Batch System introduction ¶

Batch systems keeps track of available system resources and takes care of scheduling jobs of multiple users running their tasks simultaneously. It typically organizes submitted jobs into some sort of prioritized queue. The batch system is also used to enforce local system resource usage and job scheduling policies.

HPC2N currently has one cluster which accepts local batch jobs, Kebnekaise. It is running Slurm. It is an Open Source job scheduler, which provides three key functions.

- First, it allocates to users, exclusive or non-exclusive access to resources for some period of time.
- Second, it provides a framework for starting, executing, and monitoring work on a set of allocated nodes (the cluster).
- Third, it manages a queue of pending jobs, in order to distribute work across resources according to policies.

Slurm is designed to handle thousands of nodes in a single cluster, and can sustain throughput of 120,000 jobs per hour.

You can find more general information about what a cluster and a batch system is in the tutorial "Beginners's intro to clusters".

Slurm commands and information

There are many more commands than the ones we have chosen to look at here, but these are the most commonly used ones. You can find more information on the Slurm homepage: Slurm documentation.

You can run programs either by giving all the commands on the command line or by submitting a job script.

Using a job script is often recommended:

- If you ask for the resources on the command line, you will wait for the program to run before you can use the window again (unless you can send it to the background with &).
- If you use a job script you have an easy record of the commands you used, to reuse or edit for later use.

Go here for description of writing a submit file.

salloc - requesting an interactive allocation ¶

Using `salloc`, you get an interactive shell to run your jobs in, when your resources have been allocated. Note that you cannot use the window while you wait for - perhaps - a long time before the allocation starts.

This example asks to allocate 1 node with 4 processors for 1 hour and 30 minutes. When the resources are available, you will get an interactive shell with those resources available for use.

```
$ salloc -A <your project> -N 1 -n 4 --time=1:30:00
```

Note that you will still be on the login node when the prompt returns and you MUST use `srun` to run your job on the allocated resources.

```
$ srun -n 2 my_program
```

Serial, OpenMP, MPI, hybrid jobs, GPU jobs - all can be submitted either using an interactive shell started with salloc, or through a job submission file.

sbatch - submitting jobs to the batch system

Submitting a job script avoids having to wait for the job to start before being able to continue working. While it still may take a long time before the job runs (depending on load on the machine and your project's priority), you can use the window in the meantime and you do not have to sit and be ready to use it when the job start. Remember, if you do it as an interactive job, your allocated run time starts when the job starts.

The commands `sbatch` and `srun` can be used to allocate and then start multiple tasks on multiple nodes, where each task is a separate process executing the same program. By default, Slurm allocates one processor per task, but starts tasks on multiple processors as necessary. You can, however, specify these yourself, and does not have to follow the default.

The command `sbatch JOBSCRIPT` submits your submit file JOBSCRIPT to the batch system and returns directly with the jobid of the job.

The most important options to understand when writing a submit file are the following:

- `-n` request a certain number of (MPI-)tasks, mostly used for MPI based codes
- `-c` request a certain number of cores per (MPI-)task or for OpenMP or threaded codes, must be <= #cores on the targeted node
- `–ntasks-per-node` request that a specific number of (MPI-)tasks should run on each node of the job
- `-N` request that the job has a specific number of nodes allocated to it

Note: `-c x –ntasks-per-node` must also be <= #cores on the targeted node.

More information about the different nodes types can be found in the section: The different parts of the batch system.

More information about other parameters and job submission files can be found in the section: Slurm submit file design.

squeue - viewing the state of the batch queue

To view the queue one can use the `squeue` command. The most useful options are:

- `–me` to look at your own jobs
- `-t pd` to look at pending jobs, i.e. not yet started ones
- `-t r` to look at running jobs

these can be combined like `–me -t r` to look at your running jobs.

job-usage - get url to see details of your job (not a Slurm command)

We have a command `job-usage` that will produce an URL to a page with detailed runtime information of a running job.

```
job-usage <jobid>
```

The web page will show things like CPU (and GPU when appropriate), memory, and disk-io usage for the duration of the job.

scancel - cancel a job

To cancel a job, either pending or already running, one uses `scancel`. One can cancel either a specific job, all the users jobs or jobs with a specific name. To see all the possibilities check out the man page, `man scancel`.

Usage is simply: `scancel jobid`

Squeue “Reason” explained

Due to the way we have setup the batch system, jobs can sometimes display unexpected states in the squeue output. Here we describe some of the ones that can usually be ignored since they won’t normally affect a jobs ability to start.

- **AssocGrpBillingRunMinutes**: The project’s currently active resources plus the pending job’s requested resources exceeds the limit of one monthly allocation being in use at any point in time. The pending job will eventually start, unless it, by itself, is larger than one monthly allocation for the project.
- **BadConstraints**: Some contraint set on the job (-C/–contraint) is not available on some of the partitions the job has been sent to by our remapping script. This is normally not a problem since the batch system will only send the job to a partition which fulfills the contraint. However, if the jobs constraints can’t be fulfilled by any partition the job will stay in this state and never start.

The different parts of the batch system

The batch system is made up of several different types of resources. At the top we have partitions, which are similar to queues. Each partition is made up of a specific set of nodes. At HPC2N we have three classes of partitions, one for CPU-only nodes, one for GPU nodes and one for large memory nodes. Each node type also has a set of features that can be used to select which node(s) the job should run on.

The three types of nodes also have corresponding resources one must apply for in SUPR to be able to use them.

Partitions

There is only a single partition, `batch`, that users can submit jobs to. The system then figures out, based on requested features which actual partition(s) the job should be sent to.

Since there is only one partition available for users to submit jobs to, you should now remove any use of `#SBATCH -p` or `sbatch -p`. The most common use of `-p` was for targeting the LargeMemory nodes, this is now done using a feature request like this:

```
#SBATCH -C largemem
```

See below for more info about features.

Information about the Slurm nodes and partitions can be found using this command:

```
$ sinfo
```

Unless otherwise specified the system will allocate resources on the first available node(s) that the job can run on. It will try to use as few resources as possible, i.e. use nodes with less memory per core before nodes with more memory etc.

Nodes

Kebnekaise have CPU-only, GPU enabled and large memory nodes.

The CPU-only nodes are:

- 2 x 14 core Intel skylake
  - 6785 MB memory / core

    - 48 nodes
- 2 x 64 core AMD zen3
    - 8020 MB / core
    - 1 node
- 2 x 128 core AMD zen4
    - 2516 MB / core
    - 8 nodes

To request node(s) with a specific type of CPU use the available features

The GPU enabled nodes are:

- 2 x 14 core Intel skylake
    - 6785 MB memory / core
    - 2 x Nvidia V100
    - 10 nodes
- 2 x 24 core AMD zen3
    - 10600 MB / core
    - 2 x Nvidia A100
    - 2 nodes
- 2 x 24 core AMD zen3
    - 10600 MB / core
    - 2 x AMD MI100
    - 1 node
- 2 x 24 core AMD zen4
    - 6630 MB / core
    - 2 x Nvidia A6000
    - 1 node
- 2 x 24 core AMD zen4
    - 6630 MB / core
    - 2 x Nvidia L40s
    - 10 nodes
- 2 x 48 core AMD zen4
    - 6630 MB / core
    - 4 x Nvidia H100 SXM5
    - 2 nodes
- 2 x 32 core AMD zen4
    - 11968 MB / core
    - 6 x Nvidia L40s
    - 2 nodes
    - Can only use 10 cores/GPU
- 2 x 32 core AMD zen4

- 11968 MB / core
- 8 x Nvidia A40
- 1 nodes

See the next section regarding requesting GPUs.

"Cost" of the different GPUs

GPU resources are accounted in GPU-hours. Each GPU type comes with a different "cost" when used. The base is which ever type of GPU is currently the oldest and that GPU will have a cost of 1 GPU-hour per hour used.

The current costs are:

- V100 : 1
- A40 : 1.5
- A6000 : 1.5
- L40s : 1.7
- A100 : 2
- H100 : 3
- MI100 : 1.8

The large memory nodes are:

- 4 x 18 core Intel broadwell
  - 41666 MB memory / core
  - 8 nodes

Requesting GPUs

To use GPU resources one has to explicitly ask for one or more GPUs. Requests for GPUs can be done either in total for the job or per node of the job.

```
#SBATCH --gpus=1
```

or

```
#SBATCH --gpus-per-node=1
```

One can ask for a specific GPU type by using

```
#SBATCH --gpus=l40s:1
```

or any of the other versions.

To request node(s) with a specific type of GPU use the available features or the generic GPU features.

Requesting specific features (i.e. setting contraints on the job)

To make it possible to target nodes in more detail there are a couple of features defined on each group of nodes. To select a feature one can use the `-C` option to `sbatch` or `salloc`

I.e. to constrain the job to nodes with the `zen4` feature use:

```
#SBATCH -C zen4
```

Features can be combined using “and” or “or” like this:

```
#SBATCH -C 'zen4&GPU_AI'
```

for a node with a zen4 CPU and a GPU with AI features, or

```
#SBATCH -C 'zen3|zen4'
```

to make sure the job runs on either a zen3 or a zen4 cpu.

We have at least the following features defined depending on type of node.

For selecting type of CPU:

- intel_cpu
- broadwell
- skylake
- amd_cpu
- zen3
- zen4

For selecting type of GPU:

- v100
- a40
- a6000
- a100
- l40s
- h100
- mi100

The above list can be used either as a specifier to `-gpu=type:number` or as a constraint together with an unspecified gpu request `-gpu=number`

For selecting GPUs with certain features:

- nvidia_gpu (Any Nvidia GPU)
- amd_gpu (Any AMD GPU)
- GPU_SP (GPU with single precision capability)
- GPU_DP (GPU with double precision capability)
- GPU_AI (GPU with AI features, like half precisions and lower)
- GPU_ML (GPU with ML features, like half precisions and lower)

For selecting large memory nodes:

- largemem

Useful environment variables in a job

These are a few of the important and most useful environment variables available in a job.

- **SLURM_JOB_NUM_NODES**: the number of nodes you got.
- **SLURM_NTASKS**: contains the number of task slots allocated. Note that this is only set if one of the `-ntasks` options was used.
- **SLURM_CPUS_PER_TASK**: the number of cores allocated per task. Note that this is only set if the `-c` option was used.
- **SLURM_CPUS_ON_NODE**: the total number of cores allocated on the node for the job.
- **SLURM_JOB_ID**: contains the id of the current job.

Batch system Policies

The batch system policy is fairly simple, and currently states that:

- A job is not allowed to run longer than 7 days (604800 s) regardless of the allocated CPU time.
- A job will start when the resources you have asked for are available (it takes longer to get more cores etc.), and your priority is high enough, compared to others. How high a priority your job has, depends on
  1. your allocation
  2. whether or not you, or others using the same project, have run a lot of jobs recently. If you have, then your priority becomes lower.
- The sum of the size (remaining-runtime * number-of-cores) of all running jobs must be less than the monthly allocation.

Warning

If you submit a job that takes up more than your monthly allocation (remember running jobs take away from that), then your job will be pending with **"Reason=AssociationResourceLimit"** or **"Reason=AssocGrpBillingRunMinutes"** until enough running jobs have finished.

Also, a job cannot start if it asks for more than your total monthly allocation.

You can see the current priority of your project (and that of others), by running the command `sshare` and look for the column marked `'Fairshare'` - that shows your groups current priority.

The fairshare usage weight decays gradually over 50 days, meaning that jobs older than 50 days does not count towards priority.

Remember

When and if a job starts depends on which resources it is requesting. If a job is asking for, say, 10 nodes and only 8 are currently available, the job will have to wait for resources to free up. Meanwhile, other jobs with lower requirements will be allowed to start as long as they do not affect the start time of higher priority jobs.

Detailed description of scheduling

The Slurm scheduler divides the job queue in two parts.

1. **Running jobs.** These are the jobs that are currently running.
2. **Pending jobs.** These are the jobs that are being considered for scheduling, or (for policy reasons like rules and limits), are not (yet) being considered for scheduling.

Basically what happens when a job is submitted is this.

1. The job is put in the correct part of the queue (pending) according to the policy rules.
2. The scheduler checks if any jobs that were previously breaking policy rules, can now be considered for scheduling.
3. The scheduler calculates a new priority for all the jobs in the pending part.
4. If there are available processor resources the highest priority job(s) will be started.
5. If the highest priority job cannot be started for lack of resources, the next job that fits, without changing the predicted startwindow for any higher priority jobs, will be started (so called backfilling).

Calculating priority

When a job is submitted, the Slurm batch scheduler assigns it an initial priority. The priority value will be recalculated periodically while the job is waiting, until the job gets to the head of the queue. This happens as soon as the needed resources are available, provided no jobs with higher priority and matching available resources exists. When a job gets to the head of the queue, and the needed resources are available, the job will be started.

At HPC2N, Slurm assigns job priority based on the Multi-factor Job Priority scheduling. As it is currently set up, only one thing influence job priority:

- Fair-share: the difference between the portion of the computing resource that has been promised and the amount of resources that has been consumed by a group

Weights has been assigned to the above factors in such a way that fair-share is the dominant factor.

The following formula is used to calculate a job's priority:

```
Job_priority = 1000000 * (fair-share_factor)
```

Priority is then calculated as a weighted sum of these.

The `fair-share_factor` is dependent on several things, mainly:

- Which project account you are running in.
- How much you and other members of your project have been running. This fair share weight decays over 50 days, as mentioned earlier.

You can see the current value of your jobs fairshare factors with this command

```
sprio -l -u <username>
```

and your and your project's current fairshare value with

```
sshare -l -u <username>
```

Note

- these values change over time, as you and your project members use resources, others submit jobs, and time passes.
- the job will NOT rise in priority just due to sitting in the queue for a long time. No priority is calculated merely due to age of the job.

For more information about how fair-share is calculated in Slurm, please see: http://slurm.schedmd.com/priority_multifactor.html.

Batch scripts (Job Submission Files)

The official name for batch scripts in Slurm is Job Submission Files, but here we will use both names interchangeably. If you search the internet, you will find several other names used, including Slurm submit file, batch submit file, batch script, job script.

A job submission file can contain any of the commands that you would otherwise issue yourself from the command line. It is, for example, possible to both compile and run a program and also to set any necessary environment values (though remember that Slurm exports the environment variables in your shell per default, so you can also just set them all there before submitting the job).

The results from compiling or running your programs can generally be seen after the job has completed, though as Slurm will write to the output file during the run, some results will be available quicker.

Outputs and any errors will per default be placed in the directory you are running from, though this can be changed.

Note that this directory should preferrably be placed under your project storage, since your home directory only has 25 GB of space.

Changing the output and error file

Both output and errors will, by default, be combined into a file named `slurm-JOBID.out`. You can send them to other/separate files with these commands:

```
#SBATCH --error=job.%J.err
#SBATCH --output=job.%J.out
```

In the example above, you get files named `job.JOBID.err` and `job.JOBID.out`, you can of course give them other names, but if you are running several batch jobs, remember to name them so they are not over-written by later jobs (use the `%J` environment variable).

A job submission file can either be very simple, with most of the job attributes specified on the command line, or it may consist of several Slurm directives, comments and executable statements. A Slurm directive provides a way of specifying job attributes in addition to the command line options.

Naming: You can name your script anything, including the suffix. It does not matter. Just name it something that makes sense to you and helps you remember what the script is for. The standard is to name it with a suffix of `.sbatch` or `.sh`.

Note that you have to always include `#!/bin/bash` at the beginning of the script, since `bash` is the only supported shell. Some things may work under other shells, but not everything.

Example

This example batch script is for a job with 8 MPI tasks, and separated output and error files.

```
#!/bin/bash
# The name of the account you are running in, mandatory.
#SBATCH -A hpc2nXXXX-YYY
# Request resources - here for eight MPI tasks
#SBATCH -n 8
# Request runtime for the job (HHH:MM:SS) where 168 hours is the maximum. Here asking for 1
#SBATCH --time=00:15:00
# Set the names for the error and output files
#SBATCH --error=job.%J.err
#SBATCH --output=job.%J.out

# Clear the environment from any previously loaded modules
module purge > /dev/null 2>&1

# Load the module environment suitable for the job - here foss/2021b
module load foss/2021b

# And finally run the job - use srun for MPI jobs, but not for serial jobs
srun ./my_mpi_program
```

Note

- One (or more) `#` in front of a text line means it is a comment, with the exception of the string `#SBATCH`. `#SBATCH` is used to signify a Slurm directive. In order to comment out these, you need to put one more `#` in front of the `#SBATCH`.
- It is important to use capital letters for `#SBATCH`. Otherwise the line will be considered a comment, and ignored.

Let us go through the most commonly used arguments:

- **-A PROJ-ID**: The project that should be accounted. It is a simple conversion from the SUPR project id. You can also find your project account with the command `projinfo`. The PROJ-ID argument is of the form
  - hpc2nXXXX-YYY (HPC2N local project)
- **-N**: number of nodes. If this is not given, enough will be allocated to fullfill the requirements of -n and/or -c. A range can be given. If you ask for, say, 1-1, then you will get 1 and only 1 node, no matter what you ask for otherwise. It will also assure that all the processors will be allocated on the same node.

- **-n**: number of tasks.
- **-c**: cores per task. Request that a specific number of cores be allocated to each task. This can be useful if the job is multi-threaded and requires more than one core per task for optimal performance. The default is one core per task.

See `man sbatch` or the section on Job Submit file design for more commands for the script.

Overview

Processing of the job script

- When the job is submitted, the lines of the script file are scanned for directives.
- An initial line in the script that begins with the characters `"#!"` will be ignored at this point and scanning will start with the next line.
- Scanning will continue until the first executable line, that is a line that is not blank, not a directive line, nor a line whose first non-white space character is `"#"`.
- If directives occur on subsequent lines, they will be ignored.
- The remainder of the directive line consists of the options to slurm in the same syntax as they appear on the command line.
- The option character is to be preceded with the `"-"` character, or `"--"` in the long form.
- If an option is present in both a directive and on the command line, that option and its argument, if any, will be ignored in the directive. The command line takes precedence!
- If an option is present in a directive and not on the command line, that option and its argument, if any, will be processed as if it had occurred on the command line.

There are several examples of Slurm job submission files in a later section.

Slurm Submit File Design

To best use the resources with Slurm you need to have some basic information about the application you want to run.

Slurm will do its best to fit your job into the cluster, but you have to give it some hints of what you want it to do.

The parameters described below can be given directly as arguments to `srun` and `sbatch`.

If you don't give Slurm enough information, it will try to fit your job for best throughput (lowest possible queue time). This approach will not always give the best performance for your job (or, indeed, allow the job to run, in some cases).

To get the best performance, you will need to know the following:

- Your account
- The number of tasks
- The number of cores per task
- The number of tasks per node
- Memory usage
- The run/wallclock time
- The number of nodes
- Number of GPUs needed

Some extra parameters that might be usefull:

- Sending output to files
- Send mail on job changes
- Exclusive
- Constraints

For basic examples for different types, see the example section:

- Basic serial job
- Basic MPI job
- Basic OpenMP job
- Basic MPI + OpenMP job
- Accessing the GPU resources (Kebnekaise)
- Multiple jobs
- Job arrays

Some applications may have special needs, in order to get them running at full speed.

Look at the application specific pages (under software for more information about any such special requirements.

Some commonly used programs are listed below.

Important

If you do not use constraints to ask only for specific types of nodes, you may get any available node types that match your job's requirement.

Not all modules/versions exist on all node architectures! Make sure you check that the module you load in a batch script actually exist on the node(s) your job end up on (see Different parts of the batch system). If you are not contraining the node types, pick module(s) and versions that exist on all node types.

The regular login node (`kebnekaise.hpc2n.umu.se`) can be used to see the modules available on broadwell and skylake nodes.

The AMD login node (`kebnekaise-amd.hpc2n.umu.se`) can be used to see the modules available on the Zen3/Zen4 nodes.

First line in submit file

The submit file must start with:

```
#!/bin/bash
```

This is required for the module system to work. There are other possibilities, but this is the only one we fully support.

Your account (-A)

The account is your project id, this is mandatory.

Example (HPC2N):

```
#SBATCH -A hpc2nXXXX-YYY
```

Example (NAISS):

```
#SBATCH -A naissXXXX-YY-ZZZ
```

You can find your project id by running:

```
$ projinfo
```

The number of tasks (-n)

The number of tasks is for most usecases the number of processes you want to start. The default value is one (1).

An example could be the number of MPI tasks or the number of serial programs you want to start.

Example:

```
#SBATCH -n 28
```

The number of cores per task (-c)

If your application is multi threaded (OpenMP/...) this number indicates the number of cores each task can use.

The default value is one (1).

Example:

```
#SBATCH -c 14
```

The number of tasks per node (–ntasks-per-node)

If your application requires more than the maximum number of available cores in one node (for instance 28 on the skylake nodes on Kebnekaise) it might be wise to set the number of tasks per node, depending on your job. This is the (minimum) number of tasks allocated per node.

Remember

(The total number of cores) = (the number of tasks) x (the number of cores per task).

There are 28 cores per node on the Skylake nodes on Kebnekaise, so this is the maximum number of tasks per node for those nodes. The largemem nodes have 72 cores. On Kebnekaise the number of cores depend on which type of nodes you are running. For more information, see the Kebnekaise hardware page.

Note

If you don't set this option, Slurm will try to spread the task(s) over as few available nodes as possible.

As an example, on a Skylake node (28 cores), this can result in a job with 22 tasks on one node, and 6 on another.

If you let slurm spread your job it is more likely to start faster, but the performance of the job might be hurting.

If you are using more than 28 cores (regular skylake node on kebnekaise) and are unsure of how your application behaves, it is probably a good thing to put an even spread over the number of required nodes.

There is usually no need to tell Slurm how many nodes that your job needs. It will do the math.

Example:

```
#SBATCH --ntasks-per-node=24
```

Memory usage

RAM per core	
Kebnekaise (broadwell)	4460 MB
Kebnekaise (skylake)	6785 MB
Kebnekaise largemem	41666 MB
Kebnekaise (Zen3, CPU)	8020 MB
Kebnekaise (Zen3, with GPU)	10600 MB
Kebnekaise (Zen3, A100)	10600 MB
Kebnekaise (Zen4, CPU)	2516 MB
Kebnekaise (Zen4, with GPU)	6630 MB

Each core has a limited amount of memory available. If your job requires more memory than the default, you can allocate more cores for your task with `(-c)`.

If, for instance, you need 8000MB/task on a Kebnekaise Skylake node, set `"-c 2"`.

Example:

```
# I need 2 x 6785 MB (13570MB) of memory for my job.
#SBATCH -c 2
```

This will allocate two (2) cores with 6785 MB each. If your code is not multi-threaded (using only one core per task) the other one will just add its memory to your job.

If your job requires more memory / node on Kebnekaise, there are a limited number of nodes with 3072000MB memory, which you may be allowed to use (you apply for it as a separate resource when you make your project proposal in SUPR). They are accessed by selecting the largemem feature of the cluster. You do this by setting: `-C largemem`.

Example:

```
#SBATCH -C largemem
```

The run/wallclock time (--time, --time-min)

If you know the runtime (wall clock time) of your job, it is beneficial to set this value as accurately as possible.

Note

- Smaller jobs are more likely to fit into slots of unused space faster. Do not ask for (much) too long walltime.
- Please add some extra time to account for variances in the system. The job will end when the walltime is done, even if your calculation is not!
- The maximum allowed runtime of any job is seven (7) days.

The format is:

- D-HH:MM:SS (D=Day(s), HH=Hour(s), MM=Minute(s), SS=Second(s))

Example:

```
# Runtime limit 2 days, 12hours
#SBATCH --time 2-12:00:00
```

You can also use the `--time-min` option to set a minimum time for your job. If you use this, Slurm will try to find a slot with more than `--time-min` and less than `--time`. This is useful if your job does periodic checkpoints of data and can restart from that point. This technique can be used to fill openings in the system, that no big jobs can fill, and so allows for better throughput of your jobs.

Example:

```
# Runtime limit 2 days, 12hours
#SBATCH --time 2-12:00:00
#
# Minimum runtime limit 1 days, 12hours
#SBATCH --time-min 1-12:00:00
```

The number of nodes (-N)

It is possible to set the number of nodes that slurm should allocate for your job.

This should only be used together with `--ntasks-per-node` or with `--exclusive`.

But in almost every case it is better to let slurm calculate the number of nodes required for your job, from the number of tasks, the number of cores per task, and the number of tasks per node.

Number of GPUs needed

Note

- Your project need to have time on the GPU nodes to use them, as they are considered a separate resource.
- To request GPU resources one has to include a gpu request in the submit file. The general format is:

  ```
  #SBATCH --gpu=TYPE-OF-CARD:x
  ```

  where TYPE-OF-CARD is one of the available GPU types, v100, a40, a6000, l40s, h100, or not used at all to just get any free GPU.

Sending output to files (--output/--error)

The output (`stdout`) and error (`stderr`) output from your program can be collected with the help of the `--output` and `--error` options to `sbatch`.

Example:

```
# Send stderr of my program into <jobid>.error
#SBATCH --error=%J.error

# Send stdout of my program into <jobid>.output
#SBATCH --output=%J.output
```

The files in the example will end up in the working directory of you job.

In the above example, we use the Slurm environment variable `%J`, which contains the JOB-ID. This is very useful since that means we will always get a new output file or error file, and we do not risk overwriting a previous output.

Send mail on job changes (--mail-type)

Slurm can send mail to you when certain event types occur. Valid type values are: BEGIN, END, FAIL, REQUEUE, and ALL (any state change).

Example:

```
# Send mail when job ends
#SBATCH --mail-type=END
```

Warning

We recommend that you do NOT include a command for the batch system to send an email when the job has finished, particularly if you are running large amounts of jobs. The reason for this is that many mail servers have a limit and may block accounts (or domains) temporarily if they send too many mails.

Instead use

```
scontrol show job <jobid>
```

or

```
squeue -l -u <username>
```

to see the status of your job(s).

Exclusive (--exclusive)

In some use-cases it is usefull to ask for the complete node (not allowing any other jobs, including your own, to share).

`--exclusive` can be used with `-N` (number of nodes) to get all the cores, and memory, on the node(s) exclusively for your job.

Example:

```
# Request complete nodes
#SBATCH --exclusive
```

Constraints

If you need to run only on a specific type of nodes, then you can do this with the `--constraint` flag:

- `#SBATCH --constraint=skylake`
- `#SBATCH --constraint=GPU_DP`

Common programs which have special requirements

- AMBER
- COMSOL
- Gaussian
- MATLAB
- VASP

Basic examples of job submission files ¶

The examples below generally assume you are submitting the job from the same directory your program is located in - otherwise you need to give the full path.

Note

Submit with: `sbatch MYJOB.sh`, where MYJOB.sh is whatever name you gave your submit script.

Remember - the submission files/scripts and all programs called by them, must be executable!

The project ID `hpc2nXXXX-YYY` is used in the examples. Please replace it with your own project ID.

Serial jobs

Serial job on Kebnekaise, compiler toolchain 'foss/2021b'

```
#!/bin/bash
# Project id - change to your own!
#SBATCH -A hpc2nXXXX-YYY
# Asking for 1 core
#SBATCH -n 1
# Asking for a walltime of 5 min
#SBATCH --time=00:05:00
# Purge modules before loading new ones in a script.
ml purge > /dev/null 2>&1
ml foss/2021b

./my_serial_program
```

Running two executables per node (two serial jobs)

```
#!/bin/bash
#SBATCH -A hpc2nXXXX-YYY
#SBATCH -n 2
#SBATCH --time=00:30:00

# Clear the environment from any previously loaded modules
module purge > /dev/null 2>&1

# Load the module environment suitable for the job - here foss/2021b
module load foss/2021b

# Use '&' to start the first job in the background
srun -n 1 ./job1 &
srun -n 1 ./job2

# Use 'wait' as a barrier to collect both executables when they are done. If not the batch
wait
```

The scripts job1 and job2 could be any script or executable that is a serial code. The drawback with this examnple is that any output from job1 or job2 will get mixed up in the batch jobs output file. You can handle this by naming the output for each of the jobs or by job1 and job2 being programs that create output to file directly.

Running two executables per node (two serial jobs) - separate output files

```
#!/bin/bash
#SBATCH -A hpc2nXXXX-YYY
#SBATCH -n 2
#SBATCH --time=00:30:00

# Clear the environment from any previously loaded modules
module purge > /dev/null 2>&1

# Load the module environment suitable for the job - here foss/2021b
module load foss/2021b

# Use '&' to start the first job in the background
srun -n 1 ./job1 > myoutput1 2>&1 &
srun -n 1 ./job2 > myoutput2 2>&1

# Use 'wait' as a barrier to collect both executables when they are done. If not the batch
wait
```

Naming output/error files. Serial program

```
#!/bin/bash
#SBATCH -A hpc2nXXXX-YYY
#SBATCH -n 1
#SBATCH --time=00:05:00
#SBATCH --error=job.%J.err
#SBATCH --output=job.%J.out

# Clear the environment from any previously loaded modules
module purge > /dev/null 2>&1

# Load the module environment suitable for the job - here foss/2021b
module load foss/2021b

# And finally run the job
./my_program
```

Normally, Slurm produces one output file called slurm-JOBID.out containing the combined standard output and errors from the run (though files created by the program itself will of course also be created). If you wish to rename the output and error files, and get them in separate files, you can do something similar to this example.

Using the environment variable `%J` (contains the JOBID) in your output/error files will ensure you get unique files with each run and so avoids getting the files overwritten.

OpenMP jobs

This example shows a 28 core OpenMP Job (maximum size for a regular Skylake node on Kebnekaise).

```
#!/bin/bash
# Example with 28 cores for OpenMP
#
# Project/Account - change to your own
#SBATCH -A hpc2nXXXX-YYY
#
# Number of cores
#SBATCH -c 28
#
# Runtime of this jobs is less then 12 hours.
#SBATCH --time=12:00:00
#
# Clear the environment from any previously loaded modules
module purge > /dev/null 2>&1

# Load the module environment suitable for the job - here foss/2021b
module load foss/2021b

# Set OMP_NUM_THREADS to the same value as -c
# with a fallback in case it isn't set.
# SLURM_CPUS_PER_TASK is set to the value of -c, but only if -c is explicitly set
if [ -n "$SLURM_CPUS_PER_TASK" ]; then
   omp_threads=$SLURM_CPUS_PER_TASK
else
omp_threads=1
fi
export OMP_NUM_THREADS=$omp_threads

./openmp_program
```

If you wanted to run the above job, but only use some of the cores for running on (to perhaps use more memory than what is available on 1 core), you can submit with

```
sbatch -c 14 MYJOB.sh
```

MPI jobs

MPI job on Kebnekaise, compiler toolchain 'foss/2021b'

```
#!/bin/bash
#SBATCH -A hpc2nXXXX-YYY
#SBATCH -n 14
#SBATCH --time=00:05:00
##SBATCH --exclusive

module purge > /dev/null 2>&1
ml foss/2021b

srun ./my parallel program
```

The `--exclusive` flag means your job will not share the node with any other job (including your own). It is commented out in the above example, but you can remove one of the `#`'s to activate it if you, for instance, need the entire bandwidth of the node.

Note that this will mean 1) it will likely take longer for the job to start as you need to wait until a full node is available 2) your project will be accounted (time) for the entire node.

Output from the above MPI job on Kebnekaise, run on 14 cores

Note: 14 cores is one NUMA island

```
b-an01 [~/slurm]$ cat slurm-15952.out
The following modules were not unloaded:
(Use "module --force purge" to unload all):
1) systemdefault 2) snicenvironment
Processor 12 of 14: Hello World!
Processor 5 of 14: Hello World!
Processor 9 of 14: Hello World!
Processor 4 of 14: Hello World!
Processor 11 of 14: Hello World!
Processor 13 of 14: Hello World!
Processor 0 of 14: Hello World!
Processor 1 of 14: Hello World!
Processor 2 of 14: Hello World!
Processor 3 of 14: Hello World!
Processor 6 of 14: Hello World!
Processor 7 of 14: Hello World!
Processor 8 of 14: Hello World!
Processor 10 of 14: Hello World!
```

MPI job on Kebnekaise, compiler toolchain 'foss/2021b' and more memory

```
#!/bin/bash
#SBATCH -A hpc2nXXXX-YYY
#SBATCH -N 2
#SBATCH --time=00:05:00
#SBATCH --exclusive
# This job example needs 8GB of memory per mpi-task (=mpi ranks, =cores)
# and since the amount of memory on the regular Skylak nodes is
# 6750MB per core, when using all 28 cores we have to use 2 nodes and
# only half the cores
#SBATCH -c 2
# Make sure we run on a Skylake node
#SBATCH --constraint=skylake

module purge > /dev/null 2>&1
ml foss/2021b

srun ./my parallel program
```

The `--exclusive` flag means your job will not share the node with any other job (including your own). It is commented out in the above example, but you can remove one of the `#`'s to activate it if you, for instance, need the entire bandwidth of the node. Using `--exclusive` also ensures that all the cores on the node will be available to the job so you could use the memory from all of them for instance.

Note that using this flag will mean 1) it will likely take longer for the job to start as you need to wait until a full node is available 2) your project will be accounted (time) for the entire node.

Running fewer MPI tasks than the cores you have available

```
#!/bin/bash
# Account name to run under
#SBATCH -A hpc2nXXXX-YYY
# Give a sensible name for the job
#SBATCH -J my_job_name
# ask for 4 full nodes
#SBATCH -N 4
#SBATCH --exclusive
# ask for 1 day and 3 hours of run time
#SBATCH -t 1-03:00:00

# Clear the environment from any previously loaded modules
module purge > /dev/null 2>&1

# Load the module environment suitable for the job - here foss/2021b
module load foss/2021b

# run only 1 MPI task/process on a node, regardless of how many cores the nodes have.
srun -n 4 --ntasks-per-node=1 ./my_mpi_program
```

Hybrid MPI/OpenMP jobs

This example shows a hybrid MPI/OpenMP job with 4 tasks and 28 cores per task.

```
#!/bin/bash
# Example with 4 tasks and 28 cores per task for MPI+OpenMP
#
# Project/Account - change to your own
#SBATCH -A hpc2nXXXX-YYY
#
# Number of MPI tasks
#SBATCH -n 4
#
# Number of cores per task (regular Skylake nodes have 28 cores)
#SBATCH -c 28
#
# Runtime of this job example is less then 12 hours.
#SBATCH --time=12:00:00
#
# Clear the environment from any previously loaded modules
module purge > /dev/null 2>&1

# Load the module environment suitable for the job - here foss/2021b
module load foss/2019a

# Set OMP_NUM_THREADS to the same value as -c
# with a fallback in case it isn't set.
# SLURM_CPUS_PER_TASK is set to the value of -c, but only if -c is explicitly set
if [ -n "$SLURM_CPUS_PER_TASK" ]; then
   omp_threads=$SLURM_CPUS_PER_TASK
else
   omp_threads=1
fi
export OMP_NUM_THREADS=$omp_threads

# Running the program
srun --cpu_bind=cores ./mpi_openmp_program
```

Multiple jobs

This section show examples of starting multiple jobs within the same submit file.

Starting more than one serial job in the same submit file

```
#!/bin/bash
#SBATCH -A hpc2nXXXX-YYY
#SBATCH -n 5
#SBATCH --time=00:15:00

module purge > /dev/null 2>&1
ml foss/2021b

srun -n 1 ./job1.batch &
srun -n 1 ./job2.batch &
srun -n 1 ./job3.batch &
srun -n 1 ./job4.batch &
srun -n 1 ./job5.batch
wait
```

All the jobs are serial jobs, and as they are started at the same time, you need to make sure each has a core to run on. The time you ask for must be long enough that even the longest of the jobs have time to finish. Remember the `wait` at the end. If you do not include this, then the batch job will finish when the first of the jobs finishes instead of waiting for all to finish. Alsi notice the `&` at the end of each command to run a job.

Multiple Parallel Jobs Sequentially

```
#!/bin/bash
#SBATCH -A hpc2nXXXX-YYY
#SBATCH -c 28
# Remember to ask for enough time for all jobs to complete
#SBATCH --time=02:00:00

module purge > /dev/null 2>&1
ml foss/2021b

srun -n 14 ./a.out
srun -n 14 -c 2 ./b.out
srun -n 28 ./c.out
```

Here the jobs are parallel, but they run sequentially which means that you only need enough cores that you have enough for the job that uses the most cores. In this example, the first uses 14, the second 28 (because each task uses 2 cores), the third uses 28. So you need 28 here.

We assume the programs `a.out`, `b.out`, and `c.out` all generate their own output files. Otherwise the output will go to the `slurm-JOBID.out` file.

Note: -n tasks per default uses 1 core per task, unless you use -c to say you want more cores per task.

Multiple Parallel Jobs Sequentially with named output copied elsewhere

```
#!/bin/bash
#SBATCH -A hpc2nXXXX-YYY
#SBATCH -c 28
# Remember to ask for enough time for all jobs to complete
#SBATCH --time=02:00:00

module purge > /dev/null 2>&1
ml foss/2021b

srun -n 14 ./a.out > myoutput1 2>&1
cp myoutput1 /proj/nobackup/MYSTORAGE/mydatadir
srun -n 14 -c 2 ./b.out > myoutput2 2>&1
cp myoutput2 /proj/nobackup/MYSTORAGE/mydatadir
srun -n 28 ./c.out > myoutput3 2>&1
cp myoutput3 /proj/nobackup/MYSTORAGE/mydatadir
```

Here the jobs we run and the needed cores and tasks are the same as in the example above, but the output of each one is handled in the script, sending it to a specific output file.

In addition, this example also shows how you can copy the output to somewhere else (the directory `mydatadir` located under your project storage which here is called `MYSTORAGE`).

Multiple Parallel Jobs Simultaneously

```
#!/bin/bash
#SBATCH -A hpc2nXXXX-YYY
# Total number of cores the jobs need
#SBATCH -n 56
#SBATCH --time=02:00:00

module purge > /dev/null 2>&1
ml foss/2021b

srun -n 14 --cpu bind=cores --exclusive ./a.out &
srun -n 28 --cpu bind=cores --exclusive ./b.out &
srun -n 14 --cpu bind=cores --exclusive ./c.out &
wait
```

In this example I am starting 3 jobs within the same jobs. You can put as many as you want, of course. Make sure you ask for enough cores that all jobs can run at the same time, and have enough memory. Of course, this will also work for serial jobs - just remove the srun from the command line.

Remember to ask for enough time for all of the jobs to complete, even the longest.

Job arrays

```
#!/bin/bash
#SBATCH -A hpc2nXXXX-YYY
#SBATCH -j my_array_job
#SBATCH -n 1
#SBATCH -t 01:00:00
# Submit a job array with index values between 1 and 10
#SBATCH --array=1-10

./myapplication $SLURM_ARRAY_TASK_ID
```

Job arrays for a Python job

hello-world-array.py | Batch script

GPU Jobs

GPU job on Kebnekaise, asking for 2 V100 cards

```
#!/bin/bash
#SBATCH -A hpc2nXXXX-YYY
# Expected time for job to complete
#SBATCH --time=00:10:00
# Number of GPU cards needed. Here asking for 2 V100 cards
#SBATCH --gpus-per-node=v100:2

module purge > /dev/null 2>&1
ml foss/2023b CUDA/12.4.0

./my-program
```

Expiring projects

If you have jobs still in the queue when your project expires, your job will not be removed, but they will not start to run.

You will have to remove them yourself, or, if you have a new project, you can change the project account for the job with this command

```
scontrol update job=<jobid> account=<newproject>
```

Job arrays - Slurm

Job arrays in Slurm lets you run many jobs with the same job script. If you have many data files that you would normally have processed in multiple jobs, job arrays could be an alternative way to instead generate many job scripts for each run and submit it one by one.

Example

```
#!/bin/bash
#SBATCH -A hpc2nXXXX-YYY
#SBATCH -j my_array_job
#SBATCH -n 1
#SBATCH -t 01:00:00
# Submit a job array with index values between 1 and 10
#SBATCH --array=1-10

./myapplication $SLURM_ARRAY_TASK_ID
```

Job arrays are only supported for batch jobs and the array index values are specified using the `--array` or `-a` option of the `sbatch` command. The option argument can be specific array index values, a range of index values, and an optional step size (see next examples).

Note

The minimum index value is zero and the maximum value is a Slurm configuration parameter (`MaxArraySize` minus one). Jobs which are part of a job array will have the environment variable `SLURM_ARRAY_TASK_ID` set to its array index value.

Example

```
#!/bin/bash
#SBATCH -A hpc2nXXXX-YYY
#SBATCH -j my_array_job
#SBATCH -n 1
#SBATCH -t 01:00:00
# Submit a job array with index values of 1, 3, 5 and 7
#SBATCH --array=1,3,5,7

./myapplication $SLURM_ARRAY_TASK_ID
```

Example

```
#!/bin/bash
#SBATCH -A hpc2nXXXX-YYY
#SBATCH -j my_array_job
#SBATCH -n 1
#SBATCH -t 01:00:00
# Submit a job array with index values between 1 and 7
# with a step size of 2 (i.e. 1, 3, 5 and 7)
#SBATCH --array=1-7:2

./myapplication $SLURM_ARRAY_TASK_ID
```

Job dependencies - Slurm

A job can be given the constraint that it only starts after another job has finished.

Example

Assume we have two Jobs, A and B. We want Job B to start after Job A has successfully completed.

First we start Job A (with submit file jobA.sh) by submitting it via `sbatch`:

```
$ sbatch jobA.sh
```

Making note of the assigned job-id for Job A (JOBID-JobA), we then submit Job B (with submit file jobB.sh) with the added condition that it only starts after Job A has successfully completed:

```
$ sbatch --dependency=afterok:JOBID-JobA jobB.sh
```

If we want Job B to start after several other Jobs have completed, we can specify additional jobs, using a ':' as a delimiter:

```
$ sbatch --dependency=afterok:JOBID-JobA:JOBID-JobC:JOBID-JobD jobB.sh
```

We can also tell slurm to run Job B, even if Job A fails, like so:

```
sbatch --dependency=afterany:JOBID-JobA jobB.sh
```

For more information, consult the `man` page for `sbatch`.

Queue and job status

To see status of partitions and nodes, use

```
$ sinfo
```

To get the status of all Slurm jobs

```
$ squeue
```

To get the status of all your Slurm jobs

```
$ squeue --me
```

Get detailed information about an individual job

```
$ scontrol show job JOBID
```

You get the JOBID either when you submit the job with `sbatch`, or from checking with `squeue -u USERNAME`.

Job Submission

There are three ways to run a job with Slurm: command line, job submission file, and interactively.

Command line ¶

A job can simply be submitted from the command line with `srun`.

Example

```
$ srun -A hpc2nXXXX-YYY -N 2 --exclusive --time=00:30:00 my_program
```

- This example asks for exclusive use of two nodes to run the program `my_program`
- It also asks for a time limit of 30 minutes.
- Since the number of tasks has not been specified, it assumes the default of one task per node.
- Note that the `--exclusive` parameter guarantees no other jobs will run on the allocated nodes. Without the `--exclusive` parameter, Slurm would only allocate the minimum assignable resources for each node.
- The job is run in the project hpc2nXXXX-YYY (change to your own project).

When submitting the job this way, you give all the commands on the command line, and then you wait for the job to pass through the job queue, run, and complete before the shell prompt returns, allowing you to continue typing commands.

This is a good way to run quick jobs and get accustomed to how Slurm works, but it is not the recommended way of running longer programs, or MPI programs; these types of jobs should run as a batch job with a Job Submission File. This also has the advantage of letting you easily see what you did last time you submitted a job.

Job Submission File

Instead of submitting the program directly to Slurm with srun from the command line, you can submit a batch job with sbatch. This has the advantage of you not having to wait for the job to start before you can use your shell prompt.

Before submitting a batch job, you first write a **job submission file**, which is an executable shell script. It contains all the environment setup, commands and arguments to run your job (other programs, MPI applications, srun commands, shell commands, etc). When your job submission file

is ready, you submit it to the job queue with `sbatch`. `sbatch` will add your job to the queue, returning immediately so you can continue to use your shell prompt. The job will run when resources become available.

When the job is complete, you will, if not specified otherwise with directives, get a file named `slurm-JOBID.out` containing the output from your job. This file will be placed in the same directory that you submitted your job from. `JOBID` is the id of the job, which is returned when you submitted the job (or found from `squeue -u USERNAME`.

The following example submits a job to the default batch partition:

```
$ sbatch jobXsubmit.sh
```

Interactive

If you would like to allocate resources on the cluster and then have the flexibility of using those resources in an interactive manner, you can use the command `salloc` to allow interactive use of resources allocated to your job. This can be useful for debugging, in addition to debugging tools like DDT (which uses normal batch jobs and not interactive allocations).

First, you make a request for resources with `salloc`, like in this example:

```
$ salloc -A hpc2nXXXX-YYY -n 4 --time=1:30:00
```

The example above will allocate resources for up to 4 simultaneous tasks for 1 hour and 30 minutes. You need to give your project id as well (change hpc2nXXXX-YYY to your own project id).

Your request enters the job queue just like any other job, and `salloc` will tell you that it is waiting for the requested resources. When `salloc` tells you that your job has been allocated resources, you can interactively run programs on those resources with `srun` for as long as the time you asked for. The commands you run with `srun` will then be executed on the resources your job has been allocated.

Note

After `salloc` tells you that your job resources have been granted, you are still using a shell on the login node. You must submit all commands with `srun` to have them run on your job's allocated resources. Commands run without srun will be executed on the login node.

This is demonstrated in the examples below.

1 node, resources for 4 parallel tasks, on a Kebnekaise compute node

Here we run without prefacing with `srun` and as you can see we run on the login node:

```
b-an01 [~]$ salloc -n 4 --time=1:00:00 -A hpc2nXXXX-YYY
salloc: Pending job allocation 10248860
salloc: job 10248860 queued and waiting for resources
salloc: job 10248860 has been allocated resources
salloc: Granted job allocation 10248860
b-an01 [~]$  echo $SLURM_NODELIST
b-cn0206
b-an01 [~]$ srun hostname
b-cn0206.hpc2n.umu.se
b-cn0206.hpc2n.umu.se
b-cn0206.hpc2n.umu.se
b-cn0206.hpc2n.umu.se
b-an01 [~]$ hostname
b-an01.hpc2n.umu.se
```

2 nodes, resources for 4 parallel tasks, on Kebnekaise

Here we run with `srun` and that means we run on the allocated compute node:

```
b-an01 [~]$ salloc -N 2 -n 4 --time=00:10:00 hpc2nXXXX-YYY
salloc: Pending job allocation 10248865
salloc: job 10248865 queued and waiting for resources
salloc: job 10248865 has been allocated resources
salloc: Granted job allocation 10248865
b-an01 [~]$ echo $SLURM_NODELIST
b-cn[0205,0105]
b-an01 [~]$ srun hostname
b-cn0205.hpc2n.umu.se
b-cn0205.hpc2n.umu.se
b-cn0205.hpc2n.umu.se
b-cn0105.hpc2n.umu.se
b-an01 [~]$
```

Slurm determined where to allocate resources for the 4 tasks on the 2 nodes. In this case, three tasks were run on b-cn0205, and one on b-cn0105. If needed, you can control how many tasks you want to run on each node with `--ntask-per-node=NUMBER`.

Note

Another type of interactive runs is to run through Jupyter. You can read more about that in the Jupyter on Kebnekaise tutorial.