Modules & Application Software

The number of software packages that are installed together with the operating system on cluster nodes is kept light on purpose. Additional packages and applications are provided by a module system, which enables you to easily customise your working environment on the cluster. This module system is called Lmod[1] . Furthermore, we can provide different versions of the software which you can use on demand. Loading a module, software specific settings are applied, e.g. changing environment variables like PATH, LD_LIBRARY_PATH and MANPATH.

Alternatively, you can manage software packages on the cluster yourself by building software from source, by means of EasyBuild or by using Singularity containers. Python packages can also be installed using the Conda manager. The first three possibilities, in addition to the module system, are described in the current section, whereas Conda usage in the cluster is explained in this section.

We have adopted a systematic software naming and versioning convention in conjunction with the software installation system `EasyBuild` [2] .

Software installation on the cluster utilizes a hierarchical software module naming scheme. This means that the command `module avail` does not display all installed software modules right away. Instead, only the modules that are immediately available for loading are displayed. More modules become available after their prerequisite modules are loaded. Specifically, loading a compiler module or MPI implementation module will make available all the software built with those applications. This way, he hope the prerequisites for certain software become apparent.

At the top level of the module hierarchy, there are modules for compilers, toolchains and software applications that come as a binary and thus do not depend on compilers. Toolchain modules organize compilers, MPI implementations and numerical libraries. Currently the following toolchain modules are available:

- Compiler only toolchains
  - GCC: GCC updated binutils
  - `iccifort`: Intel compilers, GCC
- Compiler & MPI toolchains
  - `gompi`: GCC, OpenMPI
  - `iimpi`: `iccifort`, Intel MPI
  - `iompi`: `iccifort`, OpenMPI
- Compiler & MPI & numerical libraries toolchains
  - `foss`: `gompi`, OpenBLAS, FFTW, ScaLAPACK
  - `intel`: `iimpi`, Intel MKL
  - `iomkl`: `iompi`, Intel MKL

Note that Intel compilers newer than 2020.x are provided by the toolchain module `intel-compilers`. It is strongly recommended to use this module as after 2023 the intel compiler modules `iccifort` will be removed.

Working with modules

This section explains how to use software modules.

List the entire list of possible modules

```
module spider
```

The same in a more compact list

```
module -t spider
```

Search for specific modules that have “string” in their name

```
module spider string
```

Detailed information about a particular version of a module (including instructions on how to load the module)

```
module spider name/version
```

Searches for all module names and descriptions that contain the specified string

```
module key string
```

List modules immediately available to load

```
module avail
```

Some software modules are hidden from the `avail` and `spider` commands. These are mostly modules for system library packages that other user applications depend on. To list hidden modules, you may provide the `–show-hidden` option to the `avail` and `spider` commands:

```
module --show-hidden avail
module --show-hidden spider
```

A hidden module has a dot (`.`) in front of its version numbers (eg. `zlib/.1.2.8`).

List currently loaded modules

```
module list
```

Load a specific version of a module

```
module load name/version
```

If only a name is given, the command will load the default version which is marked with a (D) in the

`module avail` listing (usually the latest version). Loading a module may automatically load other modules it depends on.

It is not possible to load two versions of the same module at the same time.

To switch between different modules

```
module swap old new
```

To unload the specified module from the current environment

```
module unload name
```

To clean your environment of all loaded modules

```
module purge
```

Show what environment variables the module will set

```
module show name/version
```

Save the current list of modules to “name” collection for later use

```
module save name
```

Restore modules from collection “name”

```
module restore name
```

List of saved collections

```
module savelist
```

To get the complete list of options provided by Lmod through the command `module` type the following

```
module help
```

Exercise: Working with modules

As an example, we show how to load the `gnuplot` module.

List loaded modules

```
module list

No modules loaded
```

Find available gnuplot versions

```
 module -t spider gnuplot

gnuplot/4.6.0
gnuplot/5.0.3
```

Determine how to load the selected `gnuplot/5.0.3` module

```
 module spider gnuplot/5.0.3

----------------------------------------------------------------------------
----
  gnuplot: gnuplot/5.0.3
----------------------------------------------------------------------------
----
    Description:
      Portable interactive, function plotting utility - Homepage:
http://gnuplot.sourceforge.net/

    This module can only be loaded through the following modules:

      GCC/4.9.3-2.25  OpenMPI/1.10.2

    Help:
      Portable interactive, function plotting utility - Homepage:
http://gnuplot.sourceforge.net/
```

Load required modules

```
 module load GCC/4.9.3-2.25  OpenMPI/1.10.2

Module for GCCcore, version .4.9.3 loaded
Module for binutils, version .2.25 loaded
Module for GCC, version 4.9.3-2.25 loaded
Module for numactl, version .2.0.11 loaded
Module for hwloc, version .1.11.2 loaded
Module for OpenMPI, version 1.10.2 loaded
```

And finally load the selected `gnuplot` module

```
 module load gnuplot/5.0.3

Module for OpenBLAS, version 0.2.15-LAPACK-3.6.0 loaded
Module for FFTW, version 3.3.4 loaded
Module for ScaLAPACK, version 2.0.2-OpenBLAS-0.2.15-LAPACK-3.6.0 loaded
Module for bzip2, version .1.0.6 loaded
Module for zlib, version .1.2.8 loaded
.............
.............
```

In order to simplify the procedure of loading the gnuplot module, the current list of loaded modules can be saved in a “mygnuplot” collection (the name string “mygnuplot” is, of course, arbitrary) and then loaded again when needed as follows:

Save loaded modules to “mygnuplot”

```
 module save mygnuplot

Saved current collection of modules to: mygnuplot
```

If “mygnuplot” not is specified, the name “default” will be used.

Remove all loaded modules (or open a new shell)

```
 module purge

Module for gnuplot, version 5.0.3 unloaded
Module for Qt, version 4.8.7 unloaded
Module for libXt, version .1.1.5 unloaded
............
............
```

List currently loaded modules. This selection is empty now.

```
 module list

No modules loaded
```

List saved collections

```
 module savelist

Named collection list:
  1) mygnuplot
```

Load `gnuplot` module again

```
 module restore mygnuplot

Restoring modules to user's mygnuplot
Module for GCCcore, version .4.9.3 loaded
Module for binutils, version .2.25 loaded
Module for GCC, version 4.9.3-2.25 loaded
Module for numactl, version .2.0.11 loaded
............
............
```

List of available software

In this section, you will find user guides for some of the software packages installed in the cluster. The guides provided can, of course, not replace documentation that comes with the application. Please read that as well.

A wide variety of application software is available in the cluster system. These applications are located on a central storage system that is accessible by the module system Lmod via an NFS export. Issue the command `module spider` on the cluster system or visit the page for a comprehensive list of available software. If you really need a different version of an already installed application, or one that is currently not installed, please get in touch. The main prerequisite for use of a software within the cluster system is its availability for Linux. Furthermore, if the application requires a license, we need to clarify additional questions.

Some select Windows applications can also be executed on the cluster system with the help of Wine or Singularity containers. For information on Singularity, see Singularity Containers.

A current list of available software

Usage instructions

- Abaqus
- ANSYS / CFX
- Code-Server (VS Code) on LUIS Cluster
- COMSOL
- Conda
- CPMD
- FEKO
- Gaussian & GaussView
- Jupyter on LUIS Cluster
- LIGGGHTS
- MATLAB
- mpiFileUtils
- NFFT
- uv – Python package manager

Build software from source code

Note: We recommend using EasyBuild (see next section) if you want to make your software's build process reproducible and accessible through a module environment that EasyBuild automatically creates. EasyBuild comes with pre-configured recipes for installing thousands of scientific applications.

Sub-clusters of the cluster system have different CPU architectures that provide different instruction set capabilities/extensions. The command `lcpuarchs` (available on the login nodes) lists all available CPU types.

```
login03:~$ lcpuarchs -v

CPU arch names    Cluster partitions
--------------    ------------------
avx2              LUIS[haku,lena,vis,jhub]
                  FCH[ai,ai,ainlp,ainlp,gih,pci,fi,imuk,p4d,p4d]

avx512
LUIS[gpu,gpu,gpu,gpu.test,mpp.single,mpp.share,amo,taurus,smp,smp]
FCH[tnt,tnt,ai.h100,gih,isd,isd,isd,isd,stahl,iwes,iwes,enos,pcikoe,pcikoe,p
cikoe,pcikoe,pcikoe,isu,phd,phdgpu,itp,itp,imes.gpu,imes]

LUIS - LUIS clusters. Availabe to all users around the clock
FCH  - Research Cluster Housing nodes. Available to all users
              on weekdays from 20:00 to 08:00 and on weekends

CPU of this machine: avx512

For more verbose output type: lcpuarchs -vv
```

The technical sequence of these architectures is `(oldest)-sse-avx-avx2-avx512-(newest)`. Cpus capable of executing instructions from newer instruction sets usually are able to execute commands from older extensions, so e.g. `avx512` includes `sse`.

Software compiled to use a newer cpu instruction set will usually not typically abort with an "**Illegal instruction**" error when run on an older cpu. The important message here is that compilers may automatically set flags for the platform you are currently working on. If you compile your program on a node providing `avx512` instructions (e. g. the `amo` sub-cluster) using the `gcc` compiler option `-march=native`, the program will usually not run on older nodes that are only equipped with cpus providing, say, `avx` instructions. To check which instruction set extensions a cpu architecture provides, you can run the command "lscpu", which lists them in the "flags" section.

This section explains how to build a software on the cluster system to avoid the aforementioned issue if you want to be able to submit jobs to all compute nodes without specifying the CPU type. Beware, though, that compatibility usually comes at a price and allowing a compiler to use the newer instruction sets will usually boost performance. Depending on your workload, the effects/speedup on newer cpu architectures may even be called "drastic". But there's usually no better way to tell than testing.

In the example below we want to compile a sample software `my-soft` in version `3.1`.

In your HOME (or, perhaps better, in your $SOFTWARE directory, if all members of your project want to use the software) directory, create `build/install` directories for each available CPU architecture listed by the command `lcpuarchs -s`, as well as a directory `source` to storing the installation sources

Note: you can usually refer to a variable using `$variable_name` and all will be well. In the following

examples, however, we demonstrate the use of curly brackets around the variable's designator, which will ensure proper separation of variables even in case of ambiguities (which in theory could occur in long paths composed out of several variables). For all normal purposes, `$HOME` and `${HOME}` or `$LUIS_CPU_ARCH` and `${LUIS_CPU_ARCH}` will be equivalent. If, however, you use spaces in your directories (like `dir="my directory"`), this will not be sufficient, you'll then also need to put double quotation marks " around the variable (like in `cd "${dir}"`).

```
 login03:~$ mkdir -p ${HOME}/sw/source
 login03:~$ eval "mkdir -p ${HOME}/sw/{$(lcpuarchs -ss)}/my-
soft/3.1/{build,install}"
```

Copy software installation archive to the source directory

```
 login03:~$ mv my-soft-3.1.tar.gz ${HOME}/sw/source
```

Build `my-soft` for each available CPU architecture by submitting an interactive job to each compute node type requesting the proper CPU type. For example, to compile `my-soft` for `avx512` nodes, first submit an interactive job requesting the `avx512` feature:

```
 login03:~$ salloc --nodes=1 --constraint=CPU_ARCH:avx512 --cpus-per-task=4
--time=6:00:00 --mem=16G
```

Then unpack and build the software. Note the environment variable `${LUIS_CPU_ARCH}` that contains the cpu instruction set of the compute node reserved.

```
 taurus-n034:~$ tar -zxvf ${HOME}/sw/source/my-soft-3.1.tgz -C
$HOME/sw/${LUIS_CPU_ARCH}/my-soft/3.1/build
 taurus-n034:~$ cd ${HOME}/sw/${LUIS_CPU_ARCH}/my-soft/3.1/build
 taurus-n034:~$ ./configure --prefix=${HOME}/sw/${LUIS_CPU_ARCH}/my-
soft/3.1/install && make && make install
```

Finally, use the environment variable `${LUIS_CPU_ARCH}` in your job scripts to access the correct installation path of `my-soft` executable for the current compute node. Note that you may need to set/update the `${LD_LIBRARY_PATH}` environment variable to point to the location of your software's shared libraries.

my-soft-job.sh

```
#!/bin/bash -l
#SBATCH --job-name=my-soft
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=16
#SBATCH --mem=60G
#SBATCH --time=12:00:00
#SBATCH --constraint="[CPU_ARCH:avx512|CPU_ARCH:avx2]"
#SBATCH --output my-soft-job_%j.out
#SBATCH --error my-soft-job_%j.err
#SBATCH --mail-user=myemail@....uni-hannover.de
#SBATCH --mail-type=BEGIN,END,FAIL
```

```
# Change to work dir
cd ${SLURM_SUBMIT_DIR}

# Load modules
module load my_necessary_modules

# run my_soft
export LD_LIBRARY_PATH=${HOME}/sw/${LUIS_CPU_ARCH}/my-
soft/3.1/install/lib:${LD_LIBRARY_PATH}
srun $HOME/sw/${LUIS_CPU_ARCH}/my-soft/3.1/install/bin/my-soft.exe --
input file.input
```

You can certainly consider combining the software build and execution steps into a single batch job script. However, it is recommended that you first perform the build steps **interactively** before adding them to a job script to ensure the software compiles without errors. For example, such a job script might look like this:

my-soft-job.sh

```
#!/bin/bash -l
#SBATCH --job-name=my-soft
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=32
#SBATCH --mem=120G
#SBATCH --time=12:00:00
#SBATCH --constraint=CPU_ARCH:avx512
#SBATCH --output my-soft-job_%j.out
#SBATCH --error my-soft-job_%j.err
#SBATCH --mail-user=myemail@....uni-hannover.de
#SBATCH --mail-type=BEGIN,END,FAIL

# Change to work dir
cd ${SLURM_SUBMIT_DIR}

# Load modules
module load my_necessary_modules

# install software if the executable does not exist
[ -e "${HOME}/sw/${LUIS_CPU_ARCH}/my-soft/3.1/install/bin/my-soft.exe"
] || {
  mkdir -p ${HOME}/sw/${LUIS_CPU_ARCH}/mysof/3.1/{build,install}
  tar -zxvf ${HOME}/sw/source/my-soft-3.1.tgz -C
${HOME}/sw/${LUIS_CPU_ARCH}/my-soft/3.1/build
  cd $HOME/sw/${LUIS_CPU_ARCH}/my-soft/3.1/build
  ./configure --prefix=${HOME}/sw/${LUIS_CPU_ARCH}/my-soft/3.1/install
  make
  make install
}
```

```
# run my_soft
export LD_LIBRARY_PATH=${HOME}/sw/${LUIS_CPU_ARCH}/my-
soft/3.1/install/lib:${LD_LIBRARY_PATH}
srun ${HOME}/sw/${LUIS_CPU_ARCH}/my-soft/3.1/install/bin/my-soft.exe --
input file.input
```

EasyBuild

Note: If you want to manually build the software from source code, please refer to the section above.

EasyBuild is a software build and installation framework that allows you to manage (scientific) software on High Performance Computing (HPC) systems in an efficient way.

EasyBuild framework

The EasyBuild framework is available in the cluster through the module `EasyBuild-custom`. This module defines the location of the EasyBuild configuration files, recipes and installation directories. You can load the module using the command:

```
module load EasyBuild-custom
```

EasyBuild software and modules will be installed by default under the following directory:

```
$HOME/my.soft/software/${LUIS_CPU_ARCH}
$HOME/my.soft/modules/${LUIS_CPU_ARCH}
```

Here, the variable ARCH, which stores the CPU type of the machine on which the above module load command was executed, will currently be either `haswell`, `sandybridge` or `skylake`. The command `lcpuarchs` executed on the cluster login nodes lists all currently available values of ARCH. You can override the default software and module installation directory, and the location of your EasyBuild configuration files (`MY_EASYBUILD_REPOSITORY`) by exporting the following environment variables before loading the EasyBuild module:

```
export EASYBUILD_INSTALLPATH=/your/preferred/installation/dir
export MY_EASYBUILD_REPOSITORY=/your/easybuild/repository/dir
module load EasyBuild-custom
```

If other project members should also have access to the software, the recommended location is a subdirectory in $SOFTWARE.

How to build your software

After you load the EasyBuild environment as explained in the section above, you will have the

command `eb` available to build your code using EasyBuild. If you want to build the code using a given configuration `<filename>.eb` and resolving dependencies, you will use the flag `-r` as in the example below:

```
eb <filename>.eb -r
```

The build command just needs the configuration file name with the extension `.eb` and not the full path, provided that the configuration file is in your search path: the command `eb --show-config` will print the variable `robot-paths` that holds the search path. More options are available - please have a look at the short help message typing `eb -h`. For instance, using the search flag `-S`, you can check if any EasyBuild configuration file already exists for a given program name:

```
eb -S <program_name>
```

You will be able to load the modules created by EasyBuild in the directory defined by the `EASYBUILD_INSTALLPATH` variable using the following commands:

```
module use $EASYBUILD_INSTALLPATH/modules/${LUIS_CPU_ARCH}/all
module load <modulename>/version
```

The command `module use` will prepend the selected directory to your `MODULEPATH` environment variable, therefore the command `module avail` will show modules of your software as well.

If you want the software module to be automatically available when opening a new shell in the cluster, modify your `~/.bashrc` file as follows:

```
 echo 'export EASYBUILD_INSTALLPATH=/your/preferred/installation/dir' >>
~/.bashrc
 echo 'module use $EASYBUILD_INSTALLPATH/modules/${LUIS_CPU_ARCH}/all' >>
~/.bashrc
```

Note that to preserve the dollar sign in the second line above, the string must be enclosed in single quotes.

Further Reading

- EasyBuild documentation
- Easyconfigs repository

1)

https://lmod.readthedocs.io/en/latest/010_user.html

2)

https://easybuild.readthedocs.io/en/latest/

From:
https://docs.cluster.uni-hannover.de/ - **Cluster Docs**

Permanent link:
https://docs.cluster.uni-hannover.de/doku.php/guide/to_pdf/400_modules_and_application_software

Last update: **2026/09/15 16:33**