-user=
#SBATCH --mail-type=ALL
#SBATCH --output=job.out
#SBATCH --error=job.err
#SBATCH --time=00:02:00

#SBATCH --partition=dp-cn
#SBATCH --nodes=1
#SBATCH --ntasks=12
#SBATCH --ntasks-per-node=12
#SBATCH --cpus-per-task=1

#SBATCH packjob

#SBATCH --partition=dp-dam
#SBATCH --constraint=
#SBATCH --nodes=1
#SBATCH --ntasks=12
#SBATCH --ntasks-per-node=12
#SBATCH --cpus-per-task=1

srun ./app_cn : ./app_dam
```

Here the `packjob` keyword allows to define Slurm parameter for each sub-job of the heterogeneous job. Some Slurm options can be defined once at the beginning of the script and are automatically propagated to all sub-jobs of the heterogeneous job, while some others (i.e. `--nodes` or `--ntasks`) must be defined for each sub-job. You can find a list of the propagated options on the [?Slurm documentation](#).

When submitting a heterogeneous job with this colon notation using ParaStationMPI, a unique `MPI_COMM_WORLD` is created, spanning across the two partitions. If this is not desired, one can use the `--pack-group` key to submit independent job steps to the different node-groups of a heterogeneous allocation:

```
srun --pack-group=0 ./app_cn ; srun --pack-group=1 ./app_dam
```

Using this configuration implies that inter-communication must be established manually by the applications during run time, if needed.

For more information about heterogeneous jobs please refer to the [?relevant page](#) of the Slurm documentation.

Heterogeneous jobs with MPI communication across modules

In order to establish MPI communication across modules using different interconnect technologies, some special Gateway nodes must be used. On the DEEP-EST system, MPI communication across gateways is needed only between Infiniband and Extoll interconnects.

Attention: Only [ParaStation?](#) MPI supports MPI communication across gateway nodes.

Attention: During the first part of 2020, only the DAM nodes will have Extoll interconnect, while the CM and the ESB nodes will be connected via Infiniband. This will change later during the course of the project (expected Summer 2020), when the ESB will be equipped with Extoll connectivity (Infiniband, which will be removed from the ESB and left only for the CM).

A general description of how the user can request and use gateway nodes is provided at [?this section](#) of the JURECA documentation.

Attention: some information provided on the JURECA documentation do not apply for the DEEP system. In particular:

- as of 09/01/2020, the DEEP system has 1 gateway node. In the next weeks at least one additional gateway node will be installed.
- As of 09/01/2020 the gateway nodes are exclusive to the job requesting them. Given the limited number of gateway nodes available on the system, this may change in the future.
- The `xenv` utility (necessary on JURECA to load modules for different architectures - Haswell and KNL) is needed on DEEP only to load the `extoll` module on the DAM and ESB nodes (the `extoll` module is not available on the CM. Trying to load it there will produce an error and cause the job to fail). All the other modules can be loaded via the usual `module load` or `ml` command on the batch script before the `srun` command. If desired, `xenv` can still be used to load different set of modules for different sub-jobs of a heterogeneous jobs.

Available Partitions

Please note that there is no default partition configured. In order to run a job, you have to specify one of the following partitions, using the `--partition=...` switch:

- `dp-cn`: The DEEP-EST cluster nodes
- `dp-dam`: The DEEP-EST DAM nodes
- `sdv`: The DEEP-ER sdv nodes
- `knl`: The DEEP-ER knl nodes (all of them, regardless of cpu and configuration)
- `knl256`: the 256-core knls
- `knl272`: the 272-core knls
- `snc4`: the knls configured in SNC-4 mode
- `ml-gpu`: the machine learning nodes equipped with 4 Nvidia Tesla V100 GPUs each
- `extoll`: the sdv nodes in the extoll fabric (**KNL nodes not on Extoll connectivity anymore!**)
- `dam`: prototype dam nodes, two of which equipped with Intel Arria 10G FPGAs.

Anytime, you can list the state of the partitions with the `sinfo` command. The properties of a partition can be seen using

```
scontrol show partition <partition>
```

Information on past jobs and accounting

The `sacct` command can be used to enquire the Slurm database about a past job.

FAQ

Is there a cheat sheet for all main Slurm commands?

Yes, it is available [?here](#).

Why's my job not running?

You can check the state of your job with

```
scontrol show job <job id>
```

In the output, look for the `Reason` field.

You can check the existing reservations using

```
scontrol show res
```

How can I check which jobs are running in the machine?

Please use the `squeue` command.

How do I do chain jobs with dependencies?

Please confer the `sbatch/srun` man page, especially the

```
-d, --dependency=<dependency_list>
```

entry.

Also, jobs can be chained after they have been submitted using the `scontrol` command by updating their `Dependency` field.

How can check the status of partitions and nodes?

The main command to use is `sinfo`. By default, when called alone, `sinfo` will list the available partitions and the number of nodes in each partition in a given status. For example:

```
[deamicisl@deepv hybridhello]$ sinfo
PARTITION    AVAIL  TIMELIMIT  NODES  STATE NODELIST
sdv           up    20:00:00    16    idle deeper-sdv[01-16]
knl           up    20:00:00     1  drain knl01
knl           up    20:00:00     3    idle knl[04-06]
knl256        up    20:00:00     1  drain knl01
knl256        up    20:00:00     1    idle knl05
knl272        up    20:00:00     2    idle knl[04,06]
snc4          up    20:00:00     1    idle knl05
dam           up    20:00:00     1  down* protodam01
dam           up    20:00:00     3    idle protodam[02-04]
extoll        up    20:00:00    16    idle deeper-sdv[01-16]
ml-gpu        up    20:00:00     1    idle ml-gpu01
dp-cn         up    20:00:00     1  drain dp-cn49
dp-cn         up    20:00:00     2  alloc dp-cn[01,50]
dp-cn         up    20:00:00    47    idle dp-cn[02-48]
dp-dam        up    20:00:00     1  drain* dp-dam01
dp-dam        up    20:00:00     1  drain dp-dam02
dp-dam        up    20:00:00    14  down dp-dam[03-16]
dp-sdv-esb    up    20:00:00     2    idle dp-sdv-esb[01-02]
psgw-cluster  up    20:00:00     1  down* nfgw01
psgw-booster  up    20:00:00     1  down* nfgw02
debug         up    20:00:00     1  drain* dp-dam01
debug         up    20:00:00     1  down* protodam01
debug         up    20:00:00     3  drain dp-cn49,dp-dam02,knl01
debug         up    20:00:00    14  down dp-dam[03-16]
debug         up    20:00:00     2  alloc dp-cn[01,50]
debug         up    20:00:00    69    idle deeper-sdv[01-16],dp-cn[02-48],knl[04-06],protodam[02-04]
```

Please refer to the man page for `sinfo` for more information.

Can I join stderr and stdout like it was done with `-joe` in Torque?

Not directly. In your batch script, redirect stdout and stderr to the same file:

```
...
#SBATCH -o /point/to/the/common/logfile-%j.log
#SBATCH -e /point/to/the/common/logfile-%j.log
...
```

(The %j will place the job id in the output file). N.B. It might be more efficient to redirect the output of your script's commands to a dedicated file.

What is the default binding/pinning behaviour on DEEP?

DEEP uses a ParTec-modified version of Slurm called psslurm. In psslurm, the options concerning binding and pinning are different from the ones provided in Vanilla Slurm. By default, psslurm will use a *by rank* pinning strategy, assigning each Slurm task to a different physical thread on the node starting from OS processor 0. For example:

```
[deamicisl@deepv hybridhello]$ OMP_NUM_THREADS=1 srun -N 1 -n 4 -p dp-cn ./HybridHello | sort -k9n -k11n
Hello from node dp-cn50, core 0; AKA rank 0, thread 0
Hello from node dp-cn50, core 1; AKA rank 1, thread 0
Hello from node dp-cn50, core 2; AKA rank 2, thread 0
Hello from node dp-cn50, core 3; AKA rank 3, thread 0
```

Attention: please be aware that the psslurm affinity settings only affect the tasks spawned by Slurm. When using threaded applications, the thread affinity will be inherited from the task affinity of the process originally spawned by Slurm. For example, for a hybrid MPI-OpenMP application:

```
[deamicisl@deepv hybridhello]$ OMP_NUM_THREADS=4 srun -N 1 -n 4 -c 4 -p dp-dam ./HybridHello | sort -k9n -k11n
Hello from node dp-dam01, core 0-3; AKA rank 0, thread 0
Hello from node dp-dam01, core 0-3; AKA rank 0, thread 1
Hello from node dp-dam01, core 0-3; AKA rank 0, thread 2
Hello from node dp-dam01, core 0-3; AKA rank 0, thread 3
Hello from node dp-dam01, core 4-7; AKA rank 1, thread 0
Hello from node dp-dam01, core 4-7; AKA rank 1, thread 1
Hello from node dp-dam01, core 4-7; AKA rank 1, thread 2
Hello from node dp-dam01, core 4-7; AKA rank 1, thread 3
Hello from node dp-dam01, core 8-11; AKA rank 2, thread 0
Hello from node dp-dam01, core 8-11; AKA rank 2, thread 1
Hello from node dp-dam01, core 8-11; AKA rank 2, thread 2
Hello from node dp-dam01, core 8-11; AKA rank 2, thread 3
Hello from node dp-dam01, core 12-15; AKA rank 3, thread 0
Hello from node dp-dam01, core 12-15; AKA rank 3, thread 1
Hello from node dp-dam01, core 12-15; AKA rank 3, thread 2
Hello from node dp-dam01, core 12-15; AKA rank 3, thread 3
```

Be sure to explicitly set the thread affinity settings in your script (e.g. exporting environment variables) or directly in your code. Taking the previous example:

```
[deamicisl@deepv hybridhello]$ OMP_NUM_THREADS=4 OMP_PROC_BIND=close srun -N 1 -n 4 -c 4 -p dp-dam ./HybridHello | sort -k9n -k11n
Hello from node dp-dam01, core 0; AKA rank 0, thread 0
Hello from node dp-dam01, core 1; AKA rank 0, thread 1
Hello from node dp-dam01, core 2; AKA rank 0, thread 2
Hello from node dp-dam01, core 3; AKA rank 0, thread 3
Hello from node dp-dam01, core 4; AKA rank 1, thread 0
Hello from node dp-dam01, core 5; AKA rank 1, thread 1
Hello from node dp-dam01, core 6; AKA rank 1, thread 2
Hello from node dp-dam01, core 7; AKA rank 1, thread 3
Hello from node dp-dam01, core 8; AKA rank 2, thread 0
Hello from node dp-dam01, core 9; AKA rank 2, thread 1
Hello from node dp-dam01, core 10; AKA rank 2, thread 2
Hello from node dp-dam01, core 11; AKA rank 2, thread 3
Hello from node dp-dam01, core 12; AKA rank 3, thread 0
Hello from node dp-dam01, core 13; AKA rank 3, thread 1
Hello from node dp-dam01, core 14; AKA rank 3, thread 2
Hello from node dp-dam01, core 15; AKA rank 3, thread 3
```

Please refer to the [following page](#) on the JURECA documentation for more information about how to affect affinity on the DEEP system using psslurm options. Please be aware that different partitions on DEEP have different number of sockets per node and cores/threads per socket with respect to JURECA. Please refer to the [System overview](#) or run the `lstopo-no-graphics` on the compute nodes to get more information about the hardware configuration on the different modules.

How do I use SMT on the DEEP CPUs?

On DEEP, SMT is enabled by default on all nodes. Please be aware that on all JSC systems (including DEEP), each hardware thread is exposed by the OS as a separate CPU. For a n -core node, with m hardware threads per core, the OS cores from 0 to $n-1$ will correspond to the first hardware thread of all hardware cores (from all sockets), the OS cores from n to $2n-1$ to the second hardware thread of the hardware cores, and so on.

For instance, on a Cluster node (with two sockets with 12 cores each, with 2 hardware threads per core):

[illegible]

[illegible]

The PU P#X are the Processing Units numbers exposed by the OS.

To exploit SMT, simply run a job using a number of tasks*threads_per_task higher than the number of physical cores available on a node. Please refer to the [relevant page](#) of the JURECA documentation for more information on how to use SMT on the DEEP nodes.

Attention: currently the only way of assign Slurm tasks to hardware threads belonging to the same hardware core is to use the `--cpu-bind` option of psslurm using `mask_cpu` to provide affinity masks for each task. For example:

```
[deamicis1@deepv hybridhello]$ OMP_NUM_THREADS=2 OMP_PROC_BIND=close OMP_PLACES=threads srun -N 1 -n 2 -p dp-dam --cpu-bin
Hello from node dp-dam01, core 0; AKA rank 0, thread 0
Hello from node dp-dam01, core 48; AKA rank 0, thread 1
Hello from node dp-dam01, core 1; AKA rank 1, thread 0
Hello from node dp-dam01, core 49; AKA rank 1, thread 1
```

This can be cumbersome for jobs using a large number of tasks per node. In such cases, a tool like [?hwloc](#) (currently available on the compute nodes, but not on the login node!) can be used to calculate the affinity masks to be passed to psslurm.