

**Using the job submission system on Alarik and Erik**

Joachim Hein, Jonas Lindemann, Anders Sjöström, Magnus Ullner

*Document under active development - Check back frequently!*

A more in-depth guide to the job submission system on Alarik and Erik.

# Contents

[Contents](#h.za6xtq2gyps4)

[SLURM - the batch system on Alarik and Erik](#h.gcjytusu3x17)

[Job submission](#h.qpdylg2d54mp)

[First example for a job submission](#h.ixwy1oqi0kbg)

[The job script and sbatch](#h.hjzswlkcoi3g)

[The three parts of a job script](#h.pnv7n4e7zfkv)

[Resource statements for all jobs](#h.szaf2awyxrer)

[Walltime](#h.8ndyyr6y81ve)

[Job naming](#h.hrjlj0vfjjda)

[Specifying a project for users with multiple projects](#h.f3gibfpxd1ke)

[Specifying memory requirements](#h.dcd20vpuu4k1)

[Controlling job output](#h.jvtofv3j1o5c)

[Notification](#h.f95hqwba3q2s)

[Job dependencies](#h.lx2kqjsa4eh5)

[Test queue](#h.eiu9sjdaomjb)

[Extra fat nodes on Alarik](#h.707oe99acfmi)

[Fat, extra fat and MIC nodes on Erik](#h.c6zu7j3ztwaa)

[Resource statements for multiprocessor](#h.66a7sor54qag)

[Terminology around nodes, processors, cores, tasks](#h.9w1xr1vndcr)

[Outline: Resource requests for multiprocessor jobs](#h.56jto5jnyjsx)

[Exclusive node access](#h.fp8s4npfppo)

[Specifying the number of nodes required for the job](#h.7snigg89ot0s)

[Specifying the number of tasks per node](#h.er0uychvqu33)

[Specifying the number of threads for a shared-memory job](#h.smijzro848m)

[Resource statements for hybrid programs using distributed and shared memory](#h.2sxjqtv36oim)

[Specifying the number of cores to be required by the job](#h.crlix3van4jr)

[Program execution environment](#h.9z6d7xbmrbq8)

[Job execution environment](#h.rxse0tthfj6u)

[Compiler modules](#h.bbqwxxfguymv)

[SLURM variables](#h.nl10c9t5z9ye)

[SNIC variables](#h.i2zkcn931bag)

[Using the node local disks to improve I/O performance](#h.q6fievrilbxd)

[Launching MPI jobs in OpenMPI](#h.w592soyy5mgv)

[Submitting, monitoring and manipulating jobs in SLURM](#h.dn3gxz9xnme2)

[Submitting with sbatch](#h.396houio4kin)

[Starting executables within SLURM with srun](#h.9qo9ttnpacsf)

[Monitoring with squeue](#h.yuzzje2w0q5i)

[Terminating jobs with scancel](#h.6sdhks1csphx)

[Example job scripts](#h.z8ycrofm6rsx)

[Job scripts using the node local disk](#h.1ecx43je7wqd)

[Basic run script](#h.39hwr05aah26)

[Version for codes requiring more memory than 2000 MB](#h.fxrow6rptdn)

[Running multiple serial jobs within a single job submission](#h.kcdsziq5el1s)

[The master script](#h.k6uun6op1udh)

[The worker script](#h.ip8za35kb7mn)

[Monitoring the progress of your multi-job submission](#h.q6q6tbtg3rnz)

[MPI job using 16 tasks per node](#h.bpcdd2petdlh)

[Modifications required for file I/O on all nodes](#h.7e8bw4nub67x)

[MPI jobs using fewer than 16 tasks per node](#h.60zd1wbg775f)

[OpenMP jobs using shared memory](#h.wkdd1xqoth9f)

[Thread binding for OpenMP codes](#h.4yniiqbls3o2)

[Thread binding with the GNU compilers](#h.jsco3uzba0zc)

[Thread binding with the open64 compiler](#h.h9srrpq6zh39)

[Thread binding with the Intel compiler](#h.co8b7n80sekz)

[Hybrid-jobs using threads within an MPI framework](#h.swnlr2ojpdks)

[Things to try for MPI-OpenMP hybrids with 16 threads per task](#h.rj9yj5fcwak0)

[Interactive access to compute nodes](#h.8xpfkrdxslrt)

[Starting an interactive session](#h.2hev4fwv4zx)

[Modules and environment variables](#h.vhlqikdwx0x)

[Known issues with the interactive command](#h.3ge5ixcnrxse)

# SLURM - the batch system on Alarik and Erik

On a modern HPC system efficient management of the compute resources is absolutely crucial for the system to perform. Alarik and Erik are the first Lunarc systems to deploy SLURM (**S**imple **L**inux **U**tility for **R**esource **M**anagement) as resource manager. For your program to be executed you have to describe to SLURM the resources required by your program, the name of your program and the command line arguments your program may require. SLURM also allows monitoring and manipulation of the progress of your programs execution.

This document contains two key parts. The [first part](#id.asft9ff8nuhu) describes in-depth the job submission system and its options. The [second part](#id.6w4ezxvvga34) gives example scripts for the most common use cases. They hopefully serve as a good starting point when creating submission scripts fitting your needs and requirements.

# Job submission

## First example for a job submission

### The job script and sbatch

You register your program with SLURM for execution using the **sbatch** command. This is done easiest by using a *job description file*. The job description file is also know as a *job script*.

A very simple job script, looks as follows:

#!/bin/sh

#SBATCH -t 00:05:00

echo "hello"

Write this into a file. In the following we assume the file is named echo\_script.sh, but in principle any name will do. You can now send the script for execution using the sbatch command. This will execute the “program” echo on the backend.

sbatch echo\_script.sh

This should deliver a screen output similar to

[fred@alarik Serial]$ sbatch echo\_script.sh

Submitted batch job 7185

Where 7185 is the job number assigned by SLURM. Once your job has executed you will find a file slurm-7185.out in your directory which contains the output and error messages from your program.

### The three parts of a job script

The example echo\_script.sh shows the three parts every job script requires

1. Shell specification
2. Resource statement
3. Body containing a UNIX script

In our example each part consists of a single line. The first line of our example contains the shell specification, in most cases the sh-shell as used here is just fine. The second line starting with #SBATCH specifies the resources needed. In our case it asks for 10 minutes of computer time. If the jobs hasn’t finished after that time, SLURM will terminate it. Job scripts typically contain more than one of these statements, specifying e.g. more than one processor or more memory. The most commonly used resource statements at Lunarc will be explained below. The resource statements are followed by a list of programs and UNIX commands to be executed on the system. This is actually a normal UNIX script and everything you can do in a UNIX script can be done here as well. In our example the script consists out of the UNIX echo command.

## Resource statements for all jobs

We now describe a number of statements which are most commonly used to specify resource requirements for all kind of jobs. Refer to “man sbatch” for more information.

### Walltime

The walltime attribute specifies the time requested for completing the job. The time is not cpu-time but the total time, as measured by a normal clock. In the previous example the time requested was 0 hours 5 minutes and 0 seconds. Walltime is specified in seconds or using the following notation:

Hours:Minutes:Seconds

If your calculation hasn’t finished once the specified time has elapsed, SLURM will terminate your job. It is therefore **good practise** to specify a bit more time than you anticipate your job to take. This makes sure that you still get your results, even the jobs is slowed by some interference, e.g. waiting for a write to a shared file system to finish. However don’t specify excessive extra time. Due to scheduling constraints, jobs asking for less time will typically spend less time in the queue, waiting for their execution. This also provides safety against depletion of your allocation. If, e.g., your job hangs, SLURM will terminate your job and the project will be charged less time if the walltime margin is not excessive.

To specify your walltime requirements write a statement like

#SBATCH -t 00:10:00

into your job script.

The maximum walltime for any job on Alarik is 168h, which is the same as 7 days. On Erik the maximum walltime for any job is 48h.

### Job naming

All jobs are given both a job identifier and a name, for easier identification in the batch-system. The default name given to a job is the file name of the submit script, which can make it difficult to identify your job, if you use a standard name for your submit scripts. You can give your job a name from inside the script by using the -J option:

#SBATCH -J parameterTest

This will name your job “parameterTest”.

### Specifying a project for users with multiple projects

Most users are members of only one project. These users do not need to specify a project in their in their submission script. The Lunarc set-up will automatically use that project for accounting.

A few users are members of more than project. In this case the system would not know which project to charge for the run, so you need to specify the project using the -A option:

#SBATCH -A snic2015-x-xxx

Replace the “snic2015-x-xxx” with the string naming your project. You can inquire the correct string using the projinfo command or the SUPR system.

### Specifying memory requirements

Alarik has 32 GB of memory installed on the small memory nodes and 64 GB of memory on the large memory nodes. The default memory request per core on the system is 2000 MB (a sixteenth of 32GB). If more then 2000 MB per core is needed it has to be requested explictly with the **--mem-per-cpu** option of sbatch. In this case you also have to request allocation on a large memory node using the **-C mem64GB** option of sbatch. The following show an example how to request 4000 MB or main memory per compute core used:

#SBATCH -C mem64GB

#SBATCH --mem-per-cpu=4000

When requesting more than 2000 MB of memory, your jobs may spend a longer time in the queue, waiting for execution, since it needs to wait for run-slot(s) on the large memory nodes to become available. When requesting more then 4000 MB per processing core, your jobs will be charged at a higher rate. In this case some processing cores have to remain idle since you are using more than your fair share of memory.

Erik has 64 Gb of memory on the standard nodes. Each node has two CPUs with eight cores each. The default memory request per core is therefore 4000 MB of memory. As in the case of Alarik, if more than 4000MB of memory per core is needed it has to be described as above.

### Controlling job output

By default, the output which your job writes to stdout and stderr is written to a file named

slurm\_%j.out

The %j in the file name will be replaced by the jobnumber SLURM assigns to your job. This ensures that the output file from your job is unique and different jobs do not interfere with each other's output file.

In many cases the default file name is not convenient. You might want to have a file name which is more descriptive of the job that is actually running - you might even want to include important meta-data, such as physical parameters, into the output filename(s). This can be achieved by using the -o and -e options of sbatch. The -o option specifies the file containing the stdout and the -e option the file containing the stderr. It is good practise to include the %j string into the filenames. That will prevent jobs from overwriting each other's output files. The following gives an example:

#SBATCH -o calcflow\_m1\_%j.out

#SBATCH -e calcflow\_m1\_%j.err

You can give the same filename for both options to get stdout and stderr written to the same file.

### Notification

SLURM on the systems can send you email if the status of your job changes as it progresses through the job queue. To use this feature you need to specify the email address using the --mail-user option and specify the event you want to get notified about using the --mail-type option. The following

#SBATCH --mail-user=fred@institute.se

#SBATCH --mail-type=END

Will send an email to the address fred@institute.se once the job has ended. Valid type values, selecting the event you can get notified about, are BEGIN, END, FAIL, REQUEUE, and ALL (any state change).

### Job dependencies

To describe job dependencies, use the -d option of sbatch. This is particularly useful for job dependencies, in workflows.

To illustrate this consider the following example. You require a serial job to create a mesh for your simulation. Once this has finished, you want to start a parallel job, which uses the mesh. You first submit the mesh creation job using sbatch

[fred@alarik Simcode]$ sbatch run\_mesh.sh

Submitted batch job 8042

As discussed, sbatch returns you a jobid, 8042 in this example. You use this to declare your dependency when submitting the simulation job to the queue

[fred@alarik Simcode]$ sbatch -d afterok:8042 run\_sim.sh

Submitted batch job 8043

When using squeue to monitor job 8043, this should now be in status pending (PD) with the reason of dependency. Another common use case for this functionality is a simulation requiring many days of computer times being split into a number of submissions.

### Test queue

To run short tests, it is possible to request extra high priority on Alarik with the help of

#SBATCH --qos=test

For one such job, the maximum walltime is 1 h and the maximum number of nodes is two and a user is only allowed to run two such jobs simultaneously. A system of floating reservations is used to free two nodes every second hour between 8.00 and 20.00 to reduce the queue time for test jobs. The way it works also means that the shorter the test job, the more likely it is to start sooner rather than later. It is not allowed to use qos=test for series of production runs.

On Erik there is one two-GPU node reserved for tests in a partition of its own, which is specified with

#SBATCH -p test

Like on Alarik, the maximum walltime is 1 h.

### Extra fat nodes on Alarik

Alarik has four nodes with 48 cores and 128 GB memory. To access them, the partition extra has to be specified

#SBATCH -p extra

Furthermore, the amount of memory per requested core also needs to be given.

#SBATCH --mem-per-cpu=<memory in MB>

Otherwise, the default value of 2 000 MB will be set at and if more is used, slurm can kill the job. For example, to run on the 48 cores of a single node and use a total of 128 000 MB of memory (for more on multiprocessor statements, see the next section):

#SBATCH -N 1

#SBATCH --tasks-per-node=48

#SBATCH --mem-per-cpu=3000

#SBATCH -p extra

### Fat, extra fat and MIC nodes on Erik

Erik has 7 nodes with 4 GPUs (and 96 GB of memory). To access them, the partition fat has to be specified

#SBATCH -p fat

One node is equipped with 8 GPUs (and 96 GB of memory), which is in partition extra

#SBATCH -p extra

There is also one node with two Xeon Phi (MIC) cards in the partition mic

#SBATCH -p mic

There is also one node with two Nvidia K80 cards in the partition new

#SBATCH -p new

If no -p option is specified, normal nodes with two Nvidia K20 cards will be allocated to the job.

## Resource statements for multiprocessor

In HPC it is very common to have many processing elements working on a job. The extra processing power can be utilised to process large problems beyond the capabilities of a single processing element. It can also be used to swiftly perform a number of calculations within a single job submission.

### Terminology around nodes, processors, cores, tasks

There is a a lot of structure within modern HPC equipment. For the purposes of this user guide we will stick to the following terminology:

|  |  |  |  |
| --- | --- | --- | --- |
| **Term** | **Explanation** | **Number on Alarik** | **Number on Erik** |
| Node | A physical computer | **Standard**:  200  **Extra**:  4 | **Standard**:  16  **Fat**:  7  **Extra**:  1  **Mic:**  1  **New:**  1 |
| Processor | This denotes a the multi-core processor, housing many processing elements | **Standard**:  2 per node  **Extra**:  4 per node | 2 per node |
| GPU | This denotes a nvidia co-processor | 0 | **Standard**:  2 per node  **Fat**:  4 per node  **Extra**:  8 per node  **Mic:**  0 per node  **New:**  2 cards per node  2 logical per card  4 logical per node |
| Socket | This is the “plug” the processor gets plugged into. Used as a synonym for the processor | **Standard**:  2 per node  **Extra**:  4 per node | 2 per node |
| Core | Individual processing element | **Standard**:  16 per node  8 per processor  **Extra**:  48 per node  12 per processor | 16 per node  8 per processor |
| Task | This is a software concept. It denotes a process, which is an instance of a running program. It has its own data and instruction stream(s). It can fork multiple threads to increase the computational speed. Serial programs and pure MPI programs do not spawn threads. | User controls in job script | User controls in job script |
| Thread | This is also a software concept. A thread is a stream of instructions executed on the hardware. It is part of a task and shares resources such as data with other threads within the same task. | User controls in job script | User controls in job script |

### Outline: Resource requests for multiprocessor jobs

When running multi processor jobs on the Lunarc clusters, one should specify:

1. The number of nodes required by the jobs
2. The number of computational tasks per node
3. The number of threads spawned by each task

For a pure MPI job or when processing a large number of serial jobs in a so called task farm, one will typically only specify the items 1 and 2, while for a threaded job, using e.g. OpenMP or Java, one will typically only specify items 1 and 3.

It is typically not advisable to have the product of items 2 and 3 exceeding the number of cores per node, which is 16 for standard Alarik and Erik compute nodes. On the Alarik extra compute nodes this number is 48. In most cases users requesting multiple nodes will want the product to equal the number of cores per node. The syntax how to control nodes, tasks per node and threads per task is explaned below.

### Exclusive node access

For parallel codes using MPI or OpenMP it is typically best to keep interference on the nodes at a minimum, that is to have exclusive access to the nodes you are using. This also applies to specialist and experimental work, which would interfere very badly with other user’s codes on the nodes. Adding

#SBATCH --exclusive

to your job script will ensure that SLURM will allocate dedicated nodes to your job. Obviously your project gets charged for the full costs of the nodes you are using, that is in case of Alarik and Erik 16 cores per node.

### Specifying the number of nodes required for the job

In SLURM one requests the number of nodes for a job with the **-N** option. The following statement requests four nodes for your job:

#SBATCH -N 4

**Important:** without using either the --tasks-per-node or the --cpus-per-task options of sbatch, this will reserve a single core per node, so four in total, which is most likely not what you want.

### Specifying the number of tasks per node

Use the --tasks-per-node of sbatch to specify the number of tasks you require per node. Most multinode job will set this equal to the number of cores availble per node. The following example asks for 16 task per node:

#SBATCH --tasks-per-node=16

This should be used together with the -N option, specifying the number of nodes to be used. The default value for the number of tasks per node is 1. For example to specify the requirements for an MPI job with 64 tasks or multiprocessor job using 64 processors to process a larger number of serial jobs one would specify

#SBATCH -N 4

#SBATCH --tasks-per-node=16

When using fewer than 16 tasks per node and you want to prevent other user’s jobs sharing your node, you need to consider using the --exclusive option. If --exclusive is not specified, SLURM might place other tasks onto your node.

### Specifying the number of threads for a shared-memory job

If you want to run shared-memory applications using threads, e.g. OpenMP parallised code or Java applications, you need to specify the number of threads you require per task. This can be done with the --tasks-per-node option of sbatch.

For a standard shared-memory program, which doesn’t also use distributed memory programming models such as MPI, one is restricted to a single node. On that node, one can request as many threads as there are cores on the node. On the standard Alarik compute nodes one can efficiently use up to 16 threads. Use the following resource statement:

#SBATCH -N 1

#SBATCH --tasks-per-node=16

If your program is only efficient at a lower thread count, you may want to use e.g.:

#SBATCH -N 1

#SBATCH --tasks-per-node=4

if you only want to use four threads. The Alarik extra nodes with 48 cores allow for very wide shared-memory jobs:

#SBATCH -N 1

#SBATCH --tasks-per-node=48

#SBATCH --mem-per-cpu=3000

#SBATCH -p extra

### Resource statements for hybrid programs using distributed and shared memory

So-called hybrid programs, using both distributed and shared-memory techniques have recently become popular. For example: for a program using 32 MPI tasks, each task spawning 2 OpenMP threads one would require 4 nodes and place eight tasks on each node. The number of threads per task is given by --cpus-per-task. The resource statement would look as follows:

#SBATCH -N 4

#SBATCH --tasks-per-node=8

#SBATCH --cpus-per-task=2

### Specifying the number of cores to be required by the job

In special cases, such as using very unusal numbers of tasks, the **-n** option of sbatch to specify the number of cores might become useful. When running a pure MPI program this option corresponds to the **number of tasks** required for your program. The following statement in a job script would reserve 63 cores for your job

#SBATCH -N 4

#SBATCH --tasks-per-node=16

#SBATCH -n 63

Please consider using the --exclusive option of sbatch to avoid SLURM spreading your job on more nodes than necessary and placing other user’s jobs on nodes utilising fewer than 16 cores for your job. Other user’s jobs could via shared node resources (memory bus, cache, FPU, …) interfere with your job and introduce undue operational noise. Such noise is something parallel program execution can be extremely sensitive to.

## Program execution environment

### Job execution environment

When submitting your job to SLURM using sbatch, your entire environment including the currently loaded modules gets copied. This behaviour is different from earlier Lunarc machines, including Platon. On Alarik, when hitting sbatch:

* Make sure that the loaded modules and any environment variable you may have set will not be in conflict with the environment expected by the job script

### Compiler modules

On Alarik we automatically load a modern version of the GCC compiler, which supports the deployed AMD Opteron processors. At the time of writing this is version 4.6.2 of GCC. If you prefer using a different compiler, you can add the desired module, e.g., version 12.1 of the Intel compiler

module add intel/12.1

If different modules have files with the same names in the search path, those of the module added last will be picked. Generally this is not a problem, but the compiler wrappers in the openmpi modules have the same names and it safest to only have one loaded at a time.

On Erik the same compilers as on Alarik are present. Note that the processors on Erik are of the Intel Xeon type and thus utilize the mkl as supplied.

### SLURM variables

*To come*

### SNIC variables

The SNIC meta-centres have agreed on a set of environment variables which should improve the portability of (parts of) job-scripts between SNIC sites. On Alarik the following variables are set by the system:

|  |  |  |  |
| --- | --- | --- | --- |
| **Environment variable** | **Explanation** | **Value on Alarik** | **Value on Erik** |
| SNIC\_SITE | Identifying the SNIC site you are using | lunarc | lunarc |
| SNIC\_RESOURCE | Identifying the compute resource you are using | alarik | erik |
| SNIC\_BACKUP | User directory which is:   * Regularly backed up against accidental deletion * Typically extremely limited space * Use for e.g. precious source code | /home/<user> | /home/<user> |
| SNIC\_NOBACKUP | User directory which is:   * Accessible on all Lunarc systems * Outliving individual systems * For storing larger amounts of data * Not backed up against accidental deletion * Protected against disk failure (RAID configuration) * On Alarik: the primary root directory for job management (job scripts, input/output data) | /lunarc  /nobackup  /users/<user> | /lunarc  /nobackup  /users/<user> |
| SNIC\_TMP | Directory for best performance during a job  At Lunarc:   * Local disk on nodes * Storing temporary data during job execution * High bandwidth * Automatically deleted * Transfer data with long-term value to SNIC\_NOBACKUP before job has finished | *jobid dependent* | *jobid dependent* |

## Using the node local disks to improve I/O performance

On Alarik and Erik, all nodes have a local disk. This disk offers superior bandwidth when compared to accessing your home space or the /lunarc/nobackup centre storage. In particular when files are read or written repeatedly during execution it is advisable to copy the input data onto the local disk prior to job execution and copy the result files back to the submission directory once your program has finished. During its execution, your program would then read and write to local disk.

In case of Alarik and Erik, the submission directory typically resides on the /lunarc/nobackup centre storage. All data left on the node local disks **will be deleted** when your job has finished. You need to copy everything of interest to a more permanent storage space such as /lunarc/nobackup. If a job is terminated prematurely, for example, if it exceeds the requested walltime, the files on the local disk (in $SNIC\_TMP) will be lost.

Files that would still be useful can be listed in a special file **$SNIC\_TMP/slurm\_save\_files**. Filenames are assumed to be relative to $SNIC\_TMP and should be separated by spaces or listed on separate lines. Wildcards are allowed. These files will be copied from the local disk where $SNIC\_TMP/slurm\_save\_files exists to the submission directory regardless whether the job ends as planned or is deleted, unless there is a problem with the disk or node itself. Note that the slurm\_save\_files feature is unique to Lunarc.

For the required UNIX scripting you should use the following environment variables. Example scripts using this technique are provided in the example section of this document. Contact the help desk if you have specific requirements and require consultation.

|  |  |
| --- | --- |
| **Variable** | **Addressed Volume** |
| SNIC\_TMP | node local disk  copy your input data here and start your program from here |
| TMPDIR | node local disk  Many applications use this environment variable to locate a disk volume for temporary scratch space. If your application follows that convention nothing needs to be done. |
| SLURM\_SUBMIT\_DIR | submission directory  where you ran sbatch |

## Launching MPI jobs in OpenMPI

To execute message passing parallel jobs these should be built against one of the MPI libraries provided by the support team as a module. To execute an MPI job, your job script should do the following

* Load MPI module relevant for the compiler you are using
* Start the program with mpirun
* On Alarik the correct binding is crucial to achieve good performance. When using 16 task per node, we recommend using the -bind-to-core option of mpirun. When using fewer than 16 tasks we recommend experimenting whether not using binding helps or hinders performance.

## Submitting, monitoring and manipulating jobs in SLURM

### Submitting with sbatch

One uses the command sbatch to submit a job script to the batch system for execution. SLURM will reply with the jobid number. The job will then be held in the queue until the requested resources become available. A typical use case looks as follows:

[fred@alarik MPItest]$ sbatch runjob.sh

Submitted batch job 7197

User fred submitted the script runjob.sh to the job queue and got the jobid 7197 assigned to it.

### Starting executables within SLURM with srun

The command srun allows to start executables in a way managed by SLURM. This is particularly effective if you want to process a large number of jobs within a single submission to the batch system. A use case of srun to start many serial jobs in a single multicore submission scipt is discussed in the [example section](#id.bdphbddpef0).

### Monitoring with squeue

The command squeue will show you the current state of the job queue. The standard output, created by calling squeue without any options looks as follows:

JOBID PARTITION NAME USER ST TIME NODES NODELIST(REASON)

7303 snic hybrid\_n fred PD 0:00 32 (Priority)

7302 snic hybrid\_n fred PD 0:00 32 (Priority)

7301 snic hybrid\_n fred PD 0:00 32 (Resources)

7304 snic preproce karl PD 0:00 6 (Priority)

7300 snic hybrid\_n fred R 0:24 32 an[001-032]

7305 snic preproce karl R 0:37 6 an[081-086]

7306 snic hybrid\_n fred R 0:37 6 an[081-086]

7307 snic testsimu sven R 0:07 1 an081

The first column gives the jobid, the third the job names, followed by the userid. The column labeled “ST” gives the job state. The most important states are:

|  |  |
| --- | --- |
| **Symbol** | **Meaning** |
| R | running |
| PD | pending, awaiting resources |
| CG | completing |

The state column is followed by the time used by the job and number of nodes utilised by the job. For running jobs the last column gives the names of the nodes utilised or if the job is waiting a reason why it is not executing.

The squeue command is highly configurable. Useful options include -u myid, which lists all jobs of the user myid and also the --start option. The latter gives the current estimate of when SLURM expects the job to start. Note, that this can shift in either direction, depending on e.g. jobs finishing earlier than specified or jobs with higher priority getting added to the job queue.

The command jobinfo is a script that sorts the output of squeue into running and waiting jobs. It also shows additional information, such as how long running jobs have left and in some cases when waiting jobs are expected to start.

### Terminating jobs with scancel

It is frequently required to remove jobs from the queue. This might be that you discover a problem in your job specification or intermediate results of running job indicating that something went wrong. Use scancel to remove a job from the job queue. To do so you need the jobid, which is best queried with the squeue command. To remove the job with the jobid 7103 from the job queue type

scancel 7103

# Example job scripts

In this section we provide sample scripts for typical use cases.

## Job scripts using the node local disk

### Basic run script

As discussed the node local disk provides better I/O-bandwidth than the other file systems available on Alarik. The following script assumes the program processor reads the file **input.dat** and produces a file **result.dat**.

This example executes a single serial program and is suitable for the occasional serial job. If you need to process a large number of serial jobs, we request you bundle them into a single submission. Refer to the section “[Running multiple serial jobs within a single job submission](#id.bdphbddpef0)” for a scripting example.

The script copies the input data and the program executable from the submission directory to the node local disk, executes the program on the node local disk and copies the result file back to the submission directory for safe keeping. The individual steps are highlighted by comments starting with a “#”. These comment lines can be kept in the file.

This is the Lunarc standard example and represents **recommended practise** for a basic serial job. You need to customise the file to suit your specific needs. The script is suitable for jobs consuming no more than 2000 MB of main memory.

#!/bin/bash

#

# job time, change for what your job requires

#SBATCH -t 00:10:00

#

# job name

#SBATCH -J data\_process

#

# filenames stdout and stderr - customise, include %j

#SBATCH -o process\_%j.out

#SBATCH -e process\_%j.err

# write this script to stdout-file - useful for scripting errors

cat $0

# copy the input data and program to node local disk

# customise for your input file(s) and program name

cp -p input.dat processor $SNIC\_TMP

# change to the execution directory

cd $SNIC\_TMP

# run the program

# customise for your program name and add arguments if required

./processor

# rescue the results to the submission directory

# customise for your result file(s)

cp -p result.dat $SLURM\_SUBMIT\_DIR

We recommend to be selective about the files you copy between the submission directory and the local node disk. If you have multiple input and result files you need to modify the copy statements accordingly. The above example assumes your program has been compiled with the GCC compiler loaded by default. If it has been compiled with a different compiler you need to load the compiler module by adding a line similar to

module add intel/12.1

If you are running on Erik, it is necessary to add the support for the GPU with the line

module add cuda

prior to the line ./processor. You need to consult with the person who build the executable for you. Lunarc provided modules typically complain if the wrong compiler is loaded and are hence self-documenting.

### Version for codes requiring more memory than 2000 MB

If your program requires more memory than 2000 MB, use the following script. This example is set up to use 4000 MB. If you need even more you can request this, but your runs will be charged to your project at a higher rate, since other cores have to remain idle. The comments on the previous example also apply here

#!/bin/bash

#

# job time, change for what your job requires

#SBATCH -t 00:10:00

#

# job name

#SBATCH -J data\_process

#

# filenames stdout and stderr - customise, include %j

#SBATCH -o process\_%j.out

#SBATCH -e process\_%j.err

#

# requesting a large memory node and 4000 MB or main memory

#SBATCH -C mem64GB

#SBATCH --mem-per-cpu=4000

# write this script to stdout-file - useful for scripting errors

cat $0

# copy the input data and program to node local disk

# customise for your input file(s) and program name

cp -p input.dat processor $SNIC\_TMP

# change to the execution directory

cd $SNIC\_TMP

# run the program

# customise for your program name and add arguments if required

./processor

# rescue the results to the submission directory

# customise for your result file(s)

cp -p result.dat $SLURM\_SUBMIT\_DIR

Since fewer nodes are equipped with 64 GB of memory, you have to allow for longer queueing times until resource become available.

## Running multiple serial jobs within a single job submission

When you need to run many serial jobs, similar to the ones [described above](#id.81n49hdgiv29), these should be bundled together and submitted to the job queue in a small number of submissions or even a single submission. With SLURM is perfectly reasonable to run several hundred individual jobs in a single submission. To speed up the processing of your jobs, you can ask for the cores from a number of nodes. The concept is known as a **task-farm**. The individual job are known as **job-steps**.

The following is an example processing 200 such jobs using 16 cores from a single node. The scripting use two scripts, the master script and the worker script. The Master script requests the resources (number of cores, job time, ...) and then registers 200 copies of the worker script with SLURM using the command srun. The worker script is a modification of the [basic script](#id.oyajyndi4e55) described above.

In our example this will then start sixteen jobs on the sixteen cores you requested. Once a job has finished, it will take an unprocessed job and place it on the idle core for processing. This will continue until all jobs are processed. The ordering of the jobs can not be relied on.

For our example the entire setup assumes the submission directory has 200 sub-directories, named job\_0, job\_1, job\_2, …, job\_199. Each of the directories contains the input data and the program executable to be run.

Keep the number of jobs-steps at a reasonable level. Recent testing by the Lunarc support team has shown that, when including a sleep statement inside the do loop the setup can be used to processes 800 jobs.

### The master script

The master script describes the resources required and registers, once running the worker tasks with SLURM. In most cases modifying the number of cores needed, the total job time and the number of jobs to be processed should be all that is required.

#!/bin/sh

# requesting the number of cores needed

#SBATCH -N 1

#SBATCH --tasks-per-node=16

#SBATCH --exclusive

#

# job time, change for what your job farm requires

#SBATCH -t 20:00:00

#

# job name and output file names

#SBATCH -J jobFarm

#SBATCH -o res\_jobFarm\_%j.out

#SBATCH -e res\_jobFarm\_%j.out

cat $0

# set the number of jobs - change for your requirements

export NB\_of\_jobs=200

# Loop over the job number

for ((i=0; i<$NB\_of\_jobs; i++))

do

srun -Q --exclusive -n 1 -N 1 \

workScript.sh $i &> worker\_${SLURM\_JOB\_ID}\_${i} &

sleep 1

done

# keep the wait statement, it is important!

wait

The script assumes that the job is described in a script file “workScript.sh”, which takes a single number identifying the job directory to be accessed as a command line argument. Please note the “sleep 1” command inside the do loop. In our testing this greatly enhances the stability by submitting the actual jobs over a longer period of time. With this statement included the script was able to successfully handle up to about 800 outstanding jobs on 16 and 32 cores. For reasons of job reliability, we therefore recommend not to process more than 800 jobs in a single script. However it is possible to process significantly larger job numbers than 800 by carefully tuning sleep-time and core count in relation to the average job-time.

**Remarks:** When using srun inside a batch script many srun-options act differently compared to using srun within a different environment. Consult the man-page of srun for documentation and contact the Lunarc help desk if your require further consultancy.

### The worker script

This outlines the worker script. Compared to the script describing a [single serial job](#id.oyajyndi4e55), a few modifications are required:

* To avoid access conflicts between the individual jobs, each job creates a job private sub-directory on the node local disk.
* The input file(s) are expected in the sub\_directories job\_0, job\_1, job\_2, … of the submission directory. The result file(s) will also be placed in these directories.
* The example assumes a single input file and single result file. If you have multiple input and/or result files modifications are needed, as are modifications for that actual names of your file
* The present set up allows for different executables for each job-stop. The script assumes to find an executable named “processor” in the same location as the input file(s). If you all job steps use the same executable the scripts can be simplified.
* Once a job-step has finished and the result file has been copied back, the job private sub-directory on the node local disk is removed to prevent the disc from overflow.

If you are using the above master script, the script should be named “workScript.sh”.

#!/bin/sh

# document this script to stdout (assumes redirection from caller)

cat $0

# receive my worker number

export WRK\_NB=$1

# create worker-private subdirectory in $SNIC\_TMP

export WRK\_DIR=$SNIC\_TMP/WRK\_${WRK\_NB}

mkdir $WRK\_DIR

# create a variable to address the "job directory"

export JOB\_DIR=$SLURM\_SUBMIT\_DIR/job\_${WRK\_NB}

# now copy the input data and program from there

cd $JOB\_DIR

cp -p input.dat processor $WRK\_DIR

# change to the execution directory

cd $WRK\_DIR

# run the program

./processor

# rescue the results back to job directory

cp -p result.dat ${JOB\_DIR}

# clean up the local disk and remove the worker-private directory

cd $SNIC\_TMP

rm -rf WRK\_${WRK\_NB}

### Monitoring the progress of your multi-job submission

Using the -s option of sbatch you can monitor the progression of the individual job-steps of your multi-job submission. Please keep in mind, that the step number SLURM assigns to your job and the one you assign typically differs from the loop index used in the master script.

The below is an output from squeue when running a script processing 500 jobs on 32 cores. The jobid of the job is 8070. The output shows the job-steps the script is presently processing

[fred@alarik MultiSerialTest]$ squeue -j 8070 -s

STEPID NAME PARTITION USER TIME NODELIST

8070.130 small\_ex snic fred 2:09 an074

8070.133 small\_ex snic fred 2:02 an073

8070.135 small\_ex snic fred 1:55 an074

8070.136 small\_ex snic fred 1:41 an073

8070.139 small\_ex snic fred 1:41 an073

8070.140 small\_ex snic fred 1:41 an073

8070.143 small\_ex snic fred 1:41 an073

8070.144 small\_ex snic fred 1:41 an074

8070.147 small\_ex snic fred 1:41 an074

8070.148 small\_ex snic fred 1:41 an074

8070.151 small\_ex snic fred 1:41 an074

8070.155 small\_ex snic fred 1:38 an074

8070.156 small\_ex snic fred 1:35 an074

8070.157 small\_ex snic fred 1:34 an073

8070.158 small\_ex snic fred 1:34 an073

8070.159 small\_ex snic fred 1:34 an073

8070.161 small\_ex snic fred 1:34 an073

8070.164 small\_ex snic fred 1:33 an074

8070.165 small\_ex snic fred 1:33 an074

8070.168 small\_ex snic fred 1:32 an073

8070.170 small\_ex snic fred 1:26 an073

8070.171 small\_ex snic fred 1:12 an073

8070.172 small\_ex snic fred 1:12 an073

8070.175 small\_ex snic fred 1:11 an074

8070.176 small\_ex snic fred 1:11 an074

8070.179 small\_ex snic fred 1:11 an074

8070.184 small\_ex snic fred 1:04 an074

8070.185 small\_ex snic fred 0:42 an073

8070.190 small\_ex snic fred 0:35 an073

8070.193 small\_ex snic fred 0:35 an074

8070.194 small\_ex snic fred 0:13 an073

8070.195 small\_ex snic fred 0:13 an074

## MPI job using 16 tasks per node

Most MPI jobs achieve best cost efficiency when deploying 16 tasks per node, that is one task per core. Benchmarking by the Lunarc team showed that these jobs typically require binding to achieve good performance. The binding offered by the OpenMPI library works satisfactory.

The resource request is very easy in this case. Ask for a number of cores equivalent to the number of tasks you want to run. We recommend using the --exclusive option to avoid getting unrelated jobs placed on the last node in case the number of cores requested doesn’t divide by the number of cores per node. The following is an example submission script to run the MPI application simula\_mpi with 64 tasks on 4 nodes. Notice you do not need to specify the node count.

#!/bin/sh

# requesting the number of cores needed on exclusive nodes

#SBATCH -N 4

#SBATCH --tasks-per-node=16

#SBATCH --exclusive

#

# job time, change for what your job requires

#SBATCH -t 0:30:0

#

# job name

#SBATCH -J simula\_n64

#

# filenames stdout and stderr - customise, include %j

#SBATCH -o simula\_n64\_%j.out

#SBATCH -e simula\_n64\_%j.out

# write this script to stdout-file - useful for scripting errors

cat $0

# Example assumes we need the intel runtime and OpenMPI library

# customise for the libraries your executable needs

module add intel/13.0

module add openmpi/1.6.2/intel/13.0

# Copying the executable onto the local disks of the nodes

srun -n $SLURM\_NNODES -N $SLURM\_NNODES cp -p simula\_mpi $SNIC\_TMP

# Copy the input file onto the headnode - if your MPI program

# reads from all tasks, you need to do the above srun construct

# again

cp -p input.dat $SNIC\_TMP

# change to local disk and start the mpi executable

cd $SNIC\_TMP

mpirun -bind-to-core simula\_mpi

# Copy result files back - example assumes only task zero writes

# if in your application result files are written on all nodes

# you need to initiate a copy on each node via srun

cp -p result.dat $SLURM\_SUBMIT\_DIR

This script assumes you are using up to 2000 MB of memory per task. If you need more, adding the two lines

#SBATCH -C mem64GB

#SBATCH --mem-per-cpu=4000

to the script will allow for using up to 4000 MB. Since fewer nodes are equipped with 64 GB of memory, you have to allow for longer queueing times until resource become available.

### Modifications required for file I/O on all nodes

As discussed in the comments of the sample script, the script assumes that only MPI-task 0 on the head node reads the input file and writes to the output file. If for your MPI application every MPI task reads the input file(s), replace the line

cp -p input.dat $SNIC\_TMP

with

srun -n $SLURM\_NNODES -N $SLURM\_NNODES cp -p input.dat $SNIC\_TMP

and the file gets copied onto the local disk of each node. Matters are slightly more complex, if your output is written from all tasks. We assume the output files can be wild-carded as result\_\*.dat. Copying these files back to the submission directory can be achieved creating a script, which is placed on all nodes and subsequently executed on all nodes. The following addition to the submission script will create the script and place it on all your nodes

cat <<EOF > copyfile.sh

#!/bin/sh

cp -p result\*.dat $SLURM\_SUBMIT\_DIR

EOF

chmod u+x copyfile.sh

srun -n $SLURM\_NNODES -N $SLURM\_NNODES cp copyfile.sh $SNIC\_TMP

This needs inserting into the script before the “cd $SNIC\_TMP” statement. Once this is in place you can copy your result files by replacing the line

cp -p result.dat $SLURM\_SUBMIT\_DIR

with the line

srun -n $SLURM\_NNODES -N $SLURM\_NNODES copyfile.sh

## MPI jobs using fewer than 16 tasks per node

If you want to use fewer than 16 task per nodes to e.g. give more resources to the individual task, you can use the -N and --task-per-node options of sbatch. We recommend not to use the -n option in this case. This example is for 4 nodes with 8 tasks each, a total of 32 tasks. In our experience, in this case and when using --exclusive it is typically advantageous to not use binding. Though we encourage experimenting with your own application.

#!/bin/sh

# requesting the number of nodes and cores needed, exclusive nodes

#SBATCH -N 4

#SBATCH --tasks-per-node=8

#SBATCH --mem-per-cpu=8000

#SBATCH -C mem64GB

#SBATCH --exclusive

#

# job time, change for what your job requires

#SBATCH -t 0:30:0

#

# job name

#SBATCH -J simula\_n64

#

# filenames stdout and stderr - customise, include %j

#SBATCH -o simula\_n64\_%j.out

#SBATCH -e simula\_n64\_%j.out

# write this script to stdout-file - useful for scripting errors

cat $0

# Example assumes we need the intel runtime and OpenMPI library

# customise for the libraries your executable needs

module add intel/13.0

module add openmpi/1.6.2/intel/13.0

# Copying the executable onto the local disks of the nodes

srun -n $SLURM\_NNODES -N $SLURM\_NNODES cp -p simula\_mpi $SNIC\_TMP

# Copy the input file onto the headnode - if your MPI program

# reads from all tasks, you need to do the above srun construct

# again

cp -p input.dat $SNIC\_TMP

# change to local disk and start the mpi executable

cd $SNIC\_TMP

mpirun simula\_mpi

# Copy result files back - example assumes only task zero writes

# if in your application result files are written on all nodes

# you need to initiate a copy on each node via srun

cp -p result.dat $SLURM\_SUBMIT\_DIR

## OpenMP jobs using shared memory

To run a shared memory code using OpenMP on Alarik, you specify the number of cores you require using --tasks-per-node option of sbatch. In this case you have to request placement on a single node with the “-N 1” option. In this example we call the executable “processor\_omp” to emphasis that this need to be compiled with OpenMP support. Unless you are doing something special, you are not required to specify the environment variable OMP\_NUM\_THREADS. The example script uses the techniques described for the [basic run script](#id.oyajyndi4e55) to engage the node local disk.

#!/bin/bash

#

# Specify the number of threads - request all on 1 node

#SBATCH -N 1

#SBATCH --tasks-per-node=16

#

# job time, change for what your job requires

#SBATCH -t 00:10:00

#

# job name

#SBATCH -J data\_process

#

# filenames stdout and stderr - customise, include %j

#SBATCH -o process\_omp\_%j.out

#SBATCH -e process\_omp\_%j.err

# write this script to stdout-file - useful for scripting errors

cat $0

# copy the input data and program to node local disk

# customise for your input file(s) and program name

cp -p input.dat processor\_omp $SNIC\_TMP

# change to the execution directory

cd $SNIC\_TMP

# run the program

# customise for your program name and add arguments if required

./processor\_omp

# rescue the results to the submission directory

# customise for your result file(s)

cp -p result.dat $SLURM\_SUBMIT\_DIR

This script allows to use 2000 MB of main memory per requested core. If you need more memory, this can be requested by:

#SBATCH -C mem64GB

#SBATCH --mem-per-cpu=4000

This will increase you memory request to 4000 MB per requested core.

### Thread binding for OpenMP codes

The Alarik nodes deploy a cache-coherent non-uniform-memory access architecture (cc-numa). Many scientific simulation codes gain significant performance benefits on a cc-numa architecture when the user binds the threads to physical cores of the hardware. This inhibits thread migration and improves memory locality. Unfortunately invoking thread binding is not standartised. Depending on the OpenMP runtime library the user needs to modify different environment variables to bind his threads.

#### Thread binding with the GNU compilers

By default the GNU compiler suite (gcc/gfortran) does not bind threads to cores. To engage thread binding, you need to set the environment variable GOMP\_CPU\_AFFINITY and provide this with a binding list. When setting

export GOMP\_CPU\_AFFINITY=”0-15”

in your submission script, prior to starting your OpenMP application, this will bind the threads to the 16 cores in the node. The above will bind thread 0 to core 0, thread 1 to core 1 and so on.

**More advanced remark:** If you want to utilise only 8 cores from a node and asking for exclusive node access (#SBATCH --exclusive), it might be a good idea to place threads on every second core only. This will give you more memory bandwidth and make sure you are utilising all FPUs of the Interlagos architecture. This can be achieved by setting:

export GOMP\_CPU\_AFFINITY=”0-14:2”

or

export GOMP\_CPU\_AFFINITY=”0 2 4 6 8 10 12 14”

It depend on details of your application, whether or not this helps performance. Also note, when asking for a exclusive access to a note, you will be charged for the full node, whether or not you use all cores.

**Important pitfall:** If you set GOMP\_CPU\_AFFINITY=0 this will bind all threads to core 0. You will see extremely poor performance in this case.

#### Thread binding with the open64 compiler

OpenMP code compiled with the **open64** compiler will use thread binding on Alarik. In standard use cases this will actually boost performance. However in special situation, e.g. when using fewer threads than the size of your partition, you might see a performance boost by not using thread binding. To do so you need to set the environment variable “O64\_OMP\_SET\_AFFINITY=false”

#### Thread binding with the Intel compiler

Versions 12.1 and 13.0 of the **Intel** compiler do not support thread binding when used on the AMD processors deployed on Alarik. Starting from version 13.1 the Intel compile does support thread binding on the AMD processors deployed on Alarik. Obviously all versions of the Intel compiler support thread binding on the Intel processors deployed on Erik.

For version 13.1 of the Intel compiler thread is controlled by setting the environment variable KMP\_AFFINITY. The value

export KMP\_AFFINITY=granularity=fine,compact

might be a good starting point for your experimentation.

## Hybrid-jobs using threads within an MPI framework

A cluster with multicore nodes such as Alarik is a natural environment to execute parallel codes deploying both MPI and OpenMP threads. When running such applications the optimal number of MPI-tasks and OpenMP threads to place on a node can depend highly on the details of the application. In particular for application which make many references to main memory and the programmer has not implemented a proper “first touch data allocation” it is typically **best to have 2 or 4 threads** per MPI task on an Alarik node. Together with a proper binding of your MPI tasks to the “numa-islands”, this will ensure memory locality for your code. For the below syntax you have to use **version 1.8.3 or newer** of the OpenMPI library.

In the following we give a simple example script to run a MPI-OpenMP hybrid named simul\_hyb on 2 nodes using 8 tasks and 4 threads per task. The tasks and their threads will be bound to the *numa-islands*, minimising cc-numa effects.

#!/bin/sh

# requesting number of nodes (-N option)

# number of mpi-tasks per node

# and number of threads per task exclusive nodes

#SBATCH -N 2

#SBATCH --tasks-per-node=4

#SBATCH --cpus-per-task=4

#SBATCH --exclusive

# time required

#SBATCH -t 01:00:00

#SBATCH -J hybrid\_run

# filenames stdout and stderr - customise, include %j

#SBATCH -o simula\_N2t4c4\_%j.out

#SBATCH -e simula\_N2t4c4\_%j.out

cat $0

# Example assumes we need the intel runtime and OpenMPI library

# customise for the libraries your executable needs

module add intel/15.0

module add openmpi/1.8.3/intel/15.0

# Copying the executable onto the local disks of the nodes

srun -n $SLURM\_NNODES -N $SLURM\_NNODES cp -p simul\_hyb $SNIC\_TMP

# Copy the input file onto the headnode - if your MPI program

# reads from all tasks, you need to do the above srun construct

# again

cp -p input.dat $SNIC\_TMP

cd $SNIC\_TMP

# setting number of OpenMP threads and ask for thread binding

export OMP\_NUM\_THREADS=$SLURM\_CPUS\_PER\_TASK

export OMP\_PROC\_BIND=true

mpiexec --map-by ppr:$SLURM\_NTASKS\_PER\_NODE:node:PE=$SLURM\_CPUS\_PER\_TASK \ simul\_hyb

# Copy result files back - example assumes only task zero writes

# if in your application result files are written on all nodes

# you need to initiate a copy on each node via srun

cp -p result.dat $SLURM\_SUBMIT\_DIR

The example assumes that MPI task 0 is the only task reading and writing input files. If your application reads and writes data on all nodes, you need to study the [modifications](#id.hpgejkt8dzry) described in the MPI section.

As discussed, the above binds the tasks and their threads to the numa-islands of the Alarik architecture. Alariks numa-islands have four cores, therefore the script is best used with 2 or four threads per MPI task. This results in one or two MPI tasks per numa islands.

### Things to try for MPI-OpenMP hybrids with 16 threads per task

While using more than 4 threads per MPI task on the Alarik system can result in reduced performance due to cc-numa effects, there are situations when using 16 threads per task can be required (e.g. special algorithms or extreme memory requirements per MPI task).

When running 16 threads per MPI task, that is a single MPI task per Alarik node, you might want to experiment with starting your job without specifying binding on mpiexec, that is remove the -bind-to-core, but utilise the [OpenMP thread binding](#id.zfbd8w955ujk) techniques described in the OpenMP sample section.

# Interactive access to compute nodes

Sometimes it is desirable to have an interactive login to the compute nodes of the cluster. Extensive code testing is a typical use case.

## Starting an interactive session

To start an interactive session you need to use the “interactive” command. This will request the required resources from the resource pool for you and start the interactive session once the resources are available.

Use the following command to start an interactive session asking for 32 cores lasting 60 minutes

interactive -n 32 -t 60

On Alarik and Eric this will be allocated on multiple nodes, since the nodes have only 16 cores available. The interactive session will last until either the requested time, 60 minutes in the above example, has expired or you manually exit the interactive shell. Your account gets charged with the wall time duration of your interactive session, independent of the amount of computation you do. In the above example, if your session lasts until it expires after 60 min, you get charged for 32 cpu hours. If you terminate your session after 1/2 hour, you would get charged 16 cpu hours.

The interactive command supports most command line options of the sbatch command. Please refer to the man pages of sbatch.

## Modules and environment variables

Loaded modules and environment are not always exported properly to your interactive session. Once placed in the interactive session, we recommend users to reload **all** the modules they require. That is despite the “modules list” command claiming they are still loaded.

You also need to check whether environment variables still have the required values. If the software you are using has a set-up script, you might need to re-run that script.

## Known issues with the interactive command

None at the time of writing.