Module System

1.0 Modules

To be able to deal with numerous software/libraries/compilers and their versions, Vorna, Ukko2, and Kale use a module system. With the module system, you can easily customize your working environment to be exactly as you want it.

The modules system modifies the environment variables of the user's shell so that the correct versions of executables are in the path and the linker can find the correct version of needed libraries. For example, the command *mpicc* points to different compilers depending on the loaded module. Loaded modules do propagate to the compute nodes like any other environment variables when you submit a slurm job.

It is recommended to do a 'module purge' followed by 'module load <intended module list here>' in the batch script immediately after #SBATCH -lines. This guarantees that your login host module environment is purged and replaced with the intended execution environment. You can always use *module save|restore* to construct sets of preloaded modules.

Note that not all modules are compatible with each other. Most of the modules have dependencies set, and they check for compatibility, but there may be rare exceptions.

Not all module repositories are set as system defaults. You can control the available repositories, see HPC Environment User Guide#3.1SoftwareModules for details.

1.1 Module system, programs, and modules

To list all available modules that are compatible with the modules that you have currently loaded:

```
module avail
```

To load a module:

```
module load <modulename>
```

To switch or swap between two modules:

```
module swap <module1> <module2>
```

To list all loaded modules:

```
module list
```

To get info about a module:

```
module help <module-name>
```

To list all available modules:

module spider

If you add the name or part of the name as an argument, you will get more information about matching modules. If a version number is provided, you also see that module's requirements.

```
module spider int
```

To unload a module:

```
module unload <module-name>
```

To unload all modules:

```
module purge
```

You can conveniently save loaded sets of modules into bundles, which you may then restore at any time. Without arguments, the set is default.

```
module save <name of set | default>
```

To restore module sets:

```
module restore <name of set | default>
```

1.2 Creating modules for your own software

As you are aware by now, we have quite a few modules available across the clusters. That said, our goal is to concentrate on the development tools, libraries, and packages that have a larger user base so you may be missing some specific, more rare software.

The good news is that you can create modules by yourself, and it isn't even complicated. You might wish to do it for better version control if you have multiple versions of software you'd like to use. Also, modules help you keep organized when you have several applications under development, or in use. Additionally, once you create a few, why limit it there? Of course, you can also create modules for your entire research group to use. Fundamental principles are simple.

First, we have to create a path for the modules to reside. This will prepend the path to the $MODULEPATH environment variable and they will have precedence over the system modules:

```
module use /proj/$USER/MyModules
```

If you would like not to use the path just created, just say:

```
module unuse /proj/$USER/MyModules
```

Now, let's start creating some modules. First, we could create the directory for all of the module files to reside:

```
mkdir /proj/$USER/MyModules
```

Then, the binaries will need a home (*ordinarily you would have a bit more than just binaries*). Pathname and the version number has a purpose, evident a bit later:

```
mkdir /proj/$USER/MySoftware/1.0.0/bin
```

...*and compile Hello world! there*...

Then the hard part, which is to create the module .lua -file itself. The idea of the .lua file is to point the module command where all the necessary bits and pieces are and to change your $PATH variable accordingly. There are some extra information bits included, which are not necessary but adds a nice touch.

```
help ([[For detailed instructions, go to:
 https://...
]])

whatis("Version: 1.0.0")
whatis("Keywords: Own Software")
whatis("URL: ")
whatis("Description: This is Hello World!")

setenv(       "HELLOPATH",        "/proj/juhaheli/MySoftware/1.0.0/bin")
prepend_path( "PATH",            "/proj/juhaheli/MySoftware/1.0.0/bin")
prepend_path( "LD_LIBRARY_PATH","/proj/juhaheli/MySoftware/1.0.0/lib")
```

Note that in our case there is no lib path in use, but I've added it in the example because more elaborate programs probably would require at least some libraries.

Finally, to use the creation, you have to say:

```
module use /proj/$USER/MyModules
```

And now if you say (*--ignore_cache just to make sure your list is not the cached one*):

```
module --ignore_cache avail
```

The output would be like this. *Note that the directory version number is neatly included - if wondering why directory naming was done as it was*:

```
--------------------------------------------------------------------------/proj/me/MyModules ---------------------------------------------------------------------------
  hello/1.0.0
```

You will see your own module on the top of the module list. Now you can load the modules just with the regular command:

```
module load hello
```

and then start using it.

To have your newly created modules available automatically upon your login, you would need to include this in your startup scripts (*in case of bash, this would be .profile in your home directory*).

```
module use /proj/$USER/MyModules
```

More advanced modules could include elaborate dependencies from other modules and they can be written in. In this case, once our example module is loaded, it would automatically load the required dependencies. You can use:

```
module show hello
```

To see the actual module file of our own, or by replacing the hello with the appropriate module name, any other module for further examples.

Congratulations! You have created your first module.

2.0 Virtual Environments

To create a virtual environment, in this example Python 3.5.2:

Purging modules

If not sure if correct modules are loaded, or not sure about dependencies, it is useful to purge and load the correct set.

```
cd /proj/$USER

module purge

module load Python/3.5.2-foss-2016b

mkdir python352 # any directory name, for your virtual environment

virtualenv python352

cd python352

source bin/activate

pip install biopython numpy tensorflow keras # etc, any python packages you need
```

You may also install additional packages with pip inside the virtualenv:

```
./configure --prefix=$HOME/python352

make

make install
```

For example, Tensorflow does need a virtual environment to be set up.

If a module or a program you need is missing, contact IT support at it4science@helsinki.fi

3.0 Additional Resources

NERSC has very good resources about modules, including examples if you wish to create your own.

If you wish to build your own modules.