Job Processing with SLURM on SuperMUC-NG

- **General**
- **List of relevant commands**
- **Queues (SLURM partitions) and their limits**
- **srun and mpiexec**
  - `salloc / srun` for interactive processing
- **sbatch Command / `#SBATCH` option**
  - **Batch Job Examples**
  - **General options applicable for all jobs**
  - **Options for resources and execution (select and click to expand)**
  - **Input Environment Variables**
  - **Output Environment Variables**
  - **File Patterns**
  - **Useful commands**
- **Guidelines for resource selection**
- **Specific Topics (jobfarming, constraints)**
- **SLURM Documentation**

see also:
- Job farming with SLURM
- List of SLURM Constraints and its Usage

## General

The batch system on SuperMUC-NG is the open-source workload manager SLURM (Simple Linux Utility for Resource management). For details about the SLURM batch system, see [Slurm Workload Manager](#).

Submit hosts are usually login nodes that permit to submit and manage batch jobs.

Intel processors on SuperMUC-NG support the hyperthreading mode which might increase the performance of your application. With hyperthreading, you have to increase the number of MPI tasks per node from 48 to 96 in your job script. Please be aware that with 96 MPI tasks per node each process gets only half of the memory by default. If you need more memory, you have to specify it in your job script and use the fat nodes (see example batch scripts).

### List of relevant commands

<table>
<thead>
<tr>
<th>Command's name</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sbatch</code></td>
<td>submit a job script</td>
</tr>
<tr>
<td><code>scancel</code></td>
<td>delete or terminate a queued or running job</td>
</tr>
<tr>
<td><code>squeue</code></td>
<td>print table of submitted jobs and their state. <strong>Note:</strong> non-privileged users can only see their own jobs.</td>
</tr>
<tr>
<td><code>salloc</code></td>
<td>create an interactive SLURM shell</td>
</tr>
<tr>
<td><code>srun</code></td>
<td>execute argument command on the resources assigned to a job. <strong>Note:</strong> must be executed inside an active job (script or interactive environment). <code>mpiexec</code> is an alternative and preferred on LRZ system</td>
</tr>
<tr>
<td><code>sinfo</code></td>
<td>provide overview of cluster status</td>
</tr>
<tr>
<td><code>scontrol</code></td>
<td>query and modify SLURM state</td>
</tr>
</tbody>
</table>

`sacct` is currently not working.

### Queues (SLURM partitions) and their limits

- Batch queues are called partitions in SLURM.
- The allocation granularity is multiples of one node (only complete nodes are allocated and accounted for).
- Scheduling and prioritization is based on a multifactor scheme including wait time, job size, partition, and required quality of service.

The following partitions are available. Check with `sinfo` for more details and special partitions:
<table>
<thead>
<tr>
<th>partition</th>
<th>min-max nodes per job</th>
<th>max usable memory</th>
<th>cores per node</th>
<th>max run time (hours)</th>
<th>max running jobs per user</th>
<th>max submitted jobs per user (qos)</th>
</tr>
</thead>
<tbody>
<tr>
<td>test (also used for interactive access with salloc)</td>
<td>1-16</td>
<td>90 GB</td>
<td>48</td>
<td>0.5</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>micro</td>
<td>1-16</td>
<td>90 GB</td>
<td>48</td>
<td>48</td>
<td>20</td>
<td>30</td>
</tr>
<tr>
<td>general</td>
<td>17-768</td>
<td>90 GB</td>
<td>48</td>
<td>48</td>
<td>5</td>
<td>20</td>
</tr>
<tr>
<td>large</td>
<td>769-3072 (half of system)</td>
<td>90 GB</td>
<td>48</td>
<td>24 to be increased</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>fat</td>
<td>1-128</td>
<td>740 GB</td>
<td>48</td>
<td>48</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>mixed (not yet available)</td>
<td>64-3072</td>
<td>90 GB and 760 GB</td>
<td>48</td>
<td>12 to be increased</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

**srun and mpiexec**

With SLURM `srun` command 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 are defined with srun command the options from sbatch or salloc are inherited.

Note: `srun` at LRZ is defined as the alias

```bash
srun='I_MPI_PMI_LIBRARY=/usr/lib64/ibpmi.so /usr/bin/srun'
```

Since aliases are not inherited the alias is only available in the login shell or in the initial batch script, everywhere else it falls back to /usr/bin/srun. Use the full syntax in these cases.

Note: `mpiexec` is the preferred and supported way to start applications. `srun` might fail for hyperthreaded applications.

**salloc / srun for interactive processing**

`salloc` is used the allocated nodes for interactive processing. The options for resource specification in salloc/srun/sbatch are the same. Currently, at least `--account`, `--time` and `--partition` must be specified! `srun` can be used instead of `mpiexec`. `srun` or `mpiexec` execute on the nodes previously allocated by the `salloc`.

There is no advantage by using `salloc` over `sbatch` `--partition=test` in terms of wait time.

```bash
login node> salloc -t 00:30:00 -p test -A <my-desired-project-id> -N 4
salloc: Pending job allocation 48417
salloc: job 48417 queued and waiting for resources
salloc: job 48417 has been allocated resources
salloc: Granted job allocation 48417
salloc: Waiting for resource configuration
salloc: Nodes 101r01c01s[03-06] are ready for job

i01r01c01s03> srun --nodes=2 --ntasks=2 --partition=test hostname
i01r01c01s03.sng.lrz.de
i01r01c01s03.sng.lrz.de

i01r01c01s03> mpiexec -n 5 hostname
i01r01c01s03.sng.lrz.de
i01r01c01s03.sng.lrz.de
i01r01c01s03.sng.lrz.de
i01r01c01s03.sng.lrz.de
i01r01c01s03.sng.lrz.de

i01r01c01s03> srun --nodes=2 --ntasks=2 --partition=test hostname
i01r01c01s03.sng.lrz.de
i01r01c01s03.sng.lrz.de
i01r01c01s03.sng.lrz.de
i01r01c01s03.sng.lrz.de
i01r01c01s03.sng.lrz.de

i01r01c01s03> mpiexec -n 5 hostname
i01r01c01s03.sng.lrz.de
i01r01c01s03.sng.lrz.de
i01r01c01s03.sng.lrz.de
i01r01c01s03.sng.lrz.de
i01r01c01s03.sng.lrz.de

i01r01c01s03> exit
exit
salloc: Relinquishing job allocation 48417
login node>
```
sbatch Command / #SBATCH option

Batch job options and resources can be given as command line switches to `sbatch` (in which case they override script-provided values), or they can be embedded into a SLURM job script as a comment line of the form.

Batch Job Examples

<table>
<thead>
<tr>
<th>General options applicable for all jobs</th>
</tr>
</thead>
<tbody>
<tr>
<td>#!/bin/bash</td>
</tr>
<tr>
<td># Job Name and Files (also --job-name)</td>
</tr>
<tr>
<td>#SBATCH --J jobname</td>
</tr>
<tr>
<td>#SBATCH -o ./%x.%j.out</td>
</tr>
<tr>
<td>#SBATCH -e ./%x.%j.err</td>
</tr>
<tr>
<td>#SBATCH -D ./</td>
</tr>
<tr>
<td>#SBATCH -D ./</td>
</tr>
<tr>
<td>#SBATCH --mail-type=END</td>
</tr>
<tr>
<td>#SBATCH --mail-user=insert_your_email</td>
</tr>
<tr>
<td>#SBATCH --mail-type=END</td>
</tr>
<tr>
<td>#SBATCH --account=insert_your_projectID_here</td>
</tr>
<tr>
<td>#SBATCH --constraint=scratch&amp;work</td>
</tr>
<tr>
<td>#SBATCH --constraint=scratch&amp;work</td>
</tr>
<tr>
<td>#SBATCH --export=NONE</td>
</tr>
<tr>
<td>#SBATCH --get-user-env</td>
</tr>
</tbody>
</table>
| #SBATCH --constraint=

Hints and Explanations:

Replacement patterns in filenames:
- %J: jobid.stepid of the running job. (e.g. "128.0")
- %j: jobid of the running job.
- %s: stepid of the running job.
- %t: task identifier (rank) relative to current job. this will create a separate IO file per task.
- %u: User name.
- %x: Job name.
- %a: Job array ID

Notification types:
- NONE, BEGIN, END, FAIL, REQUEUE
- queue/no-requeue: Whether the job should eligible to being requeue or not. When a job is requeued, the batch script is initiated from its beginning, no-requeue specifies that the batch job should never be requeued under any circumstances.

environment:
- Do not export the variables of the submitting shell into the job (which would make debugging of errors nearly impossible for LRZ).
- get-user-env will set Environment variable as during Login.

account:
- Resources used by this job are subtracted from budget of this project. The billing unit is core-hours.
- Make sure that you use the right project.

constraint (optional):
- Nodes can have features. Users can specify which of these features are required by their job using the constraint option. Only nodes having features matching the job constraints will be used to satisfy the request. Multiple constraints may be specified with AND (&), OR (|), matching OR, resource counts, etc. The availability of specific file systems can be specified as a constraint, giving the LRZ the opportunity to start jobs which do not need all. See:

  > List of SLURM Constraints and its Usage

slurm_setup

specific setting which cannot be set in the SLURM prologue
Options for resources and execution
(select and click to expand)

Resource Specifications:

`nodes=<minnodes[-maxnodes]>`
Request that a minimum of `minnodes` nodes be allocated to this job. A maximum node count may also be specified with `maxnodes`. If only one number is specified, this is used as both the minimum and maximum node count. The default behavior is to allocate enough nodes to satisfy the requirements of the `ntasks` and `cpus-per-task` options.

`ntasks`:
The default is one task per node, but note that the `cpus-per-task` option will change this default.

`ntasks-per-node`:
Request that `ntasks` be invoked on each node. If used with the `ntasks` option, the `ntasks` option will take precedence and the `ntasks-per-node` will be treated as a maximum count of tasks per node. Meant to be used with the `nodes` option.

`ntasks-per-core`:
Request that the maximum `ntasks` be invoked on each core.

`cpus-per-task`:
Without this option, the controller will just try to allocate one core per task.

`sbatch` --
Attempt to invoke on each node. If used with the `ntasks` option, the `ntasks` option will take precedence and the `ntasks-per-node` will be treated as a count of tasks per node. Meant to be used with the `nodes` option.

`sbatch` --
Maximum count of switches desired for the job allocation and optionally the maximum time to wait for that number of switches. + Use this option only for very large Jobs.

`array`:
Submit a job array, multiple jobs to be executed with identical parameters.

`mpiexec`:
In most cases `mpiexec` can be used without specifying the number of tasks, because this is inherited from the `sbatch` command. Slurm output variables can also be used e.g.,

```
mpiexec -n $SLURM_NTASKS ./myprog
```

If SLURM can detect the number of tasks form its settings it is sufficient to use `mpiexec` without further parameters e.g.,

```
mpiexec ./myprog
```

Execution Specification:

By default, the system may dynamically change the clock frequency of CPUs during the runtime of a job to optimise for energy consumption (for more details, see Energy Aware Runtime (EAR)). This makes profiling or benchmark measurements difficult and unstable. Users can enforce a fixed default frequency by switching EAR off:

```
sbatch --ear=off
```
#SBATCH --partition=general
#SBATCH --nodes=128
#SBATCH --ntasks-per-node=96
#Note: Needs specific MPI version > 2019
#Run the program:
#Pinning
#Task0->CPU0, Task1->CPU48 ...
#CPU0 and CPU48 are on the same physical core) !
#Task2->CPU1, Task3->CPU49
mpiexec -n $SLURM_NTASKS ./myprog

#Optional: use more explicit pinning (spreading)
#I_MPI_PIN_PROCESSOR_LIST=0-95
#Task0->CPU0, Task1->CPU1,...
#CPU2, ... CPU4
#... (general part)
#SBATCH --
partition=general
#Number of
nodes and MPI
tasks per
node:
#SBATCH --
nodes=128
#SBATCH --
ntasks-per-
node=8

#Run the
program:
export
OMP_NUM_THREADS
S=6
#Default
Pinning:
#Thread0
/Task0 ->CPU0
or CPU48
#Thread1
/Task0 ->CPU1
or CPU49
#Thread2
/Task0 ->CPU2
or CPU50
mpiexec -n
$SLURM_NTASKS
./myprog

#export
I_MPI_PIN_CELL
=core
#export
I_MPI_PIN_DOMA
IN=omp:compact
#Thread0
/Task0 ->CPU0,
Thread1
/Task0 ->CPU1
#Thread2
/Task0 ->CPU2,
Thread3
/Task0 ->CPU3
#...
#Thread0
/Task1 ->
CPU6, Thread1
/Task1 ->CPU7

#explicit
pinning with
masks
#export
I_MPI_PIN_CELL
=unit
#export
I_MPI_PIN_DOMA
IN=\[3F,FC0, 3F000,FC0000, etc.\]
#SBATCH -- partition=gene
#SBATCH -- nodes=128
#SBATCH -- ntasks-per-node=8

# Run the program:
export OMP_NUM_THREADS=24
mpiexec -n $SLURM_NTASKS ./myprog

# Optional:
Resulting in same pinning as default
export I_MPI_PIN_CELL=unit
export I_MPI_PIN_DOMA=omp:compact

# Thread0 /Task0 -> CPU0, Thread1 /Task0 -> CPU1
# Thread5 /Task0 -> CPU5
!!!
# Thread6 /Task0 -> CPU48,
# Thread7 /Task0 -> CPU49

# explicit pinning with masks
export I_MPI_PIN_CELL=unit
export I_MPI_PIN_DOMA="[FFFFFF, FFFFF0000000, etc.]"
#!/bin/bash

#SBATCH --partition=fat
#SBATCH --nodes=64
#SBATCH --ntasks-per-node=48
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=48

# Run the program:
mpirun -n $SLURM_NTASKS ./myprog

#SBATCH --partition=micro
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=48

# Run the program:
export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK
./myprog

# Default:
KMP_AFFINITY=granularity=thread,compact,1,0
#Thread0 -> CPU0,
#Thread1 -> CPU1
#SBATCH --partition=large
Max number of islands and max waittime
#SBATCH --switches=2@24:00:00
#SBATCH --nodes=1024
#SBATCH --ntasks-per-node=48
mpiexec -n $SLURM_NTASKS ./myprog

#SBATCH --partition=general
Number of nodes and MPI tasks per node:
#SBATCH --nodes=10
#SBATCH --ntasks=480
#SBATCH --cpus-per-task=1
#SBATCH --array=1-10
mpiexec -n $SLURM_NTASKS ./myprog <in.$SLURM_ARRAY_TASK_ID

#SBATCH --ear=off
...
#!/bin/bash
# Chain of
# batch jobs
with
dependencies
NR_OF_JOBS=6
JOB_SCRIPT=./my_batch_script
echo "Submitting
chain of
jobs for
batch script
$JOB_SCRIPT"
#submit and
get JOBID
JOBID=$(sbatch $JOB_SCRIPT 2>&1 |
awk '{print $(NF)}' )
I=1
while [ $I -lt $NR_OF_JOBS ]
do
   JOBID=$(sbatch --dependency=afterok:$JOBID
$JOB_SCRIPT 2>&1 |
awk '{print $(NF)}' )
   I=$(($I+1 ))
done

dependency=<dependency_list>
Defer the start of this job until the specified dependencies have been satisfied completed.< dependency_list> is of the form:
- after:job_id[:job_id...] job can begin execution after the specified jobs have been begun execution.
- afterany:job_id[:job_id...] job can begin execution after the specified jobs have terminated.
- afternotok:job_id[:job_id...] job can begin execution after the specified jobs have terminated in some failed state (non-zero exit code, node failure, timed out, etc).
- afterok:job_id[:job_id...] job can begin execution after the specified jobs have successfully executed (ran to completion with an exit code of zero

Input Environment Variables

Upon startup, sbatch will read and handle the options set in the following environment variables. Note that environment variables will override any options set in a batch script, and command line options will override any environment variables. Some which may be used by you in $HOME/.profile:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>SBATCH_ACCOUNT</td>
<td>--account</td>
</tr>
<tr>
<td>SBATCH_JOB_NAME</td>
<td>--jobid</td>
</tr>
<tr>
<td>SBATCH_REQUEUE</td>
<td>--requeue</td>
</tr>
<tr>
<td>SBATCH_NOREQUEUE</td>
<td>--no-requeue</td>
</tr>
</tbody>
</table>

Output Environment Variables

The Slurm controller will sets the variables in the environment of the batch script

<table>
<thead>
<tr>
<th>Variable</th>
<th>Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLURM_JOB_ACCOUNT</td>
<td>Account name associated of the job allocation</td>
</tr>
</tbody>
</table>
**File Patterns**

`sbatch` allows for a filename pattern to contain one or more replacement symbols, which are a percent sign “%” followed by a letter.

**Example:**
```
#SBATCH -o %x.%j.out
```

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Expansion</th>
</tr>
</thead>
<tbody>
<tr>
<td>%j</td>
<td>jobid of the running job</td>
</tr>
<tr>
<td>%J</td>
<td>jobid.stepid of the running job</td>
</tr>
<tr>
<td>%a</td>
<td>Job array ID (index)</td>
</tr>
<tr>
<td>%u</td>
<td>User name</td>
</tr>
<tr>
<td>%x</td>
<td>Job Name</td>
</tr>
<tr>
<td>%t</td>
<td>task identifier (rank) relative to current job. This will create a separate IO file per task.</td>
</tr>
</tbody>
</table>

**Useful commands**

Show the estimated start time of a job: `squeue --start [-u <userID>]`

**Guidelines for resource selection**

### Processing Mode

- Jobs that only use one or at most a few hardware cores perform serial processing and are not supported on SuperMUC-NG. Use the SuperMUC-Cloud for such purposes.
- Bunches of multiple independent tasks can be bundled into one job, using one or more nodes.

### Run time limits

- Please note that all job classes impose a maximum run time limit. It can be adjusted downward for any individual job. Since the scheduler uses a backfill algorithm, the better you specify a realistic runtime limit, the better throughput of your job may be achieved.

### Islands/Switches

- When a tree topology is used, this defines the maximum count of switches desired for the job allocation and optionally the maximum time to wait for that number of switches. If Slurm finds an allocation containing more switches than the count specified, the job remains pending until it either finds an allocation with desired (lower) switch count or the time limit expires. It there is no switch count limit, there is no delay in starting the job. This trades off better performance vs. shorter wait time in the queue.

### Memory Requirements

- The total memory available in user space for the set of nodes requested by the job must not be exceeded.
- The memory used on each individual node must not be exceeded by all tasks run on that node.
- Applications exist for which the memory usage is unsymmetric. In this case it may become necessary to work with a variable number of tasks per node. One relevant scenario is a master-worker scheme where the master may need an order of magnitude more memory and therefore requires a node of its own, while worker nodes can share a node. LRZ provides the “mixed” partition for using thin and fat nodes concurrently.

### Disk and I/O Requirements
The disk and I/O requirements are not controlled by the batch scheduling system, but rely on parallel shared file systems, which provide system-global services with respect to bandwidth - this means that the total I/O bandwidth is shared between all users. The consequence is that all I/O may be significantly slowed down if heavily used by multiple users at the same time, or even - for large scale parallel jobs - by a single user. At present, LRZ can not make any Quality of Service assurance for I/O bandwidth.

The appropriate usage of the parallel file systems is essential.

Please consult File Systems of SuperMUC-NG for more detailed technical information.

Licences

- Some jobs may make use of licensed software, either from the LRZ software application stack, or of software installed in the user’s HOME directory. In many cases, the software needs to access a license server because there exist limits on how many instances of the software may run and who may access it at all.
- There is no connection from SuperMUC-NG to the outside. Check with LRZ if you are in need of such licenses.
- LRZ is currently not able to manage license contingents. The reason is that a significant additional effort is required, not only with suitable configuration of SLURM, but also with how the license servers are managed. The situation implies that a job will fail if the usage limit of a licensed software is exceeded when the job starts.

Conversion of scripts from LoadLeveler and other Workload Managers table

- see: List of the most common command, environment variables, and job specification options used by the major workload management systems

Specific Topics (jobfarming, constraints)

- Job farming with SLURM
- List of SLURM Constraints and its Usage

SLURM Documentation

- SLURM Workload Manager at LRZ
- Command/option Summary (two pages)
- Documentation for SLURM at SchedMD
- The manual pages slurm(1), sinfo(1), squeue(1), scontrol(1), scancel(1), sview(1)