Argon Software List

General Information

The sub pages of this page list the software packages that are installed on Argon. Note that combining our HPC systems, and making Argon "live forever", necessitated some changes in how the software is managed. Previously, the stack would be viable for the life of a cluster system but that is no longer feasible given the long lived nature of Argon. In the new HPC model there needs to be a way to update all levels of the software stack while maintaining a consistent set of dependencies, while still maintaining older versions of software should they be needed. This is accomplished by updating the entire stack of software but maintaining the previous stacks as well. These stacks are versioned so that they can be tracked and documented as a set. The software packages are managed via Environment Modules. One extra level of the environment module hierarchy was introduced to manage this, with a versioned "stack" module. The original set of packages will be maintained as the "legacy" stack. The sets of packages are made visible by loading a "stack" module. To set the default stack module, and possibly other environment modules to be loaded by default, set a default module set.

Concomitant with enabling this scheme, the package building framework was changed from an in-house developed system to a community project based framework. One of the effects of this was a change to the naming scheme. The Legacy modules were named as the upstream project named them, as much as possible, in terms of format. With the new framework everything is named with lowercase. For example, R becomes r.

In addition, extendable packages, such as R and Python, now have each package installed as a module. This is different behavior from the Legacy stack where packages were all loaded together for R and Python. This should make the packages that are available more visible when using module avail while logged into Argon. In addition, it minimizes the number of non-python/R modules that have to be loaded into your environment. While the trade off is that more modules are presented, it allows for more flexibility in managing dependencies. It also means that one can load just the Python/R packages that are needed, and allows for using different versions if need be.

Also note that each package has a compiler suffix. This generally means that the package was compiled with that compiler but in cases where the package was not compiled, it is just a placeholder to have consistency as a dependency for other packages built with that compiler. We do not create hierarchical levels for compilers so make sure to look at the compiler string when loading modules, if applicable.

All modules can be seen with the module spider command. Once a stack is loaded then the module avail command can be used to see what modules are available in the loaded stack. For example,

```bash
$ module spider bcftools
```

```
---
bcftools:
---

Versions:
bcftools/1.3.1
bcftools/1.9_gcc-4.9.4
bcftools/1.9_gcc-9.1.0
bcftools/1.10.2_gcc-4.9.4
bcftools/1.10.2_gcc-8.3.0
bcftools/1.10.2_gcc-9.2.0
bcftools/1.10.2_intel-19.0.5.281
```

```
----
For detailed information about a specific "bcftools" module (including how to load the modules) use the module's full name. For example:

```
$ module spider bcftools/1.9_gcc-9.1.0
```
```
```
---
```

Following the example given from the output of module spider we can drill down.

Note that if you want to use the packrat package in these newer versions, you must separately load the "r-packrat" environment module for the corresponding stack R module. See Adding R programs to a personal library.
$ module spider bcftools/1.9_gcc-9.1.0

--------------------------------------------------------------------------------------------------------------------
----
bcftools: bcftools/1.9_gcc-9.1.0
--------------------------------------------------------------------------------------------------------------------
----

You will need to load all module(s) on any one of the lines below before the "bcftools/1.9_gcc-9.1.0" module is available to load.

stack/2019.1

So we can load that version of bcftools by first loading stack/2019.1 module.

$ module load stack/2019.1
$ module load bcftools/1.9_gcc-9.1.0

Note that newer versions of packages will become the default after loading a "stack" module so be careful with versions.

The module system can also automatically swap versions for you but be careful to check that it is loading the module versions that you want. This will only work if you have not specified an explicit version. For example,

$ module purge
$ module load bcftools
$ module list bcftools
   1) bcftools/1.3.1

$ module load stack/2019.1

The following have been reloaded with a version change:
   1) bcftools/1.3.1 => bcftools/1.9_gcc-9.1.0      3) zlib/1.2.11 => zlib/1.2.11_gcc-9.1.0
   2) htslib/1.3.2 => htslib/1.9_gcc-9.1.0

$ module list bcftools

Currently Loaded Modules Matching: bcftools
   1) bcftools/1.9_gcc-9.1.0

You can swap one stack for another. Continuing with the bcftools example.
$ module list bcftools

Currently Loaded Modules Matching: bcftools
  1) bcftools/1.9_gcc-9.1.0

$ module load stack/2020.1

Inactive Modules:
  1) libzip/1.2.0_gcc-9.1.0

The following have been reloaded with a version change:
  1) bcftools/1.9_gcc-9.1.0 => bcftools/1.10.2_intel-19.0.5.281
  2) bzip2/1.0.6_gcc-9.1.0 => bzip2/1.0.8_intel-19.0.5.281
  3) curl/7.52.1 => curl/7.63.0_intel-19.0.5.281
  4) gsl/2.3 => gsl/2.5_intel-19.0.5.281
  5) htslib/1.9_gcc-9.1.0 => htslib/1.10.2_intel-19.0.5.281
  6) stack/2019.1 => stack/2020.1
  7) xz/5.2.4_gcc-9.1.0 => xz/5.2.4_intel-19.0.5.281
  8) zlib/1.2.11_gcc-9.1.0 => zlib/1.2.11_intel-19.0.5.281

$ module list bcftools

Currently Loaded Modules Matching: bcftools
  1) bcftools/1.10.2_intel-19.0.5.281

The above also shows why you have to verify that the auto swapping is correct as you can see that now the compiler has changed, which may not be what you want.

To see the python modules available after loading a versioned stack module,

```
module -r spider ^py--
or
module -r avail ^py--
```

Likewise, for R,

```
module -r spider ^r--
or
module -r avail ^r--
```

**Versioned Software Stacks**

These software stacks are made available by loading the environment module that corresponds to the stack version. That will make the modules available to be loaded. For instance,

```
module load stack/2019.1
```

will load the 2019.1 version of the stack module and then make the modules in that stack available. It is important to note that the modules in a stack will not show up in the output of

```
module avail
```

unless the stack module is loaded. Those modules will, however, be visible with the

```
module list
```
General Information

The sub pages of this page list the software packages that are installed on Argon. Note that combining our HPC systems, and making Argon "live forever", necessitated some changes in how the software is managed. Previously, the stack would be viable for the life of a cluster system but that is no longer feasible given the long lived nature of Argon. In the new HPC model there needs to be a way to update all levels of the software stack while maintaining a consistent set of dependencies, while still maintaining older versions of software should they be needed. This is accomplished by updating the entire stack of software but maintaining the previous stacks as well. These stacks are versioned so that they can be tracked and documented as a set. The software packages are managed via Environment Modules. One extra level of the environment module hierarchy was introduced to manage this, with a versioned "stack" module. The original set of packages will be maintained as the "legacy" stack. The sets of packages are made visible by loading a \"stack\" module. To set the default stack module, and possibly other environment modules to be loaded by default, set a default module set.

Concomitant with enabling this scheme, the package building framework was changed from an in-house developed system to a community project based framework. One of the effects of this was a change to the naming scheme. The Legacy modules were named as the upstream project named them, as much as possible, in terms of format. With the new framework everything is named with lowercase. For example, R becomes r.

In addition, extendable packages, such as R and Python, now have each package installed as a module. This is different behavior from the Legacy stack where packages were all loaded together for R and Python. This should make the packages that are available more visible when using \texttt{module avail} while logged into Argon. In addition, it minimizes the number of non-python/R modules that have to be loaded into your environment. While the trade off is that more modules are presented, it allows for more flexibility in managing dependencies. It also means that one can load just the Python/R packages that are needed, and allows for using different versions if need be.

Also note that each package has a compiler suffix. This generally means that the package was compiled with that compiler but in cases where the package was not compiled, it is just a placeholder to have consistency as a dependency for other packages built with that compiler. We do not create hierarchical levels for compilers so make sure to look at the compiler string when loading modules, if applicable.

All modules can be seen with the \texttt{module spider} command. Once a stack is loaded then the \texttt{module avail} command can be used to see what modules are available in the loaded stack. For example,

\begin{verbatim}
$ module spider bcftools

----------------------------------------------------------------------------------------------------------------
----
bcftools:
----------------------------------------------------------------------------------------------------------------
----
  Versions:
  bcftools/1.3.1
  bcftools/1.9_gcc-4.9.4
  bcftools/1.9_gcc-9.1.0
  bcftools/1.10.2_gcc-4.9.4
  bcftools/1.10.2_gcc-8.3.0
  bcftools/1.10.2_gcc-9.2.0
  bcftools/1.10.2_intel-19.0.5.281

----------------------------------------------------------------------------------------------------------------
----
For detailed information about a specific "bcftools" module (including how to load the modules) use the module's full name.
For example:

  $ module spider bcftools/1.9_gcc-9.1.0

----------------------------------------------------------------------------------------------------------------
----
\end{verbatim}

Following the example given from the output of \texttt{module spider} we can drill down.
You will need to load all module(s) on any one of the lines below before the "bcftools/1.9_gcc-9.1.0" module is available to load.

stack/2019.1

So we can load that version of bcftools by first loading stack/2019.1 module.

$ module load stack/2019.1
$ module load bcftools/1.9_gcc-9.1.0

Note that newer versions of packages will become the default after loading a "stack" module so be careful with versions.

The module system can also automatically swap versions for you but be careful to check that it is loading the module versions that you want. This will only work if you have not specified an explicit version. For example,

$ module purge
$ module load bcftools
$ module list bcftools

Currently Loaded Modules Matching: bcftools
  1) bcftools/1.3.1

$ module load stack/2019.1

The following have been reloaded with a version change:
  1) bcftools/1.3.1 => bcftools/1.9_gcc-9.1.0       3) zlib/1.2.11 => zlib/1.2.11_gcc-9.1.0
  2) htslib/1.3.2 => htslib/1.9_gcc-9.1.0

$ module list bcftools

Currently Loaded Modules Matching: bcftools
  1) bcftools/1.9_gcc-9.1.0

You can swap one stack for another. Continuing with the bcftools example.
$ module list bcftools

Currently Loaded Modules Matching: bcftools
  1) bcftools/1.9_gcc-9.1.0

$ module load stack/2020.1

Inactive Modules:
  1) libzip/1.2.0_gcc-9.1.0

The following have been reloaded with a version change:
  1) bcftools/1.9_gcc-9.1.0 => bcftools/1.10.2_intel-19.0.5.281
  2) bzip2/1.0.6_gcc-9.1.0 => bzip2/1.0.8_intel-19.0.5.281
  3) curl/7.52.1 => curl/7.63.0_intel-19.0.5.281
  4) gsl/2.3 => gsl/2.5_intel-19.0.5.281
  5) htslib/1.9_gcc-9.1.0 => htslib/1.10.2_intel-19.0.5.281
  6) stack/2019.1 => stack/2020.1
  7) xz/5.2.4_gcc-9.1.0 => xz/5.2.4_intel-19.0.5.281
  8) zlib/1.2.11_gcc-9.1.0 => zlib/1.2.11_intel-19.0.5.281

$ module list bcftools

Currently Loaded Modules Matching: bcftools
  1) bcftools/1.10.2_intel-19.0.5.281

The above also shows why you have to verify that the auto swapping is correct as you can see that now the compiler has changed, which may not be what you want.

To see the python modules available after loading a versioned stack module,

```
module -r spider ^py--
or
module -r avail ^py--
```

Likewise, for R,

```
module -r spider ^r--
or
module -r avail ^r--
```

**Versioned Software Stacks**

These software stacks are made available by loading the environment module that corresponds to the stack version. That will make the modules available to be loaded. For instance,

```
module load stack/2019.1
```

will load the 2019.1 version of the stack module and then make the modules in that stack available. It is important to note that the modules in a stack will not show up in the output of

```
module avail
```

unless the stack module is loaded. Those modules will, however, be visible with the
General Information

The sub pages of this page list the software packages that are installed on Argon. Note that combining our HPC systems, and making Argon "live forever", necessitated some changes in how the software is managed. Previously, the stack would be viable for the life of a cluster system but that is no longer feasible given the long lived nature of Argon. In the new HPC model there needs to be a way to update all levels of the software stack while maintaining a consistent set of dependencies, while still maintaining older versions of software should they be needed. This is accomplished by updating the entire stack of software but maintaining the previous stacks as well. These stacks are versioned so that they can be tracked and documented as a set. The software packages are managed via Environment Modules. One extra level of the environment module hierarchy was introduced to manage this, with a versioned "stack" module. The original set of packages will be maintained as the "legacy" stack. The sets of packages are made visible by loading a "stack" module.

To set the default stack module, and possibly other environment modules to be loaded by default, set a default module set.

Concomitant with enabling this scheme, the package building framework was changed from an in-house developed system to a community project based framework. One of the effects of this was a change to the naming scheme. The Legacy modules were named as the upstream project named them, as much as possible, in terms of format. With the new framework everything is named with lowercase. For example, R becomes r.

In addition, extendable packages, such as R and Python, now have each package installed as a module. This is different behavior from the Legacy stack where packages were all loaded together for R and Python. This should make the packages that are available more visible when using \texttt{module avail} while logged into Argon. In addition, it minimizes the number of non-python/R modules that have to be loaded into your environment. While the trade off is that more modules are presented, it allows for more flexibility in managing dependencies. It also means that one can load just the Python/R packages that are needed, and allows for using different versions if need be.

Also note that each package has a compiler suffix. This generally means that the package was compiled with that compiler but in cases where the package was not compiled, it is just a placeholder to have consistency as a dependency for other packages built with that compiler. We do not create hierarchical levels for compilers so make sure to look at the compiler string when loading modules, if applicable.

All modules can be seen with the \texttt{module spider} command. Once a stack is loaded then the \texttt{module avail} command can be used to see what modules are available in the loaded stack. For example,

```
$ module spider bcftools
```

```
-----------------------------------------------
----
bcftools:
-----------------------------------------------

    Versions:
        bcftools/1.3.1
        bcftools/1.9_gcc-4.9.4
        bcftools/1.9_gcc-9.1.0
        bcftools/1.10.2_gcc-4.9.4
        bcftools/1.10.2_gcc-8.3.0
        bcftools/1.10.2_gcc-9.2.0
        bcftools/1.10.2_intel-19.0.5.281

-----------------------------------------------

    For detailed information about a specific "bcftools" module (including how to load the modules) use the module's full name.
    For example:

    $ module spider bcftools/1.9_gcc-9.1.0
```

Following the example given from the output of \texttt{module spider} we can drill down.
$ module spider bcftools/1.9_gcc-9.1.0

---
bcftools: bcftools/1.9_gcc-9.1.0
---

You will need to load all module(s) on any one of the lines below before the "bcftools/1.9_gcc-9.1.0" module is available to load.

stack/2019.1

So we can load that version of bcftools by first loading stack/2019.1 module.

$ module load stack/2019.1
$ module load bcftools/1.9_gcc-9.1.0

Note that newer versions of packages will become the default after loading a "stack" module so be careful with versions.

The module system can also automatically swap versions for you but be careful to check that it is loading the module versions that you want. This will only work if you have not specified an explicit version. For example,

$ module purge
$ module load bcftools
$ module list bcftools

Currently Loaded Modules Matching: bcftools
1) bcftools/1.3.1

$ module load stack/2019.1

The following have been reloaded with a version change:
1) bcftools/1.3.1 => bcftools/1.9_gcc-9.1.0
2) htslib/1.3.2 => htslib/1.9_gcc-9.1.0
3) zlib/1.2.11 => zlib/1.2.11_gcc-9.1.0

$ module list bcftools

Currently Loaded Modules Matching: bcftools
1) bcftools/1.9_gcc-9.1.0

You can swap one stack for another. Continuing with the bcftools example.
$ module list bcftools

Currently Loaded Modules Matching: bcftools
  1) bcftools/1.9_gcc-9.1.0

$ module load stack/2020.1

Inactive Modules:
  1) libzip/1.2.0_gcc-9.1.0

The following have been reloaded with a version change:
  1) bcftools/1.9_gcc-9.1.0 => bcftools/1.10.2_intel-19.0.5.281
  2) bzip2/1.0.6_gcc-9.1.0 => bzip2/1.0.8_intel-19.0.5.281
  3) curl/7.52.1 => curl/7.63.0_intel-19.0.5.281
  4) gsl/2.3 => gsl/2.5_intel-19.0.5.281
  5) htslib/1.9_gcc-9.1.0 => htslib/1.10.2_intel-19.0.5.281
  6) stack/2019.1 => stack/2020.1
  7) xz/5.2.4_gcc-9.1.0 => xz/5.2.4_intel-19.0.5.281
  8) zlib/1.2.11_gcc-9.1.0 => zlib/1.2.11_intel-19.0.5.281

$ module list bcftools

Currently Loaded Modules Matching: bcftools
  1) bcftools/1.10.2_intel-19.0.5.281

The above also shows why you have to verify that the auto swapping is correct as you can see that now the compiler has changed, which may not be what you want.

To see the python modules available after loading a versioned stack module,

```bash
module -r spider ^py--
or
module -r avail ^py--
```

Likewise, for R,

```bash
module -r spider ^r--
or
module -r avail ^r--
```

**Versioned Software Stacks**

These software stacks are made available by loading the environment module that corresponds to the stack version. That will make the modules available to be loaded. For instance,

```bash
module load stack/2019.1
```

will load the 2019.1 version of the stack module and then make the modules in that stack available. It is important to note that the modules in a stack will not show up in the output of

```bash
module avail
```

unless the stack module is loaded. Those modules will, however, be visible with the
Optimizations

Beginning with the 2020.2 stack, the environment modules point to packages that were compiled with optimizations specific to the micro-architectures of the node type the modules are loaded on. This means that the binaries are not identical between different node types. The login nodes are different in that the modules loaded on login nodes will point to the lowest common micro-architecture. This means that if you use the module packages for development on a login node, the resultant binaries will run on all of the Argon node types. However, if you use the modules for development on a compute node, the resultant binaries will link against libraries built for the node type used, and may fail to run on less capable compute nodes. To avoid that, there is a `stack/2020.2-base_arch` module for the base architecture that can be used. If developing software on a compute node, load the `stack/2020.2-base_arch` module rather than the `stack/2020.2` module. That will point to binaries of the base architecture and binaries linked to these will run on all compute nodes. That is assuming that no additional compiler flags were added during the compilation that would create host specific optimizations. Another reason that you may want to use the `stack/2020.2-base_arch` module is to ensure that the binaries used at run time are identical across different nodes.