Lmod Documentation

Release 8.7.37

Robert McLay

Apr 13, 2024

CONTENTS

CHAPTER

ONE

MONTHLY ZOOM MEETING

NOTE Lmod is holding Monthly Zoom meeting to discuss various topics. Typically it is the first Tuesday of the Month at 9:30 U.S. Central (which is 14:30 UTC or 15:30 UTC in the winter months). Beginners are welcome. There is always a Q/A session at the beginning. Topic announcements are sent to the Lmod mailing list.

See: https://github.com/TACC/Lmod/wiki for details.

1.1 PURPOSE

Lmod is a Lua based module system that easily handles the MODULEPATH Hierarchical problem. Environment Modules provide a convenient way to dynamically change the users' environment through modulefiles. This includes easily adding or removing directories to the PATH environment variable. Modulefiles for Library packages provide environment variables that specify where the library and header files can be found.

1.2 OVERVIEW

This guide is written to explain what Environment Modules are and why they are very useful for both users and system administrators. Lmod is an implementation of Environment Modules, much of what is said here is true for any environment modules system but there are many features which are unique to Lmod.

Environment Modules provide a convenient way to dynamically change the users' environment through modulefiles. This includes easily adding or removing directories to the PATH environment variable.

A modulefile contains the necessary information to allow a user to run a particular application or provide access to a particular library. All of this can be done dynamically without logging out and back in. Modulefiles for applications modify the user's path to make access easy. Modulefiles for Library packages provide environment variables that specify where the library and header files can be found.

Packages can be loaded and unloaded cleanly through the module system. All the popular shells are supported: bash, ksh, rc, csh, tcsh, fish, zsh. Also available for perl, python, lisp, cmake, and R.

It is also very easy to switch between different versions of a package or remove it.

1.3 Lmod Web Sites

- Documentation: https://lmod.readthedocs.io/en/latest/
- GitHub: https://github.com/TACC/Lmod
- SourceForge: https://lmod.sf.net
- TACC Homepage: https://www.tacc.utexas.edu/research-development/tacc-projects/lmod
- Lmod Test Suite: https://github.com/rtmclay/Lmod_test_suite
- Join Lmod Mailing list: https://sourceforge.net/projects/lmod/lists/lmod-users

The most up-to-date source is at github. There is a secondary git repo found at SourceForge. Both repos are the same. Stable releases in the form of tar files can be found at sourceforge. All label versions found at the git repos have passed Lmod's regression test suite.

1.4 Introduction to Lmod

If you are new to Lmod then please read the User Guide and possibly the Frequently Asked Questions Guide. Users who wish to read about how to have their own personal modules should read the Advanced User Guide.

1.4.1 User Guide for Lmod

The guide here explains how to use modules. The User's tour of the module command covers the basic uses of modules. The other sections can be read at a later date as issues come up. The Advance user's guide is for users needing to create their own modulefiles.

User's Tour of the Module Command

The module command sets the appropriate environment variable independent of the user's shell. Typically the system will load a default set of modules. A user can list the modules loaded by:

```
$ module list
```

To find out what modules are available to be loaded a user can do:

```
$ module avail
```

To load packages a user simply does:

```
$ module load package1 package2 ...
```

To unload packages a user does:

```
$ module unload package1 package2 ...
```

A user might wish to change from one compiler to another:

```
$ module swap gcc intel
```

The above command is short for:

```
$ module unload gcc
$ module load intel
```

A user may wish to go back to an initial set of modules:

```
$ module reset
```

This will unload all currently loaded modules, including the sticky ones, then load the list of modules specified by LMOD_SYSTEM_DEFAULT_MODULES. There is a related command:

```
$ module restore
```

This command will also unload all currently loaded modules, including the sticky ones, and then load the system default unless the user has a default collection. See *User Collections* for more details.

If there are many modules on a system, it can be difficult to see what modules are available to load. Lmod provides the overview command to provide a concise listing. For example:

```
$ module overview

------------------ /opt/apps/modulefiles/Core -----------------
StdEnv    (1)   hashrf    (2)   papi        (2)   xalt     (1)
ddt       (1)   intel     (2)   singularity (2)
git       (1)   noweb     (1)   valgrind    (1)

--------------- /opt/apps/lmod/lmod/modulefiles/Core ----------
lmod (1)   settarg (1)
```

This shows the short name of the module (i.e. git, or singularity) and the number in the parenthesis is the number of versions for each. This list above shows that there is one version of git and two versions of singularity.

If a module is not available then an error message is produced:

```
$ module load packageXYZ
Warning: Failed to load: packageXYZ
```

It is possible to try to load a module with no error message if it does not exist. Any other failures to load will be reported.:

```
$ module try-load packageXYZ
```

Modulefiles can contain help messages. To access a modulefile's help do:

```
$ module help packageName
```

To get a list of all the commands that module knows about do:

```
$ module help
```

The module avail command has search capabilities:

```
$ module avail cc
```

will list for any modulefile where the name contains the string "cc".

Users may wish to test whether certain modules are already loaded:

```
$ module is-loaded packageName1 packageName2 ...
```

Lmod will return a true status if all modules are loaded and a false status if one is not. Note that Lmod is setting the status bit, there is nothing printed out. This means that one can do the following:

```
$ if module is-loaded pkg ; then echo "pkg is loaded"; fi
```

Users also may wish to test whether certain modules can be loaded with the current $MODULEPATH:

```
$ module is-avail packageName1 packageName2 ...
```

Lmod will a true status if all modules are available and false if one can not be loaded. Again this command sets the status bit.

Modulefiles can have a description section known as "whatis". It is accessed by:

```
$ module whatis pmetis
pmetis/3.1  : Name: ParMETIS
pmetis/3.1  : Version: 3.1
pmetis/3.1  : Category: library, mathematics
pmetis/3.1  : Description: Parallel graph partitioning..
```

There is a keyword search tool:

```
$ module keyword word1 word2 ...
```

This will search any help message or whatis description for the word(s) given on the command line.

Another way to search for modules is with the "module spider" command. This command searches the entire list of possible modules. The difference between "module avail" and "module spider" is explained in the "Module Hierarchy" and "Searching for Modules" section.:

```
$ module spider
```

Users can also find which categories a site provides:

```
$ module category

------------------ List of Categories --------------------
Compiler           Programming tools        library
Graph partitioner  System Environment/Base  mpi
MPI library        Visual Tool              tools
```

To know which modules, users can pick one or more name from the list of categories with the number of modules provide:

```
$ module category library

---------------------- MPI library -----------------------
mpich (30)   openmpi (6)

----------------------- library --------------------------
boost     (1)   hdf5  (18)   pdtoolkit (1)    pmetis (6)
fftw2     (6)   metis (2)    petsc     (13)   tau    (3)
gotoblas2 (1)   papi  (1)    phdf5     (40)
```

Here we see that there are 30 versions of mpich and 6 version of openmpi. Also that category name given, in this case "library" does partial matches and is case-insensitive.

Specifying modules to load

Modules are a way to ask for a certain version of a package. For example a site might have two or more versions of the gcc compiler collection (say versions 7.1 and 8.2). So a user may load:

```
$ module load gcc
```

or:

```
$ module load gcc/7.1
```

In the second case, Lmod will load gcc version 7.1 where as in the first case Lmod will load the default version of gcc which normally be 8.2 unless the site marks 7.1 as the default.

In this user guide, we call **gcc/7.1** the **fullName** of the module and **gcc** as the **shortName**. We also call what the user asked for as the **userName** which could either be the **fullName** or the **shortName** depending on what the user typed on the command line.

Showing the contents of a module

There are several ways to use the show sub-command to show the contents of a modulefile. The first is to show the module functions instead of executing them:

```
$ module show gcc
```

This shows the functions such as **setenv** () or **prepend_path** () but nothing else. If you want to know the contents of the modulefile you can use:

```
$ module --raw show gcc
```

This will show the raw text of the modulefile. This is same as printing the modulefile, but here Lmod will find the modulefile for you. If you want to know just the location of a modulefile do:

```
$ module --redirect --location show gcc
```

You will probably use the –redirect option so that the output goes to stdout and not stderr.

If you want to know how Lmod will parse a TCL modulefile you can do:

```
$ tclsh $LMOD_DIR/tcl2lua.tcl  <path_to_TCL_modulefile>
```

This useful when there is some question on how Lmod will treat a TCL modulefile.

ml: A convenient tool

For those of you who can't type the *mdoule*, *moduel*, err *module* command correctly, Lmod has a tool for you. With **ml** you won't have to type the module command again. The two most common commands are *module list* and *module load <something>* and **ml** does both:

```
$ ml
```

means *module list*. And:

```
$ ml foo
```

means *module load foo* while:

```
$ ml -bar
```

means *module unload bar*. It won't come as a surprise that you can combine them:

```
$ ml foo -bar
```

means *module unload bar; module load foo*. You can do all the module commands:

```
$ ml spider
$ ml avail
$ ml show foo
```

If you ever have to load a module name *spider* you can do:

```
$ ml load spider
```

If you are ever forced to type the **module** command instead of **ml** then that is a bug and should be reported.

clearLmod: Complete remove Lmod setup

It is rare, but sometimes a user might need to remove the Lmod setup from their current shell. This command can be used with bash/zsh/csh/tcsh to remove the Lmod setup:

```
$ clearLmod
```

This command prints a message telling the user what it has done. This message can be silented with:

```
$ clearLmod --quiet
```

SAFETY FEATURES

(1): Users can only have one version active: The One Name Rule

If a user does:

```
$ module avail xyz

--------------- /opt/apps/modulefiles ----------------
```

(continues on next page)

(continued from previous page)

```
xyz/8.1   xyz/11.1 (D)   xyz/12.1

$ module load xyz
$ module load xyz/12.0
```

The first load command will load the 11.1 version of xyz. In the second load, the module command knows that the user already has xyz/11.1 loaded so it unloads that and then loads xyz/12.0. This protection is only available with Lmod.

This is known as the *One Name* rule. This feature is core to how Lmod works and there is no way to override this.

(2) : Users can only load one compiler or MPI stack at a time.

Lmod provides an additional level of protection. If each of the compiler modulefiles add a line:

```
family("compiler")
```

Then Lmod will not load another compiler modulefile. Another benefit of the modulefile family directive is that an environment variable "LMOD_FAMILY_COMPILER" is assigned the name (and not the version). This can be useful specifying different options for different compilers. In the High Performance Computing (HPC) world, the message passing interface (MPI) libraries are important. The mpi modulefiles can contain a family("MPI") directive which will prevent users from loading more than one MPI implementation at a time. Also the environment variable "LMOD_FAMILY_MPI" is defined to the name of the mpi library.

Module Hierarchy

Libraries built with one compiler need to be linked with applications with the same compiler version. If sites are going to provide libraries, then there will be more than one version of the library, one for each compiler version. Therefore, whether it is the Boost library or an mpi library, there are multiple versions.

There are two main choices for system administrators. For the XYZ library compiled with either the UCC compiler or the GCC compiler, there could be the xyz-ucc modulefile and the xyz-gcc module file. This gets much more complicated when there are multiple versions of the XYZ library and different compilers. How does one label the various versions of the library and the compiler? Even if one makes sense of the version labeling, when a user changes compilers, the user will have to remember to unload the ucc and the xyz-ucc modulefiles when changing to gcc and xyz-gcc. If users have mismatched modules, their programs are going to fail in very mysterious ways.

A much saner strategy is to use a module hierarchy. Each compiler module adds to the MODULEPATH a compiler version modulefile directory. Only modulefiles that exist in that directory are packages that have been built with that compiler. When a user loads a particular compiler, that user only sees modulefile(s) that are valid for that compiler.

Similarly, applications that use libraries depending on MPI implementations must be built with the same compiler - MPI pairing. This leads to modulefile hierarchy. Therefore, as users start with the minimum set of loaded modules, all they will see are compilers, not any of the packages that depend on a compiler. Once they load a compiler they will see the modules that depend on that compiler. After choosing an MPI implementation, the modules that depend on that compiler-MPI pairing will be available. One of the nice features of Lmod is that it handles the hierarchy easily. If a user swaps compilers, then Lmod automatically unloads any modules that depends on the old compiler and reloads those modules that are dependent on the new compiler.

```
$ module list

1) gcc/4.4.5 2) boost/1.45.0

```

(continues on next page)

(continued from previous page)

```
$ module swap gcc ucc

Due to MODULEPATH changes the follow modules have been reloaded: 1) boost
```

If a modulefile is not available with the new compiler, then the module is marked as inactive. Every time MODULEPATH changes, Lmod attempts to reload any inactive modules.

Searching For Modules

When a user enters:

```
$ module avail
```

Lmod reports only the modules that are in the current MODULEPATH. Those are the only modules that the user can load. If there is a modulefile hierarchy, then a package the user wants may be available but not with the current compiler version. Lmod offers a new command:

```
$ module spider
```

which lists all possible modules and not just the modules that can be seen in the current MODULEPATH. This command has three modes. The first mode is:

```
$ module spider

lmod: lmod/lmod
Lmod: An Environment Module System

ucc: ucc/11.1, ucc/12.0, ...
Ucc: the ultimate compiler collection

xyz: xyz/0.19, xyz/0.20, xyz/0.31
xyz: Solves any x or y or z problem.
```

This is a compact listing of all the possible modules on the system. The second mode describes a particular module:

```
$ module spider ucc
----------------------------------------------------------------------------
ucc:
----------------------------------------------------------------------------

Description:
Ucc: the ultimate compiler collection

Versions:
ucc/11.1
ucc/12.0
```

The third mode reports on a particular module version and where it can be found:

```
$ module spider parmetis/3.1.1
----------------------------------------------------------------------------
parmetis: parmetis/3.1.1
```

(continues on next page)

(continued from previous page)

```
----------------------------------------------------------------------------
Description:
Parallel graph partitioning and fill-reduction matrix ordering routines

This module can be loaded through the following modules:
ucc/12.0, openmpi/1.4.3
ucc/11.1, openmpi/1.4.3
gcc/4.4.5, openmpi/1.4.3

Help:
The parmetis module defines the following environment variables: ...
The module parmetis/3.1.1 has been compiled by three different versions of the ucc␣
↪compiler and one MPI implementation.
```

Controlling Modules During Login

Normally when a user logs in, there are a standard set of modules that are automatically loaded. Users can override and add to this standard set in two ways. The first is adding module commands to their personal startup files. The second way is through the "module save" command.

To add module commands to users' startup scripts requires a few steps. Bash users can put the module commands in either their `~/.profile` file or their `~/.bashrc` file. It is simplest to place the following in their `~/.profile` file:

```
if [ -f ~/.bashrc ]; then
   .   ~/.bashrc
fi
```

and place the following in their `~/.bashrc` file:

```
if [ -z "$BASHRC_READ" ]; then
   export BASHRC_READ=1
   # Place any module commands here
   # module load git
fi
```

By wrapping the module command in an if test, the module commands need only be read in once. Any sub-shell will inherit the PATH and other environment variables automatically. On login shells the `~/.profile` file is read which, in the above setup, causes the `~/.bashrc` file to be read. On interactive non-login shells, the `~/.bashrc` file is read instead. Obviously, having this setup means that module commands need only be added in one file and not two.

Csh users need only specify the module commands in their `~/.cshrc` file as that file is always sourced:

```
if ( ! $?CSHRC_READ ) then
   setenv CSHRC_READ 1
   # Place any module command here
   # module load git
endif
```

User Collections

User defined initial list of login modules:

Assuming that the system administrators have installed Lmod correctly, there is a second way which is much easier to set up. A user logs in with the standard modules loaded. Then the user modifies the default setup through the standard module commands:

```
$ module unload XYZ
$ module swap gcc ucc
$ module load git
```

Once users have the desired modules load then they issue:

```
$ module save
```

This creates a file called `~/.config/lmod/default` which has the list of desired modules. Note only the current set of modules is recorded the in the collection. If module X loads module A and the user deletes module A before doing `module save collectionName` then module A will NOT be loaded when the collection is restored. All load(), always_load(), depends_on() statements inside the modulefiles are ignored when restoring a collection. Instead Lmod loads just the list of modulefiles stored in the collection.

Once this is set-up a user can issue:

```
$ module restore
```

and only the desired modules will be loaded. If Lmod is setup correctly (see *Providing A Standard Set Of Modules for all Users*) then the default collection will be the user's initial set of modules.

If a user doesn't have a default collection, the Lmod purges ALL currently loaded modules, including the sticky ones, and loads the list of module specified by LMOD_SYSTEM_DEFAULT_MODULES just like the `module reset` command.

Users can have as many collections as they like. They can save to a named collection with:

```
$ module save <collection_name>
```

and restore that named collection with:

```
$ module restore <collection_name>
```

A user can print the contents of a collection with:

```
$ module describe <collection_name>
```

A user can list the collections they have with:

```
$ module savelist
```

Finally a user can disable a collection with:

```
$ module disable <collection_name>
```

If no `collection_name` is given then the default is disabled. Note that the collection is not remove just renamed. If a user disables the foo collection, the file foo is renamed to foo~. To restore the foo collection, a user will have to do the following:

```
$ cd ~/.config/lmod;  mv foo~ foo
```

Rules for loading modules from a collection

Lmod has rules on what modules to load when restoring a collection. Remember that **userName** is what the user asked for, the **fullName** is the exact module name and **shortName** is name of the package (e.g. gcc, fftw3).

1. Lmod records the fullName and the userName in the collection.
2. If the userName is the same as the fullName then it loads fullName independent of the default.
3. if the userName is not the same as the fullName then it loads the default.
4. Unless LMOD_PIN_VERSIONS=yes then the fullName is always loaded.

In other words if a user does:

```
$ module --force purge; module load A B C
$ module save
```

then "**module restore**" will load the default A, B, and C. So if the default for module A changed between when the collection was saved and then restored, a new version of A will be loaded. This assumes that LMOD_PIN_VERSIONS is not set. If it is set or Lmod is configured that way then if A/1.1, B/2.4 and C/3.3 are the default then those modules will be loaded in the future independent of what the defaults are in the future.

On the other hand:

```
$ module --force purge; module load A/1.0 B/2.3 C/3.4
$ module save
```

then "**module restore**" will load the A/1.0, B/2.3, and C/3.4 independent of what the defaults are now or in the future.

User Collections on shared home file systems

If your site has a shared home file system, then things become a little more complicated. A shared home file system means that your site has a single home file system shared between two or more clusters. See *Lmod on Shared Home File Systems* for a system administrators point of view.

If you have a collection on one cluster it needs to be independent of another cluster. Your site should set $LMOD_SYSTEM_NAME uniquely for each cluster. Suppose you have cluster A and B. Then $LMOD_SYSTEM_NAME will be either A or B. A default collection will be named "default.A" for the A cluster and "default.B" for the B cluster. The names a user sees will have the extension removed. In other words on the A cluster a user would see:

```
$ module savelist

  1) default
```

where the default file is named "default.A".

Showing hidden modules

Sites modules (or user personal modules) can be hidden from normal "module avail" or "module spider" through different mechanisms. See *Hidden Modules*

To see hidden modules, one can do:

```
$ module --show_hidden avail
$ module --show_hidden spider
```

1.4.2 An Introduction to Writing Modulefiles

This is a different kind of introduction to Lmod. Here we will remind you what Lmod is doing to change the environment via modulefiles. Then we will start with the four functions that are typically needed for any modulefile. From there we will talk about intermediate level module functions when things get more complicated. Finally we will discuss the advanced module functions to flexibly control your site via modules. All the Lua module functions available are described at *Lua Modulefile Functions*. This discussion shows how they can be used.

A Reminder of what Lmod is doing

All Lmod is doing is changing the environment. Suppose you want to use the "ddt" debugger installed on your system which is made available to you via the module. If you try to execute ddt without the module loaded you get:

```
$ ddt
bash: command not found: ddt

$ module load ddt
$ ddt
```

After the ddt module is loaded, executing **ddt** now works. Let's remind ourselves why this works. If you try checking the environment before loading the ddt modulefile:

```
$ env | grep -i ddt
$ module load ddt
$ env | grep -i ddt

DDTPATH=/opt/apps/ddt/5.0.1/bin
LD_LIBRARY_PATH=/opt/apps/ddt/5.0.1/lib:...
PATH=/opt/apps/ddt/5.0.1/bin:...

$ module unload ddt
$ env | grep -i ddt
$
```

The first time we check the environment we find that there is no **ddt** stored there. But the second time there we see that the PATH and LD_LIBRARY_PATH have been modified. Note that we have shorten the path-like variables to show the important changes. There are also several environment variables which have been set. After unloading the module all the references for ddt have been removed. We can see what the modulefile looks like by doing:

```
$ module show ddt

help([[
```

(continues on next page)

(continued from previous page)

```
For detailed instructions, go to:
   https://...

]])
whatis("Version: 5.0.1")
whatis("Keywords: System, Utility")
whatis("URL: http://content.allinea.com/downloads/userguide.pdf")
whatis("Description: Parallel, graphical, symbolic debugger")

setenv(       "DDTPATH",        "/opt/apps/ddt/5.0.1/bin")
prepend_path( "PATH",           "/opt/apps/ddt/5.0.1/bin")
prepend_path( "LD_LIBRARY_PATH","/opt/apps/ddt/5.0.1/lib")
```

Modulefiles state the actions that need to happen when loading. For example the above modulefile uses **setenv** and **prepend_path** to set environment variables and prepend to the **PATH**. If the above modulefile is unloaded then the **setenv** actually unsets the environment variable. The **prepend_path** removes the element from the **PATH** variable. That is unload causes the functions to be reversed.

Basic Modulefiles

There are two main module functions required, namely **setenv** and **prepend_path**; and two functions to provide documentation **help** and **whatis**. The modulefile for ddt shown above contains all the basics required to create one. Suppose you are writing this module file for ddt version 5.0.1 and you are placing it in the standard location for your site, namely */apps/modulefiles* and this directory is already in **MODULEPATH**. Then in the directory */apps/modulefiles/ddt* you create a file called *5.0.1.lua* which contains the modulefile shown above.

This is the typical way of setting a modulefile up. Namely the package name is the name of the directory, *ddt*, and version name, *5.0.1* is the name of the file with the *.lua* extension added. We add the lua extension to all modulefile written in Lua. All modulefiles without the lua extension are assumed to be written in TCL.

If another version of ddt becomes available, say *5.1.2*, we create another file called *5.1.2.lua* to become the new modulefile for the new version of *ddt*.

When a user does *module help ddt*, the arguments to the **help** function are written out to the user. The **whatis** function provides a way to describe the function of the application or library. This data can be used by search tools such as **module keyword** *search_words*. Here at TACC we also use that data to provide search capability via the web interface to modules we provide.

Intermediate Level Modulefiles

The four basic functions describe above is all that is necessary for the majority of modulefiles for application and libraries. The intermediate level is designed to describe some situations that come up as you need to provide more than just packages modulefile but need to set up a system.

Meta Modules

Some sites create a single module to load a default set of modules for all users to start from. This is typically called a meta module because it loads other modules. As an example of that, we here at TACC have created the TACC module to provide a default compiler, mpi stack and other modules:

```
help([[
The TACC modulefile defines ...
]])

-- 1 --
if (os.getenv("USER") ~= "root") then
  append_path("PATH",  ".")
end

-- 2 --
load("intel", "mvapich2")

-- 3 --
try_load("xalt")

-- 4 --
-- Environment change - assume single threaded.
if (mode() == "load" and os.getenv("OMP_NUM_THREADS") == nil) then
  setenv("OMP_NUM_THREADS","1")
end
```

This modulefile shows the use of four new functions. The first one is **append_path**. This function is similar to **prepend_path** except that the value is placed at the end of the path-like variable instead of the beginning. We add `.` to our user's path at the end, except for root. This way our new users don't get surprised with some programs in their current directory that do not run. We used the **os.getenv** function built-in to Lua to get the value of environment variable "USER".

The second function is **load**, this function loads the modulefiles specified. This function takes one or more names. Here we are specifying a default compiler and mpi stack. The third function is **try_load**, which is similar to **load** except that there is no error reported if the module can't be found. Any other errors found during loading will be reported.

The fourth block of code shows how we set **OMP_NUM_THREADS**. We wish to set **OMP_NUM_THREADS** to have a default value of 1, but only if the value hasn't already been set and we only want to do this when the module is being loaded and not at any other time. So when this module is loaded for the first time **mode()** will return "load" and **OMP_NUM_THREADS** won't have a value. The **setenv** will set it to 1. If the TACC module is unloaded, the **mode()** will be "unload" so the if test will be false and therefore the **setenv** will not be reversed. If the user changes **OMP_NUM_THREADS** and reloads the TACC modulefile, their value won't change because **os.getenv("OMP_NUM_THREADS")** will return a non-nil value, therefore the **setenv** command won't run. Now this may not be the best way to handle this. It might be better to set **OMP_NUM_THREADS** in a file that is sourced in /etc/profile.d/ and have all the important properties. Namely that there will be a default value that the user can change. However this example shows how to do something tricky in a modulefile.

Typically meta modules are a single file and not versioned. So the TACC modulefile can be found at */apps/modulefiles/TACC.lua*. There is no requirement that this will be this way but it has worked well in practice.

Modules with dependencies

Suppose that you have a package which needs libraries or an application. For example the octave application needs gnuplot. Let's assume that you have a separate applications for both. Inside the octave module you can do:

```
prereq("gnuplot")
...
```

So if you execute:

```
$ module unload gnuplot
$ module load octave
$ module load gnuplot octave
$ module unload octave
```

The second module command will fail, but the third one will succeed because we have met the prerequisites. The advantage of using prereq is after fourth module command is executed, the gnuplot module will be loaded.

This can be contrasted with including the load of gnuplot in the octave modulefile:

```
load("gnuplot")
...
```

This simplifies the loading of the octave module. The trouble is that when a user does the following:

```
$ module load   gnuplot
$ module load   octave
$ module unload octave
```

is that after unloading *octave*, the *gnuplot* module is also unloaded. It seems better to either use the **prereq** function shown above or use the **always_load** function in the octave module:

```
always_load("gnuplot")
...
```

Then when a user does:

```
$ module load   gnuplot
$ module load   octave
$ module unload octave
```

The *gnuplot* module will still be loaded after unloading *octave*. This will lead to the least confusion to users.

Fancy dependencies

Sometimes an application may depend on another application but it has to be a certain version or newer. Lmod can support this with the **atleast** modifier to both **load**, **always_load** or **prereq**. For example:

```
-- Use either the always_load or prereq but not both:

prereq(     atleast("gnuplot","5.0"))
always_load(atleast("gnuplot","5.0"))
```

The **atleast** modifier to **prereq** or **always_load** will succeed if the version of gnuplot is 5.0 or greater.

Assigning Properties

Modules can have properties that will be displayed in a *module list* or *module avail*. Properties can be anything but they must be specified in the *lmodrc.lua* file. You are free to add to the list. For example, to specify a module to be experimental all you need to do is:

```
add_property("state","experimental")
```

Any properties you set must be defined in the **lmodrc.lua** file. In the source tree the properties are in init/lmodrc.lua. A more detailed discussion of the lmodrc.lua file can be found at *Module Properties*

Pushenv

Lmod allows you to save the state in a stack hidden in the environment. So if you want to set the **CC** environment variable to contain the name of the compiler.:

```
-- gcc --
pushenv("CC","gcc")

-- mpich --
pushenv("CC","mpicc")
```

If the user executes the following:

```
#                                      SETENV         PUSHENV
$ export CC=cc;         echo $CC  # -> CC=cc          CC=cc
$ module load   gcc;    echo $CC  # -> CC=gcc         CC=gcc
$ module load   mpich;  echo $CC  # -> CC=mpicc       CC=mpicc
$ module unload mpich;  echo $CC  # -> CC is unset    CC=gcc
$ module unload gcc;    echo $CC  # -> CC is unset    CC=cc
```

We see that the value of **CC** is maintained as a stack variable when we use *pushenv* but not when we use *setenv*.

Setting aliases and shell functions

Sometimes you want to set an alias as part of a module. For example the visit program requires the version to be specified when running it. So for version 2.9 of visit, the alias is set:

```
set_alias("visit","visit -v 2.9")
```

Whether this will expand correctly depends on the shell. While C-shell allows argument expansion in aliases, Bash and Zsh do not. Bash and Zsh use shell functions instead. For example the ml shell function can be set like this:

```
local bashStr = 'eval $($LMOD_DIR/ml_cmd "$@")'
local cshStr  = "eval `$LMOD_DIR/ml_cmd $*`"
set_shell_function("ml",bashStr,cshStr)
```

Please note that aliases in bash are not expanded for non-interactive shells. This means that it won't work in bash shell scripts. Please change the shell alias to use the `set_shell_function` instead. Shell functions do work in both interactive and non-interactive shells.

1.4.3 Frequently Asked Questions

How does the module command work?

We know that the child program inherits the parents' environment but not the other way around. So it is very surprising that a command can change the current shell's environment. The trick here is that the module command is a two part process. The module shell function in bash is:

```
$ type module
module() { eval $($LMOD_CMD bash "$@") }
```

Where $LMOD_CMD points to your lmod command (say /apps/lmod/lmod/libexec/lmod). So if you have a module file (foo/1.0) that contains:

```
setenv("FOO", "BAR")
```

then "$LMOD_CMD bash load foo/1.0" the following string written to stdout:

```
export FOO=BAR
...
```

The eval command read that output from stdout and changes the current shell's environment. Any text written to stderr bypasses the eval and written to the terminal.

What are the environment variables _ModuleTable001_, _ModuleTable002_, etc doing it in the environment?

The module command remembers its state in the environment through these variables. The way Lmod does it is through a Lua table called ModuleTable:

```
ModuleTable = {
   mT = {
     git = { ... }
   }
}
```

This table contains quotes and commas and must be store in environment. To prevent problems the various shells, the table is encoded into base64 and split into blocks of 256 characters. These variables are decoded at the start of Lmod. You can see what the module table contains with:

```
$ module --mt
```

How does one debug a modulefile?

There are two methods. Method 1: If you are writing a Lua modulefile then you can write messages to stderr with and run the module command normally:

```
local a = "text"
io.stderr:write("Message ",a,"\n")
```

Method 2: Take the output directly from Lmod. You can put print() statements in your modulefile and do:

```
$ $LMOD_CMD bash load *modulefile*
```

Why doesn't % `module avail |& grep ucc` work under tcsh and works under bash?

It is a bug in the way tcsh handles evals. This works:

```
% (module avail) |& grep ucc
```

However, in all shells it is better to use:

```
% module avail ucc
```

instead as this will only output modules that have "ucc" in their name.

Why does Lmod require a static location of lua? Why shouldn't a site allow Lmod to use the lua found in the path?

The short answer is that it is possible but for general use it is not a good idea. If you change the first line of the lmod script to be:

```
"#!/usr/bin/env lua"
```

and all the other Lmod executable scripts to do the same then all the scripts would use the lua found in the user's $PATH. There are other things that would have to change as well. Lmod carefully sets two env. vars: LUA_PATH and LUA_CPATH to be compatible with the Lua that was used to install lmod.

So why does Lmod go to this much trouble? Why doesn't Lmod just use the lua found in the path and not set LUA_PATH and LUA_CPATH. Earlier version of Lmod did not. The answer is users. Lmod has to protect itself from every user out there. What if a user installs their own Lua? What if your system install of Lua is version 5.1 but your user wants to install the latest version of Lua. The libraries that Lmod depends on change with version. What if a user installs their own version lua but doesn't install the required libraries for Lmod to work (lua-posix, lfs) or they install their own library called lfs but it does something completely different. Lmod would fail with very strange errors.

To sum up. Lmod is very careful to use the Lua that was used to install it and the necessary libraries. Lmod is also very careful to set LUA_PATH and LUA_CPATH internally so that user changes to those env. variables don't affect how Lmod runs.

Can I disable the pager output?

Yes, you can. Just set the environment variable `LMOD_PAGER` to **none**.

Why are messages printed to standard error and not standard out?

The module command is an alias under tcsh and a shell routine under all other shells. There is an lmod command which writes out commands such as export FOO="bar and baz" and messages are written to standard error. The text written to standard out is evaluated so that the text strings make changes to the current environment. See next question for a different way to handle Lmod messages.

Can I force the output of list, avail and spider to go to stdout instead of stderr?

Bash and Zsh user can set the environment variable `LMOD_REDIRECT` to **yes**. Sites can configure Lmod to work this way by default. However, no matter how Lmod is set-up, this will not work with tcsh/csh due to limitations of this shell.

How can I use grep easily with the module command?

If your site doesn't send the output of stdout, you can still use this trick when you need to grep the output of module command. Here are some examples:

```
$ module -t --redirect avail       | grep foo
$ module --raw --redirect show foo | grep bar
$ module -t --redirect spider      | grep baz
```

Can I ignore the spider cache files when doing `module avail`?

Yes you can:

```
$ module --ignore_cache avail
```

or you can set:

```
$ export LMOD_IGNORE_CACHE=1
```

to make Lmod ignore caches as long as the variable is set.

I have created a module and "module avail" can't find it. What do I do?

Assuming that the modulefile is in MODULEPATH then you have an out-of-date cache. Try running:

```
$ module --ignore_cache avail
```

If this does find it then you might have an old personal spider cache. To clear it do:

```
$ rm -rf ~/.cache/lmod
```

If "module avail" doesn't find it now, then the system spider cache is out-of-date. Please ask your system administrator to update the cache. If you are the system administrator then please read *System Spider Cache* and *User Spider Cache*

Why doesn't the module command work in shell scripts?

It will if the following steps are taken. First the script must be a bash script and not a shell script, so start the script with `#!/bin/bash`. The second is that the environment variable BASH_ENV must point to a file which defines the module command. The simplest way is having `BASH_ENV` point to `/opt/apps/lmod/lmod/init/bash` or wherever this file is located on your system. This is done by the standard install. Finally Lmod exports the module command for Bash shell users.

How do I use the initializing shell script that comes with this application with Lmod?

New in Lmod 8.6+, a modulefile can contain **source_sh** ("shell", "shell_script arg1 arg2 …") to source a shell script by automatically converting it into module commands. Sites can use $LMOD_DIR/sh_to_modulefile to convert the script once. See *Shell scripts and Lmod* for details.

Why is the output of `module avail` not filling the width of the terminal?

If the output of `module avail` is 80 characters wide, then Lmod can't find the width of the terminal and instead uses the default size (80). If you do `module --config`, you'll see a line:

> Active lua-term true

If it says **false** instead then lua-term is not installed. One way this happens is to build Lmod on one computer system that has a system lua-term installed and the package on another where lua-term isn't installed on the system.

Why isn't the module defined when using the **screen** program?

The screen program starts a non-login interactive shell. The Bash shell startup doesn't start sourcing /etc/profile and therefore the `/etc/profile.d/*.sh` scripts for non-login interactive shells. You can patch bash and fix `/etc/bashrc` (see *Issues with Bash* for a solution) or you can fix your `~/.bashrc` to source `/etc/profile.d/*.sh`

You may be better off using **tmux** instead. It starts a login shell.

Why does `LD_LIBRARY_PATH` get cleared when using the **screen** program?

The screen program is a guid program. That means it runs as the group of the program and not the group associated with the user. For security reason all of these kinds of program clear `LD_LIBRARY_PATH`. This unsetting of `LD_LIBRARY_PATH` is done by the Unix operating system and not Lmod.

You may be better off using **tmux** instead. It is a regular program.

How can you write TCL files that can be safely used with both Lmod and Tmod?

For example the hide-version command only works Lmod and could be found in ~/.modulerc. This could be read by both Tmod and Lmod. You can prevent Tmod from executing Lmod only code in the following way:

```
#%Module
if { [info exists ::env(LMOD_VERSION_MAJOR)] } {
   hide-version CUDA/8.8.8
}
```

Lmod defines the environment variable LMOD_VERSION_MAJOR during its execution. This trick can also be used in a TCL modulefile to set the family function:

```
#%Module
...
if { [info exists ::env(LMOD_VERSION_MAJOR)] } {
   family compiler
}
```

As of Lmod 8.4.8+ you can also use the TCL global variable ModuleTool:

```
#%Module
...
if ( [ info exists ::ModuleTool ] && $::ModuleTool == "Lmod" } {
   family compiler
}
```

How can I get the shell functions created by modules in bash shell scripts such as job submission scripts?

First, please make sure that shell functions and alias works correctly in bash interactive sub-shells. If they don't then your site is not set up correctly.

Once that works then change the first line of the shell script to be:

#!/bin/bash -l

Note that is a minus ell not minus one. This will cause the startup scripts to be sourced before the first executable statement in the script.

Why do modules sometimes get loaded when I execute `module use <path>`?

A main principal is that when $MODULEPATH changes, Lmod checks all the currently loaded modules. If any of thoses modules would not have been chosen then each is swapped for the new choice.

How to use module commands inside a Makefile?

A user might wish to use module commands inside a Makefile. Here is a generic way that would work with both Tmod and Lmod. Both Lmod and Tmod define MODULESHOME to point to the top of the module install directory and both tools use the same initialization method to define the module command.

Here is an example Makefile that shows a user listing their currently loaded modules:

```
module_list:
	source $$MODULESHOME/init/bash; module list
```

What to do if new modules are missing when doing `module avail`?

If your site adds a new modulefile to the site's $MODULEPATH but are unable to see it with `module avail`?

It is likely that your site is having an spider cache issue. If you see different results from the following commands then that is the problem:

```
$ module --ignore_cache avail
$ module                avail
```

If you see a difference between the above two commands, delete (if it exists) the user's spider cache:

```
$ rm -rf ~/.cache/lmod ~/.lmod.d/__cache__
```

and try again. If that still leads to a difference then there is a out-of-date system spider cache. Please see *System Spider Cache* on how to set up and update a system spider cache. This issue can happen with a user's personal spider cache. Please see *User Spider Cache* for more details.

How to edit a modulefile?

Lmod does not provide a way to directly edit modulefiles. Typically modulefiles are owned by the system so cannot be editted by users. However, Lmod does provide a convenient way to locate modules which could be used for a bash/zsh shell function:

```
function edit_modulefile () {
   $EDITOR $(module --redirect --location $1)
}
```

If my startup shell is bash or tcsh and I start zsh, why do I get a message that is like: "/etc/zsh/zshrc:48: compinit: function definition file not found"

Lmod supports both zsh and ksh. Both these shells use shell variable FPATH but in very different ways. The issue is that some bash or tcsh users run ksh scripts and need access to the module command. In the K-shell, the environment variable FPATH is exported and is the path to where the module shell function is found. Z-shell also uses FPATH to point to tool like compinit and others. By exporting FPATH, Z-shell does not change the value of FPATH which means that the zsh user can not find all the functions that make it work.

The solution is to add "unset FPATH" in your startup files before "exec zsh".

Why doesn't setting CC in compiler and mpi modulefiles always work reliably in a software hierarchy?

When Lmod sees changes to $MODULEPATH it immediately checks to see whether all the currently loaded modules should still be loaded. This means that setting CC with a pushenv() before modifying

$MODULEPATH is unsafe. The correct way to modify $MODULEPATH is in the beginning of the modulefile. So for the gcc modulefile you might have:

```
prepend_path("MODULEPATH","...")
pushenv("CC","gcc")
```

and not the other way around.

1.4.4 Advanced User Guide for Personal Modulefiles

This advanced guide is for users wishing to create modulefiles for their own software. The reasons are simple:

1. Install a newer version of open source software than is currently available.
2. Easily change the version of applications or libraries under their own development.
3. Better documentation for what software is available.

One could create new version of some software and place it in your personal PATH and forget about it. At least when it is in a module, it will be listed in the loaded modules. It will also appear in the list of available software via `module avail`

User Created Modules

Users can create their own modules. The first step is to add to the module path:

```
$ module use /path/to/personal/modulefiles
```

This will prepend `/path/to/personal/modulefiles` to the `MODULEPATH` environment variable. This means that any modulefiles defined here will be used instead of the system modules. There is a possible exception where defaults will override this selection. (See the next section *Finding Modules With Same Name* for more details).

Suppose that the user creates a directory called `$HOME/modulefiles` and they want a personal copy of the "git" package. They do the usual "tar, configure, make, make install" steps:

```
$ wget https://www.kernel.org/pub/software/scm/git/git-2.6.2.tar.gz
$ tar xf git-2.6.2.tar.gz
$ cd git-2.6.2
$ ./configure --prefix=$HOME/pkg/git/2.6.2
$ make
$ make install
```

This document has assumed that 2.6.2 is the current version of git, it will need to be replaced with the current version. To create a modulefile for git one does:

```
$ cd ~/modulefiles
$ mkdir git
$ cd git
$ cat > 2.6.2.lua
local home    = os.getenv("HOME")
local version = myModuleVersion()
local pkgName = myModuleName()
local pkg     = pathJoin(home,"pkg",pkgName,version,"bin")
prepend_path("PATH", pkg)
^D
```

Starting first from the name: git/2.6.2.lua, modulefiles with the .lua extension are assumed to be written in lua and files without this extension are assumed to be written in TCL. This modulefile for git adds `~/pkg/git/2.6.2/bin` to the user's path so that the personal version of git can be found. Note that the use of the functions **myModuleName()** and **myModuleVersion()** allows the module to be generic and not hard-wired to a particular module file. We have used the *cat* command to quickly create this lua modulefile. Obviously, this file can easily created by your favorite editor.

The first line reads the user's HOME directory from the environment. The second line uses the "pathJoin" function provided from Lmod. It joins strings together with the appropriate number of "/". The last line calls the "prepend_path" function to add the path to git to the user's path.

Finally the user can do:

```
$ module use $HOME/modulefiles
$ module load git
$ type git
~/pkg/git/2.6.2/bin/git
```

For git to be available on future logins, users need to add the following to their startup scripts or a saved collection:

```
$ module use $HOME/modulefiles
$ module load git
```

The modulefiles can be stored in different directories. There is an environment variable `MODULEPATH` which controls that. Modulefiles that are listed in an earlier directory are found before ones in later directories. This is similar to command searching in the `PATH` variable. There can be several versions of a command. The first one found in the `PATH` is used.

Finding Modules With Same Name

Suppose the user has created a "git" module to provide the latest available. At a later date, the system administrators add a newer version of "git"

```
$ module avail git
--------------- /home/user/modulefiles ----------------
git/2.6.2

--------------- /opt/apps/modulefiles ----------------
git/1.7.4   git/2.0.1   git/3.5.4 (D)

$ module load git
```

The load command will load `git/3.5.4` because it is the highest version.

If a user wishes to make their own version of git the default module, they will have to mark it as a default. Marking a module as a default is discussed in section *Marking a Version as Default*.

1.4.5 New Features in Lmod

hook.register(<hook_name>, func, <action>)
(Lmod 8.7.25) The hook.register function now takes a optional third argument: action. The legal actions are the followning strings: "replace", "append", "prepend". See *Registering Multiple Hook functions* for more details.

/etc/lmod/.modulespath
(Lmod 8.7.24) Sites can use /etc/lmod/.modulespath or set $LMOD_MODULEPATH_INIT during their site's startup scripts to control where Lmod finds the .modulespath file.

check_module_tree_syntax
(Lmod 8.4.3+) check_module_tree_syntax is a command to find syntax errors in your modulefiles. It also reports when there are modulefile directories that have multiple ways of marking a default. See *Checking Syntax in Your Sites Modulefiles* for more details.

module category
(Lmod 8.7.14+) Community provided feature from PR #600. This command has two levels. 1) "module category" list all the categories in the entire module tree; 2) "module category <name1> <name2> ..." list the all the modules that have list that category. Note that this listing is case insensitive. So "Library" and "library" will both be printed out when running **module category library**

Collection are written to both ~/.config/lmod and ~/.lmod.d
(Lmod 8.7.13+) Lmod is transitioning away from using the ~/.lmod.d directory. During transition, collection files are written to both directories. Sites can use the –with-useDotConfigDirOnly=yes or set the env. var. **LMOD_USE_DOT_CONFIG_ONLY** to "yes" to complete the transition. Lmod 9+ will force the default to be yes. This means that Lmod will only write to ~/.config/lmod but always read from both directories.

User cache files are now written to ~/.cache/lmod:
(Lmod 8.7.12+) The old user cache directory was ~/.lmod.d/.cache and it has now changed to ~/.cache/lmod/*. Lmod is transitioning away from using the ~/.lmod.d directory. Also the –with-useDotFiles configuration option has been remove as it is nolonger needed.

Controlling installed permission:
(Lmod 8.7.11+): Lmod now uses your umask to set permissions unless your user id (id -u) is less than UID_MIN (from /etc/login.defs) or UID_MIN=500 by default. In that case, it uses a umask of 022. This translates to 755 for executable files and 644 for all others. Sites can override this at configure time with **–with-mode=MODE**.

LMOD_SYSTEM_DEFAULT_MODULES:
(Lmod 8.7.7+) If a site has no system default modules they can set this variable to **__NO_SYSTEM_DEFAULT_MODULES__**. This will allow **module reset** to purge all modules and reset $MODULEPATH to the system default.

Dynamic Spider Cache Support:
(Lmod 8.7.4+): If a modulefile changes $MODULEPATH, it is marked as dynamic and is re-read when performing a *module spider*. This is to allow sites to dynamically add in modulefiles in user directories. This can be turned off at configure time or with by setting LMOD_DYNAMIC_SPIDER_CACHE=no.

Warning when reading too many non-modulefile:
Lmod (8.7.4+): To catch directory that are full of non-modulefiles, Lmod count the number of regular files that do not start with a ".". If there are more than 100, Lmod reports a warning.

Loading a modulefile too many times:
(Lmod 8.7.3+): Lmod throws an error if any modulefile is loaded 500 time or more in a single module command. This is to prevent infinite load loops.

An unload cannot fail:
As of Lmod 8.7+, an error found during unload is treated as a warning.

puts stdout:
This TCL modulefile will generate its output at the end of the modulefile evaluation and not the beginning. New

in Lmod 8.7. This matches Tmod 4.1

puts prestdout:
This TCL modulefile will generate its output at the beginning of the modulefile evaluation. New in Lmod 8.7. This matches Tmod 5.1.

LmodBreak:
LmodBreak() modulefile function causes the evaluation of the current modulefile to stop and all changed in the user's environment to be ignored from the current modulefile. However, all other modulefiles are evaluated. In TCL modulefiles, this command is **break**. Break works normally inside a loop. If a bare break is found outside a loop, it cause the current modulefile stop its processing. New in Lmod 8.7+

Note As of Lmod 8.6.16: LmodBreak()/break does nothing when unloading.

/etc/lmod/lmod_config.lua:
Lmod looks for a file named lmod_config.lua in the LMOD_CONFIG_DIR which is by default /etc/lmod/lmod_config.lua. This file allows sites configure lmod through lua instead of setting environment variables by using the cosmit:assigin() functions:

```
cosmit:assign("LMOD_SITE_NAME","XYZZY")
```

See *Configuring Lmod with /etc/lmod/lmod_config.lua:* for details. New in Lmod 8.6+

LMOD_QUARANTINE_VARS:

This is an environment variable containing a list of environment variables that Lmod will not change the value of. Note that this only applies to non-path variables. Variables such as PATH or LD_LIBRARY_PATH are ignored in this variable. In other words, Lmod could change any path like variables. New in Lmod 8.6+

source_sh ("shellName","shell_script arg1 ..."):
source a shell script as part of a module. Supported shellNames are *sh*, *dash*, *bash*, *zsh*, *csh*, *tcsh*, *ksh*. When loading, Lmod automatically converts the shell script into module commands and saves the module commands in the environment. It does this by sourcing the shell script string in a subshell and comparing the environment before and after sourcing the shell script string. When unloading, the saved module commands from the environment are used.

See *Shell scripts and Lmod* for details (New in version 8.6)

sh_to_modulefile:
New in version 8.6, it now track converts alias and shell functions as well as environment variables into a modulefile.

LMOD_SITE_MODULEPATH:
An colon separated list of directories to be prepended to $MODULEPATH before the first call to the Lmod command. See *Installing Lmod* for details New in version 8.5.18

module overview:

Similar to **module avail**, this command prints the just module name and not the name and the version. The number of versions is printed next to the module name: For example:

```
$ module overview

------------ /apps/modulefiles/Core -------------
StdEnv    (1)   hashrf    (2)   papi        (2)
ddt       (1)   intel     (2)   singularity (2)
git       (1)   noweb     (1)   valgrind    (1)
```

New in version 8.5.10+

sh_to_modulefile:
New in version 8.6, it now track converts alias and shell functions as well as environment variables into a modulefile.

extension():
New in version 8.2.5+, Lmod provides a module function which says that these packages available when this module is loaded. See *Module Extensions* for details.

depends_on():
A safe way to have one module load another. See *Dependent Modules* for details. New in version 7.5.12+

1.5 Installing Lmod

Anyone wishing to install Lmod on a personal computer or for a system should read the Installation Guide as well as the Transitioning to Lmod Guide. The rest of the guides can be read as needed.

1.5.1 Installing Lua and Lmod

Environment modules simplify customizing the users' shell environment and it can be done dynamically. Users load modules as they see fit. It is completely under their control. Environment Modules or simply modules provide a simple contract or interface between the system administrators and users. System administrators provide modules and users get to choose which to load.

There have been environment module systems for quite a while. See http://modules.sourceforge.net/ for a TCL based module system and see http://www.lysator.liu.se/cmod for another module system. Here we describe Lmod, which is a completely new module system written in Lua. For those who have used modules before, Lmod automatically reads TCL modulefiles. Lmod has some important features over other module system, namely a built-in solution to hierarchical modulefiles and provides additional safety features to users as described in the User Guide.

The hierarchical modulefiles are used to solve the issue of system pre-built libraries. User applications using these libraries must be built with the same compiler as the libraries. If a site provides more than one compiler, then for each compiler version there will be separate versions of the libraries. Lmod provides built-in control making sure that compilers and pre-built libraries stay matched. The rest of the pages here describe how to install Lmod, how to provide the module command to users during the login process and some discussion on how to install optional software and the associated modules.

The goal of installing Lmod is when completed, any user will have the module command defined and a preset list of modules will be loaded. The module command should work without modifying the users startup files (`~/.bashrc`, `~/.profile`, `~/.cshrc`, or `~/.zshenv`). The module command should be available for login shells, interactive shells, and non-interactive shells. The command `ssh YOUR_HOST module list` should work. This will require some understanding of the system startup procedure for various shells which is covered here.

Installing Lua

In this document, it is assumed that all optional software is going to be installed in /opt/apps. The installation of Lmod requires installing lua as well. On some system, it is possible to install Lmod directly with your package manager. It is available with recent fedora, debian and ubuntu distributions.

Install lua-X.Y.Z.tar.gz

One choice is to install the lua-X.Y.Z.tar.gz file. This tar ball contains lua and the required libraries. This can be downloaded from https://sourceforge.net/projects/lmod/files/:

```
$ wget https://sourceforge.net/projects/lmod/files/lua-5.1.4.9.tar.bz2
```

The current version is 5.1.4.9 but it may change in the future. If the above wget doesn't work then please go to sourceforce.net and down from the web interface. The lua package can be installed using the following commands:

```
$ tar xf lua-X.Y.Z.tar.bz2
$ cd lua-X.Y.Z
$ ./configure --prefix=/opt/apps/lua/X.Y.Z
$ make; make install
$ cd /opt/apps/lua; ln -s X.Y.Z lua
$ mkdir /usr/local/bin; ln -s /opt/apps/lua/lua/bin/lua /usr/local/bin
```

The last command is optional, but you will have to somehow put the `lua` command in your path. Also obviously, please replace X.Y.Z with the actual version (say 5.1.4.9)

If you use this option you do **not** need to use your package manager or install luarocks. Instead please read the section on how to install Lmod.

Using Your Package Manager for Redhat/Centos

If you didn't install the lua tar ball described above then you can use your package manager for your OS to install Lua. You will also need the luaposix package

Centos may require looking the EPEL repo. At TACC we install the following rpms for our Centos based systems:

```
lua-5.1.4-15.el7.x86_64
lua-bitop-1.0.2-3.el7.x86_64
lua-devel-5.1.4-15.el7.x86_64
lua-json-1.3.2-2.el7.noarch
lua-lpeg-0.12-1.el7.x86_64
lua-posix-32-2.el7.x86_64
lua-term-0.03-3.el7.x86_64
```

You will also need the tcl and tcl-devel packages as well.:

```
tcl-8.5.13-8.el7.x86_64
tcl-devel-8.5.13-8.el7.x86_64
```

Please note that the devel packages such as tcl-devel and lua-devel are only required to build Lmod. They are not required to run the lmod package. Note as well that the tcl-devel for Centos or tcl-dev for ubuntu is only required if you configure Lmod using –with-fastTCLInterp=yes.

Using Your Package Manager for Ubuntu or Debian systems

For modern debian and debian derivatives, you can get the necessary dependencies by executing the following commands. Note that for this to work you may need to repair your /etc/apt/sources.list. Check to see if your sources.list file as uncommmented "deb-src" lines. Once this is correct, please execute the following lines:

```
$ sudo apt update
$ sudo apt -y build-dep lmod
$ lua_ver=$(which lua | xargs realpath -e | xargs basename)
$ sudo apt -y install lib${lua_ver}-dev tcl-dev
```

Otherwise please install the following packages:

```
lua5.3
lua-bit32:amd64
lua-posix:amd64
lua-posix-dev
liblua5.3-0:amd64
liblua5.3-dev:amd64
tcl
tcl-dev
tcl8.6
tcl8.6-dev:amd64
libtcl8.6:amd64
```

For Ubuntu 18.04, you will need to make lua 5.3 the default using **update-alternatives** and fix a lua posix symlink:

```
sudo update-alternatives --install /usr/bin/lua \
    lua-interpreter /usr/bin/lua5.3 130 \
    --slave /usr/share/man/man1/lua.1.gz lua-manual \
    /usr/share/man/man1/lua5.3.1.gz
sudo update-alternatives --install /usr/bin/luac \
    lua-compiler /usr/bin/luac5.3 130 \
    --slave /usr/share/man/man1/luac.1.gz lua-compiler-manual \
    /usr/share/man/man1/luac5.3.1.gz
sudo ln -s /usr/lib/x86_64-linux-gnu/liblua5.3-posix.so \
    /usr/lib/x86_64-linux-gnu/lua/5.3/posix.so
```

Using Luarocks

If you have installed lua but still need luaposix, you can install the `luarocks` program from your package manager or directly from https://luarocks.org/. The `luarocks` programs can install many lua packages including the ones required for Lmod.

```
$ luarocks install luaposix
```

Now you have to make the lua packages installed by luarocks to be known by lua. On our Centos system, Lua knowns about the following for *.lua files:

```
$ lua -e 'print(package.path)'
./?.lua;/usr/share/lua/5.1/?.lua;/usr/share/lua/5.1/?/init.lua;/usr/lib64/lua/5.1/?.lua;/
↪usr/lib64/lua/5.1/?/init.lua;
```

and the following for shared libraries:

```
$ lua -e 'print(package.cpath)'
./?.so;/usr/lib64/lua/5.1/?.so;/usr/lib64/lua/5.1/loadall.so;
```

Assuming that luarocks has installed things in its default location (/usr/local/. . .) then you'll need to do:

```
LUAROCKS_PREFIX=/usr/local
export LUA_PATH="$LUAROCKS_PREFIX/share/lua/5.1/?.lua;$LUAROCKS_PREFIX/share/lua/5.1/?/
→init.lua;;"
export LUA_CPATH="$LUAROCKS_PREFIX/lib/lua/5.1/?.so;;"
```

Please change LUAROCKS_PREFIX to match your site. The exporting of LUA_PATH and LUA_CPATH must be done before any module commands. It is very important that the double trailing semicolon are there. They are replaced by the built-in system path for Lua.

Using Ansible

There is a *ready-to-use Ansible role <https://galaxy.ansible.com/idiv-biodiversity/lmod/>* that allows you to install Lmod conveniently from Ansible. The role was written with installation on HPC clusters in mind, i.e. it is possible to install Lmod into a global, networked file system share on only a single host, while all other hosts install just the Lmod dependencies and the shell configuration files. Nevertheless, it is of course possible to install Lmod with this role on a single server. Also, the role supports the transition to Lmod as described in *How to Transition to Lmod (or how to test Lmod without installing it for all)*.

You can find the complete role documentation *here <https://github.com/idiv-biodiversity/ansible-role-lmod#ansible-role-lmod>*.

Why does Lmod install differently?

Lmod automatically creates a version directory for itself. So, for example, if the installation prefix is set to `/apps`, and the current version is `X.Y.Z`, installation will create `/apps/lmod` and `/apps/lmod/X.Y.Z`. This way of configuring is different from most packages. There are two reasons for this:

1. Lmod is designed to have just one version of it running at one time. Users will not be switching version during the course of their interaction in a shell.
2. By making the symbolic link the startup scripts in /etc/profile.d do not have to change. They just refer to `/apps/lmod/lmod/...` and not `/apps/lmod/X.Y.Z/...`

Installing Lmod

Lmod has a large number of configuration options. They are discussed in the Configuring Lmod Guide. This section here will describe how to get Lmod installed quickly by using the defaults:

Note: If you have a large number of modulefiles or a slow parallel filesystem please read the Configure Lmod Guide on how to set-up the spider caching system. This will greatly speed up `module avail` and `module spider`

To install Lmod, you'll want to carefully read the following. If you want Lmod version X.Y installed in `/opt/apps/lmod/X.Y`, just do:

```
$ ./configure --prefix=/opt/apps
$ make install
```

The installation will also create a link to `/apps/lmod/lmod`. The symbolic link is created to ease upgrades to Lmod itself, as numbered versions can be installed side-by-side, testing can be done on the new version, and when all is ready, only the symbolic link needs changing.

To create such a testing installation, you can use:

```
$ make pre-install
```

which does everything but create the symbolic link.

In the `init` directory of the source code, there are `profile.in` and `cshrc.in` templates. During the installation phase, the path to lua is added and `profile` and `cshrc` are written to the `/apps/lmod/lmod/init` directory. These files are created assuming that your modulefiles are going to be located in `/apps/modulefiles/$LMOD_sys` and `/apps/modulefiles/Core`, where `$LMOD_sys` is what the command "`uname`" reports, (e.g., Linux, Darwin). The layout of modulefiles is discussed later.

Note: Obviously you will want to modify the profile.in and cshrc.in files to suit your system.

Sites that want to use the .modulespath file have 3 choices on how to specify where the .modulespath file is located in order of priority:

1. Set the LMOD_MODULEPATH_INIT environmant variable to point to a file.
2. Use /etc/lmod/.modulespath
3. */apps/lmod/lmod/init/.modulespath* ` or configure with *–with-ModulePathInit=…* ` to point to any file.

The format of this file is:

```
# comments are allowed as well as wildcards
/apps/modulefiles/\*
/apps/other_modulefiles
```

If this file exists then MODULEPATH_ROOT method is not used.

Another way for a site to add their own directories $MODULEPATH is to define it before z00_lmod.* is sourced. Care is required so that $MODULEPATH is changed on the login shell but not subsequent sub-shells.

Also sites can set the environment variable $LMOD_SITE_MODULEPATH with a colon separate list of directories which will be prepended to $MODULEPATH. This variable is used in /etc/profile.d/z00_lmod.* So it must be defined before the z00_lmod.* file is source. ($LMOD_SITE_MODULEPATH is new in Lmod 8.5.18)

Note: It is important to define $MODULEPATH before z00_lmod.* is run by either using `.modulepath` or setting $LMOD_SITE_MODULEPATH or $MODULEPATH. Do not use **module use …** statements in later /etc/profile.d/* files. This is because **module reset** returns $MODULEPATH to the value defined when lmod is first executed, which will be when z00_lmod.* is run.

The `profile` file is the Lmod initialization script for the bash and zsh shells, the `cshrc` file is for tcsh and csh shells, and the `profile.fish` file is for the fish shell, etc. Please copy or link the `profile` and `cshrc` files to `/etc/profile.d`, and optionally the fish file to `/etc/fish/conf.d`:

```
$ ln -s /opt/apps/lmod/lmod/init/profile        /etc/profile.d/z00_lmod.sh
$ ln -s /opt/apps/lmod/lmod/init/cshrc          /etc/profile.d/z00_lmod.csh
$ ln -s /opt/apps/lmod/lmod/init/profile.fish   /etc/fish/conf.d/z00_lmod.fish
```

To test the setup, you just need to login as a user. Or if you are already logged in, please logout and log back in so that the startup files in `/etc/profile.d/*.sh` will be sourced. The module command should be set and `MODULEPATH` should be defined. Bash or Zsh users should see something like:

```
$ type module
module ()
{
  eval $($LMOD_CMD bash $*)
}

$ echo $LMOD_CMD
/opt/apps/lmod/lmod/libexec/lmod

$ echo $MODULEPATH
/opt/apps/modulefiles/Linux:/opt/apps/modulefiles/Core
```

Similar for csh users:

```
% which module
module: alias to eval `/opt/apps/lmod/lmod/libexec/lmod tcsh !*`

% echo $MODULEPATH
/opt/apps/modulefiles/Linux:/opt/apps/modulefiles/Core
```

If you do not see the module alias then please read the next section.

Integrating module Into Users' Shells

Bash:

On login, the bash shell first reads `/etc/profile`, and if `profiles.d` is activated, that in turn should source all the *.sh files in `/etc/profile.d` with something like:

```
if [ -d /etc/profile.d ]; then
  for i in /etc/profile.d/*.sh; do
    if [ -r $i ]; then
      . $i
    fi
  done
fi
```

Similarly, the system BASHRC file should source all the *.sh files in `/etc/profile.d` as well.

Bash under Ubuntu:

Sites that run Ubuntu and have bash users should consider adding the following to their /etc/bash.bashrc:

```
if ! shopt -q login_shell; then
  if [ -d /etc/profile.d ]; then
    for i in /etc/profile.d/*.sh; do
      if [ -r $i ]; then
        . $i
      fi
    done
  fi
fi
```

This is useful because non-login interactive shells only source /etc/bash.bashrc and this file doesn't normally source the files in /etc/profile.d/*.sh.

Bash Shell Scripts:

Bash shell scripts do not source any system or user files before starting execution. Instead it looks for the environment variable BASH_ENV. It treats the contents as a filename and sources it before starting a bash script.

Bash Script Note:

It is important to remember that all bash scripts should start with:

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

Starting with:

```
#!/bin/sh
```

won't define the module command, even if sh is linked to bash. Bash will run those scripts in shell emulation mode and won't source the file that BASH_ENV points to.

Csh:

Csh users have an easier time with the module command setup. The system cshrc file is always sourced on every invocation of the shell. The system cshrc file is typically called: `/etc/csh.cshrc`. This file should source all the *.csh files in `/etc/profile.d`:

```
if ( -d /etc/profile.d ) then
  set nonomatch
  foreach i (/etc/profile.d/*.csh)
    source $i
  end
  unset nonomatch
endif
```

Zsh:

Zsh users have an easy time with the module command setup as well. The system zshenv file is sourced on all shell invocations. This system file can be in a number of places but is typically in `/etc/zshenv` or `/etc/zsh/zshenv` and should have:

```
if [ -d /etc/profile.d ]; then
  setopt no_nomatch
  for i in /etc/profile.d/*.sh; do
    if [ -r $i ]; then
      . $i
    fi
  done
  setopt nomatch
fi
```

Ksh:

Ksh users as of Lmod version 8.3.12+ now have full support as long as the site installs Lmod with *–supportKsh=yes*. Lmod now defines FPATH to be the directory for the shell function commands such as module and ml that provide the module commands.

Note: Zsh users who wish to run ksh scripts that have module commands in them will have to export the FPATH variable as FPATH is normally a local variable and not exported in zsh.

Rc:

Rc shells ignore the */etc/profile.d* directory and source */lib/rcmain* on startup. Global initialization should be placed in that file. Login shells (started with *rc -l*) additionally read the user's *$HOME/.rcrc* file.

Running the following line on startup will set up the module commands for rc users:

` mod=/opt/apps/lmod/lmod/init/profile.rc if(test -r $mod) . $mod `

Fish:

Fish users have several standard places searched for scripts. The system location is usually `/etc/fish/conf.d` and the user location is usually `~/.config/fish/conf.d/`. Fish users are provided a special profile file, `init/profile.fish`, that should be linked into one of these places with a suitable name. For example, a local user for fish might want:

```
$ ln -s /opt/apps/lmod/lmod/init/profile.fish ~/.config/fish/conf.d/z00_lmod.fish
```

A site might set:

```
$ ln -s /opt/apps/lmod/lmod/init/profile.fish /etc/fish/conf.d/z00_lmod.fish
```

Issues with Bash

Interactive Non-login shells

The Bash startup procedure for interactive non-login shells is complicated and varies between Operating Systems. In particular, Redhat & Centos distributions of Linux as well as Mac OS X have no system bashrc read during startup whereas Debian based distributions do source a system bashrc. One easy way to tell how bash is set up is to execute the following:

```
$ strings `type -p bash` | grep bashrc
```

If the entire result of the command is:

```
~/.bashrc
```

then you know that your bash shell doesn't source a system bashrc file.

If you want to have the same behavior between both interactive shells (login or non-login) and your system doesn't source a system bashrc, then you have two choices:

1. Patch bash so that it does source a system bashrc. See `contrib/bash_patch` for details on how to do that.
2. Expect all of your bash users to have the following in their `~/.bashrc`

   ```
   if [ -f /etc/bashrc ]; then
      . /etc/bashrc
   fi
   ```

As a side note, we at TACC patched bash for a different reason which may apply to your site. When an MPI job starts, it logs into each node with an interactive non-login shell. When we had no system bashrc file, many of our fortran 90 programs failed because they required `ulimit -s unlimited` which makes the stack size unlimited. *By patching bash, we could guarantee that it was set by the system on each node.* Sites will have to chose which of the two above methods they wish to deal with this deficiency in bash.

You may have to also change the /etc/bashrc (or /etc/bash.bashrc) file so that it sources /etc/profile.d/*.sh for non-login shells.

Bash Shell Scripts

Bash shell scripts, unlike Csh or Zsh scripts, do not source any system or user files. Instead, if the environment variable, `BASH_ENV` is set and points to a file then this file is sourced before the start of bash script. So by default Lmod sets `BASH_ENV` to point to the bash script which defines the module command.

It may seem counter-intuitive but Csh and Zsh users running bash shell scripts will want BASH_ENV set so that the module command will work in their bash scripts.

A bash script is one that starts as the very first line:

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

A script that has nothing special or starts with:

```
#!/bin/sh
```

is a shell script. And even if `/bin/sh` points to `/bin/bash` bash runs in a compatibility mode and doesn't honor `BASH_ENV`.

To combat this Lmod exports the definition of the module command. This means that even /bin/sh scripts will have the module command defined when run by a Bash User. However, a Csh or Zsh user running a bash script will still need to set `BASH_ENV` and run bash scripts. They won't have the module command defined if they run a sh script.

1.5.2 How to Transition to Lmod (or how to test Lmod without installing it for all)

In the *Installing Lua and Lmod* document, we described how to install Lua and Lmod for all. Sites which are currently running another environment module system will likely wish to test and then transition from their old module system to Lmod. This can be done smoothly by switching over all users on a set date.

It is important to remember the following facts:

- Lmod reads modulefiles written in TCL. There is typically no need to translate modulefiles written in TCL into Lua. Lmod does this for you automatically.
- Some users can run Lmod while others use the old environment module system.
- However, no user can run both at the same time in the same shell.

Obviously, since you are installing Lmod in your own account, this is a good way to test Lmod without committing your site to switch. Part of this document will describe TACC's transition experience.

Steps for Testing Lmod in your account

1. Install Lua
2. Install Lmod in your account
3. Build the list modules required
4. Purge modules using old module command
5. Reload modules using Lmod

Install Lua

The previous document described how to install Lua. If your system doesn't provide a package for Lua, then it is probably easiest to install the lua tarball found at sourceforge.net using the following command:

```
$ wget https://sourceforge.net/projects/lmod/files/lua-W.X.Y.Z.tar.gz
```

where you replace the *W.X.Y.Z* with the current version (i.e. 5.1.4.8).

Many Linux distributions already have a lua package and it may even be installed automatically. For example, recent Centos and other Redhat based distributions automatically install Lua as part of the rpm tools.

Once you have lua installed and in your path, you'll need the luafilesystem and luaposix libraries to complete the requirements. See the previous document on how to install these libraries via your package manager or luarocks.

Install Lmod

Please follow the previous document on how to install Lmod. Let's assume that you have installed Lmod in your own account like this:

```
$ ./configure --prefix=$HOME/pkg
$ make install
```

This will install Lmod in **$HOME/pkg/lmod/x.y.z** and make a symbolic link to **$HOME/pkg/lmod/lmod**.

Build the list of modules required

Many sites provide a default set of modules. When testing, you'll want to be able to load those list of modules using Lmod. Using your old module system, login and do:

```
$ module list

Currently Loaded Modules:

1) a1      3) A     5) b2     7) B
2) a2      4) b1    6) b3
```

It turns out that both the latest version of TCL/C modules and the pure TCL script list a module that loads other modules later in the list. In this made up case we unload module B and notice that unloading the B module also unloads modules b3, b2 and b1. Then unloading the A module also unloads modules a2 and a1. In this case then we would set:

```
export LMOD_SYSTEM_DEFAULT_MODULES=A:B
```

Purge modules using old module command

Execute:

```
$ module purge
```

to unload the currently loaded modules **using the old module command**.

Reload modules using Lmod

Once all modules have been purged and the environment variable LMOD_SYSTEM_DEFAULT_MODULES has been set, all that is required is to redefine the module command to use Lmod and to restore the default set of modules by:

```
$ export BASH_ENV=$HOME/pkg/lmod/lmod/init/bash
$ source $BASH_ENV
```

This will define the module command. Finally the default set of modules can be loaded:

```
$ module --initial_load restore
```

This command first looks to see if there is a default collection in **~/.lmod.d/default**. If that file isn't found then it uses the value of variable LMOD_SYSTEM_DEFAULT_MODULES as a list of module to load.

If you have gotten this far then you have installed Lmod in your account. Congratulations!

Please test your system. Try to load your most complicated modulefiles. See if **module avail**, **module spider** works and so on.

If you have trouble loading certain TCL modulefiles then read the **How Lmod reads TCL modulefiles** to see why you might have problems.

An example of how this can be done in your bash startup scripts

All the comments above can be combined into a complete example:

```
if [ -z "$_INIT_LMOD" ]; then
   type module > /dev/null 2>&1
   if [ "$?" -eq 0 ]; then
     clearLmod --quiet           # Purge all modules and completely remove old Lmod setup
   fi
   export _INIT_LMOD=1           # guard variable is crucial, to avoid breaking existing␣
↪modules settings

   export MODULEPATH=...                          # define  MODULEPATH
   export BASH_ENV=$HOME/pkg/lmod/lmod/init/bash # Point to the new definition of Lmod

   source $BASH_ENV                               # Redefine the module command to point
                                                  # to the new Lmod
   export LMOD_SYSTEM_DEFAULT_MODULES=...         # Colon separated list of modules
                                                  # to load at startup
   module --initial_load --no_redirect restore   # load either modules listed above or␣
↪the
                                                  # user's ~/.lmod.d/default module␣
↪collection
else
   source $BASH_ENV                               # redefine the module command for sub-
↪shell
   module refresh                                 # reload all modules but only activate␣
↪the "set_alias"
                                                  # functions.
fi
```

Obviously, you will have to define **MODULEPATH** and **LMOD_SYSTEM_DEFAULT_MODULES** to match your site setup. The reason for the guard variable **_INIT_LMOD** is that the module command and the initialization of the modules is only done in the initial login shell. On any sub-shells, the module command gets defined (again). Finally the **module refresh** command is called to define any alias or shell functions in any of the currently loaded modules.

How to Transition to Lmod: Staff & Power User Testing

Once you have tested Lmod personally and wish to transition your site to use Lmod, I recommend the following strategy for staff and friendly/power users for testing:

1. Install Lua and Lmod in system locations
2. Install */etc/profile.d/z00_lmod.sh* to redefine the module command
3. Load system default modules (if any) after previous steps
4. Only users who have a file named *~/.lmod* use Lmod
5. At TACC, we did this for 6 months.

Using this strategy, you can have extended testing without exposing Lmod to any user which hasn't opted-in.

How to Deploy Lmod

Once Staff testing is complete and you are ready to deploy Lmod to your users it is quite easy to switch to an opt-out strategy:

1. Change */etc/profile.d/z00_lmod.sh* so that everyone is using Lmod
2. If a user has a ~/.no.lmod then they continue to use your original module system
3. At TACC we did this for another 6 months
4. We broke Environment Module support with the family directive
5. We now only support Lmod
6. Both transitions generated very few tickets (2+2)

1.5.3 Lua Modulefile Functions

Lua is an efficient language built on simple syntax. Readers wanting to know more about lua can see http://www.lua.org/. This simple description given here should be sufficient to write all but the most complex modulefiles.

It is important to understand that modulefiles are written in the positive. That is, one writes the actions necessary to activate the package. A modulefile contains commands to add to the PATH or set environment variables. When loading a modulefile the commands are followed. When unloading a modulefile the actions are reversed. That is the element that was added to the PATH during loading, is removed during unloading. The environment variables set during loading are unset during unloading.

prepend_path ("*PATH*", "*/path/to/pkg/bin*"):
prepend to a path-like variable the value.

prepend_path ("*PATH*", "*/path/to/pkg/bin*", "*delim*"):
prepend to a path-like variable the value. It is possible to add a third argument to be the delimiter. By default is is ":", the delimiter can be any single character for example " " or ";"

prepend_path {"*PATH*", "*/path/to/pkg/bin*", delim="*delim*", priority=*num*}:
prepend to a path-like variable the value. One can use this form **with braces {} instead of parens** () to specify both a priority a non-default delimiter.

append_path ("*PATH*", "*/path/to/pkg/bin*"):
append to a path-like variable the value.

append_path ("*PATH*", "*/path/to/pkg/bin*", "*delim*"):
append to a path-like variable the value. It is possible to add a third argument to be the delimiter. By default is is ":", the delimiter can be any single character for example " " or ";"

append_path {"*PATH*", "*/path/to/pkg/bin*", delim="*delim*", priority=*num*}:
append to a path-like variable the value. One can use this form **with braces {} instead of parens** () to specify both a priority a non-default delimiter.

remove_path ("*PATH*", "*/path/to/pkg/bin*"):
remove value from a path-like variable for both load and unload modes.

remove_path ("*PATH*", "*/path/to/pkg/bin*" , "*delim*"):
remove value from a path-like variable for both load and unload modes. It is possible to add a third argument to be the delimiter. By default is is ":", the delimiter can be any single character for example " " or ";"

setenv ("NAME", "*value*"):
assigns to the environment variable "NAME" the value. Do not use this function to assign the initial to a path-like variable. Use **append_path** or **prepend_path** instead.

pushenv ("NAME", "*value*"):
sets **NAME** to *value* just like **setenv**. In addition it saves the previous value in a hidden environment variable. This way the previous state can be returned when a module is unloaded. **pushenv** ("FOO",false) will clear "FOO" and the pop will return the previous value.

add_property ("NAME", "*value*"):
See *Module Properties* for how to use this function.

remove_property ("NAME", "*value*"):
See *Module Properties* for how to use this function.

unsetenv ("NAME"):
unset the value associated with "NAME". This command is a no-op when the mode is unload.

whatis ("STRING"):
The whatis command can be called repeatedly with different strings. See the Administrator Guide for more details.

help ([[*help string*]]):
What is printed out when the help command is called. Note that the *help string* can be multi-lined.

pathJoin ("/a", "b/c/", "d/"):
builds a path: "/a/b/c/d", It combines any number of strings with one slash and removes excess slashes. Note that trailing slash is removed. If you need a trailing slash then do **pathJoin("/a", "b/c") .. "/"** to get "/a/b/c/".

depends_on ("pkgA", "pkgB", "pkgC"):
Loads all modules. When unloading only dependent modules are unloaded. See *Dependent Modules* for details.

load ("pkgA", "pkgB", "pkgC"):
load all modules. Report error if unable to load.

load_any ("pkgA", "pkgB", "pkgC"):
loads the first module found. Report error if unable to load any of the modules. When unloading all modules are marked to be unloaded.

try_load ("pkgA", "pkgB", "pkgC"):
load all modules. No errors reported if unable to load. Any other errors will be reported.

complete ("shellName","name","args"):
Bash and tcsh support the complete function. Note that the shellName must match the name of the shell given on the Lmod command. There is no error if the shell names do not match. The command is ignored. See rt/complete/mf/spack/1.0.lua for an example.

source_sh ("shellName","shell_script arg1 ...")
source a shell script as part of a module. Supported shellNames are *sh*, *dash*, *bash*, *zsh*, *csh*, *tcsh*, *ksh*. When loading, Lmod automatically converts the shell script into module commands and saves the module commands in the environment. It does this by sourcing the shell script string in a subshell and comparing the environment before and after sourcing the shell script string. When unloading, the saved module commands from the environment are used.

Note that shell script string must not change between loading and unloading as the full string is used to reference the saved module commands.

Other shells could be supported with help from the community that uses that shell. (New in version 8.6)

This feature was introduced in Tmod 4.6 and was shamelessly studied and re-implemented in Lmod 8.6+.

LmodBreak (msg):
LmodBreak() modulefile function causes the evaluation of the current modulefile to stop and all changed in the user's environment to be ignored from the current modulefile. However, all other modulefiles are evaluated. In TCL modulefiles it is **break**.

In other words, this function does not stop, where as **LmodError()** stops all evaluations. New in Lmod 8.6+

Note As of Lmod 8.6.16: LmodBreak() does nothing when unloading.

userInGroups ("group1", "group2", . . .):
Returns true if user is root or a member of one of the groups listed.

mgrload (required, active_object):
load a single module file. If required is true then error out if not found. If false then no message is generated. Returns true if successful. See *Using mgrload function* for details.

always_load ("pkgA", "pkgB", "pkgC"):
load all modules. However, when this command is reversed, it does nothing.

set_alias ("name", "value"):
define an alias to name with value.

unload ("pkgA", "pkgB"):
In both load and unload mode, the modulefiles are unloaded. It is not an error to unload modules that where not loaded.

family ("name"):
A user can only have one family "name" loaded at a time. For example family("compiler") would mean that a user could only have one compiler loaded at a time.

prereq ("name1", "name2"):
The current modulefile will only load if **all** the listed modules are already loaded.

prereq_any ("name1", "name2"):
The current modulefile will only load if **any** of the listed modules are already loaded.

conflict ("name1", "name2"):
The current modulefile will only load if all listed modules are NOT loaded.

extensions ("numpy/2.1, scipy/3.2, foo/1.3"):
This module provides the following extensions. Place the list of extensions as a single string.

requireFullName ():
This function throws an error if module name specified by the user is not the fullName. Typically used as:

```
if (mode() == "load") then requireFullName() end
```

haveDynamicMPATH ():
This function tells that Lmod that this module has a dynamic $MODULEPATH when building the spider cache. See *The spider tool* for details.

Extra functions

The entries below describe several useful commands that come with Lmod that can be used in modulefiles.

os.getenv ("NAME"):
Get the value for the environment variable called "NAME". Note that if "NAME" is not set in the environment, then it is probably best to do:

```
local foo=os.getenv("FOO") or ""
```

otherwise `foo` will have the value of `nil`.

os.exit(number):
Exits a modulefile. Note that no environment variables are changed when this command is evaluated.

capture ("string"):
Run the "string" as a command and capture the output. This function uses the value of LD_PRELOAD and LD_LIBRARY_PATH found when Lmod is configured. Use **subprocess** if you wish to use the current values. There may be a trailing newline in the result which is your responsibility to remove or otherwise handle.:

```
local nprocs = capture("nprocs"):gsub("\n$","")
```

subprocess ("string")
Run the "string" as a command and capture the output. There may be a trailing newline in the result which is your responsibility to remove or otherwise handle.

isFile ("name"):
Returns true if "name" is any file type except directory.

isDir ("name"):
Returns true if "name" is a directory.

splitFileName ("name"):
Returns both the directory and the file name. `local d,f=splitFileName("/a/b/c.ext")`. Then `d="/a/b"`, `f="c.ext"`

LmodMessage ("string", ...):
Prints a message to the user.

LmodWarning ("string", ...):
Prints a warning message to the user.

LmodError ("string", "..."):
Print Error string and exit without loading the modulefile.

Note that LmodError() is treated as a warning when unloading as of Lmod 8.6.16

mode ():
Returns the string "load" when a modulefile is being loaded, "unload" when unloading, and "spider" when a modulefile is processed builting the spider cache which is used by *module avail* and *module spider*.

isloaded ("NAME"):
Return true when module "NAME" is loaded or is in the middle of a load. Use isPending() to distinguish between loaded or pending.

isPending ("NAME"):
Return true when module "NAME" is in the middle of a load(). This function is rarely needed. It can be useful when checking if one depends_on() package is currently being loaded.

isAvail ("NAME"):
Return true when "NAME" is possible to load. Note that it probably better to use the **try_load** () instead:

```
if ( not isloaded("foo") ) then try_load("foo") end
```

LmodVersion ():
The version of lmod.

execute {cmd="<*any command*>", modeA={"load"}}
Run any command with a certain mode. For example **execute** {cmd="ulimit -s unlimited",modeA={"load"}} will run the command **ulimit -s unlimited** as the last thing that the loading the module will do.

Modifier functions to prereq and loads

atleast ("name", "version"):
This modifier function will only succeed if the module is "version" or newer. See the between function for adding a "<" to modify the search criteria.

between ("name", "v1", "v2"): This modifier function will only
succeed if the module's version is equal to or between "v1" and "v2". Note that version "1.2" is the same as "1.2.0.0.0….". This means that between("foo","2.7","3.0") would include "foo/3.0" but not "foo/3.0.0.1". You can add a "<" to either the lower or upper version boundary to specify less than instead of "<=". So between("foo","2.7<","<3.0") would want any module greater than 2.7 and less than 3.0.

latest ("name"):
This modifier function will only succeed if the module has the highest version on the system.

Introspection Functions

The following functions allow for more generic modulefiles by finding the name and version of a modulefile.

myModuleName ():
Returns the name of the current modulefile without the version.

myModuleVersion ():
Returns the version of the current modulefile.

myModuleFullName ():
Returns the name and version of the current modulefile.

myModuleUsrName ():
Returns the name the user specified to load a module. So it could be the name or the name and version.

myFileName ():
Returns the absolute file name of the current modulefile.

myShellName ():
Returns the name of the shell the user specified on the command line.

myShellType ():
Returns the shellType based on the name of the shell the user specified on the command line. It returns sh for sh, bash, zsh, csh for csh, tcsh. Otherwise it is the same as **myShellName** ().

hierarchyA ("fullName", level):
Returns the hierarchy of the current module. See the section on Generic Modules for more details.

Math Functions

math.floor (): math floor function

math.ceil (): math ceil function

math.max (): math max function

math.min (): math min function

Special Functions

inherit (): imports the contents of exact same name module also
found in the module tree. (See *A Personal Hierarchy Mirroring the System Hierarchy* for an explanation.)

1.5.4 TCL Modulefile Functions

Lmod reads modulefiles written in either Lua or TCL. Lmod has to interpret the TCL modulefiles into Lua and then evaluate the Lua file. So it is always faster to interpret Lua file rather than TCL files.

Here are a list of TCL commands that are provided in addition to the standard TCL language.

add-property NAME value :
See *Module Properties* for how to use this command.

always-load A B C :
Load one or more modules. When this command is used in a modulefile that is being unloaded, this command has does nothing. See the **load** command for the

append-path NAME path :
Append **path** to environment variable **NAME**.

append-path –*delim char* NAME path *priority* :
Append **path** to environment variable **NAME**. Using the option –*delim char* to change the separator from ':' to *char* . Also the last optional argument can specify a priority. (See *Specifying Priorities for PATH entries* for details.)

complete shell name arg1 arg2 … "
Bash and tcsh support the complete function. Note that the shellName must match the name of the shell given on the Lmod command. There is no error if the shell names do not match. The command is ignored. See rt/complete/mf/tcl_spack/1.0.lua for an example.

conflict A B :
The current modulefile will only load if all listed modules are NOT loaded.

depends-on A B :
Loads all modules. When unloading only dependent modules are unloaded. See *Dependent Modules* for details.

break msg :
This command causes the evaluation of the current modulefile to stop and all changed in the user's environment to be ignored from the current modulefile. However, all other modulefiles are evaluated.

In other words, this command does not stop the operation, where as **exit** stops all evaluations. New in Lmod 8.6+

Note As of Lmod 8.6.16: break does nothing when unloading.

exit *number* :
Exits the module. No changes in the environment occur if this command is found.

extensions "numpy/2.1 scipy/3.2" :
This module provides the following extensions. Place the list of extensions as a single string.

family NAME :
A user can only have one family "name" loaded at a time. For example family("compiler") would mean that a user could only have one compiler loaded at a time.

is-loaded NAME :
Return true when module "NAME" is loaded.

module *command args* :
This command performs different actions depending on *command*:

- **add *A B* :**
  load one or more modules
- **load *A B* :**
  load one or more modules
- **try-load *A B* :**
  load one or more modules but does not report an error if not found.
- **load-any *A B* :**
  load any one of the following modulefiles
- **swap *A B* :**
  unload *A* and load *B*
- **switch *A B* :**
  unload *A* and load *B*
- **unload *A B* :**
  unload one or more modules.
- **del *A B* :**
  unload one or more modules.
- **rm *A B* :**
  unload one or more modules.
- **use *path* :**
  Add *path* to MODULEPATH
- **unuse *path* :**
  remove *path* to MODULEPATH

module-info *string* :
This command returns different things depending what *string* is:

mode : is the current mode: "load", "remove" or "display"

shell : The current shell specified by the user

shelltype : It has the value of "sh", "csh", "perl", "python", "lisp", "fish", "cmake", or "r".

flags : always returns 0

name : The fullname of the module.

user : always returns 0.

symbols : always returns 0.

specified : User specified name on command line.

module-whatis *string* :
The whatis command can be called repeatedly with different strings. See the Administrator Guide for more details.

prepend-path NAME path :
prepend to a path-like variable the value.

prepend-path *–delim char* NAME path *priority* :
prepend **path** to environment variable **NAME**. Using the option *–delim char* to change the separator from ':' to *char*. Also the last optional argument can specify a priority which is a number. (See *Specifying Priorities for PATH entries* for details.)

prereq A B:
The current modulefile will only load if **any** of the listed modules are already loaded.

pushenv NAME *value* :
sets **NAME** to *value* just like **setenv**. In addition it saves the previous value in a hidden environment variable. This way the previous state can be returned when a module is unloaded. **pushenv** ("FOO",false) will clear "FOO" and the pop will return the previous value.

remove-path NAME *value* :
remove value from a path-like variable for both load and unload modes.

remove-property NAME *value* :
See *Module Properties* for how to use this command.

reportError *string* :
Report an error and abort processing of the modulefile.

Note: During unloading, this command reports the error message but does not abort the processing of the modulefile. (as of Lmod 8.6.16+)

require-fullname :
Reports an error if the user specified name is not the fullname of the module (e.g. **module load gcc/10.1** vs **module load gcc**. Typically used in TCL modulefile as follows:

```
if { [ module-info mode load ] } {
    require-fullname
}
```

source-sh *shellName shell_script arg1 …*
source a shell script as part of a module. Supported shellNames are *sh*, *dash*, *bash*, *zsh*, *csh*, *tcsh*, *ksh*. When loading, Lmod automatically converts the shell script into module commands and saves the module commands in the environment. It does this by sourcing the shell script string in a subshell and comparing the environment before and after sourcing the shell script string. When unloading, the saved module commands from the environment are used. Aliases and shell functions are tracked.

Note that shell script string must not change between loading and unloading as the full string is used to reference the saved module commands.

Other shells could be supported with help from the community that uses that shell. (New in version 8.6)

This feature was introduced in Tmod 4.6 and was shamelessly studied and re-implemented in Lmod 8.6+.

set-alias NAME *value* :
Define an alias to **NAME** with *value*.

setenv NAME *value* :
Assigns to the environment variable "NAME" the value. Do not use this function to assign the initial to a path-like variable. Use **append_path** or **prepend_path** instead.

unset-alias NAME *value* :
Removes the **NAME** alias.

unsetenv NAME *value* :
unsets the **NAME** env. var.

versioncmp *version-string1 version-string2* :
Returns -1, 0, 1 if the version string are less-than, equal or greater than. Note that this command knows that 1.10 is newer than 1.8.

is-avail *name* :
Return 1 if the name is available for loading, 0 if not. (As of Lmod 8.6+)

haveDynamicMPATH :
This function tells that Lmod that this module has a dynamic $MODULEPATH when building the spider cache. See *The spider tool* for details.

TCL Modulefile Functions NOT SUPPORTED

atleast :
It is not possible to use the atleast function inside a TCL modulefile

between :
It is not possible to use the between function inside a TCL modulefile

latest :
It is not possible to use the latest function inside a TCL modulefile

TCL Global Variables

The following TCL global variables are set inside modulefiles and .modulerc and .version files.

ModuleTool
[This is the string "Lmod". This works for Lmod] 8.4.8+. This variable also exists in Tmod version 4.7 or greater and reports "Modules".

ModuleToolVersion
[This is the current version of Lmod. This] works for Lmod 8.4.8+ This variable also exists in Tmod version 4.7 or greater.

1.5.5 Lmod Environment variables

Environment variables defined by Lmod startup files

LMOD_CMD : The path to the installed lmod command.

LMOD_DIR
[The directory that contains the installed lmod] command. This environment variable is usefull for running the **spider** command: i.e. $LMOD_DIR/spider. This is the libexec directory

LMOD_PKG
[This the path the directory that contains the libexec,] init etc directories.

LMOD_ROOT : the parent directory of LMOD_PKG.

LMOD_sys : Typically what uname -s returns.

LMOD_VERSION : The current version of Lmod.

LMOD_SETTARG_FULL_SUPPORT
[If this environment variable is set] then when the settarg module is loaded several shell functions are defined such as "targ". See the *Combining modules and build systems with settarg* for more details.

Lmod Environment variables defined when evaluating a modulefile

LMOD_VERSION_MAJOR
[The current major version. If it is X.Y.Z] then X is returned (i.e. 10.14.17 -> 10) (exists for Lmod 5.1.5+)

LMOD_VERSION_MINOR
[The current minor version. If it is X.Y.Z] then Y is returned (i.e. 10.14.17 -> 14) (exists for Lmod 5.1.5+)

LMOD_VERSION_SUBMINOR
[The current subminor version. If it is X.Y.Z] then Z is returned (i.e. 10.14.17 -> 17) (exists for Lmod 5.1.5+)

ModuleTool
[This environment variable is defined to be] **Lmod**. (exists for Lmod 8.4.7+) It is defined in Tmod version 4.7+ as "Modules"

ModuleToolVersion
[Current Version of Lmod (exists for Lmod] 8.4.7+). It also reports the version of Tmod as of version 4.7 or later.

Environment variables to change Lmod behavior

See *Configuring Lmod for your site* for the list of env. variables that change the behavior of Lmod.

1.5.6 Module names and module naming conventions

The modulefiles contain commands that change the user's environment to make an application or library available to a user. So loading a modulefile named gcc/7.1 should hopefully make the programs Gnu Compiler Collection (i.e. gcc, gfortran and g++ compilers) available to a user.

So the `gcc/7.1` module is a file named either 7.1 or 7.1.lua depending on whether the commands are written in TCL or Lua respectively and this file is in the `gcc` directory.

To see a more complete module layout, let's assume your site has has a MODULEPATH set to:

```
/apps/modulefiles/Core:/apps/modulefiles/Other
```

And the directory layout from /apps/modulefiles is:

```
$ cd /apps/modulefiles; tree -a
.
├── Core
│   ├── A
│   │   ├── 1.0
│   │   └── 2.0.lua
│   ├── gcc
│   │   ├── 5.4
│   │   └── 7.1.lua
│   └── StdEnv.lua
└── Other
    ├── C
```

(continues on next page)

(continued from previous page)

```
    │   ├── 3.3.lua
    │   └── 3.4.lua
    └── D
        └── 4.0.lua
```

In the above structure there are 8 modulefiles. So the 5 modulefiles under Core are: A/1.0, A/2.0, gcc/5.4, gcc/7.1 and StdEnv. Note that Lmod's name of a modulefile doesn't include the .lua extension. Also note that both A and gcc have two versions where as StdEnv has a name with no version.

fullName == sn/version

The fullName of a modulefile is complete name which is made of two parts: sn and version. The sn is the shortname and represents the minumum name that can be used to specify a module. So for the `gcc/7.1` modulefile. The fullName is `gcc/7.1` and the sn is `gcc` and the version is `7.1`. This naming convention is known as NAME/VERSION and is abbreviated N/V. There are two more complicated naming schemes known as Category/Name/Version (a.k.a. C/N/V) and Name/Version/Version (a.k.a N/V/V). In all three naming schemes, a modulefile has a fullName which is split into sn / version. The split between the sn and version will be different depending on which naming scheme is used. Sometimes the version doesn't exist like for StdEnv module. In this case the fullName is the same as the sn.

In next section *How Lmod Picks which Modulefiles to Load*, we explain how a Lmod takes an `sn` and determines which modulefile to load. But in short, Lmod picks the marked default or if there is no default then Lmod picks the "highest" version.

Special Names for modulefiles

If the version has a leading dot then this modulefile is hidden. That is, it can be loaded but it won't be shown by `module avail`. If the sn name has a leading dot, such as `.X` then every version of this module will be hidden. So `.X/1.0` is hidden and so is `Y/.2.0`.

Because of the way that Lmod passes information between module command invocations, a modulefile **CANNOT** have two or more leading underscores. So naming a modulefile `_A/1.0` is acceptable but having two underscore such as `__B/1.0` is not!

Module Naming scheme: Category/Name/Version (C/N/V)

Sites may wish to group similar modules into categories. For example, all the biology packages grouped together, like bowtie and tophat packages be named: `bio/bowtie/3.4` or `bio/tophat/2.7`. When the fullName is `bio/bowtie/3.4` then the sn for is `bio/bowtie` and the version is `3.4`.

Sites should think carefully about chosing to using C/N/V. This can make it easier for users to know which modules provide say physics, chemistry or biology applications but it does lead to great deal more typing of which tab completion provided by the bash or zsh shells can only do so much.

Sites can have meta-module inside a category. A meta-module is a module that has no version. For example, suppose you have the following C/N/V modulefile structure:

```
$ cd /apps/modulefiles/Other; tree -a
.
└── bio
    ├── bowtie
    │   └── 3.1
    ├── genomics.lua
```

(continues on next page)

(continued from previous page)

```
    └── tophat
        └── 7.2
```

In this case there are three modules `bio/bowtie/3.1`, `bio/tophat/7.2`. The `bio/genomic` module is a meta module where the fullName is the same as the sn and it has no version.

The rule that Lmod uses to determine which modules are meta modules and which have a version is the following. If a file is in a directory that has other sub-directories (other than . and ..), the file is a meta-module. So the genomic.lua file is part of the sn for bio/genomics because the genomics.lua file is in a sub-directory that has the directories `bowtie` and `tophat`.

Sites can mix N/V and C/N/V layouts, Lmod will be able to decide the sn and versions by walking directory tree. In general, the fullName, will be divided into directories names to become the sn and the version will be the file. So for the fullName `bio/tophat/7.2` the directores bio and tophat become the sn, `bio/tophat` and the version is `7.2`.

Lmod supports as many directory levels as site likes. For example, a site could have a modulefile named `A/B/C/D/1.1` where the sn name is `A/B/C/D` and the version is `1.1`.

Module Naming scheme: Name/Version/Version (N/V/V)

A site many wish to have directories has part of the version. This could be used to have 32 or 64 bit versions of the same package. So for example a site might wish to have a modules named `acme/32/4.2` and `acme/64/4.2` where sn is `acme` and the versions are `32/4.2` and `64/4.2`.

By default, Lmod assumes that the version is just the file, so lmod needs to be told where the sn / version split is. This can be done by creating an empty `.version` or `.modulerc` file where a site wants the split to be. For the above example, the following layout make `acme` be the sn:

```
$ tree -a
.
└── acme
    ├── .version
    ├── 32
    │   └── 4.2
    └── 64
        └── 4.2
```

because there is a .version at the same level as the 32 and 64 directories. With the .version file, the fullName is `acme/64/4.2` and the sn is `acme` and the version `64/4.2`. If the .version file was removed then the sn would be `acme/64` and the version would be `4.2`.

Sites are can name modules with as many directories as they like. For example a site can have a module named:

```
mpi/mpich/64/3.1/048
```

If there is an empty file at `mpi/mpich/.version` then the sn would be `mpi/mpich` and the version would be `64/3.1/048`.

1.5.7 How Lmod Picks which Modulefiles to Load

Lmod uses the directories listed in `MODULEPATH` to find the modulefiles to load. Suppose that you have a single directory `/opt/apps/modulefiles` that has the following files and directories:

```
/opt/apps/modulefiles

StdEnv.lua  ucc/  xyz/

./ucc:
8.1.lua  8.2.lua

./xyz:
10.1.lua
```

Lmod will report the following directory tree like this:

```
---------- /opt/apps/modulefiles -----------
StdEnv    ucc/8.1    ucc/8.2 (D)    xyz/10.1
```

We note that the `.lua` extension has not been reported above. The `.lua` extension tells Lmod that the contents of the file are written in the Lua language. All other files are assumed to be written in TCL.

Here the name of the file or directory under `/opt/apps/modulefiles` is the name of the module. The normal way to specify a module is to create a directory to be the name of the module and the file(s) under that directory are the version(s). So we have created `ucc` and `xyz` directories to be the names of the module. There are two version files under `ucc` and one version file under `xyz`.

The `StdEnv.lua` file is another way to specify a module. This file is a module with no version associated with it. These are typically used as a meta-module: a module that loads other modules.

N/V: Picking modules when there are multiple directories in MODULEPATH

The follow rules apply when the module layout is either Name/Version (N/V) or Category/Name/Version (C/N/V). The rules are a little different for Name/Version/Version (N/V/V) as described in the section below. When there are multiple directories specified in MODULEPATH, the rules get more complicated on what modulefile to load. Lmod uses the following rules to locate a modulefile:

1. The user may specify *C/N*/default or *N*/default which is exactly the same as *C/N* or *N*. Namely Lmod removes the string "/default" and then continues as if it was never there.
2. It looks for an exact match in all `MODULEPATH` directories. If there are multiple exact matches, then Lmod picks the marked default. Otherwise, it picks the first match it finds. This is true of hidden modules. Specifying the fullName of a module will load it.
3. If a site has "extended defaults" enabled and a user types in part of the version then that part is used select the "best" of that version if any exist. Note that if user enters "abc/1" then it will match "abc/1.*" or "abc/1-*" but not "abc/17.*"
4. If the name doesn't contain a version then Lmod looks for a marked default in the first directory that has one. A marked default which is also hidden will be loaded.
5. Finally it looks for the "Highest" Version in all `MODULEPATH` directories. If there are two or more modulefiles with the "Highest" version then the first one in `MODULEPATH` order will be picked. Note that hidden modulefiles are ignored when choosing the "Highest".
6. As a side node, if there are two version files, one with a `.lua` extension and one without, the lua file will be used over the other one. It will be like the other file is not there.

As an example, suppose you have the following module tree:

```
---------- /home/user/modulefiles -----------
xyz/11.1  xyz/11.2


---------- /opt/apps/modulefiles ------------
StdEnv    ucc/8.1    ucc/8.2         xyz/10.1

---------- /opt/apps/mfiles -----------------
ucc/8.3 (D)  xyz/12.0   xyz/12.1 (D) xyz/12.2
```

If a user does the following command:

```
$ module load ucc/8.2 xyz
```

then ucc/8.2 will be loaded because the user specified a particular version and xyz/12.1 will be loaded because it is the highest version across all directories in `MODULEPATH`.

If a user does the following command:

```
$ module load xyz/11
```

then xyz/11.2 will be loaded because it is the highest of the xyz/11.* modulefiles. Note that:

```
$ module load xyz/12
```

will load xyz/12.1 not xyz/12.2 because 12.1 is the marked default and is therefore the highest "xyz/12.*".

Overriding a marked default

Suppose you wish to make a personal version of a modules. It could be that you are testing your own xyz/12.1 module. If you have the following module layout:

```
---------- /home/user/modulefiles -----------
xyz/11.1  xyz/11.2   xyz/12.1


---------- /opt/apps/modulefiles ------------
StdEnv    ucc/8.1    ucc/8.2         xyz/10.1

---------- /opt/apps/mfiles -----------------
ucc/8.3 (D)  xyz/12.0   xyz/12.1 (D) xyz/12.2
```

If you try to load xyz/12.1, the modulefile in /opt/apps/mfiles will be loaded and not the one in /home/user/modulefiles.

If you wish to load the modulefile in /home/user/modulefiles then you'll need to make that one a marked default. The next section shows how to accomplish this.

You should verify that the marked default has moved:

```
---------- /home/user/modulefiles -----------
xyz/11.1  xyz/11.2   xyz/12.1 (D)
```

(continues on next page)

(continued from previous page)

```
---------- /opt/apps/modulefiles ------------
StdEnv    ucc/8.1    ucc/8.2         xyz/10.1

---------- /opt/apps/mfiles -----------------
ucc/8.3 (D)  xyz/12.0   xyz/12.1     xyz/12.2
```

Marking a Version as Default

Suppose you have several versions of the mythical UCC compiler suite:

```
$ module avail ucc
---------- /opt/apps/modulefiles/Core -----------
ucc/8.1   ucc/9.2   ucc/11.1   ucc/12.2 (D)
```

and you would like to make the 11.1 version the default. Lmod searches three different ways to mark a version as a default in the following order. The first way is to make a symbolic link between a file named "`default`" and the desired default version.:

```
$ cd /opt/apps/modulefiles/Core/ucc; ln -s 11.1.lua default
```

A second way to mark a default is with a .modulerc.lua file in the same directory as the modulefiles:

```
module_version("ucc/11.1", "default")
```

A third way to mark a default is with a .modulerc file in the same directory as the modulefiles:

```
#%Module
module-version ucc/11.1 default
```

There is a fourth method to pick the default module. If you create a .version file in the ucc directory that contains:

```
#%Module
set   ModulesVersion   "11.1"
```

Please note that a .modulerc.lua, .modulerc or .version file must be in the same directory as the 11.1.lua file in order for Lmod to read it.

Using any of the above three ways will change the default to version 11.1.

```
$ module avail ucc
---------- /opt/apps/modulefiles/Core -----------
ucc/8.1   ucc/9.2   ucc/11.1 (D)   ucc/12.2
```

Lmod Order of Marking a Default

As stated above, there are four files used to mark a default:

1. default symlink
2. .modulerc.lua
3. .modulerc
4. .version

Lmod searches in this order. If it finds a number earlier in the list then the other are ignored. In other words if your site as both a default symlink and a .modulerc.lua file then the default file is used and the .modulerc.lua file is ignored. Sites can check duplicate ways of marking a default (among other checks) with:

```
$ $LMOD_DIR/check_module_tree_syntax $MODULEPATH
```

Highest Version

If there is no marked default then Lmod chooses the "Highest" version across all directories:

```
$ module avail ucc

---------- /opt/apps/modulefiles/Core -----------
ucc/8.1   ucc/9.2   ucc/11.1   ucc/12.2

---------- /opt/apps/modulefiles/New -----------
ucc/13.2 (D)
```

The "Highest" version is by version number sorting. So Lmod "knows" that the following versions are sorted from lowest to highest:

```
  2.4dev1
    2.4a1
 2.4beta2
   2.4rc1
      2.4
  2.4.0.0
    2.4-1
2.4.0.0.1
    2.4.1
```

N/V/V: Picking modules when there are multiple directories in MODULEPATH

The rules are different when the module layout is Name/Version/Version (N/V/V). The rules for N/V can be called `Find Best` where as N/V/V is `Find First`. Note that if any one of the directories in `MODULEPATH` are in N/V/V format, the whole tree is searched with N/V/V rules. Below are the rules that Lmod uses to locate a modulefile when in N/V/V mode:

1. The user may specify *N*/default as *N/V*/default which is exactly the same as *N* or *N/V*. Namely Lmod removes the string "/default" and then continues as if it was never there.
2. If there is an exact match and the exact match is marked default then this marked default is chosen no matter which directory it is in.

3. It looks for an exact match in all `MODULEPATH` directories. It picks the first match it finds.
4. If there is no exact match then Lmod finds the first match for the names that it has. It matches by directory name. No partial matches are on directory names
5. In the directory that is found above the first marked default is found.
6. If there are no marked defaults, then the "highest" is chosen.
7. The two above rules are followed at each directory level.
8. If a site has "extended defaults" enabled and a user types in part of the version then Lmod picks the "best" of the modulefiles that match. Partial matching is only available for the last part of the version.

For example with the following module tree where foo is the name of the module and rest are version information:

```
----- /apps/modulefiles/A ----------------
foo/2/1  foo/2/4    foo/3/1    foo/3/2 (D)

----- /apps/modulefiles/B ----------------
foo/3/3    foo/3/4

----- /apps/modulefiles/C ----------------
bar/32/3.0.1   bar/32/3.0.4   bar/32/3.1.5
```

Then the commands `module load foo` and `module load foo/3` would both load `foo/3/2`. The command `module load foo/2` would load `foo/2/4`.

When searching for `foo`, Lmod finds it in the `A` directory. Then seeing a choice between `2` and `3` it picks `3` as it is higher. Then in the `foo/3` directory it choses `2` as it is higher than `1`. To load any other `foo` module, the full name will have to specified.

The commands `module load bar` and `module load bar/32/3.1` would load `bar/32/3.1.5`. And the command `module load bar/32/3.0` would load `bar/32/3.0.4`. Note that command `module load bar/3` would fail to load any modules.

Marking a directory as default in an N/V/V layout

There are three ways to mark a directory as a default: Using a `default` symlink, or the use of either the `.modulerc` or `.version` files. Since it is possible (but not recommended) to have all three possibilities, This is the same technique that was used before to mark a particular version file when in an N/V layout. Lmod choses the setting of the default directory in the following order:

1. `default` symlink
2. `.modulerc.lua`
3. `.modulerc`
4. `.version`

Suppose that you have the following architecture split with (32,64,128) bit libraries and you want the 64 directory to be the default. With the following structure:

```
----- /apps/modulefiles/A ----------------
foo/32/1    foo/64/1      foo/128/1
foo/32/4    foo/64/2 (D)  foo/128/2
```

You can have a symlink for `/apps/modulefiles/A/foo/default` which points to `/apps/modulefiles/A/foo/64`. Or you can have the contents of `/apps/modulefiles/A/foo/.modulerc` contain:

```
#%Module
module-version 64 default
```

or you can have the contents of `/apps/modulefiles/A/foo/.modulerc.lua` contain:

```
module_version("64","default")
```

or you can have the contents of `/apps/modulefiles/A/foo/.version` contain:

```
#%Module
set ModulesVersion "64"
```

Normally the 128 directory would be chosen as the default directory as 128 is higher than 64 or 32 but any one of these files forces Lmod to pick 64 over the other directories.

Why do N/V/V module layouts use `Find First` over `Find Best`?

The main problem here is that of the default directories. There is no sane way to pick. Suppose that you have the following structure:

```
----- /apps/modulefiles/A ----------------
foo/32/1    foo/64/1      foo/128/1
foo/32/4    foo/64/2 (D)  foo/128/2

----- /apps/modulefiles/B ----------------
foo/32/5    foo/64/3      foo/128/3
foo/32/6    foo/64/4      foo/128/4
```

And where the default directory in `A` in `64` and in `B` it is `32`. When trying to load `foo/64` the site has marked `64` the default in `A` where as it is not in `B`. Does that mean that `foo/64/2` is "higher" that `foo/64/4` or not. There is no clear reason to pick one over the other so Lmod has chosen `Find First` for N/V/V module layouts.

For sites that are mixing N/V and N/V/V module layouts they may wish to change Lmod to use the find first rule in all cases. See *Configuring Lmod for your site* to see how to configure Lmod for find first.

Autoswapping Rules

When Lmod autoswaps hierarchical dependencies, it uses the following rules:

1. If a user loads a default module, then Lmod will reload the default even if the module version has changed.
2. If a user loads a module with the version specified then Lmod will only load the exact same version when swapping dependencies.

For example a user loads the intel and boost library:

```
$ module purge; module load intel boost; module list

Currently Loaded Modules:
1) intel/15.0.2  2) boost/1.57.0
```

Now swapping the Intel compiler suite for the Gnu compiler suite:

```
The following have been reloaded with a version change:
1) boost/1.57.0 => boost/1.56.0
```

Here boost has been reloaded with a different version because the default is different in the gcc hierarchy. However if the user does:

```
$ module purge; module load intel boost/1.57.0; module list

 Currently Loaded Modules:
 1) intel/15.0.2  2) boost/1.57.0
```

And:

```
$ module swap intel gcc;

Inactive Modules:
1) boost/1.57.0
```

Since the user initially specified loading boost/1.57.0 then Lmod assumes that the user really wants that version. Because version 1.57.0 of boost isn't available under the gcc hierarchy, Lmod marks this boost module as inactive. This is true even though version 1.57.0 is the default version of boost under the Intel hierarchy.

1.5.8 Providing A Standard Set Of Modules for all Users

Users can be provided with an initial set of modulefiles as part of the login procedure. Once a list of modulefiles has been installed, please create a file called `StdEnv.lua` and place it in the `$MODULEPATH` list of directories, typically `/opt/apps/modulefiles/Core/StdEnv.lua`. The name is your choice, the purpose is provide a standard list of modules that get loaded during login. In `StdEnv.lua` is something like:

```
load("name1","name2","name3")
```

Using the /etc/profile.d directory system described earlier to create a file called `z01_StdEnv.sh`

```
if [ -z "$__Init_Default_Modules" ]; then
   export __Init_Default_Modules=1;

   ## ability to predefine elsewhere the default list
   LMOD_SYSTEM_DEFAULT_MODULES=${LMOD_SYSTEM_DEFAULT_MODULES:-"StdEnv"}
   export LMOD_SYSTEM_DEFAULT_MODULES
   module --initial_load --no_redirect restore
else
   module refresh
fi
```

Similar for z01_StdEnv.csh:

```
if ( ! $?__Init_Default_Modules )  then
  setenv __Init_Default_Modules 1
  if ( ! $?LMOD_SYSTEM_DEFAULT_MODULES ) then
    setenv LMOD_SYSTEM_DEFAULT_MODULES "StdEnv"
  endif
  module --initial_load restore
else
  module refresh
endif
```

The z01_Stdenv.* names are chosen because the files in /etc/profile.d are sourced in alphabetical order. These names guarantee they will run after the module command is defined.

The z01_Stdenv.sh now includes `--no_redirect`. This option prevents sites that configure Lmod to write messages to stdout to write them to stderr instead. This is important as any messages written to stdout during shell startup causes scp copies to fail. Csh/tcsh cannot write messages to stdout due to limitations in that shell.

The first time these files are sourced by a shell they will set `LMOD_SYSTEM_DEFAULT_MODULES` to `StdEnv` and then execute `module restore`. Any subshells will instead call `module refresh`. Both of these statements are important to get the correct behavior out of Lmod.

The `module restore` tries to restore the user's default collection. If that doesn't exist, it then uses contents of the variable `LMOD_SYSTEM_DEFAULT_MODULES` to find a colon separated list of Modules to load.

The `module refresh` solves an interesting problem. Sub shells inherit the environment variables of the parent but do not normally inherit the shell aliases and functions. This statement fixes this. Under a "`refresh`", all the currently loaded modules are reloaded but in a special way. Only the functions which define aliases and shell functions are active, all others functions are ignored.

The above is an example of how a site might provide a default set of modules that a user can override with a default collection. Sites are, of course, free to set up Lmod any way they like. The minimum required setup (for bash with z01_StdEnv.sh) would be:

```
if [ -z "$__Init_Default_Modules" ]; then
   export __Init_Default_Modules=1;

   module --initial_load restore
else
   module refresh
fi
```

The module restore command still depends on the environment variable LMOD_SYSTEM_DEFAULT_MODULES but that can be set somewhere else.

Lmod, LD_LIBRARY_PATH and screen

In general, it is probably better to NOT use `screen` and use `tmux` instead. The problem with `screen` is that it is a *set-group-id* (SGID) program (`tmux` is not). That means it uses the group associated with the executable and not the user's group. The main consequence of this is that the operating system removes LD_LIBRARY_PATH from the environment. This is a security feature built into the operating system.

A site could change z01_StdEnv.sh to have:

```
if [ -z "$__Init_Default_Modules" -o -z "$LD_LIBRARY_PATH" ]; then
   export __Init_Default_Modules=1;

   module --initial_load restore
else
   module refresh
fi
```

to help with the situation. This will force Lmod restore the initial set of modules (or the user's default collection). This works fine as long as the initial set of modules actually sets LD_LIBRARY_PATH. If it doesn't every interactive sub-shell will do a module restore, which is probably not what you want. For example, if you see the following then you probably want to remove the test for an empty LD_LIBRARY_PATH:

```
$ module list
Currently Loaded Modules:
   1) git/2.7     2) StdEnv

$ module load bowtie
$ bash
$ module list
Currently Loaded Modules:
   1) git/2.7     2) StdEnv
```

Running the bash shell caused the module restore to run which unloaded all modules and restored the modules back to the initial set.

1.5.9 Converting from TCL/C Environment Modules to Lmod

Sites converting from the TCL/C based Environment Modules (a.k.a Tmod) should be aware of some of the differences between Tmod and Lmod. Lmod is a complete re-implementation of the environment module concept with no code reuse from the original program. There are some differences between that can affect how easy your transition to Lmod.

One major difference between the two tools is that Lmod is written in Lua and not TCL. Lmod has to translate TCL into Lua. This means that pure TCL statements are evaluated by a program called **tcl2lua.tcl**. This program outputs lua statements. This can lead to difference between Tmod and Lmod because different order that statements are evaluated. The details are discussed at *How does Lmod convert TCL modulefile into Lua*.

Another important difference between the tools is that Lmod has the *One Name* rule. That is you can only have one “name” loaded at a time. Suppose a user tries to load gcc/5.4 and gcc/7.1 at the same time:

```
$ module load gcc/5.4
$ module load gcc/7.1

The following have been reloaded with a version change:
  1) gcc/5.4 => gcc/7.1
```

Lmod is telling you that the gcc/5.4 has been replaced by gcc/7.1. This happens automatically without anything special in the modulefiles.

Module Naming rules

The fullname of a module is split into name/version. Normally the version is just the string after the last slash. So gcc/5.4 has a name of gcc and a version of 5.4. A module named compiler/gcc/5.4 would have a name of compiler/gcc and a version of 5.4.

With Lmod 7+, sites can change the name - version split if they like. For example a site might name their modules bowtie/64/3.4 and bowtie/32/3.4 where the name of the module would be bowtie and the version would be either 64/3.4 or 32/3.4. Please see *N/V/V: Picking modules when there are multiple directories in MODULEPATH* for details.

1.6 Advanced Topics

1.6.1 How to report a bug in Lmod

Lmod has some built-in tools to make debugging possible on your site. The first feature of Lmod is the configuration report:

```
$ module --miniConfig
```

This reports how Lmod has been configured at build time as well as any `LMOD_*` environment variables set. The second tool is the debug output also built-in to Lmod:

```
$ module -D load foo 2> load.log
```

The `-D` option turns on the debug printing and will report all the steps that Lmod took to load a module called `foo`. Note that the configuration report is at the top of every debug output.

Steps to report a bug

1. Test your bug against the latest release from github. Please pull the HEAD branch.
2. Try to reduce the problem to the fewest number of modules. Shoot for 1 or 2 modulefiles if you can.
3. Run the command that fails. i.e. `module -D` *cmd modulefile* ... `2> lmod.log`
4. Combine the lmod.log file, the modulefiles from step 2, and possibly the spider cache file into a tar file.
5. Send the tar file to mclay@tacc.utexas.edu

1.6.2 Rules for PATH-like variables

Lmod provides great flexibility in handling prepending and appending to path-like variables. This is especially true when there are duplicate entries. A modulefile can modify a variable like `PATH` using `append_path()` or `prepend_path()` or their TCL equivalents. For example, suppose that `PATH=/usr/bin:/usr/local/bin` then:

```
prepend_path("PATH","/bin")
```

would change `PATH` to `/bin:/usr/bin:/usr/local/bin`. The interesting question is what happens when the following is executed:

```
prepend_path("PATH","/usr/bin")
```

That is, when `/usr/bin` is already in $PATH or any other duplicate entry.

LMOD_DUPLICATE_PATHS == yes

Lmod supports two main styles for dealing with duplicates. If $LMOD_DUPLICATE_PATHS is yes (or Lmod is configured that way). Then duplicates entries are allowed (assume PATH is empty):

```
prepend_path("PATH","/A")  --> PATH = /A
prepend_path("PATH","/B")  --> PATH = /B:/A
prepend_path("PATH","/A")  --> PATH = /A:/B:/A
```

When unloading a modulefile with prepend_path(), Lmod removes the first matching entry it finds. Reversing an append_path(), Lmod removes the last matching entry.

LMOD_DUPLICATE_PATHS == no

The default setting of Lmod is that duplicates are not allowed. When prepending, Lmod pushes the directory to be first even if it is a duplicate (assume PATH is empty):

```
append_path("PATH","/A")   --> PATH = /A
prepend_path("PATH","/B")  --> PATH = /B:/A
prepend_path("PATH","/A")  --> PATH = /A:/B
```

When duplicates are not allowed, Lmod maintains a reference count on each entry. That is, Lmod knows that "/A" has appended/prepended twice and "/B" once. This means that two prepend_path("PATH","/A") will be required to completely remove "/A" from $PATH.

LMOD_TMOD_PATH_RULE == yes

If this env. var is set (or configured), then Lmod does not change the order of entries but it does increase the reference count (assume $PATH is empty):

```
append_path("PATH","/A")   --> PATH = /A
prepend_path("PATH","/B")  --> PATH = /B:/A
prepend_path("PATH","/A")  --> PATH = /B:/A
```

Here we see that prepending "/A" does not change the order of directories in $PATH. Obviously if LMOD_TMOD_PATH_RULE is yes then duplicates are not allowed.

Special treatment for $MODULEPATH

The MODULEPATH environment variable is treated special. No duplicates entries are ever allowed even if LMOD_DUPLICATE_PATHS == yes. It always uses reference counting for PATH entries. In order to not confuse users. The command:

```
$ module unuse /foo/bar
```

will always remove the path entry, even if the reference count is greater than 1. Also a user can always clear $MODULEPATH with:

```
$ module unuse $MODULEPATH
```

Specifying Priorities for PATH entries

There are rare occasions where a site might want a directory to at the beginning of the PATH. This can be done by giving a priority to a path:

```
prepend_path{"PATH","/foo/bar",priority=100}
```

Note the use of curly braces instead of parentheses and setting priority to a number. Lmod groups the entries of the same priority together. This means that `/foo/bar` will likely be at the beginning of $PATH as long as no other entry has a higher priority.

Assuming that PATH is initially empty, here is an example:

```
prepend_path{"PATH","/foo",priority=100}  --> PATH = /foo
prepend_path("PATH","/A")                 --> PATH = /foo:/A
prepend_path("PATH","/B")                 --> PATH = /foo:/B/A
```

Lmod remembers the priority between invocations, meaning that you'll get the same results even if the following where in three separate modulefiles.

An Example of Loading and Unloading a Module

Above we showed that there are three modes for path like variables:

1. LMOD_DUPLICATE_PATH=no
2. LMOD_DUPLICATE_PATH=no, LMOD_TMOD_PATH_RULE=yes
3. LMOD_DUPLICATE_PATH=yes

Let's assume that $PATH = `/A:/B:/C` and the module FOO is:

```
prepend_path("PATH","/C")
```

then the following table shows what happens for each of the three modes when loading and unloading FOO. Note that `/A(2)` is the path entry `/A` a reference count of 2:

Action	1	2	3
original PATH	/A(1):/B(1):/C(1)	/A(1):/B(1):/C(1)	/A:/B:/C
module load FOO	/C(2):/A(1):/B(1)	/A(1):/B(1):/C(2)	/C:/A:/B:/C
module unload FOO	/C(1):/A(1):/B(1)	/A(1):/B(1):/C(1)	/A:/B:/C

For mode (1) where no duplicates are allowed, upon loading FOO path `/C` is moved to the beginning and stays there when unloading FOO. For mode (2), If a directory is already in `$PATH`, it is not moved, only the ref count is increased on load and decreased upon unload. Finally in mode (3), loading causes `/C` to be placed at the beginning and unloading removes it from the beginning.

When duplicates are allowed and unloading a module, Lmod does not remember which module inserted which directory where, it just removes the first or last entry depending on whether it was a prepend_path() or append_path() respectively. Also there is no reference counting when duplicates are allowed. It is not necessary and doesn't make sense.

1.6.3 Hidden Modules

To see hidden modules a user can do:

```
$ module --show_hidden avail
$ module --show_hidden spider
```

To hide modules, a site can name a module with a leading "." for the version or the name:

The following tree contains 3 modules, 2 hidden and one not:

```
$ tree -a modulefiles

modulefiles
├── .B
│   └── 3.0.lua
└── A
    ├── .1.0.lua
    └── 2.0.lua

$ module avail

 A/2.0 (D)

$ module --show_hidden avail

 .B/3.0 (H)    A/.1.0 (H)    A/2.0 (D)
```

It is also possible to mark modules with functions in modulerc files. See *Site and user control of defaults, aliases and hidden modules* for details on how to mark by using the modulerc files.

Remember that hidden modules can be loaded with normal commands.

Finally, if your site wishes to mark many modules as hidden, you can use the hook function isVisibleHook(). See *SitePackage.lua and hooks* for details. Also see the contrib/more_hooks/SitePackage.lua file for a worked example.

1.6.4 How to use a Software Module hierarchy

Libraries built with one compiler need to be linked with applications built with the same compiler version. For High Performance Computing there are libraries called Message Passing Interface (MPI) that allow for efficient communication between tasks on a distributed memory computer with many processors. Parallel libraries and applications must be built with a matching MPI library and compiler. To make this discussion clearer, suppose we have the intel compiler version 15.0.1 and the gnu compiler collection version 4.9.2. Also we have two MPI libraries: mpich version 3.1.2 and openmpi version 1.8.2. Finally we have a parallel library HDF5 version 1.8.13 (phdf5).

Of the many possible ways of specifying a module layout, this flat layout of modules is a reasonable way to do this:

```
$ module avail

--------------- /opt/apps/modulefiles ----------------------
gcc/4.9.2                         phdf5/gcc-4.9-mpich-3.1-1.8.13
intel/15.0.2                      phdf5/gcc-4.9-openmpi-15.0-1.8.13
mpich/gcc-4.9-3.1.2               phdf5/intel-15.0-mpich-3.1-1.8.13
mpich/intel-15.0-3.1.2            phdf5/intel-15.0-openmpi-15.0-1.8.13
```

(continues on next page)

(continued from previous page)

```
openmpi/gcc-4.9-1.8.2
openmpi/intel-15.0-1.8.2
```

In order for users to load the matching set of compilers and MPI libraries, they will have to load the matching set of modules. For example this would be correct:

```
$ module load gcc/4.9.2 openmpi/gcc-4.9-1.8.2  phdf5/gcc-4.9-openmpi-15.0-1.8.13
```

But it is quite easy to load an incompatible set of modules. Now, it is possible that the system administrators at your site might have set up `conflict` s to avoid loading mismatched modules. However, using conflicts can be fragile. What happens if a site adds a new compiler such as clang or pgi or a new mpi stack? All those module file conflict statements will have to be updated.

A different strategy is to use a software hierarchy. In this approach a user loads a compiler which extends the **MODULEPATH** to make available the modules that are built with the currently loaded compiler (similarly for the mpi stack).

Our modulefile hierarchy is stored under `/opt/apps/modulefiles/{Core,Compiler,MPI}`. The Core directory is for modules that are not dependent on Compiler or MPI implementations. The Compiler directory is for packages which are only Compiler dependent. Lastly, the MPI directory is packages which dependent on MPI-Compiler pairing. The modulefiles for the compilers are placed in the Core directory. For example the gcc version 4.9.2 file is in Core/gcc/4.9.2.lua and contains:

```
-- Setup Modulepath for packages built by this compiler
local mroot = os.getenv("MODULEPATH_ROOT")
local mdir  = pathJoin(mroot,"Compiler/gcc", "4.9")
prepend_path("MODULEPATH", mdir)
```

This code asks the environment for **MODULEPATH_ROOT** which is `/opt/apps/modulefiles` and the last two lines prepend `/opt/apps/modulefiles/Compiler/gcc/4.9` to the **MODULEPATH** .

The modulefiles for the MPI implementations are placed under the Compiler directory because they only depend on a compiler. The openmpi module file for the gcc-4.9.2 compiler is then stored at `/opt/apps/modulefiles/Compilers/gcc/4.9/openmpi/1.8.2.lua` and it contains:

```
-- Setup Modulepath for packages built by this MPI stack
local mroot = os.getenv("MODULEPATH_ROOT")
local mdir = pathJoin(mroot,"MPI/gcc", "4.9","openmpi","1.8")
prepend_path("MODULEPATH", mdir)
```

The above code will prepend `/opt/apps/modulefiles/MPI/gcc/4.9/openmpi/1.8` to the **MODULEPATH**.

The above description is a suggested way to handle the modulefile software hierarchy. The software packages themselves can be stored in many ways. For software packages, but not the modulefiles, we store them in another software hierarchy as follows:

1. Core packages: **/opt/apps/pkgName/version**
2. Compiler dependent packages: **/opt/apps/compilerName-version/pkgName/version**
3. MPI-Compiler dependent packages: **/opt/apps/compilerName-version/mpiName-version/pkgName/version**

The modulefiles also need to be stored in a software hierarchy as well.

1. Compiler dependent modulefiles: **/opt/apps/modulefiles/Compilers/compilerName/compiler-version/pkgName/version**

2. MPI dependent modulefiles: **/opt/apps/modulefiles/MPI/compilerName/compiler-version/mpiName/mpi-version/pkgName/version**

The regression testing suite that comes with the Lmod source has many examples of a software hierarchy. See the directory **rt/hierarchy/mf/** from the Lmod source tree.

When **MODULEPATH** changes, Lmod unloads any modules which are not currently in the **MODULEPATH** and then tries to reload all the previously loaded modules. Any modules which are not available are marked as inactive. Those inactive modules become active if found with new **MODULEPATH** changes.

Note: In all of the examples above, we used only the first two version numbers. In other words, we used 4.9 instead of 4.9.2 and similarly 1.8 instead of 1.8.2. It is our view that for at least compilers and MPI stacks, the third digit is typically a bug fix and doesn't require rebuilding all the dependent packages. Y.M.M.V.

1.6.5 Configuring Lmod for your site

Sites can control the behavior of Lmod at configuration time. After Lmod is configured and installed, user can also modify the behavior through environment variables. To see all the configuration options you can execute:

```
$ ./configure --help
```

in the Lmod source directory. There are a few behavior options that do not have a configuration option.

There two kinds of variables: (1) An explicit values; (2) a yes/no variable. An example of first kind is *LMOD_SITE_NAME*. This variable controls the site name (e.g. TACC). This value of variable is used directly. There is no change of case or any other changes in that string.

The second kind of variable is an yes/no variable. One example of this is LMOD_IGNORE_CACHE. When this variable is "yes", Lmod ignores any cache files and walks MODULEPATH instead.

The following settings are considered "no". Note that the string value is lowercased first, so NO, No, and nO are the same as no. ALL OTHER VALUES are treated as "yes".

1. export LMOD_IGNORE_CACHE=""
2. export LMOD_IGNORE_CACHE=0
3. export LMOD_IGNORE_CACHE=no
4. export LMOD_IGNORE_CACHE=off

Environment variables only

The following variables set actions that can only be controlled by environment variables. The actions can not be controlled through the configuration step.

LMOD_ADMIN_FILE:
[path] If set this will be a file to specify the nag message. If this variable has no value, then Lmod looks for `<install_dir>/../etc/admin.list`

LMOD_AVAIL_STYLE:
Used by the avail hook to control how avail output is handled. This is a colon separated list of names. Note that the default choice is marked by angle brackets: A:B:<C> ==> C is the default. If no angle brackets are specified then the first entry is the default (i.e. A:B:C => A is default). See *Providing Custom Labels for Avail* for more details.

LMOD_IGNORE_CACHE:
[yes/no] If set to yes then Lmod will bypass all cachefile and walk the directories in MODULEPATH instead.

LMOD_PAGER:
[string] Lmod uses a pager when not using redirect. It defaults to less. Site/Users can turn off the pager if it is set to "None".

LMOD_RTM_TESTING:
[any value] If this variable has any value it means that Lmod does nothing. This is useful when testing a personal copy of Lmod and your site has the SHELL_STARTUP_DEBUG package installed so that the invoking of the module command in the system startup will a no-op.

LMOD_SYSTEM_NAME:
[string] This variable is used to where a site is using shared home files systems. See *Lmod on Shared Home File Systems* for more details.

LMOD_SYSTEM_DEFAULT_MODULES:
[string] This variable to define a list of colon separated standard modules when the **module reset** command is issued by or for the user. If a site has no default modules then they should set this module to **__NO_SYSTEM_DEFAULT_MODULES__**.

LMOD_TRACING:
[yes/no] If set to yes then Lmod will trace the loads/unloads while the module command is running.

LMOD_MODULERC:
Also **LMOD_MODULERCFILE** and **MODULERCFILE** can be used. They all are used the same way but **LMOD_MODULERC** is used before the other two vars. The contents of one of these environment variables is to be a single file or a colon separated list of files or directories to be used to specify the system MODULERC file(s).

If a directory is specified then all the files in that directory are assumed to be MODULERC files. See *Site and user control of defaults, aliases and hidden modules* for more details.

LMOD_QUARANTINE_VARS:
A colon separated list of environment variables that Lmod will not change. Note that only non-path like variable can be added to this list. Having variables like PATH and LD_LIBRARY_PATH in this list are ignored. In other words, they can be changed by Lmod. New in Version 8.6+.

Configuration time settings that can be overridden by env. vars.

The following settings are defined by configure but can be overridden by environment variables. The brackets show the following values [kind, default: value, configuration option] where kind is either yes/no, string, path, etc, value is what the default will be. Finally the configuration option which will set the action.

LMOD_ALLOW_TCL_MFILES:
[yes/no, default: yes, –with-tcl]. Allow tcl modulefiles. Note that .version and .modulerc files still use the tcl interpreter. So setting this to no means that your site will have to use either the "default" symlink or ".modulerc.lua" to specify defaults.

LMOD_ANCIENT_TIME:
[number, default:86400, –with-ancient]. The number of seconds that the user's personal cache is considered valid.

LMOD_AUTO_SWAP:
[yes/no, default: yes, –with-autoSwap] Allows Lmod to swap any modules that use the family function such as compilers and mpi stacks.

LMOD_AVAIL_EXTENSIONS:
[yes/no, default: yes, –with-availExtensions] Display package extensions when doing "module avail".

LMOD_BASH_INITIALIZE:
[yes/no, default:yes, –with-bashInitialize] If “yes” then Lmod will disable file globbing when eval’ing the output from Lmod.

LMOD_CASE_INDEPENDENT_SORTING:
[yes/no, default: no, –with-caseIndependentSorting] Make avail and spider use case independent sorting.

LMOD_COLORIZE:
[yes/no, default: yes, –with-colorize] Let lmod write colorize message to the terminal.

LMOD_DISABLE_NAME_AUTOSWAP:
[yes/no, default: no, –with-disableNameAutoSwap] Setting this to “yes” disables the one name rule autoswapping. In other words, “module load gcc/4.7 gcc/5.2 will fail when this is set.

LMOD_DUPLICATE_PATHS:
[yes/no, default: no, –with-duplicatePaths] Allow duplicates directories in path-like variables, PATH, LD_LIBRARY_PATH, … Note that if LMOD_TMOD_PATH_RULE is “yes” then LMOD_DUPLICATE_PATH is set to “no”.

LMOD_DYNAMIC_SPIDER_CACHE:
[yes/no, default: yes, –with-dynamicSpiderCache] Allow Lmod to re-evaluate modules that change $MODULEPATH that are already in the system spider cache. This will allow for user modulefiles that are in a matching software hierarchy. There is a small speed cost to support this feature. Sites that do not use this feature may wish to disable this.

LMOD_EXTENDED_DEFAULT:
[yes/no, default: yes, –with-extendedDefault] Allow users to specify a partial match of a version. So abc/17 will try to match the “best” abc/17.*.*

LMOD_EXACT_MATCH:
[yes/no, default: no, –with-exactMatch] Requires Lmod to use fullNames for modules. This disables defaults.

LMOD_HIDDEN_ITALIC:
[yes/no, default: no, –with-hiddenItalic] Use italics for hidden modules instead of faint.

LMOD_MPATH_AVAIL:
[yes/no, default: no, –with-mpathSearch] If this is set then module avail <string> will search modulepath names.

LMOD_OVERRIDE_LANG:
[string, default: en, –with-lang] Override $LANG for Lmod error/messages/warnings.

LMOD_PIN_VERSIONS:
[yes/no, default: no, –with-pinVersions] If yes then when restoring load the same version that was chosen with the save, instead of the current default version.

LMOD_PREPEND_BLOCK:
[normal/reverse, default: normal, –with-prependBlock] Treat multiple directories passed to prepend in normal order and not reversed.

LMOD_REDIRECT:
[yes/no, default: no, –with-redirect]. Normal messages generated by “module avail”, “module list”,etc write the output to stderr. Turning redirect to “yes” will cause these messages to be written to stdout. Note this only works for bash and zsh. This will not work with csh or tcsh as there is a problem with these shells and not Lmod.

LMOD_SHORT_TIME:
[number, default: 2, –with-shortTime]. If the time to build the spider cache takes longer than this number then write the spider cache out into the user’s account. If you want to prevent the spider cache file being written to the user’s account then set this number to be large, like 86400.

LMOD_SITE_MSG_FILE:
[full path, default: <nil> –with-siteMsgFile] The Site message file. This overrides the messageDir/en.lua file so

that sites can replace some or all Lmod messages.

LMOD_SITE_NAME:
[string, default: <nil>, –with-siteName]. This is the site name, for example TACC, and not the name of the cluster. This is used with the family function.

LMOD_SYSHOST:
[string, default: <nil>, –with-syshost]. This variable can be used to help with module tracking. See *Tracking Module Usage* for details.

LMOD_TMOD_FIND_FIRST:
[yes/no, default: no, –with-tmodFindFirst]. Normally Lmod uses the FIND BEST rule to search for defaults when searching C/N/V or N/V module layouts. A site can force FIND_FIRST for C/N/V or N/V module layouts to match the FIND_FIRST rule for N/V/V module layout. See *N/V/V: Picking modules when there are multiple directories in MODULEPATH* for more details.

LMOD_TMOD_PATH_RULE:
[yes/no, default: no, –with-tmodPathRule]. Normally Lmod prepend/appends a directory in the beginning/end of the path like variable. If this is true then if path entry is already there then do not prepend/append. Note that if LMOD_TMOD_PATH_RULE is "yes" then LMOD_DUPLICATE_PATH is set to "no".

LMOD_USE_DOT_CONFIG_ONLY:
[yes/no, default: no, –with-useDotConfigDirOnly]. Lmod is transitioning from using ~/.lmod.d/ to ~/.config/lmod to store collections. By default, Lmod writes to both directories and read the most recent collection from either directory. By changing this configuration option to yes, then Lmod will write only to ~/.config/lmod but will check both directories when reading picking the most recent one.

Configuration only settings

–with-silentShellDebugging:
[yes/no, default: no] If yes then the module command will silence its output under shell debug.

–with-mode=MODE:
By default Lmod uses your umask to control the permission unless your user id (i.e. id -u) is less than 500. If your id is less than 500 then it uses a umask of 022. This translates to use permission of 755 for executables and 644 for all other files. Sites may wish to install with different permission. For example 750, or 700. Remember to set the execute bit. It will be removed for non-executable files.

Configuring Lmod with /etc/lmod/lmod_config.lua:

Lmod looks for a file named lmod_config.lua in the LMOD_CONFIG_DIR, which is by default /etc/lmod/. So normally the file is found here: /etc/lmod/lmod_config.lua. It can be configured to any value with the configure option (–with-lmodConfigDir=) or setting the environment variable LMOD_CONFIG_DIR. This file is used optionally. It is not required.

This file allows sites configure Lmod through Lua instead of setting environment variables for each shell. By using the cosmic:assign() functions this can be accomplished in one file. Here is a full example:

```
require("strict")
local cosmic       = require("Cosmic"):singleton()

cosmic:assign("LMOD_SITE_NAME",    "XYZZY")

-- Note that this directory could be anything including /etc/lmod
cosmic:assign("LMOD_PACKAGE_PATH", "/path/to/SitePackage_Dir/")
```

(continues on next page)

(continued from previous page)

```
local function echoString(s)
   io.stderr:write(s,"\n")
end

sandbox_registration {
   echoString = echoString
}
```

In the above example a site is setting its name and providing the path to the location directory where the SitePackage.lua file is. Also the simple **echoString** function has been added and is callable from modulefiles because it has been registered in the sandbox.

Sites wishing to change the default values of other Lmod configuration variables should study the src/myGlobals.lua file to see what the name of the variable is and then use the cosmic:assign() function to set the new value. For example:

```
cosmic:assign("LMOD_PIN_VERSIONS","yes")
cosmic:assign("LMOD_CACHED_LOADS","yes")
...
```

To check that your installation is correct please run:

```
$ module --config
```

to see that you got what you wanted.

1.6.6 Site and user control of defaults, aliases and hidden modules

Lmod uses a **.modulerc.lua** or **.modulerc** file to control the default version and other things as describe in *Marking a Version as Default*. Lmod also allows sites to control the defaults, aliases and hidden modules from a system location. Also users can override these defaults and add new aliases and hidden modules via a personal **~/.modulerc.lua** or **~/.modulerc** file. Note that lua files always take priority over non-lua files.

A default can be specified in three ways:

1. **default**, **.version**, **.modulerc.lua** or **.modulerc** file in the module tree as describe earlier.
2. One or more system MODULERC files: $LMOD_MODULERCFILE or $MODULERC or <INSTALL_DIR>/../etc/rc.lua
3. A **~/.modulerc.lua** or **~/.modulerc**

The highest priority for defaults are the user MODULERC files, followed by the system MODULERC file(s) and the lowest priority are the files in the module tree. In other words a user or system MODULERC file can override the default modules.

All lua modulerc files support the following commands:

module_version ("known_version","default"):
This command marks a known version to be the default. If there are duplicates, the last one applies.

module_version ("known_version","alias"):
The known version can be also known as the alias. For example: module_version("ab/7.4", "7") means that the ab/7.4 and ab/7 names the same modulefile.

module_alias (“alias”, “known_version”):
An alias can be used as a global alias. For example: module_alias(“z13”, “z/13.0.1”) says that “module load z13” will load the “z/13.0.1” modulefile.

hide_version (“fullName”):
This command will hide all “fullName” modules. So if there are multiple phdf5/1.8.6 module, then all will be marked as hidden. See the isVisible hook in *SitePackage.lua and hooks* to do more complicated hiding.

hide_modulefile (“full_path”):
This command will hide just one module located at “full_path” modules. This way only modulefile can be hidden. See the isVisible hook in *SitePackage.lua and hooks* to do more complicated hiding.

The above functions are the only functions support in .modulerc.lua files. In particular, Lmod functions like setenv(), pushenv() are not supported.

The TCL files support similar commands:

module-version known_version default
see module_version() above.

module-version known_version alias
see module_version() above.

module-alias alias known_version
see module_alias() above.

hide-version fullName
see hide_version above.

hide-modulefile full_path
see hide-modulefile above.

The above TCL commands are the only commands support in .modulerc or .version files. In particular, TCL commands like setenv(), pushenv() are not supported.

System MODULERC files

By default, Lmod looks in /path_to_lmod/lmod/../etc/rc.lua or /path_to_lmod/lmod/../etc/rc to find a system MODULERC file. The lua file takes precedence over the TCL version. Or you can set either **LMOD_MODULERCFILE** or **MODULERCFILE** to be a single file or a colon separated list. If more than one file is specified then the priority is left to right.

1.6.7 How does Lmod convert TCL modulefile into Lua

Lmod uses a TCL program called **tcl2lua.tcl** to read TCL modulefiles and convert them to lua. The whole TCL modulefile evaluated by the TCL interpreter. This is *NOT* a source to source translation.

The purpose of **tcl2lua.tcl** is to evaluate the regular TCL command but replace “module functions”, such as **prepend-path** or **setenv**, and converted to Lua functions. For example, suppose you have the following simple TCL modulefile for git:

```
#%Module
set appDir          $env(APP_DIR)
set version         2.0.3

prepend-path        PATH "$appDir/git/$version/bin"
```

Assuming that the environment variable APP_DIR is */apps* then the entire output of the **tcl2lua.tcl** program would be:

```
prepend_path("PATH", "/apps/git/2.0.3/bin")
```

Note that all the normal TCL code has been evaluated and the TCL **prepend-path** command has been converted to a lua **prepend_path** function call.

Normally this works fine. However, because Lmod does evaluate the actions of a TCL module file as a two-step process, it can cause problems. In particular, suppose you have two TCL modulefiles:

Centos:

```
#%Module
setenv SYSTEM_NAME Centos
```

And B:

```
#%Module
module load Centos

if { $env(SYSTEM_NAME) == "Centos" } {
   # do something
}
```

When Lmod tries to translate the B modulefile into lua it fails:

```
load("Centos")
LmodError([[/opt/mfiles/B: (???): can't read "env(SYSTEM_NAME)": no such variable]])
```

This is because unlike the TCL/C Module system, the **module load Centos** command is converted to a function call, but it won't load the module in time for the test to be evaluated properly.

The only solution is convert the B modulefile into a Lua modulefile (B.lua):

```
load("Centos")
if (os.getenv("SYSTEM_NAME") == "Centos") then
  -- Do something
end
```

The Centos modulefile does not have to be translated in order for this to work, just the B modulefile.

As a side note, you are free to put Lua modules in the same tree that the TCL/C Module system uses. It will ignore files that the first line is not #%Module and Lmod will pick B.lua over B.

1.6.8 Dependent Modules

Let's assume that module "X" depends on module "A". There are several ways to handle module dependency. Inside the "X" modulefile you could have one of the following choices:

1. Use `depends_on("A")`
2. Use `prereq("A")`
3. Use `load("A")`
4. Use `always_load("A")`
5. Use RPATH to make "X" know where the libraries in "A" can be found.

Let's examine these choices in order. The main issue for each of these choices is what happens when module "X" is unloaded.

depends_on("A")

This choice loads module "A" on the users behalf if it not already loaded. When module "X" is unloaded, module "A" will be unloaded if it is a dependent load. Imagine the following scenario with `depends_on("A")`:

```
$ module purge; module load X; module unload X                => unload A
$ module purge; module load A; module load X; module unload X => keep A
```

Lmod implements reference counting for modules loaded via `depends_on()` and only `depends_on()`. So if "X" and "Y" depend on "A" then:

```
$ module purge; module load X Y; module unload X   => keep A
$ module purge; module load X Y; module unload X Y => unload A
```

Complex uses of depends_on()

Sites can have complex dependencies that they might wish to express using `depends_on()`. Let's assume that module X depends on the openblas package but only when using gcc. A site might try to do the following in the X modulefile:

```
-- DO NOT DO THIS!!
if (isloaded("gcc")) then
   depends_on("openblas")
end
-- DO NOT DO THIS!!
```

Let's also assume that your site is using the hierarchy where X is a compiler dependent module and you wish to use the same modulefile for X for both the gcc and intel compiler modules. The above code in the X modulefile works correctly when loading but will fail when unloading in this common scenario: if a user tries to swap gcc for intel then the openblas module will likely be left loaded or inactive.

To simplify the discussion, let's have the user start with the following modules loaded:

```
$ module list

1) gcc/7.1  2) X/1.0  3) openblas/0.2.20
```

And lets assume that openblas is a Core module and X is a compiler dependent module. Then executing:

```
$ module swap gcc intel
```

causes gcc to be unloaded. When gcc is unloaded it removes the path in MODULEPATH to the gcc dependent modules which means that X will be unloaded and marked as inactive. The way that a module is unloaded is that the contents of the module is evaluated and most action requested are reversed. So load statements cause a module to unload and a depends_on() function is told to forgo() the modules. The isloaded() is not reversed. But as you can see since the gcc modulefile is not loaded the if statement then clause is not evaluated. This means that openblas will still be loaded.

In the case where openblas is a compiler-dependent module then it will be unloaded and marked as inactive. Either way this probably not what the site wants to happen. The trouble here is that environment that happens on load is not the case on unload.

There is another way to determine which compiler and/or mpi stack a module is in and that is its filename. This assumes that you have a rational naming convention for module locations. Using a similar technique to the one describe in *Introspection*. We can determine which compiler is in use. So if the module file is located in */apps/mfiles/Compiler/<compiler>/<compiler-version>/<app-name>/<app-version>* then we can do the following and use the hierarchyA() function in the X modulefile:

```
local hierA = hierarchyA(myModuleFullName(),1)
if (hierA[1]:find("^gcc/")) then
   depends_on("openblas")
end
```

This will work correctly for both loading and unloading. This, of course, assumes that the location of the X modulefile is something like:

```
/apps/mfiles/Compiler/gcc/7.1/X/1.0.lua
```

prereq("A")

This choice is the one you give for sophisticated users. If a user tried to load module "X" without previously loading "A" then the user will get a message telling the user that they must load "A" before loading "X". This way the dependency is explicitly handled by the user. When the user unloads "X", module "A" will remain loaded.

load("A")

This choice will always load module "A" on the users behalf. This is true even if "A" is already loaded. When module "X" is unloaded, module "A" will be unloaded as well. This may surprise some users who might want to continue using the "A" package. At least with `prereq()`, your users won't be surprised by this. Another way to handle this is the next choice.

always_load("A")

A site can chose to use `always_load()` instead. This command is a shorthand for:

```
if (mode() == "load") then
   load("A")
end
```

The TCL equivalent is:

```
if { [ module-info mode load ] } {
   module load A
}
```

This choice will always load module "A" on the users behalf. This is true even if "A" is already loaded. When module "X" is unloaded, module "A" will remain loaded.

Use RPATH

We have switched to using RPATH for library dependencies at TACC. That is when we build package X, we use the RPATH linking option to link libraries in package A as part of the X rpm. This has the disadvantage that if the A package is removed then the X package is broken. This has happened to us occasionally. In general, however, we have found that this has worked well for us.

1.6.9 Modulefile Examples from simple to complex

Most of the time a modulefile is just a collection of setting environment variables and prepending to PATH or other path like variables. However, modulefiles are actually programs so you can do a great deal if necessary.

Here we show some of the techniques that site's or user's might use in their modulefiles. In addition, there are many examples in the Lmod source tree. The `rt` directory contains the regression testing suite. Each subdirectory in `rt` is a separate test and below that there will be many modulefiles. For example see all the modulefile associated with the load test can be found in `rt/load/mf`.

Some simple modulefiles

An application modulefile might add to `$PATH`, set a few other environment variables and provide help message as well as `whatis()` strings. For example the valgrind memory usage tester might look like:

```
help([[
To use the valgrind utility on an executable called a.out:

valgrind ./a.out
]])

local version = "3.7.0"
local base    = pathJoin("/apps/valgrind",version)
prepend_path("PATH", pathJoin(base,"bin"))  -- /app/valgrind/3.7.0/bin
setenv(      "SITE_VALGRIND_DIR", base)
setenv(      "SITE_VALGRIND_INC", pathJoin(base,"include"))
setenv(      "SITE_VALGRIND_LIB", pathJoin(base,"lib"))

whatis("Name: ".. pkgName)
whatis("Version: " .. fullVersion)
whatis("Category: tools")
whatis("URL: http://www.valgrind.org")
whatis("Description: memory usage tester")
```

A library module might look like:

```
help([[
...
]])
whatis("Name: boost")
whatis("Version: 1.64")
whatis("Category: Lmod/Modulefiles")
whatis("Keywords: System, Library, C++")
whatis("URL: http://www.boost.org")
whatis("Description: Boost provides free peer-reviewed portable C++ source libraries.")

setenv("SITE_BOOST_DIR","/apps/intel17/boost/1.64")
setenv("SITE_BOOST_LIB","/apps/intel17/boost/1.64/lib")
setenv("SITE_BOOST_INC","/apps/intel17/boost/1.64/include")
setenv("SITE_BOOST_BIN","/apps/intel17/boost/1.64/bin")
setenv("BOOST_ROOT","/apps/intel17/boost/1.64")
prepend_path("LD_LIBRARY_PATH","/apps/intel17/boost/1.64/lib")
prepend_path("PATH","/apps/intel17/boost/1.64/bin")
```

Introspection

Lmod provides inspection functions that describe the name and version of a modulefile as well as the path to the modulefile. These functions provide a way to write "generic" modulefiles, i.e. modulefiles that can fill in its values based on the location of the file itself.

These ideas work best in the software hierarchy style of modulefiles. For example: suppose the following is a modulefile for Git. Its modulefile is located in the "/apps/mfiles/Core/git" directory and software is installed in "/apps/git/<version>". The following modulefile would work for every version of git:

```
local pkg = pathJoin("/apps",myModuleName(),myModuleVersion())
local bin = pathJoin(pkg,"bin"))
prepend_path("PATH",bin)

whatis("Name:         ", myModuleName())
whatis("Version:      ", myModuleVersion())
whatis("Description: ", "Git is a fast distributive version control system")
```

The contents of this modulefile can be used for multiple versions of the git software, because the local variable bin changes the location of the bin directory to match the version of the used as the name of the file. So if the module file is in */apps/mfiles/Core/git/2.3.4.lua* then the local variable *bin* will be */apps/git/2.3.4*.

Relative Paths

Suppose you are interested in modules where the module and application location are relative. Suppose that you have an $APPS directory, and below that you have modulefiles and packages, and you would like the modulefiles to find the absolute path of the package location. This can be done with the `myFileName()` function and some lua code:

```
local fn      = myFileName()                          -- 1
local full    = myModuleFullName()                    -- 2
local loc     = fn:find(full,1,true)-2                -- 3
local mdir    = fn:sub(1,loc)                         -- 4
local appsDir = mdir:gsub("(.*)/","%1")               -- 5
local pkg     = pathJoin(appsDir, full)               -- 6
```

To make this example concrete, let's assume that applications are in `/home/user/apps` and the modulefiles are in `/home/user/apps/mfiles`. So if the modulefile is located at `/home/user/apps/mfiles/git/1.2.lua`, then that is the value of `fn` at line 1. The `full` variable at line 2 will have `git/1.2`. What we want is to remove the name of the modulefile and find its parent directory. So we use Lua string member function on `fn` to find where `full` starts. In most cases `fn:find(full)` would work to find where the "git" starts in `fn` The trouble is that the Lua find function is expecting a regular expression and in particular `.` and `-` are regular expression characters. So here we are using `fn:find(full,1,true)` to tell Lua to treat each character as is with no special meaning.

Line 3 also subtracts 2. The find command reports the location of the start of the string where the "g" in "git" is, We want the value of `mdir` to be `/home/user/apps/mfiles` so we need to subtract 2. This makes `mdir` have the right value. One note is that Lua is a one based language, so locations in strings start at one.

It was important for the value of `mdir` to remove the trailing `/` so that line 5 will do its magic. We want the parent directory of `mdir`, so the regular expressions says greedily grab every character until the trailing `/` and the `%1` says to capture the string found in and use that to set `appsdir` to `/home/user/apps`. Finally we wish to set `pkg` to the location of the actual application so we combine the value of `appsdir` and `full` to set `pkg` to `/home/user/apps/git/1.2`.

The nice thing about this Lua code is that it figures out the location of the package no matter where it is, as long as the relation between apps directories and modulefiles is consistent.

Creating modules like this can be complicated. See *Debugging Modulefiles* for helpful tips.

Generic Modules with the Hierarchy

This works great for Core modules. It is a little more complicated for Compiler or MPI/Compiler dependent modules but quite useful. For a concrete example, lets cover how to handle the boost C++ library. This is obviously a compiler dependent module. Suppose you have the gnu compiler collection (gcc) and the intel compiler collection (intel), which means that you'll have a gcc version and an intel version for each version of booth.

In order to have generic modules for compiler dependent modules, there must be some conventions to make this work. A suggested way to do this is the following:

1. Core modules are placed in */apps/mfiles/Core*. These are the compilers, programs like git and so on.
2. Core software goes in */apps/<app-name>/<app-version>*. So git version 2.3.4 goes in */apps/git/2.3.4*
3. Compiler-dependent modulefiles go in */apps/mfiles/Compiler/<compiler>/<compiler-version>/<app-name>/<app-version>* using the **two-digit** rule (discussed below). So the Boost 1.55.0 modulefile built with gcc/4.8.3 would be found in */apps/mfiles/Compiler/gcc/4.8/boost/1.55.0.lua*
4. Compiler-dependent packages go in */apps/<compiler-version>/<app-name>/<app-version>*. So the same Boost 1.55.0 package built with gcc 4.8.3 would be placed in */apps/gcc-4_8/boost/1.55.0*

The above convention depends on the **two-digit** rule. For compilers and mpi stack, we are making the assumption that compiler dependent libraries built with gcc 4.8.1 can be used with gcc 4.8.3. This is not always safe but it works well enough in practice. The above convention also assumes that the boost 1.55.0 package will be placed in */apps/gcc-4_8/boost/1.55.0*. It couldn't go in */apps/gcc/4.8/. . .* because that is where the gcc 4.8 package would be placed and it is not a good idea to co-mingle two different packages in the same tree. Another possible choice would be */apps/gcc-4.8/boost/1.55.0*. It is my view that it looks too much like the gcc version 4.8 package location where as *gcc-4_8* doesn't.

With all of the above assumptions, we can now create a generic module file for compiler dependent modules such as Boost. In order to make this work, we will need to use the *hierarchyA* function. This function parses the path of the modulefile to return the pieces we need to create a generic boost modulefile:

```
hierA = hierarchyA(myModuleFullName(),1)
```

The *myModuleFullName()* function returns the full name of the module. So if the module is named **boost/1.55.0**, then that is what it will return. If your site uses module names like *lib/boost/1.55.0* then it will return that correctly as well. The *1* tells Lmod to return just one component from the path. So if the modulefile is located at */apps/mfiles/Compiler/gcc/4.8/boost/1.55.0.lua*, then *myModuleFullName()* returns **boost/1.55.0** and the *hierarchyA* function returns an array with 1 entry. In this case it returns:

```
{ "gcc/4.8" }
```

The rest of the module file then can make use to this result to form the paths:

```
local pkgName     = myModuleName()
local fullVersion = myModuleVersion()
local hierA       = hierarchyA(myModuleFullName(),1)
local compilerD   = hierA[1]:gsub("/","-"):gsub("%.","_")
local base        = pathJoin("/apps",compilerD,pkgName,fullVersion)

whatis("Name: "..pkgName)
whatis("Version "..fullVersion)
whatis("Category: library")
whatis("Description: Boost provides free peer-reviewed "..
                    " portable C++ source libraries.")
whatis("URL: http://www.boost.org")
```

(continues on next page)

(continued from previous page)

```
whatis("Keyword: library, c++")

setenv("TACC_BOOST_LIB", pathJoin(base,"lib"))
setenv("TACC_BOOST_INC", pathJoin(base,"include"))
```

The important trick is the building of the *compilerD* variable. It converts the *gcc/4.8* into *gcc-4_8*. This makes the *base* variable be: */apps/gcc-4_8/boost/1.55.0*.

Creating modules like this can be complicated. See *Debugging Modulefiles* for helpful tips.

A proposed directory structure of /apps/mfiles/Compiler would be:

```
.base/    gcc/  intel/

.base/
boost/generic.lua

gcc/4.8/boost/

1.55.0.lua ->  ../../../.base/boost/generic.lua

intel/15.0.2/boost/

1.55.0.lua -> ../../../.base/boost/generic.lua
```

In this way the *.base/boost/generic.lua* file will be the source file for all the boost version build with gcc and intel compilers.

The same technique can be applied for modulefiles for Compiler/MPI dependent packages. In this case, we will create the phdf5 modulefile. This is a parallel I/O package that allows for Hierarchical output. The modulefile is:

```
local pkgName    = myModuleName()
local pkgVersion = myModuleVersion()
local pkgNameVer = myModuleFullName()

local hierA      = hierarchyA(pkgNameVer,2)
local mpiD       = hierA[1]:gsub("/","-"):gsub("%.","_")
local compilerD  = hierA[2]:gsub("/","-"):gsub("%.","_")
local base       = pathJoin("/apps", compilerD, mpiD, pkgNameVer)

setenv(      "TACC_HDF5_DIR",   base)
setenv(      "TACC_HDF5_DOC",   pathJoin(base,"doc"))
setenv(      "TACC_HDF5_INC",   pathJoin(base,"include"))
setenv(      "TACC_HDF5_LIB",   pathJoin(base,"lib"))
setenv(      "TACC_HDF5_BIN",   pathJoin(base,"bin"))
prepend_path("PATH",            pathJoin(base,"bin"))
prepend_path("LD_LIBRARY_PATH", pathJoin(base,"lib"))

whatis("Name: Parallel HDF5")
whatis("Version: " .. pkgVersion)
whatis("Category: library, mathematics")
whatis("URL: http://www.hdfgroup.org/HDF5")
whatis("Description: General purpose library and file format for storing scientific data␣
↪(parallel I/O version)")
```

We use the same tricks as before, It is just that since the module for phdf5 built by gcc/4.8.3 and mpich/3.1.2 will be found at */apps/mfiles/MPI/gcc/4.8./mpich/3.1/phdf5/1.8.14.lua*. The results of *hierarchyA(pkgNameVer,2)* would be:

```
{ "mpich/3.1", "gcc/4.8" }
```

This is because the *hierarchyA* works back up the path two elements at a time because the full name of this package is also two elements (phdf5/1.8.14). The *base* variable now becomes:

```
/apps/gcc-4_8/mpich-3_1/phdf5/1.8.14
```

The last type of modulefile that needs to be discussed is an mpi stack modulefile such as mpich/3.1.2. This modulefile is more complicated because it has to implement the two-digit rule, build the path to the package and build the new entry to the **MODULEPATH**. The modulefile is:

```
local pkgNameVer   = myModuleFullName()
local pkgName      = myModuleName()
local fullVersion  = myModuleVersion()
local pkgV         = fullVersion:match('(%d+%.%d+)%.?')

local hierA        = hierarchyA(pkgNameVer,1)
local compilerV    = hierA[1]
local compilerD    = compilerV:gsub("/","-"):gsub("%.","_")
local base         = pathJoin("/apps",compilerD,pkgName,fullVersion)
local mpath        = pathJoin("/apps/mfiles/MPI", compilerV, pkgName, pkgV)

prepend_path("MODULEPATH", mpath)
setenv(      "TACC_MPICH_DIR", base)
setenv(      "TACC_MPICH_LIB", pathJoin(base,"lib"))
setenv(      "TACC_MPICH_BIN", pathJoin(base,"bin"))
setenv(      "TACC_MPICH_INC", pathJoin(base,"include"))

whatis("Name: "..pkgName)
whatis("Version "..fullVersion)
whatis("Category: mpi")
whatis("Description: High-Performance Portable MPI")
whatis("URL: http://www.mpich.org")
```

The **Two Digit** rule implemented by forming the *pkgV* variable. The *base* and *mpath* are:

```
base  = "/apps/gcc-4_8/mpich-3_1/phdf5/1.8.14"
mpath = "/apps/mfiles/MPI/gcc/4.8/mpich/3.1"
```

The *rt* directory contains all the regression test used by Lmod. As such they contain many examples of modulefiles. To complement this description, the *rt/hierarchy/mf* directory from the source tree contains a complete hierarchy.

1.6.10 The Interaction between Modules, MPI and Parallel Filesystems

Many sites that use modules have hundred to thousands of nodes connected to large parallel filesystems. Some care has to be taken when a user's job scripts start a parallel mpi execution to avoid problems with the parallel filesystem. Module commands in a user's ~/.bashrc or ~/.cshrc can overload or cause timeouts in a parallel filesystem.

At TACC, we do a couple of things to avoid this problem. We have seen this timeout problem for mpi programs that execute more than 1000 nodes, but when this problem occurs will depend on relative speeds of the network and the parallel filesystem.

It is helpful to outline the proceedure that we use at TACC to start a job on a compute node for a user

1. The bash user submits a job to the scheduler.
2. The current environment is captured
3. That environment passed to the shell script that starts on a compute node.
4. The user's job script starts an mpi execution.
5. The mpi execution starts an interactive non-login (and not prompt) shell on every node.
6. This non-login interactive shell sources the user's ~/.bashrc
7. Then the environment found at the start of mpi execution is passed to all nodes.

It is the sixth step that causes the problem. If there are any module load commands in ~/.bashrc, these module commands will be started on every node at about the same time. The module command has to walk the directories listed in $MODULEPATH to find the modules. This hits the parallel filesystem hard when there are thousands of nodes.

At TACC we define an environment variable call `ENVIRONMENT` (a clever name, no?) to be `BATCH` when a job is started on a compute node. We provide every user with a default ~/.bashrc that has the following section:

```
if [ -z "$__BASHRC_SOURCED__" -a "$ENVIRONMENT" != BATCH ]; then
  export __BASHRC_SOURCED__=1

  ##########################################################
  # **** PLACE MODULE COMMANDS HERE and ONLY HERE.      ****
  ##########################################################

  # module load git

fi
```

This way modules are only loaded once on the initial shell and not reloaded on subshells or on compute nodes during job submission.

Some users won't follow our guidelines so as an extra layer of protection we make the module be a no-op for bash users.

The module command for bash is defined to be:

```
module () {
   eval $( $LMOD_CMD bash "$@" ) ...
}
```

Normally, `$LMOD_CMD` points the lmod command but on compute nodes we have the following startup behavior for bash users:

```
# Compute notes set the ENVIRONMENT var to BATCH for non-interactive shells.
if [ -z "$PS1" ]; then
```

(continues on next page)

(continued from previous page)

```
  export ENVIRONMENT=BATCH

  # If we are in BATCH mode then turn off the module command.
  if [ -z "$TACC_DEBUG" ]; then
    export LMOD_CMD=':'
    export LMOD_SETTARG_CMD=':'
  fi
fi
```

By making `$LMOD_CMD` be a colon, we make the module command silently do nothing. Unfortunately, this only works for bash and not tcsh or zsh users. We want module command to work in a user's job submission script:

```
#!/bin/bash
#SBATCH ...

module load intel mvapich2
ibrun ./my_mpi_program        # start parallel executation.
```

Note that the total environment is passed by our parallel job starter ibrun. So there is no need for a user's ~/bashrc or similar file to load modules. In fact it can lead to problems if the user's current module environment loads modules that don't match the environment that ibrun pushes to the compute nodes.

Tcsh and Zsh source the startup scripts sourced in /etc/profile.d/ for shell script startup. This is how the module command is defined there. If we redefined `$LMOD_CMD` for those shell, module command would not work in tcsh or zsh scripts.

Bash uses a different technique. It only uses the value of $BASH_ENV. If that variable points to a file then that file is source. Lmod defines that variable to point to a file that defines the module command. So re-defining `$LMOD_CMD` in the startup scripts won't affect the module commands in the job submission script, just in the parallel execution when ~/.bashrc is sourced.

One drawback to redefining `$LMOD_CMD` is that if bash user tries to invoke a shell script to run in parallel any module command will silently do nothing.

1.6.11 Lmod on Shared Home File Systems

Many sites have a single Operating System and one set of modules across their cluster. If a site has more than one cluster, they may chose to have a separate home directory for each cluster. Some sites may wish to have multiple clusters share a single home directory. While this strategy has some advantages, it complicates things for your users and adminstrators. If your site has a single home directory sharing between two or more clusters, you have a shared home file system.

As a further complication, your site may or may not have a shared home file system even if you have two or more clusters. If you have separate login nodes for each cluster then you do have a shared home file system. If you have a single login which can submit jobs to different clusters then you do not have a shared home file system.

The way to think about this is each cluster is going to have at least some modules which are different. Module collections need to be unique to each cluster. The trick described below will make them unique for each cluster.

Sites that use a shared home file system across multiple clusters should take some extra steps to ensure the smooth running of Lmod. Typically each cluster will use different modules.

There are three steps that will make Lmod run smoothly on a shared home filesystem:

1. It is best to have a separate installation of Lmod on each cluster.
2. Define the environment variable "LMOD_SYSTEM_NAME" uniquely for each cluster.

3. If you build a system spider cache, then build a separate cache for each cluster.

A separate installation on each cluster is the safest way to install Lmod. It is possible to have a single installation but since there is some C code build with Lmod, this has to work on all clusters. Also the location of the Lua interpreter must be exactly the same on each cluster.

It is also recommended that you set "LMOD_SYSTEM_NAME" outside of a modulefile. It would be bad if a module purge would clear that value. When you set this variable, it makes the module collections and user spider caches unique for a given cluster.

A separate system spider cache is really the only way to go. Otherwise a "module spider" will report modules that don't exist on the current cluster. If you have a separate install of Lmod on each cluster then you can specify the location of system cache at configure time. If you don't, you can use the "LMOD_RC" environment to specify the location of the lmodrc.lua file uniquely on each cluster.

Lmod knows about the system spider cache from the lmodrc.lua file. If you install separate instances of Lmod on each cluster, Lmod builds the scDescriptT table for you. Otherwise you can modify lmodrc.lua to point to the system cache by adding scDescriptT to the end of the file:

```
scDescriptT = {
  {
    ["dir"] = "<location of your system cache directory>,
    ["timestamp"] = "<location of your timestamp file",
  },
}
```

where you have filled in the location of both the system cache directory and timestamp file.

1.6.12 User Spider Cache

In *System Spider Cache*, we described how to build a system cache. If there is no system cache available then Lmod can produce a user based spider cache. It gets written to **~/.cache/lmod**. *See below for earlier versions of Lmod.* It is designed to provide improved speed of performing doing module avail or module spider. But it is not without its problems. The first point is that if Lmod thinks any spider cache is valid, it uses it for the MODULEPATH directories it covers then it uses it instead of walking the tree.

Personal Cache rules:

1. If it can't find a valid cache then Lmod walks the tree to find all available modules and builds the spider cache in memory.
2. If the time it takes to build the cache is longer than the contents of env var. LMOD_SHORT_TIME (default 2 seconds) then Lmod writes the cache file into the ~/.cache/lmod directory.
3. A user's cache is assumed to be valid for the contents of LMOD_ANCIENT_TIME (default 86400 seconds or 24 hours) based on the date associated with the cache file.

Sites can change these defaults at configure time. Users can set the environment variables to change their personal setting.

To turn off the generation of a user cache, one can set LMOD_SHORT_TIME to some big number of seconds. For example:

```
export LMOD_SHORT_TIME=86400
```

would say that Lmod only write the user cache file if it took longer than 1 day (=86400 seconds). A second way do to this is do make the user cache directory unwritable:

```
$ rm -rf    ~/.cache/lmod
$ chmod 500 ~/.cache/lmod
```

NOTE: Lmod versions earlier than 8.7.12 wrote the cache file to **~/.lmod.d/.cache**

1.6.13 System Spider Cache

It is now very important that sites with large modulefile installations build system spider cache files. There is a shell script called "update_lmod_system_cache_files" that builds a system cache file. It also touches a file called "system.txt". Whatever the name of this file is, Lmod uses this file to know that the spider cache is up-to-date.

Lmod uses the spider cache file as a replacement for walking the directory tree to find all modulefiles in your `MODULEPATH`. This means that Lmod only knows about system modules that are found in the spider cache. Lmod won't know about any system modules that are not in this cache. (Personal module files are always found). It turns out that reading a single file is much faster than walking the directory tree.

The spider cache is used to speed up `module avail` and `module spider` and not `module load`. All the spider cache file(s) provide is a way for Lmod to know what modules exist and any properties that a modulefile might have. It does not save the contents of any modulefiles. Lmod always reads and evaluate the actual modulefile when performing loads, shows and similar commands.

The reason that Lmod does not use the cache with `module load` is that if the spider cache is out-of-date, then Lmod will not be able to load a module. Either Lmod uses the spider cache or it walks the directories in `MODULEPATH`.

A site may choose to use have the spider cache assist the `module load` command by configuring Lmod or setting the environment variable:

```
export LMOD_CACHED_LOADS=yes
```

See *Configuring Lmod for your site* for more details. Just remember that the cache file has to be up-to-date or user's won't be able to find system modulefiles! Note too, that a cache file is tied to a particular set of directories in the MODULEPATH. Lmod knows which directories in `MODULEPATH` are covered by spider cache file(s) and which are not. So having a system spider cache file and setting LMOD_CACHED_LOADS=yes will not hamper modulefiles created by users in personal directories.

While building the spider cache, each modulefile is evaluated for changes to `MODULEPATH`. Any directories added to `MODULEPATH` are also walked. This means if your site uses the software hierarchy then the new directories added by compiler or mpi stack modulefiles will also be searched.

Sites running Lmod have three choices:

1. Do not create a spider cache for system modules. This will work fine as long as the number of modules is not too large. You will know when it is time to start building a cache file when you start getting complains how long it takes to do any module commands.
2. If you have a formal procedure for installing packages on your system, then I recommend you to do the following. Have the install procedure run the update_lmod_system_cache_files script. This will create a file called "system.txt", which marks the time that the system was last updated, so that Lmod knows that the cache is still good.
3. Or you can run the update_lmod_system_cache_files script say every 30 minutes. This way the cache file is up-to-date. No new module will be unknown for more than 30 minutes.

There are two ways to specify how cache directories and timestamp files are specified. You can use "–with-spiderCacheDir=dirs" and "–with-updateSystemFn=file" to specify one or more directories with a single timestamp file:

```
./configure --with-spiderCacheDir=/opt/mData/cacheDir --with-updateSystemFn=/opt/mdata/
↪system.txt
```

If you have multiple directories each with their own timestamp file, you can list those in a file that configure will read rather than enumerating them with –with-spiderCacheDescript=file. This also enables each cache directory to have its own timestamp. The file is only used at configure time, not when Lmod runs, and is used like:

```
cacheDir1:timestamp1
cacheDir2:timestamp2
```

Lines starting with '#' and blank lines are ignored. It is best if each cache directory has its own timestamp file. This file is used by configure to modify the $LMOD_DIR/init/lmodrc.lua file. See the *An Example Setup* for a complete example.

How to decide how many system cache directories to have

The answer to this question depends on which machines "owns" which modulefiles. Many sites have a single location where their modulefiles are stored. In this case a single system cache file is all that is required.

At TACC, we need two system cache files because we have two different locations of files: one in the shared location and one on a local disk. So in our case Lmod sees two cache directories. Each node builds a spider cache of the modulefiles it "owns" and a single node (we call it master) builds a cache for the shared location.

What directories to specify?

If your site doesn't use the software hierarchy, (see *How to use a Software Module hierarchy* for more details) then just use all the directory specified in **MODULEPATH**. If you do use the hierarchy, then just specify the "Core" directories, i.e. the directories that are used to initialize Lmod but not the compiler dependent or mpi-compiler dependent directories.

How to test the Spider Cache Generation and Usage

In a couple of steps you can generate a personal spider cache and get the installed copy of Lmod to use it. The first step would be to load the lmod module and then run the **update_lmod_system_cache_files** program and place the cache in the directory *~/moduleData/cacheDir* and the time stamp file in *~/moduleData/system.txt*:

```
$ module load lmod
$ update_lmod_system_cache_files -d ~/moduleData/cacheDir -t ~/moduleData/system.txt
↪$MODULEPATH
```

If you using Lmod 6 then replace **MODULEPATH** with **LMOD_DEFAULT_MODULEPATH** instead.

Next you need to find your site's copy of lmodrc.lua. This can be found by running:

```
$ module --config
...

Active RC file(s):
------------------
/opt/apps/lmod/6.0.14/init/lmodrc.lua
```

It is likely your site will have it in a different location. Please copy that file to ~/lmodrc.lua. Then change the bottom of the file to be:

```
scDescriptT = {
  {
    ["dir"]       = "/path/to/moduleData/cacheDir",
    ["timestamp"] = "/path/to/moduleData/system.txt",
  },
}
```

where you have changed */path/to* to match your home directory. Now set:

```
$ export LMOD_RC=$HOME/lmodrc.lua
```

Then you can check to see that it works by running:

```
$ module --config
...

Cache Directory             Time Stamp File
---------------             ---------------
$HOME/moduleData/cacheDir   $HOME/moduleData/system.txt
```

Where **$HOME** is replaced by your real home directory. Now you can test that it works by doing:

```
$ module avail
```

The above command should be much faster than running without the cache:

```
$ module --ignore_cache avail
```

An Example Setup

Suppose that your site has three different modulefile trees. This can be handle in two very different ways. If each tree is on the same computer you can have one spider cache that knows about all three.

Assuming that the tree modulefile trees are named:

```
/sw/ab/modulefiles
/sw/cd/modulefiles
/sw/ef/modulefiles
```

If all tree directory trees are owned by same computer then one can configure Lmod with:

```
$ ./configure --with-spiderCacheDir=/sw/mData/cacheDir --with-updateSystemFn=/sw/mData/
↪cacheTS.txt
```

And build the cache file with:

```
$ export MODULEPATH=/sw/ab/modulefiles:/sw/cd/modulefiles:/sw/ef/modulefiles
$ update_lmod_system_cache_files -d /sw/mData/cacheDir -t /sw/mData/cacheTS.txt
↪$MODULEPATH
```

Now suppose you have the same three module directories but they reside on three different computers or are managed by three different groups. If you have three different groups managing a different module directory tree, you'll obviously want each group to manage each module tree separately.

Many sites place all their module based software on a shared disk across all nodes. Other sites might store some software locally on a node and some in a shared location. It is this scenario which requires some care when generating the spider caches.

So for any number of reasons you might have to have multiple spider cache files. In this case your site would configure Lmod with a spider cache description file (call say: spiderCacheDescript.txt) that contains:

```
/sw/ab/mData/cacheDir:/sw/ab/mData/cacheTS.txt
/sw/cd/mData/cacheDir:/sw/cd/mData/cacheTS.txt
/sw/ef/mData/cacheDir:/sw/ef/mData/cacheTS.txt
```

Next Lmod is configured with this spiderCacheDescript.txt file, which is only used to configure Lmod.:

```
$ ./configure --with-spiderCacheDescript=/path/to/spiderCacheDescript.txt
```

The configure script modifies the $LMOD_DIR/init/lmodrc.lua file so that the lmod command knows about the caches. The spiderCacheDescript.txt is never used again. Here is what the bottom of the lmodrc.lua would look like:

```
...
scDescriptT = {
    {
      ["dir"]       = "/sw/ab/mData/cacheDir",
      ["timestamp"] = "/sw/ab/mData/cacheTS.txt",
    },
    {
      ["dir"]       = "/sw/cd/mData/cacheDir",
      ["timestamp"] = "/sw/cd/mData/cacheTS.txt",
    },
    {
      ["dir"]       = "/sw/ef/mData/cacheDir",
      ["timestamp"] = "/sw/ef/mData/cacheTS.txt",
    },
}
```

Scenario 1: Three groups managing a separate module tree

Here we are assuming that all software resides on a shared but there are three group each managing a module tree.

So the "ab" group builds their spider cache as follows:

```
$ update_lmod_system_cache_files -d /sw/ab/mData/cacheDir -t /sw/ab/mData/cacheTS.txt  /
↪sw/ab/modulefiles
```

Similar the "cd" group builds their spider cache by:

```
$ update_lmod_system_cache_files -d /sw/cd/mData/cacheDir -t /sw/cd/mData/cacheTS.txt  /
↪sw/cd/modulefiles
```

and so on for each group managing their module tree. Each group has to update their spider cache if they update their module tree. If the "ab" group add new software and new modulefiles. They must update their cache file, but other groups do not have to update their caches if everything has remained the same for their modules

Scenario 2: Different computers owning different module trees

Suppose that the master node controls the directories **/sw/ab/. . .** and the **/sw/cd/. . .** on a shared disk. Then on the master node, one runs:

```
master$ update_lmod_system_cache_files -d /sw/ab/mData/cacheDir -t /sw/ab/mData/cacheTS.
↪txt  /sw/ab/modulefiles
master$ update_lmod_system_cache_files -d /sw/cd/mData/cacheDir -t /sw/cd/mData/cacheTS.
↪txt  /sw/cd/modulefiles
```

Then on each local node has a replicated copy of **/sw/ef/. . .** on a local disk. So each node has to run:

```
$ update_lmod_system_cache_files -d /sw/ef/mData/cacheDir -t /sw/ef/mData/cacheTS.txt  /
↪sw/ef/modulefiles
```

Again if any new modulefiles are added or changed, then the appropriate caches must be updated.

1.6.14 Using the module spider command

The module spider command reports all the modules that can be loaded on a system. In a flat module layout system, the *module avail* and *module spider* return similar information. In a hierarchical system, *module spider* returns all the modules that are possible where as *module avail* only reports modules that can be loaded directly.

There are three modes to module spider

1. module spider: Report all modules, known as level 0.
2. module spider <name> : Report all the versions for the modules that match <name>. This is known as level 1
3. module spider <name/version>: Report detailed information on a particular module version. This is known as level 2

Level 0: module spider

The *module spider* command by itself lists all the modules that can be found. If your site is running a flat layout, then the information displayed will be similar to what *module avail* reports. A typical output looks like:

```
$ module spider
----------------------------------------------------------------
The following is a list of the modules currently available:
----------------------------------------------------------------
  autotools: autotools/1.2
    Autoconf, automake, libtool

  boost: boost/1.54.0, boost/1.55.0, boost/1.56.0, boost/1.61.0, boost/1.62.0
    Boost provides free peer-reviewed portable C++ source libraries.

  fftw2: fftw2/2.1.5
    Numerical library, contains discrete Fourier transformation

  gcc: gcc/4.7.3, gcc/4.8.1, gcc/5.3.1, gcc/6.2.0
    The Gnu Compiler Collection

  hdf5: hdf5/1.8.12, hdf5/1.8.13, hdf5/1.8.14
```

(continues on next page)

(continued from previous page)

```
    General purpose library and file format for storing scientific data (Serial Version)

  lmod: lmod/7.5.9
    Lmod: An Environment Module System

  mpich: mpich/3.0.4-dbg, mpich/3.0.4, mpich/3.1.1-dbg, mpich/3.1.1, mpich/3.1.2-dbg, ...
    High-Performance Portable MPI

  openmpi: openmpi/1.8.2, openmpi/1.10.3, openmpi/2.0.1
    Openmpi Version of the Message Passing Interface Library

  phdf5: phdf5/1.8.12, phdf5/1.8.13, phdf5/1.8.14
    General purpose library and file format for storing scientific data (parallel I/O␣
↪version)
```

This output shows the name of the module followed by a list of the fullnames of the modules. If there are more modules then can fit on one line, the the list is truncated. Below is the description. Lmod looks for a particular whatis command in the modulefile. For example, the autotools module has a whatis function call that looks like the following in a lua modulefile:

```
whatis("Description: Autoconf, automake, libtool")
```

In a TCL modulefile it would look like:

```
module-whatis "Description: Autoconf, automake, libtool"
```

If your output of *module spider* doesn't have a description, please ask your site to consider adding an appropriate whatis line in your modulefiles.

Level 1: module spider name1 name2 . . .

You can get more details about particular modules by adding one or more names to the command to get a level 1 command. For example:

```
$ module spider hdf5

----------------------------------------------------------------------------
  hdf5:
----------------------------------------------------------------------------
    Description:
      General purpose library and file format for storing scientific data
      (Serial Version)

     Versions:
        hdf5/1.8.12
        hdf5/1.8.13
        hdf5/1.8.14
     Other possible modules matches:
        phdf5
```

Since the name "hdf5" matches a module name, Lmod only reports on the hdf5 module and not the phdf5 module. It does report that other matches are possible (such as phdf5). The reason for this is some sites name the R stat package as R. This rule is to prevent getting every module that has an ‘r’ in it. Note that the searching for modules is case insensitve. So *module spider openmpi* would match a module named *OpenMPI*.

If you search a name that only partially matches a module name then Lmod reports all matches:

```
$ module spider df5

----------------------------------------------------------------------------
  hdf5:
----------------------------------------------------------------------------
    Description:
      General purpose library and file format for storing scientific data
      (Serial Version)

     Versions:
        hdf5/1.8.12
        hdf5/1.8.13
        hdf5/1.8.14

----------------------------------------------------------------------------
  phdf5:
----------------------------------------------------------------------------
    Description:
      General purpose library and file format for storing scientific data
      (parallel I/O version)

     Versions:
        phdf5/1.8.12
        phdf5/1.8.13
        phdf5/1.8.14
```

Finally, you can perform regular expression matches with:

```
$ module -r spider '.*hdf5.*'
```

Level 2: module spider name/version

The level 2 output provides a detailed report on a module:

```
$ module  spider phdf5/1.8.14

----------------------------------------------------------------------------
  phdf5: phdf5/1.8.14
----------------------------------------------------------------------------
    Description:
      General purpose library and file format for storing scientific
      data (parallel I/O version)

    You will need to load all module(s) on any one of the lines below
    before the "phdf5/1.8.14" module is available to load.

      gcc/4.8.1  mpich/3.1.1
      gcc/4.8.1  mpich/3.1.2
```

(continues on next page)

(continued from previous page)

```
      gcc/4.8.1  openmpi/1.8.2


    Help:
       The HDF5 module defines the following environment variables:
       ...
```

1.6.15 The spider tool

Lmod provides a tool called *spider* to build the spider cache and other files that can help a site manage their modulefiles. The shell script *update_lmod_system_cache_files* described in *System Spider Cache* uses the spider command to build the spiderT.lua file which is the filename of the spider cache. The command *spider* is different from the *module spider* command. The first one will be mainly used by system administrators where as the second command is typically for users.

The *spiderT.lua* file contains information about all the modulefiles on your system that can be found from $MODULEPATH and is built by *spider*. The way this file is built is be evaluating each module and remembering the help message, the whatis calls and the location of the modulefile. It also remembers any properties set by the modulefile. It doesn't remember the module commands inside the modulefile.

When evaluating the modulefiles, Lmod looks for changes in $MODULEPATH. Each new directory added to MODULEPATH is also searched for modulefiles. This is why this command is called "spider".

There are some important points about Lmod walking the module tree. Each modulefile is evaluated separately. This means that any changes in MODULEPATH need to be static. That is new path must be defined in the modulefile. If the change in MODULEPATH is dynamic or depends on some environment variable that is dynamic, it is very likely that Lmod's *module spider* won't know about these dynamic changes to MODULEPATH.

The spiderT.lua file is used by Lmod to know any properties set in a modulefile. This information is used by *module avail*, *module spider* and *module keyword*. By default Lmod doesn't use the spider cache (aka spiderT.lua) to load modulefiles, however a site can configure Lmod to use cache based loads. This just means that a site *must* keep the spider cache up-to-date.

Lmod looks for special whatis calls to know what the description for a module is. See *Using the module spider command* for details.

Dynamic Spider Cache Support

Lmod 8.7.4+ supports for spidering user modulefiles that compiler/MPI/cuda modules could support. Suppose your site uses the software hierarchy and your site wants to allow users to be able to use **module spider** to find their modules as well. To do this something like this to your compiler/MPI/cuda modules. Suppose in your sites gcc/10.3.0 module you have:

```
prepend_path("MODULEPATH", "...") -- System compiler dependent modules
local home_root = pathJoin(os.getenv("HOME"),"myModules")
local userDir   = pathJoin(home_root,"Compiler/gcc10")
if (isDir(userDir)) then
  prepend_path("MODULEPATH",userDir)
end
```

Where users know to define their personal hierarchical for gcc/10.* packages in say *$HOME/myModules/Compiler/gcc10/*.

If your site uses a spider cache (and you rebuild this cache with Lmod 8.8+ then Lmod will reread modulefiles that modify $MODULEPATH. This rereading can be turned with configure time options.

When building the spider cache, Lmod will not find modulefiles that conditionally add to $MODULEPATH. Suppose your standard module (say StdEnv.lua) wants to allow users to have "Core" modules then you might have:

```
local home_root = pathJoin(os.getenv("HOME"),"myModules")
local userDir = pathJoin(hroot,"Core")
if (isDir(userDir)) then
   prepend_path("MODULEPATH",userDir)
end
haveDynamicMPATH()
```

The point is that since at the time of building the system spider cache, the user's home directory won't be known. In this case, your StdEnv.lua file will need the **haveDynamicMPATH()** function. Thus telling Lmod that the "StdEnv.lua" module will be reread when performing a "module spider" command and allow any user's Core modules to be found when spidering.

Software page generation

Since the package modulefiles are the gold standard of the packages your sites offers, it should be that information which should be used to generate the list of software packages. Lmod provides two kind of output depending on your sites needs. Suppose you have the simple module tree:

```
foo
├── .2.0.lua
├── 1.0.lua
├── 1.1.lua
└── default -> 1.0.lua
```

Then Lmod can produce the following information in json format:

```
$ MODULEPATH=.
$ spider -o jsonSoftwarePage $MODULEPATH | python  -mjson.tool
[
    {
        "defaultVersionName": "1.0",
        "description": "foo description",
        "package": "foo",
        "versions": [
            {
                "canonicalVersionString": "000000001.*zfinal",
                "full": "foo/.2.0",
                "help": "foo v.2.0",
                "hidden": true,
                "markedDefault": false,
                "path": "foo/.2.0.lua",
                "versionName": "1.0",
                "wV": "000000000.000000002.*zfinal"
            },
            {
                "canonicalVersionString": "000000001.000000001.*zfinal",
                "full": "foo/1.1",
                "help": "foo v1.1",
```

(continues on next page)

(continued from previous page)

```
                "markedDefault": false,
                "path": "foo/1.1.lua",
                "versionName": "1.1",
                "wV": "000000001.000000001.*zfinal"
            },
            {
                "canonicalVersionString": "000000001.*zfinal",
                "full": "foo/1.0",
                "help": "foo v1.0",
                "markedDefault": true,
                "path": "foo/1.0.lua",
                "versionName": "1.0",
                "wV": "^00000001.*zfinal"
            }
        ]
    }
]
```

The versions array block is sorted by the "wV" fields. This a weighted version of the canonicalVersionString, where the only difference is that the first character in the string is modified to know that it is marked default. Also if a module is hidden then the "hidden" field will be set to true.

The last entry in the versions array is used to set the description.

Another json output may be of interest. There is more information but it will be up to the site build the summarization that jsonSoftwarePage provides:

```
$ MODULEPATH=.
$ spider -o spider-json $MODULEPATH | python  -mjson.tool
{
    "foo": {
        "foo/.2.0.lua": {
            "Description": "foo description",
            "Version": "1.0",
            "fullName": "foo/.2.0",
            "help": "foo v.2.0",
            "hidden": true,
            "pV": "000000000.000000002.*zfinal",
            "wV": "000000000.000000002.*zfinal",
            "whatis": [
                "Description: foo description",
                "Version: .2.0",
                "Categories: foo"
            ]
        },
        "foo/1.0.lua": {
            "Description": "foo description",
            "Version": "1.0",
            "fullName": "foo/1.0",
            "help": "foo v1.0",
            "hidden": false,
            "pV": "000000001.*zfinal",
            "wV": "^00000001.*zfinal",
```

(continues on next page)

(continued from previous page)

```
            "whatis": [
                "Description: foo description",
                "Version: 1.0",
                "Categories: foo"
            ]
        },
        "foo/1.1.lua": {
            "Description": "foo description",
            "Version": "1.1",
            "fullName": "foo/1.1",
            "help": "foo v1.1",
            "hidden": false,
            "pV": "000000001.000000001.*zfinal",
            "wV": "000000001.000000001.*zfinal",
            "whatis": [
                "Description: foo description",
                "Version: 1.1",
                "Categories: foo"
            ]
        }
    }
}
```

1.6.16 Deprecating Modules

There may come a time when your site might want to mark a module for deprecation. If you track module usage, you can find the modules that are rarely used, and you can find out which users are using the modules. Once you have decided which modules are marked for removal, you can make a message be printed when the module is loaded.

Suppose you want to mark a module for later removal. You can set up a message that is reported to the user *Every time the module is loaded.*

For example:

```
$ module load abc/1.2.3

abc/1.2.3:
    This module is deprecated and will be removed from the system
    on June 19.   Please load abc/2.3.4 instead.
```

Note that this message is just text and in no way controls user access to the module. Your site will have to remove the module. This nag message is a way to let your users know that removal will happen ahead of time.

Also note that a user only gets this message when loading the module. There is no special marking in `module avail` or `module spider`.

You can create a file called "admin.list" and place it in "/path/to/lmod/etc/admin.list". Note that typically the lmod script will be in "/path/to/lmod/lmod/libexec/lmod". The etc directory is independent to the version of Lmod. You can see the location that Lmod is looking for by executing:

```
$ module --config
```

Look for "Admin File". You can also set the "LMOD_ADMIN_FILE" to point to the admin.list file.

The admin file consists of key-value pairs. For example:

```
moduleName/version:  message
<blank line>
```

Or:

```
Full_PATH_to_Modulefile: message
<blank line>
```

Or:

```
foo/1.1  | bar/1.2  | Full_Path_to_Modulefile:
message
<blank line>
```

Lmod treats assume that if a pattern starts with a '/' then it is a full path to a modulefile. If it doesn't then it is a moduleName/version string. Also, you can have several modulefiles use the same message by separating them with |

You can use Lua regular expression to also match one or modules for both the full path to a module or the module fullname. Lua regular expressions are much less powerful (see http://lua-users.org/wiki/PatternsTutorial for more details). In particular they do not support patterns like:

```
(foo|bar)
```

Lmod searches the list of modules and/or paths from top to bottom and it uses the first match it finds. So you might want to place the module patterns from specific to general in the admin file to avoid incorrect matching.

Also some care may be necessary for '.' as it matches every character so you might have to change the '.' to '%.' to match the actual period. Also '-' is a regular expression character so matching module names that have a '-' in their name requires % quoting.

The message can be as many lines as you like. The message ends with a blank line. Below is an example:

```
gcc/2.95:    This module is deprecated and will be removed from the system on Jan 1.␣
↪1999.
             Please switch to a newer compiler.

boost/1%.54%.0:
We are having issues

/opt/apps/modulefiles/Compiler/gcc/4.7.2/boost/1.55.0:
We are having issues


boost/1%.[5-7].*:
   We are having more issues.

boost%-mpi/1%.[5-7].*:
   This module will be removed.
```

Note that you don't include the .lua part when specifying the version number.

1.6.17 Module Properties

Lmod support giving modules properties. For modules written Lua, the *add_property()* function looks like:

```
add_property("key", "value")
```

In TCL, it is written as:

```
add-property key value
```

The *key* and *value* are controlled by a file called lmodrc.lua.

The Properties File: lmodrc.lua

Lmod provides a standard lmodrc.lua which is copied to the installation directory. For example Lmod version X.Y.Z is installed in /apps/lmod/X.Y.Z then lmodrc.lua would be installed in /apps/lmod/X.Y.Z/init/lmodrc.lua. During the install process this file is modified to include the location of the system spider cache.

Lmod searches the properties in several location in order given below. Assuming again that Lmod is installed in /apps/lmod/X.Y.Z then Lmod searches for the property information in the following order:

1. /apps/lmod/X.Y.Z/init/lmodrc.lua
2. /apps/lmod/etc/lmodrc.lua
3. $LMOD_CONFIG_DIR/lmodrc.lua (default /etc/lmod/lmodrc.lua)
4. /etc/lmodrc.lua
5. $HOME/.lmodrc.lua
6. $LMOD_RC

Where $LMOD_RC is an environment variable that can be set to point to any file locaiton. If there are more than one of these files exist then they are merged and not a replacement. So a site can (and should) leave the first file as is and create another one to specify site properties and Lmod will merge the information into one.

The format of this file looks like:

```
local i18n = require("i18n")
propT = {
   arch = {
      validT = { mic = 1, offload = 1, gpu = 1, },
      displayT = {
         ["mic:offload"]     = { short = "(*)",  color = "blue", full_color = false, doc␣
↪= "built for host, native MIC and offload to the MIC",  },
         ["mic"]             = { short = "(m)",  color = "blue", full_color = false, doc␣
↪= "built for host and native MIC", },
         ["offload"]         = { short = "(o)",  color = "blue", full_color = false, doc␣
↪= "built for offload to the MIC only",},
         ["gpu"]             = { short = "(g)",  color = "red" , full_color = false, doc␣
↪= "built for GPU",},
         ["gpu:mic"]         = { short = "(gm)", color = "red" , full_color = false, doc␣
↪= "built natively for MIC and GPU",},
         ["gpu:mic:offload"] = { short = "(@)",  color = "red" , full_color = false, doc␣
↪= "built natively for MIC and GPU and offload to the MIC",},
      },
```

(continues on next page)

(continued from previous page)

```
   },
   state = {
      validT = { experimental = 1, testing = 1, obsolete = 1 },
      displayT = {
         experimental  = { short = "(E)", full_color = false,  color = "blue",   doc =␣
↪i18n("ExplM"), },
         testing       = { short = "(T)", full_color = false,  color = "green", doc =␣
↪i18n("TstM"), },
         obsolete      = { short = "(O)", full_color = false,  color = "red",   doc =␣
↪i18n("ObsM"), },
      },
   },
   lmod = {
      validT = { sticky = 1 },
      displayT = {
         sticky = { short = "(S)",  color = "red",    doc = i18n("StickyM"), }
      },
   },
   status = {
      validT = { active = 1, },
      displayT = {
         active = { short = "(L)",  color = "yellow", doc = i18n("LoadedM")},
     },
   },
}
```

This file defines a table called propT. A table is a generic name for a hash table or dictionary or associative array. That is, it stores key value pairs. It is an Lmod convention that a table is named with a trailing T to remind us that it is a table.

In this case propT defines the valid keys and values that are possible for a modulefile to use with *add_property()*. In the case of the above table, the only valid keys in a modulefile would be *arch*, *state*, *lmod* and *status*.

The value for *state* controls the valid values. In particular, the only valid values for *state* are *experimental*, *testing* or *obsolete*. Please note that a modulefile can have multiple properties but each property key can have only one value. So:

```
add_property("state","testing")
add_property("state","obsolete")
```

would make the *state* property have a value of *obsolete*. On the other hand a modulefile could have two or more properties.:

```
add_property("state","testing")
add_property("lmod","sticky")
```

Lmod itself depends on the keys *lmod* and *status*. So as a site, it is expected that any lmodrc.lua file will contain these properties.

The tables *validT* and *displayT*

The function *add_property()* expects a key and value. So for the *state* key, possible value are *experimental*, *testing* or *obsolete*. Those strings must appear in two tables: the *validT* and the *displayT* tables. For example, we can see that *testing* appears both in the *validT* and *displayT* tables. This exist for checking for valid values when the *add_property()* function is called from modulefiles.

The *displayT* table controls how the property is displayed. The fields in the table controls how a property is displayed. For example:

```
testing = { short = "(T)", full_color = false,  color = "green", doc = i18n("TstM"), },
```

says that a module with this property will have a '(T)' next to its name when printed by module avail. If the terminal display has "xterm" as part of the environment variable TERM. then the 'T' will be in green. If the field *full_color* is set to *true* then the name and 'T' will be in green.

The possible color values are: *black*, *red*, *green*, *yellow*, *blue*, *magenta*, *cyan*, and *white*. In practice, since users can use light letters on dark backgrounds or dark letters on light backgrounds, sites may wish to avoid *black*, *white* and possibly *yellow*.

The arch key shows that the the values can be combined. If the value is colon separated then each string between the colons have to be valid keys.

1.6.18 Debugging Modulefiles

Most modulefiles are simple combination of a `help()` message, a couple of `setenv()` and a `prepend_path()` or two and don't require much in the way of debugging. However modulefiles are a program and might need debugging.

Using *module show* to check a modulefile

You can check how Lmod will evaluate a module file by using *module show*. Lmod evaluates a modulefile and prints out the module commands. If the modulefile is syntactically then *module show* will report the module commands such as *setenv()* and *prepend_path()* etc.

Note that if the originally modulefile is written in TCL, the output will be in Lua.

TCL modulefiles

As was discuss in *How does Lmod convert TCL modulefile into Lua*, Lmod converts TCL modulefiles into a lua modulefile by executing normal tcl commands and translates TCL module commands into lua functions. To see what Lmod does with your TCL modulefile, you can run `tcl2lua.tcl` to see the translation:

```
$ $LMOD_DIR/tcl2lua.tcl <path_to_modulefile>
```

For example, suppose you have a TCL modulefile in `~/my_modules/foo/1.0`:

```
#%Module

global env
set home $env(HOME)
set pkg "$home/foo"
prepend-path PATH $pkg/bin
setenv FOO_DIR $pkg
```

Then running the command produces:

```
$ $LMOD_DIR/tcl2lua.tcl ~/my_modules/foo/1.0

prepend_path{"PATH","/home/user/foo/bin",delim=":",priority="0"}
setenv("FOO_DIR","/home/user/foo")
```

Lua Modulefiles

It is important to remember that Lmod uses a two part process to change your environment. The lmod program produces text that is appropriate for the shell choice: bash commands for bash shell; csh commands for C-shell and so on. Then that text is evaluated by the shell to change your environment.

We can take advantage of this two part process to debug modulefiles by getting Lmod to produce the commands but not evaluate them. So starting with a simple lua modulefile called `~/my_modules/foo/1.0.lua`:

```
local home = os.getenv("HOME")
local pkg  = pathJoin(home,"foo")
io.stderr:write("home: ",home,"\n")
io.stderr:write("pkg:  ",pkg,"\n")

prepend_path("PATH",pathJoin(pkg,"bin"))
setenv("FOO",pathJoin(pkg,"bin"))
```

You can see that the above modulefile contains two extra print debugging statements that you'll want to remove after debugging. Running the Lmod command produces:

```
$ module use ~/my_modules
$ $LMOD_CMD bash load foo/1.0

home: /home/user
pkg:  /home/user/foo
FOO="/home/user/foo/bin";
export FOO;
PATH="/home/user/foo/bin:..."
export PATH;
...
```

Actually the lmod command will produce much more text. It contains other environment variables such as:

```
LOADEDMODULES="...";
export LOADEDMODULES;
__LMFILES__="...";
export __LMFILES__;
MODULEPATH="...";
export MODULEPATH;
__LMOD_REF_COUNT_PATH="/home/user/foo/bin:1;..."
export __LMOD_REF_COUNT_PATH;
PATH="/home/user/foo/bin:..."
export PATH;
_ModuleTable001_="...";
export _ModuleTable001_;
_ModuleTable_Sz_="6";
export _ModuleTable_Sz_;
```

LOADEDMODULES and __LMFILES__ are the list of modules loaded and their locations. These variables are made available to be compatible with Tmod and can be used by modulefiles. Variables like __LMOD_REF_COUNT_PATH are used to support reference counting for path like variables. Finally, Lmod uses a lua table called the _ModuleTable_ which contains the information used between Lmod invocations. This table is base64 encoded and split into 256 character blocks and stored in $_ModuleTable001, $_ModuleTable002, ...

1.6.19 SitePackage.lua and hooks

Sites may wish to alter the behavior of Lmod to suit their needs. A good place to do this is the SitePackage.lua. Anything in this file will automatically be loaded every time the Lmod command is run. This file can be used to provide common functions that can be used in a sites modulefiles. It is also a place where a site can implement their hook functions.

Hook functions are normal functions that if implemented and registered with Lmod will be called when certain operations happen inside Lmod. For example, there is a load hook. A site can register a function that is called every time a module is loaded. There are several hook functions that are discussed in *Hook functions*.

How to set up SitePackage.lua

Here are two suggestions on how to use your SitePackage.lua file:

1. Install Lmod normally and then overwrite your SitePackage.lua file over this one in the install directory. This is good for testing but it is recommended that sites use the second method for permanent changes. Otherwise changes may be lost when upgrading Lmod.
2. Create a file named "SitePackage.lua" in a different directory separate from the Lmod installed directory and it will override the one in the Lmod install directory. Then you should modify your z00_lmod.sh and z00_lmod.csh (or however you initialize the "module" command) with:

   ```
   (for bash, zsh)
   export LMOD_PACKAGE_PATH=/path/to/the/Site/Directory

   (for csh)
   setenv LMOD_PACKAGE_PATH /path/to/the/Site/Directory
   ```

Implementing functions in SitePackage.lua

For example your site might wish to provide the following function to set MODULEPATH inside your SitePackage:

```
function prependModulePath(subdir)
   local mroot = os.getenv("MODULEPATH_ROOT")
   local mdir  = pathJoin(mroot, subdir)
   prepend_path("MODULEPATH", mdir)
end
```

This function must be registered with the sandbox so that Lmod modulefiles can call it:

```
sandbox_registration{ prependModulePath    = prependModulePath }
```

Hook tips

# If you want to know which modules are being loaded, use the FrameStack:

```
local frameStk = FrameStk:singleton()
-- check if anything else gets loaded
if not frameStk:empty() then
    print("This modules has no dependencies loaded.")
end

-- the module at the top of the FrameStack is the one the user requested
local userload = (frameStk:atTop()) and "yes" or "no"
```

# If you want to know which modules are loaded, you can use the ModuleTable:

```
local mt = MT:singleton()
if mt:exists('shortname') then
    print("Module shortname is loaded")
end
```

# If you want to know the short name or path of a loaded module, you can use the ModuleTable:

```
local mname   = MName:new("mt", FullModuleName)
local sn      = mname:sn()
local version = mname:version()
```

Hook functions

The following is the list of hooks supported by Lmod. To use these hooks will require some Lua coding expertise and studying how the hooks are used in the Lmod program.

load (…):
This function is called after a modulefile is loaded in "load" mode.

unload (…):
This function is called after a modulefile is unloaded in "unload" mode.

colorize_fullName (fullName, sn):
(Returns: string) Allows a site to colorize how module names are displayed in "module av" and "module list".

parse_updateFn (…):
This hook returns the time on the timestamp file.

writeCache (…):
This hook return whether a cache should be written.

SiteName (…):
(Returns: string: SiteName) This hook is used to specify Site Name. It is used to generate family prefix: `site_FAMILY_COMPILER` `site_FAMILY_MPI`

msgHook (…):
(Return: string) Hook to print messages after avail, list, spider.

errWarnMsgHook (…):
(Returns: string) Hook to print messages after LmodError, LmodWarning, LmodMessage.

groupName (…):
(Returns: string) This hook adds the arch and os name to moduleT.lua to make it safe on shared filesystems.

category (. . .):
(Returns: array) Allows sites to control what categories are printed for the command **module category**.

avail (. . .):
Map directory names to labels

restore (. . .):
This hook is run after restore operation

startup (UsrCmd):
This hook is run when Lmod is called but before any command is run.

finalize (UsrCmd):
This hook is run just before Lmod generates its output of environment variables and aliases and shell functions and exits.

packagebasename (s_patDir, s_patLib):
This hook gives you a table with the current patterns that spider uses to construct the reverse map.

load_spider (. . .):
This hook is called when spider is evaluating a modulefile.

isVisibleHook (modT):
This hook is called when evaluating whether a module is visible or hidden. It's argument is a table containing: fullName, sn (short name), fn (file path) and isVisible (boolean) of the module.

reverseMapPathFilter (keepA, ignoreA):
(Returns: 2 arrays) This hook returns two arrays: *keepA* and *ignoreA*. The *keepA* is an array of paths patterns that a site wishes to be stored in the spider cache. The *ignoreA* is an array of path patterns to ignore in the cache. See *src/StandardPackage.lua* has an example on how to implement this hook. This hook is used to control the directories that are included/excluded in the reverseMap.

spider_decoration (table):
(Returns: table) This hook provide a way for a site to add decoration to spider level 1 output. The table passed in contains the whatis category and the property table. See *rt/uitSitePkg/mf/site_scripts/SitePackage.lua* for a complete example.

Example Hook: msgHook

A site might like to control the output of list, avail, and spider commands by adding text to the beginning or end of the generated text.

Here is an example of how to use the msgHook. So inside a site's SitePackage.lua file one would do:

```lua
local hook = require("Hook")

function myMsgHook(kind,a)
   if (kind == "avail") then

      -- Here is text that would go at the top of avail:
      table.insert(a,1,"This system has ...\n")
      table.insert(a,2,"blah blah blah ...\n")
      table.insert(a,3,"more blah blah blah ...\n")

      -- Here is text that would go at the end of avail:
      a[#a+1] = "More blah blah ...\n"
      a[#a+1] = "yet more blah blah ...\n"
   elseif (kind == "list") then
```

(continues on next page)

(continued from previous page)

```
      ...
   elseif (kind == "spider") then
      ...
   end
   return a
end

hook.register("msgHook", myMsgHook)
```

As you can see you can add text to the beginning and/or the end of the text that is generated by avail, spider and list.

Registering Multiple Hook functions

Lmod 8.7.35+ supports sites registering multiple functions for a single hook. The function **hook.register()** now takes an optional third argument to control how the functions are evaluated. For example, a site may wish to register more than one load hook:

```
local function load_hook_a(t)
    local frameStk  = require("FrameStk"):singleton()
    local mt        = frameStk:mt()
    local simpleName = string.match(t.modFullName, "(.-)/")
    LmodMessage("Load hook A called on " .. simpleName)
end

local function load_hook_b(t)
    local frameStk  = require("FrameStk"):singleton()
    local mt        = frameStk:mt()
    local simpleName = string.match(t.modFullName, "(.-)/")
    LmodMessage("Load hook B called on " .. simpleName)
end

hook.register("load", load_hook_a)
hook.register("load", load_hook_b) -- overwrites the previous hook,
-- expected output for 'module load my_software/version':
-- Load hook B called on my_software

hook.register("load", load_hook_a)
hook.register("load", load_hook_b, "replace") -- overwrites the previous hook function,
-- expected output for 'module load my_software/version':
-- Load hook B called on my_software

-- > the following will run load_hook_a then load_hook_b.
hook.register("load", load_hook_a)           -- initializes the load hook function
hook.register("load", load_hook_b, "append") -- appends to the previous hook function.
-- expected output for 'module load my_software/version':
-- Load hook A called on my_software
-- Load hook B called on my_software

-- > the following will run load_hook_b then load_hook_a
hook.register("load", load_hook_a)            -- initializes the load hook function
hook.register("load", load_hook_b, "prepend") -- prepends to the previous hook function
```

(continues on next page)

(continued from previous page)

```
-- expected output for 'module load my_software/version':
-- Load hook B called on my_software
-- Load hook A called on my_software
```

Note that if the optional third argument (the action argument) is not provided, the default behaviour is "replace".

There are some hooks (such as groupName, SiteName, etc) that require return values. The last registered hook function will be used to return the value.

1.6.20 Lmod Localization

Lmod places all messages, warning and error messages into a file. It also uses a slightly modified version of the i18n package from:

https://github.com/kikito/i18n.lua

to allow for translate into multiple languages. The standard message file is in messageDir/en.lua. Other languages are such as French, Spanish, German, and Mandarin Chinese are available. You can submit pull requests if you wish to have other languages. Lmod uses the LANG variable to pick which language to use. Note that Lmod only uses the first part of the LANG variable to choose which language to use. So there is only one version of English and one version of Spanish and so on. So a typical value of LANG is:

LANG=en_US.UTF-8

Lmod removes from the underscore on and just uses "en". A site can also configure Lmod or set LMOD_LANG to override $LANG.

Site Tailoring

It is also possible to modify the standard messages to be tailored for a site. You can leave the Language to be English but change the value of a particular message to better match your site. For example you might change message e118 (as seen in messageDir/en.lua). Please note that the message string should be on one line.:

```
e118 = "User module collection: \\"%{collection}\\" does not exist.\\n  Try \\"module␣
↪savelist\\" for possible choices.\\n",
```

by creating a file called: /path/to/site_msgs.lua:

```
return {
     site = {
         e118 = "User module collection: \\"%{collection}\\" does not exist.\\n  Try \\
↪"module savelist\\" for possible choices.  For questions see https://...\\n";,
     }
 }
```

Then configure Lmod (or set env var. LMOD_SITE_MSG_FILE) to use /path/to/site_msg.lua.

1.6.21 Integration of EasyBuild and Lmod

EB & Lmod work great together.

There are variables you should look into for tuning your system optimally, such as `EASYBUILD_MODULES_TOOL=Lmod`, `LMOD_PACKAGE_PATH` (keep track of loads), `LMOD_SYSTEM_DEFAULT_MODULES` (choose buildsets/stages etc)

Here is an example of how to ensure that your users can choose to have (or not) the default directory of EasyBuild under home (`$HOME/.local/easybuild/modules/all`) in their `$MODULEPATH`. Conveniently, by going this way users can save/restore environments via Lmod, at will.

Features obtained:

1. user modulefiles generated via EasyBuild are easy to enable - less typing, too
2. `$MODULEPATH` is managed via modulefiles in itself
3. technique is compatible with `$LMOD_SYSTEM_DEFAULT_MODULES` feature
4. `ml save/restore` work as desired

Caveats:

- This feature is super-picky on broken module environments, fi: https://bugzilla.redhat.com/show_bug.cgi?id=1326075 (testable via https://github.com/hpcugent/easybuild-framework/issues/1756) . Fix that asap, or your modulefiles experience may not be as good as it can!

How to put into use

The two examples are visible below and are ready to be copied in respective filenames `use.own.eb/append` & `use.own.eb/prepend`. Simply place both files in a directory of the existing `$MODULEPATH` and decide if you with to *append* or *prepend*:

```
$ ml av use.own

--------------------- /etc/site/modules ---------------------
   use.own.eb/append    use.own.eb/prepend (D)

------------------- /cm/local/modulefiles -------------------
   use.own

  Where:
   D:  Default Module

Use "module spider" to find all possible modules and extensions.
Use "module keyword key1 key2 ..." to search for all
possible modules matching any of the "keys".

$ ml use.own.eb/append
$ echo $MODULEPATH
/etc/site/modules:/cm/local/modulefiles:/home/user/.local/easybuild/modules/all
```

Example use.own.eb/append

```
$ cat /etc/site/modules/use.own.eb/append
#%Module1.0#####################################################################
##
## use.own.eb modulefile
##
## modulefiles/use.own.eb.  Generated by fgeorgatos
##
proc ModulesHelp { } {

      puts stderr "\tThis module file will add \$HOME/.local/easybuild/modules/all to the
↪"
      puts stderr "\tlist of directories that the module command will search"
      puts stderr "\tfor modules.  EasyBuild places your own modulefiles here."
      puts stderr "\tThis module, when loaded, will create this directory if necessary."
      puts stderr "\n\n"
}

module-whatis   "adds your EasyBuild modulefiles directory to MODULEPATH"

eval set  [ array get env HOME ]
set     ownmoddir       $HOME/.local/easybuild/modules/all

# create directory if necessary
if [ module-info mode load ] {
      if { ! [ file exists $ownmoddir ] } {
              file mkdir $ownmoddir
      }
}

module use --append $ownmoddir
```

Example use.own.eb/prepend

```
$ cat /etc/site/modules/use.own.eb/prepend
#%Module1.0#####################################################################
##
## use.own.eb modulefile
##
## modulefiles/use.own.eb.  Generated by fgeorgatos
##
proc ModulesHelp { } {

      puts stderr "\tThis module file will add \$HOME/.local/easybuild/modules/all to the
↪"
      puts stderr "\tlist of directories that the module command will search"
      puts stderr "\tfor modules.  EasyBuild places your own modulefiles here."
      puts stderr "\tThis module, when loaded, will create this directory if necessary."
}

```

(continues on next page)

(continued from previous page)

```
module-whatis   "adds your EasyBuild modulefiles directory to MODULEPATH"

eval set  [ array get env HOME ]
set     ownmoddir       $HOME/.local/easybuild/modules/all

# create directory if necessary
if [ module-info mode load ] {
      if { ! [ file exists $ownmoddir ] } {
              file mkdir $ownmoddir
      }
}

module use --prepend $ownmoddir
```

1.6.22 Providing Custom Labels for Avail

Lmod writes out the modules in alphabetical order for each directory in MODULEPATH in order:

```
$ module avail

--------------- /opt/apps/modulefiles/A-B ----------------
abc/8.1   def/11.1   ghi/2.3

--------------- /opt/apps/modulefiles/Core ------------------
xyz/8.1   xyz/11.1 (D)

--------------- /opt/apps/modulefiles/Compilers -------------
gcc/6.3  intel/17.0
```

This is very useful and informative from the system perspective, but users might prefer to have more descriptive labels and have the contents of multiple directories displayed under the same label.

Sites can replace the directory paths with any label they like. This is implemented by adding a SitePackage.lua file and calling the avail hook. See *SitePackage.lua and hooks* for how to create SitePackage.lua.

Suppose you wish to merge the Common and Core sections above into a single group named “Core Modules” and change the directory to “Compiler Modules”. The result would be:

```
$ module avail

--------------- Core Modules -------------------------
abc/8.1   def/11.1   ghi/2.3   xyz/8.1   xyz/11.1 (D)

--------------- Compiler Modules ---------------------
gcc/6.3  intel/17.0
```

To make this happen you need to do the following. Create a SitePackage.lua file containing:

```
require("strict")
local hook = require("Hook")

local mapT =
```

(continues on next page)

(continued from previous page)

```
{
   en_grouped = {
      ['/Compilers$'] = "Compilers",
      ['/Core$']      = "Core Modules",
      ['/A%-B$']      = "Core Modules",
   },
   fr_grouped = {
      ['/Compilers$'] = "Compilateurs",
      ['/Core$']      = "Modules de base",
      ['/A%-B$']      = "Modules de base",
   },
}


function avail_hook(t)
   local availStyle = masterTbl().availStyle
   local styleT     = mapT[availStyle]
   if (not availStyle or availStyle == "system" or styleT == nil) then
      return
   end

   for k,v in pairs(t) do
      for pat,label in pairs(styleT) do
         if (k:find(pat)) then
            t[k] = label
            break
         end
      end
   end
end


hook.register("avail",avail_hook)
```

Sites must specially quote the minus sign ("-") with a % because it is a regex character.

The default style for displaying module names is the *system* style.

To use the new, grouped labeling, set the *LMOD_AVAIL_STYLE* variable to be:

```
export LMOD_AVAIL_STYLE="system:<en_grouped>:fr_grouped"
```

The angle brackets define the default which in this case is en_grouped. A user can set:

```
export LMOD_AVAIL_STYLE="fr_grouped"
```

to change to the french labels.

If a site does nothing then the *system* layout will be the result. Sites control the behavior by the *LMOD_AVAIL_STYLE* environment variable. Sites can set it as follows:

```
export LMOD_AVAIL_STYLE=system:grouped
```

The first word is assumed to be the default. The word *system* is special in that Lmod will report the output in the original way. By changing the order sites can make the *grouped* output the default.:

```
export LMOD_AVAIL_STYLE=grouped:system
```

To get the original output then users would have to do *module -s system avail*. The command *module avail* would use the *grouped* avail style.

Alternately, a user may also *export LMOD_AVAIL_STYLE=system* to get the original output on all subsequent *module avail* invocations.

1.6.23 Load Storms: Long load times or Fails to Load

Sometimes it can take a long time for a module to load. Or Lmod will produce an error that it can't load a module it should. One likely possibility is that you have a `load storm`. That is Lmod is loading and reloading the same modulefiles over and over. This can happen when a module tries to load other modules and other modules do the same. For example A/1.0.lua is:

```
load("B/2.0")
load("C/2.0")
```

The module B/2.0.lua is:

```
load("C/2.0")
load("D/2.0")
```

And module C/2.0.lua is:

```
load("D/2.0")
```

The load() function always loads the requested module file even if that modulefile is already loaded.

Lmod can report what is happening. Using the -D debug flag it is possible to track what gets loaded:

```
$ module purge
$ module -D load A          2> ~/load_storm.log
$ grep 'MasterControl:.*load(' ~/load_storm.log
```

The results from the grep is:

```
MasterControl:load(mA={A}){
    MasterControl:load(mA={B/2.0}){
        MasterControl:load(mA={C/2.0}){
            MasterControl:load(mA={D/2.0}){
        MasterControl:load(mA={D/2.0}){
            MasterControl:unload(mA={D}){
            MasterControl:load(mA={D/2.0}){
    MasterControl:load(mA={C/2.0}){
        MasterControl:unload(mA={C}){
            MasterControl:unload(mA={D/2.0}){
        MasterControl:load(mA={C/2.0}){
            MasterControl:load(mA={D/2.0})
```

We can see that the `D/2.0.lua` module is loaded 4 times in this example. To avoid this problem, one can reduce the number of loads. In this case, the B/2.0.lua module can only load the C module as D is already being loaded by the C module. If this is not practical then placing guards around the load statement can also reduce the number of loads. For example changing A/1.0.lua to:

```
if (not isloaded("B/2.0")) then
   load("B/2.0")
end
if (not isloaded("C/2.0")) then
   load("C/2.0")
end
```

and similarly for the other modules will reduce loading of each module to one time. This can been seen by executing the debug module load and greping the results as before:

```
MasterControl:load(mA={A}){
   MasterControl:load(mA={B/2.0}){
      MasterControl:load(mA={C/2.0}){
         MasterControl:load(mA={D/2.0}){
```

The above guard statements won't unload the dependent module, unloading the A won't unload B, C, or D. Changing the guard statements to the following will allow for loading and unloading:

```
if (not isloaded("B/2.0") or mode() == "unload") then
   load("B/2.0")
end
if (not isloaded("C/2.0") or mode() == "unload") then
   load("C/2.0")
end
```

1.6.24 Tracing Lmod

Lmod supports a tracing option to report the major steps that Lmod performs. It can be set by either setting the `LMOD_TRACING` to `yes` or using the -T or –trace option on the command line. This can be useful to understand what happens a shell startup. By setting `LMOD_TRACING` to yes for a particular user you can see something like the following for a user which as a default collection:

```
$ module -T restore
running: module -T restore
  Using collection:      /home/user/.lmod.d/default
  Setting MODULEPATH to :/opt/apps/modulefiles/Linux:/opt/apps/modulefiles/Core
  Loading: unix (fn: /opt/apps/modulefiles/Core/unix/unix.lua)
  Loading: gcc (fn: /opt/apps/modulefiles/Darwin/gcc/5.2.lua)
  Loading: noweb (fn: /opt/apps/modulefiles/Core/noweb/2.11b.lua)
  Loading: StdEnv (fn: /opt/apps/modulefiles/Core/StdEnv.lua)
```

Using Shell Startup Debug with Tracing

Sites may wish to install the shell startup debug package from sourceforce:

```
https://sourceforge.net/projects/shellstartupdebug
```

It tracks the startup actions during shell startup. It can tell you what files in /etc/profile.d/* are started. For example:

```
% cat > ~/.init.sh
export SHELL_STARTUP_DEBUG=1
```

(continues on next page)

(continued from previous page)

```
^D
% zsh -l
/etc/zshenv{
  /etc/profile.d/lmod.sh{
  } Time = 0.1875
  /etc/profile.d/z00_lmod.sh{
  } Time = 0.2294
  /etc/profile.d/z01_StdEnv.sh{
  } Time = 0.2889
} Time = 0.2929
```

One of the tricks of this package is that it guarantees that a ~/.init.sh is read before any files in /etc/profile.d/*.sh are sourced for bash and zsh. The file ~/.init.csh is used for csh/tcsh. This means that you can also easily track Lmod startup actions:

```
% cat > ~/.init.sh
export LMOD_TRACING=yes
^D

% ssh localhost
...
running: module --initial_load --no_redirect restore
  Using collection:      /home/user/.lmod.d/default
  Setting MODULEPATH to: /opt/apps/modulefiles/Darwin:/opt/apps/modulefiles/Core
  Loading: unix (fn: /opt/apps/modulefiles/Core/unix/unix.lua)
  Loading: gcc (fn: /opt/apps/modulefiles/Darwin/gcc/5.2.lua)
  Loading: noweb (fn: /opt/apps/modulefiles/Core/noweb/2.11b.lua)
  Loading: StdEnv (fn: /opt/apps/modulefiles/Core/StdEnv.lua)
```

This way you can trace Lmod startup without having to edit any files in /etc/profile.d/* or the shell startup files.

1.6.25 Sticky Modules

Sites may wish to mark a module as *sticky*. This means that if the module is loaded then it won't be removed with a normal unload or purge. One possible use of sticky modules is where a site wants to define some environment variables that define what the architecture or operating system that allows users to use their system. The reason to make these values sticky is that the system may be difficult to use without these variables set.

Suppose you have a module named "site" that will be sticky. A lua module would look like:

```
setenv("ARCH","abc")
add_property("lmod","sticky")
```

A TCL module would look like:

```
#%Module
setenv ARCH abc
add-property lmod sticky
```

It is the `add_property()` function in Lua or `add-property` command in TCL which makes the module sticky.

If the "site" module is loaded then it can be unloaded by either command:

```
$ module --force unload site
```

or:

```
$ module --force purge
```

Since a user can unload a sticky module if they really want to. You may wish to the startup scripts (i.e. /etc/profile.d/*) instead of modules to define environment variables that you don't want users to easily change.

Lmod Implementation

Lmod unloads all requested modules, including the sticky modules. As each module is unloaded it remembers any modules which are sticky. After all modules have been unloaded, Lmod tries to load any sticky modules found from the previous step.

1.6.26 Modify Lmod behavior with SitePackage.lua

Lmod provides a standard way for sites to modify its behavior. This file is called SitePackage.lua. Anything in this file will automatically be loaded every time the Lmod command is run. Here are two suggestions on how to use your SitePackage.lua file

1. Install Lmod normally and then overwrite your SitePackage.lua file over this one in the install directory.
2. Create a file named "SitePackage.lua" in a different directory separate from the Lmod installed directory. Then you should modify your z01_lmod.sh and z01_lmod.csh (or however you initialize the "module" command) with:

    (for bash, zsh, etc) export LMOD_PACKAGE_PATH=/path/to/the/Site/Directory

    (for csh) setenv LMOD_PACKAGE_PATH /path/to/the/Site/Directory

   A "SitePackage.lua" in that directory will override the one in the Lmod install directory.

You should check to see that Lmod finds your SitePackage.lua. If you do:

```
$ module --config
```

and it reports:

```
Modules based on Lua: Version X.Y.Z  3027-02-05 16:31
    by Robert McLay mclay@tacc.utexas.edu

Description                       Value
-----------                       -----
...
Site Pkg location                 standard
```

Then you haven't set things up correctly.

Possible functions for your SitePackage.lua file

There are two common reason a site might set up this file. The main reason is to specify hook functions. If your site wishes to track module usage, you can use the load hook to have a message sent to syslog. The details are in contrib/track_module_usage/README.

The second reason is to provide common functions that all your modulefiles can use. These functions must be registered. To prevent modulefiles from calling arbitrary functions all modulefiles are evaluated in a "sandbox" which limits the functions that are called.

The following SitePackage.lua file is a simple example of how to implement a function. In this case, we are going to provide a simple prepend to the MODULEPATH that can be called by any modulefile. Suppose your site is wants to use MODULEPATH_ROOT as the top level directory and all modulepath entry are subdirectories of it. You could create a `prependModulePath()` function to simplify your modulefiles.

Here is the SitePackage.lua file:

```
require("sandbox")

function prependModulePath(subdir)
   local mroot = os.getenv("MODULEPATH_ROOT")
   local mdir  = pathJoin(mroot, subdir)
   prepend_path("MODULEPATH", mdir)
end

sandbox_registration{ prependModulePath    = prependModulePath, }
```

Then in your lua modulefiles you can use it to extend MODULEPATH:

```
-- gcc modulefile:
local version = "7.1"
prependModulePath(pathJoin("Compiler/gcc/",version))
```

Note that if you wish to use any function defined in SitePackage.lua in a TCL modulefile, you will need to modify tcl2lua.tcl to know about these functions. You'll also have to merge your changes to tcl2lua.tcl into any new version of Lmod.

There is no need to modify tcl2lua.tcl if you only add hook functions.

Using mgrload function

Suppose your site has an archecture state which controls the how the modules work. So you would like to unload all the current modules and then set an environment variable and reload. Here you want to use the **mgrload** function and not the **load** function. If you use the load function you loose the **depends_on** () state and whether a module was loaded by a user or was loaded by another module. The way this works is that you ask for the currently loaded modules with the **loaded_modules** function to get an array of "active" objects. So the following is the "AVX" architecture module:

```
if (mode() == "load") then
   local required = false
   local activeA = loaded_modules()

   for i = 1,#activeA do
      io.stderr:write("Unloading: ",activeA[i].userName,"\n")
      unload(activeA[i].userName)
   end
```

(continues on next page)

(continued from previous page)

```
   setenv("SITE_CURRENT_ARCH","avx")
   for i = 1,#activeA do
      io.stderr:write("loading: ",activeA[i].userName,"\n")
      mgrload(required, activeA[i])
   end
end
```

1.6.27 Shell scripts and Lmod

Some application provide shell scripts to initialize their use. The drawbacks of this approach is that applications would have to provide scripts for each shell and there was typically no way to unload the application. Also users of shells other than bash or (t)csh were also usually out of luck.

Lmod, like other environment module system are used by tools that are not shells such as cmake, R, perl, python, etc. So application shell scripts there weren't helpful there either. Modules provide a way to support the other shells and non-shell applications.

Lmod has provided **sh_to_modulefile** to convert scripts to modulefiles. New in version 8.6+, Lmod provides support for **source_sh** () to source shells scripts *inside* a modulefile. This provides several features at a cost. It means that it can be used by all module applications *and it can be unloaded*. The cost is that Lmod is evaluating the shell script in a subshell extract the module commands every time that module is loaded. So it is typically better to convert it once with **sh_to_modulefile**.

Converting shell scripts to modulefiles

Lmod provides a script called *sh_to_modulefile* which will convert a script to a modulefile. An example is:

```
% $LMOD_DIR/sh_to_modulefile  ./foo.sh > foo_1.0.lua
```

or:

```
% $LMOD_DIR/sh_to_modulefile  --output foo_1.0.lua ./foo.sh
```

This program defaults to generating a lua based modulefile. It is possible to generate a TCL modulefile with:

```
% $LMOD_DIR/sh_to_modulefile  --to TCL --output foo_1.0 ./foo.sh
```

See:

```
% $LMOD_DIR/sh_to_modulefile  --help
```

for all the options.

The way it works is that remembers the initial environment and runs the script. The program then compares the initial environment and generate environment. The output is a report of the environment changes.

As of version 8.6, Lmod now tracks changes to shell aliases and shell functions and writes them to the generated modulefile.

Converting scripts once with this command is usually best. However, some scripts depend on dynamic environment variable that change between users such as the values of $HOME or $USER. In this case, the use of the **source_sh** () modulefile function can be helpful.

Using source_sh ()

The feature of sourcing shell scripts inside a modulefile was introduced in Tmod 4.6+. It has be shamelessly studied and re-implemented in Lmod 8.6+. In Lmod, this feature re-uses much of the code that implements **sh_to_modulefile**. This code does the following when performing a module load.

1. Gets the current environment, shell aliases and shell functions
2. Sources the shell script with arguments
3. Compares the new environment to extract module commands

The resulting modules commands are stored in the user environment inside the module table which can be shown by running **$ module –mt**.

When unloading or showing, the module commands are extracted from the module table and used to unload the changes that the script caused. In other words, the shell script is only evaluated when loaded. not on unload.

> **Note**: Occasionly, application scripts will provide a “deactivate” that a site might be temped to use like this:
>
> ```
> if (mode() == "unload") then
>    source_sh("bash", "app_script deactivate")
> end
> ```
>
> **Do not do this!** the function **source_sh** expects to find the module function in the module table in the environment. It is better to do this for load and unload:
>
> ```
> source_sh("bash", "app_script activate")
> ```
>
> and let Lmod unload the scripts via the generated module functions.

Sites can dynamically build the shell script and argument string. It is important however that this string be the same for both load and unload because this string is part of the access method to extract the commands from the module table.

Assumptions that Lmod makes about scripts used by source_sh ()

Lmod assumes that these scripts **DO NOT** have module commands or change $MODULEPATH.

Shell script are evaluated with set -e

Shell scripts are evaluated by sh_to_modulefile and the source_sh() function with set -e in bash or equivalently in other shells. This means that execution of the shell script stops at the first time a statement in the script has a non-zero status. Turning this option on means that Lmod can know that the evaluation of the shell script has an issue. But this also means that sometimes a script would work fine without using “set -e”.

Assuming the script is a bash script, please do the following:

```
$ set -exv ; . ./my_script
```

This should tell you where the issue was found. Note that the error may be in another script sourced by “my_script”. In particular, you might have a silent error. For example, a bash script might have a line:

```
unalias some_alias_name 2> /dev/null
```

where “some_alias_name” is not currently an alias. This unalias statement returns a non-zero status but is silent because of the redirection of stderr to /dev/null. One fix might be:

```
unalias some_alias_name 2> /dev/null || true
```

Calling the shell script directly inside a modulefile

Site can also use the execute{} function inside a modulefile. This function add the shell script text at the end of the string that is evaluated by the shell. This execute command only makes sense if the evaluation is by the appropriate shell.

If a site wants to place the shell command first then they can use the print() statement as this will appear first. For example, to have the script appear before the unset environment commands then do this:

```
if (mode() == "unload") then
   print("app_script deactivate")
end
```

The string **"app_script deactivate"** will be generated before any other environment commands will generated.

1.6.28 Tracking Module Usage

Once you have a module system, it can be important to know what modules your users are using or not. This ability collect in some fashion has existed for a long time. What is new here is a complete solution: Using syslog to track module usage. Then collect this data into a database.

There are a number of steps but all are easy. The following is an overview. I am going to explain our setup and point out where you site may be different. In this setup I will be assuming that you are using a MySQL database and Rsyslog (version 5.8)

For a cluster, it is common to have each node in that cluster to send its syslog messages to a single central machine (called master in this discussion). I will be sending the module usage message through syslog to a separate machine. That separate machine will collect the module tracking data into a log file that just contains tracking data. Finally this data is written into the database.

Also provided is a script to analyze the data as shown below:

```
$ analyzeLmodDB --sqlPattern '%fftw%' counts --start '2015-01-01 --end '2015-02-01'

Module path                                            Distinct Users
-----------                                            --------------
/apps/intel13/mvapich2_1_9/modulefiles/fftw3/3.3.2               151
/apps/intel13/mvapich2_1_9/modulefiles/fftw2/2.1.5                62
/apps/intel13/impi_4_1/modulefiles/fftw3/3.3.2                    45
/apps/intel13/impi_4_1/modulefiles/fftw2/2.1.5                    19
```

This shows number of users of any fftw module for the first month of 2015.

Detailed Steps

Step 1

Use SitePackage.lua to send a message to syslog.:

```
------------------------------------------------------------------------------
-- load_hook(): Here we record the any modules loaded.

local hook    = require("Hook")
local uname   = require("posix").uname
local cosmic  = require("Cosmic"):singleton()
local syshost = cosmic:value("LMOD_SYSHOST")

local s_msgT = {}

local function load_hook(t)
   -- the arg t is a table:
   --     t.modFullName:  the module full name: (i.e: gcc/4.7.2)
   --     t.fn:           The file name: (i.e /apps/modulefiles/Core/gcc/4.7.2.lua)


   -- use syshost from configuration if set
   -- otherwise extract 2nd name from hostname: i.e. login1.stampede2.tacc.utexas.edu
   local host        = syshost
   if (not host) then
      local i,j, first
      i,j, first, host = uname("%n"):find("([^.]*)%.([^.]*)%.")
   end


   if (mode() ~= "load") then return end
   local msg         = string.format("user=%s module=%s path=%s host=%s time=%f",
                                     os.getenv("USER"), t.modFullName, t.fn, uname("%n"),
                                     epoch())
   s_msgT[t.modFullName] = msg
end

hook.register("load", load_hook)

local function report_loads()
   for k,msg in pairs(s_msgT) do
      lmod_system_execute("logger -t ModuleUsageTracking -p local0.info " .. msg)
   end
end

ExitHookA.register(report_loads)
```

This code uses two “hook” functions. The first is load_hook. This means that every load will saved. The second hook is called at exit. If there were no errors then any module loads are reported by sending a syslog message with the tag “ModuleUsageTracking”

Please read the file src/SitePackage.lua to see how to use the environment variable LMOD_PACKAGE_PATH to point to your own SitePackage.lua.

You should check to see that Lmod finds your SitePackage.lua. If you do:

```
$ module --config
```

and it reports:

```
Modules based on Lua: Version X.Y ...
    by Robert McLay mclay@tacc.utexas.edu

Description                       Value
-----------                       -----
...
Site Pkg location                 standard
```

Then you haven't set things up correctly.

Step 2

Have "master" send the tracking messages to a separate computer.

You can add the following to master's /etc/rsyslog.conf file:

```
if $programname contains 'ModuleUsageTracking' then @module_usage_tracking
&~
```

Where you change "module_usage_tracking" into a real machine name. Adding this to rsyslog.conf will direct all syslog messages to be sent to the "module_usage_tracking" machine.

Remember to restart the rsyslog daemon on master.

Step 3

On the "module_usage_tracking" machine you add to /etc/rsyslog.conf the following:

```
# read in include files
$IncludeConfig /etc/rsyslog.d/*.conf...
```

Then in /etc/rsyslog.d/moduleTracking.conf:

```
$Ruleset remote
if $programname contains 'ModuleUsageTracking' then /var/log/moduleUsage.log
$Ruleset RSYSLOG_DefaultRuleset

# provides UDP syslog reception
$ModLoad imudp
$InputUDPServerBindRuleset remote
$UDPServerRun 514
```

The above commands are in the language of rsyslog version 5.8. What this says is accept outside syslog messages on port 514 and if any are tagged with "ModuleUsageTracking" then write them to /var/log/moduleUsage.log

Remember to restart the rsyslog daemon on the "module_usage_tracking" machine.

Step 4

Create the file /etc/logrotate.d/moduleUsage:

```
/var/log/moduleUsage.log{
   missingok
   copytruncate
   rotate 4
   daily
   create 644 root root
   notifempty
}
```

This will log rotate the moduleUsage.log. Remember to restart the logrotate daemon. Note that it will be the second day before the log is rotated. On Centos machines, it seems that the log rotate happens at about 3am.

Step 5

I found the following site helpful in getting the MySQL database setup:

```
http://zetcode.com/db/mysqlpython/
```

a) Install MySQL db. Create a mysql root password. Then create an account in the database like this:

   ```
   $ mysql -u root -p
   Enter password:

   mysql> CREATE DATABASE lmod;

   mysql> CREATE USER 'lmod'@'localhost' IDENTIFIED BY 'test623';

   mysql> USE lmod;

   mysql> GRANT ALL ON lmod.* TO 'lmod'@'localhost';

   mysql> flush privileges;

   mysql> quit;
   ```

   You will want to change 'test623' to some other password. You'll also probably want to allow access to this database from outside machines as well.

b) Use the "conf_create" program from the contrib/tracking_module_usage directory to create a file containing the access information for the db:

   ```
   $ ./conf_create
   Database host:
   Database user: lmod
   Database pass:
   Database name: lmod
   ```

   Where you'll have to fill in the correct name for the database host and password. This creates a file named lmod_db.conf which is used by createDB.py, analyzeLmodDB and other programs to access the database.

c) Make sure your python knows about the MySQLdb module. Please use pip or something similar if it is unavailable.

d) Create the database by running the createDB.py program.:

```
$ ./createDB.py
```

Step 6

a) If you have more than one cluster and you want to store them in the same database you might want to modify the store_module_data program found in the contrib/tracking_module_usage directory. It assumes that host names are of the form: node_name.cluster_name.something.something and the current store_module_data program picks off the second field in the hostname. If your site names things differently you should modify that routine to match your needs.

b) I use a cron job to load the moduleUsage.log-* files. This is the script I use:

```
#!/bin/bash

PATH=<path_to_python3>:$PATH
cd ~mclay/load_module_usage

for i in /var/log/moduleUsage.log-*; do
  ./store_module_data $i
  if [ "$?" -eq 0 ]; then
    rm -f $i
  fi
done
```

Where <path_to_python3> has a python3 that can also import MySQLdb python module. If it is not already installed, you can do:

```
$ pip3 install mysqlclient
```

Also you'll probably want to change ~mclay/load_module_usage to where ever you have the store_module_data program and lmod_db.conf files.

I am running this cron job on the "module_usage_tracking" machine at 5am every morning. This is after the log rotation has been done.

Step 7

Once data is being written to the database you can now start analyzing the data. You can use SQL commands directly into the MySQL data base or you can use the supplied script found in the contrib/tracking_module_usage directory: analyseLmodDB:

```
% ./analyzeLmodDB --help
usage: analyzeLmodDB [-h] [--dbname DBNAME] [--syshost SYSHOST]
                     [--start STARTDATE] [--end ENDDATE]
                     [--sqlPattern SQLPATTERN]
                     cmdA [cmdA ...]

positional arguments:
```

(continues on next page)

(continued from previous page)

```
  cmdA                   commands: counts, usernames, modules_used_by

optional arguments:
  -h, --help             show this help message and exit
  --dbname DBNAME        lmod db name
  --syshost SYSHOST      system host name
  --start STARTDATE      start date
  --end ENDDATE          end date
  --sqlPattern SQLPATTERN sql pattern for matching
```

There are three kinds of reports this program will report. Only one command at a time.

a) counts: Report the number of distinct users of a particular module:

```
$ analyzeLmodDB --sqlPattern '%fftw%' --start '2015-01-01 --end '2015-02-01'  counts

    Module path                                          Distinct Users
    -----------                                          --------------
    /apps/intel13/mvapich2_1_9/modulefiles/fftw3/3.3.2             151
    /apps/intel13/mvapich2_1_9/modulefiles/fftw2/2.1.5              62
    /apps/intel13/impi_4_1/modulefiles/fftw3/3.3.2                  45
    /apps/intel13/impi_4_1/modulefiles/fftw2/2.1.5                  19
```

To get all modules loaded in a date range do:

```
$ analyzeLmodDB --sqlPattern '%' --start '2015-01-01 --end '2015-02-01'  counts
```

b) usernames: Report users of a particular pattern:

```
$ ./analyzeLmodDB --sqlPattern '%/apps/modulefiles/settarg%' usernames

Module path                      User Name
-----------                      ---------
/opt/apps/modulefiles/settarg/5.8      user1
/opt/apps/modulefiles/settarg/5.8      user2
/opt/apps/modulefiles/settarg/5.8      user3
/opt/apps/modulefiles/settarg/5.8.1    mclay
/opt/apps/modulefiles/settarg/5.9.1    user5
```

c) modules_used_by: Report the modules used by a particular user:

```
$ ./analyzeLmodDB --start '2015-01-01 --end '2015-02-01' --sqlPattern 'mclay'␣
↪modules_used_by

Module path                                                 User Name
-----------                                                 ---------
/opt/apps/gcc4_9/modulefiles/boost/1.55.0.lua               mclay
/opt/apps/gcc4_9/modulefiles/mvapich2/2.1                   mclay
/opt/apps/gcc4_9/mvapich2_2_1/modulefiles/phdf5/1.8.16.lua  mclay
/opt/apps/gcc4_9/mvapich2_2_1/modulefiles/pmetis/4.0.2.lua  mclay
/opt/apps/intel13/modulefiles/boost/1.55.0.lua              mclay
/opt/apps/intel13/modulefiles/mvapich2/1.9a2                mclay
```

Tracking user loads and not dependent loads

Some sites would like to track the modules loaded by users directly and not the dependent loads. If your site wished to do that then look at the directory in the source tree: **contrib/more_hooks**. In that directory is a SitePackage.lua file as well as README.md which explains how to just track user loads.

1.6.29 Combining modules and build systems with settarg

Introduction

The settarg module works with Lmod to help developers manage their compiled software projects. It does so by making it easy to switch between optimized or debug builds; change compiler or other modules; and let the build system know about the changes. The secret of settarg is that it consolidates the state of a build into one environment variable called $TARG.

One of the big advantages of developing software with a module system is the ability to easily switch between compilers and mpi stacks and other modules. Sometimes one gets a strange compiler error or runtime error with one set of tools and it is convenient try a different compiler to see if it tells you something different.

Another useful trick to switch between the level of optimization with one set of builds that have minimal optimization and the "-g" flag to include the symbol table for ease of debugging and other builds with full optimization.

Finally there are some sites which have "shared-home" filesystems. That is where two or more clusters share the same "home" directory tree. Unless they are exactly the same hardware and running the same version of the operating system, software built on one system might not work on another.

For all these reasons, it is convenient to have the built software reside in different directories. Settarg makes it easy to place all the objects, libraries and executes in a separate directory for each kind of build so that they never mix and this avoids hard to resolve bugs.

Settarg manages these environment variables but it is up to the software developer to integrate these variables in to their build system. More on how to modify a Makefile to know about all the TARG variables later.

The keys to settarg

The keys to settarg are:

1. It manages a set of environment variables including $TARG
2. It is integrated with Lmod so that changes in modules automatically changes $TARG
3. Your PATH is dynamically adjusted with $TARG
4. The title bar is dynamically managed.
5. It is easy to configure with site, user and project control.

All you need to is load the settarg module and it will control these variables including PATH that you can use to control your build process. The most important one of these is TARG short for target. It contains a string for the machine architecture, the build scenario, (opt, dbg), the compiler and the mpi module. For example, TARG might be:

```
TARG = OBJ/_x86_64_06_2d_dbg_gcc-7.1_openmpi-2.2
```

Settarg also generates other environment variables to be used to control your Makefile or other build tool. So for the above $TARG, the following variables are also set:

```
TARG_MACH             = x86_64_06_2d
TARG_BUILD_SCENARIO   = dbg
TARG_COMPILER         = gcc/7.1
TARG_COMPILER_FAMILY  = gcc
TARG_MPI              = openmpi/2.2
TARG_MPI_FAMILY       = openmpi
```

Knowing that TARG_COMPILER_FAMILY is gcc or intel can mean it is easy to set the compiler flags appropriate for each compiler.

But the most important feature of settarg is that if you change your compiler from gcc to intel/18.0.1 then TARG and the other environment variable change automatically:

```
TARG                  = OBJ/_x86_64_06_2d_dbg_intel-18.0.1_openmpi-2.2
TARG_COMPILER         = intel/18.0.1
TARG_COMPILER_FAMILY  = intel
```

and all the other variables remain the same. If you change the build scenario from `dbg` to `opt` then the following would be the result:

```
TARG                  = OBJ/_x86_64_06_2d_opt_intel-18.0.1_openmpi-2.2
TARG_BUILD_SCENARIO   = opt
```

Integration with your build system

Once we have these environment variables, we can use them to control where our software is built. It is possible to place all the objects, libraries and executable stored in the $TARG directory. If all the generated files are in the $TARG directory, then changing the compiler will result in a different TARG directory. So each $TARG directory is independent and we won't require a `make clean` between changing compilers or build scenarios.

Integration with PATH

It is useful to have your PATH point to the new $TARG directory, so settarg changes your path to include $TARG by removing the old value of $TARG and replacing it with the new value of $TARG. This way you can set $TARG, build, then run the new executable.

Settarg integration witn prompt commands

Tbe bash shell support an environment variable called "PROMPT_COMMAND". If this variable is set to the name of a shell function, then for each new shell prompt, this command is run. Similarly, zsh will run the "precmd" on every new prompt. By default the settarg module defines a shell function called precmd and if the user is using the bash shell, the PROMPT_COMMAND variable is set to "precmd".

If users have their own prompt command then, they can prevent settarg from overriding their prompt command by LMOD_SETTARG_IN_PROMPT to "no".

Xterm title bar support

If the environment variable LMOD_SETTARG_TITLE_BAR=yes and $TERM has "xterm" in the string, then loading the module settarg will turn on title bar support. A typical string in the title bar might be:

```
(D G/5.2 M/3.2)user@host: ~/bin
```

where user is your user name and host is your hostname followed by your current directory. The string in the parentheses are what settarg are providing. The "D" is dbg build scenario, the "G/5.2" is an abbreviation for the gcc/5.2 compiler module and "M/3.2" is an abbreviation for the mpich/3.2 mpi module. The abbreviations are controlled by configuration files. This string is TARG_TITLE_BAR_PAREN.

Settarg uses the following rules to define what the host is. If the environment variable $SHOST is defined then that value is use for the hostname. Otherwise the first name is used from the hostname. In other words, your host name is "login1.stampede2.tacc.utexas.edu" then "login1" will be used as the host. Obviously, if you'd rather have "login1.stampede2" be in the title bar then define SHOST to do so.

Settarg configuration

Lmod provides a default configuration for settarg in the file settarg_rc.lua. Sites may have to tailor this file to match the names of their compilers and mpi modules and other module names. Then users may wish to set their own preferences. Finally a project may wish to have specialized settings. All files are merged together in an intelligent fashion into a single configuration. They do not overwrite the previous setting. More on this in *Settarg configuration files*

Commands

The environment TARG's value is typically used as a name of the build directory. So the settarg module provides some helpful aliases to take advantage of this.

1. gettargdir: it is simply an alias for "echo $TARG"
2. cdt: Another alias: "cd $TARG"
3. settarg: How to set the build scenario and to access other features.

By default settarg has an "empty" build scenario. This can be changed by:

```
$ settarg dbg
$ settarg opt
```

Which will change TARG_BUILD_SCENARIO to "dbg" or "opt". Also:

```
$ settarg --report
```

report the state of the .settarg table after combining all the possible .settarg.lua files.

For those of you who like short commands, please configure Lmod with –with_settarg=full or set the environment variable LMOD_SETTARG_FUNCTIONS=yes before loading the settarg module. One useful command is:

```
$ targ
```

which is a short for "gettargdir". Also if you switch between build scenarios frequently may wish to define the following shortcuts for setting the build scenario:

```
dbg()  { settarg "$@" dbg;   }
opt()  { settarg "$@" opt;   }
mdbg() { settarg "$@" mdbg;  }
empty(){ settarg "$@" empty; }
```

What environment variables are defined by settarg

Below are a typical list of variables:

```
TARG_SUMMARY=x86_64_06_2d_dbg_gcc-7.1_openmpi-2.2
TARG=OBJ/_x86_64_06_2d_dbg_gcc-7.1_openmpi-2.2

TARG_TITLE_BAR=D G/7.1 O/2.2
TARG_TITLE_BAR_PAREN=(D G/7.1 O/2.2)

TARG_BUILD_SCENARIO=dbg

TARG_MACH=x86_64_06_2d

TARG_COMPILER=gcc/7.1
TARG_COMPILER_FAMILY=gcc

TARG_MPI=openmpi/2.2
TARG_MPI_FAMILY=openmpi

TARG_OS=Linux-2.6.32-279
TARG_OS_Family=Linux
TARG_HOST=stampede
```

Here is a glossary of what each of these variables mean:

TARG_SUMMARY:
The dynamic combination of items like the machine architecture, build scenario, etc. See below for how this gets built.

TARG:
This variable contains all the "interesting" items. How is put together is described later.

TARG_TITLE_BAR:
This contains everything in TARG_SUMMARY but it is abbreviated to fit the space available. This string is provided in case the user wishes to use this variable as part of their own title bar string.

TARG_TITLE_BAR_PAREN:
This is $TARG_TITLE_BAR with parentheses around the string. This variable is typically used in the xterm title bar.

TARG_BUILD_SCENARIO:
This can be used to control compiler flags so that "dbg" might mean to create a debuggable executable. Where as "opt" might mean to build a fully optimized build. To clear this field use the command `settarg empty`.

TARG_MACH:
This is the machine architecture along with the cpu family and model number in two hex numbers when on Linux system that has the pseudo file /proc/cpuinfo. The architecture is what is reported by "uname -m"

TARG_COMPILER:
The name of the compiler and version written as <compiler>/<version>

TARG_COMPILER_FAMILY:
The name of the compiler without the version.

TARG_MPI:
The name of the mpi module and version written as <mpi>/<version>

TARG_MPI_FAMILY:
The name of the mpi module without the version.

TARG_OS, TARG_OS_FAMILY:
These are the OS name and family. These variables are always defined even if there are not part of TARG_SUMMARY.

TARG_HOST:
See below on how this is extracted from *hostname -f*

Settarg configuration files

Below is a typical configuration file. This is file contains several tables in written in Lua. If you don't know Lua, it still should be easy to modify this table. just remember the comma's.

The BuildScenarioTbl table maps host name to initial Build Scenario state. So the default is "empty" which means that the TARG_BUILD_SCENARIO is undefined. If you are on "login1.stampede.tacc.utexas.edu" your default TARG_BUILD_SCENARIO will be "opt". Similarly, any host with "foo.bar.edu" will have a default scenario of "dbg".:

```
BuildScenarioTbl = {
   default              = "empty",
   ["tacc.utexas.edu"] = "opt",
   ["foo.bar.edu"]     = "dbg",
}

ModuleTbl = {
   build_scenario     = { "dbg", "opt", "empty"},
   compiler           = { "intel", "pgi", "gcc", "sun",},
   mpi                = { "mpich", "mpich2", "openmpi", "mvapich2", "impi"},
   solver             = { "petsc","trilinos"},
   profiling          = { "mpiP", "tau"},
   file_io            = { "hdf5", "netcdf", },
}

TargetList = { "mach", "build_scenario", "compiler", "mpi"}

SettargDirTemplate = { "$SETTARG_TAG1", "/", "$SETTARG_TAG2", "$TARG_SUMMARY" }

NoFamilyList = {"mach", "build_scenario"}

TitleTbl = {
   dbg                    = 'D',
   opt                    = 'O',
   impi                   = "IM",
   mvapich2               = 'M',
   openmpi                = "O",
   mpich                  = "M",
   mpich2                 = "M2",
```

(continues on next page)

(continued from previous page)

```
   intel                        = "I",
   gcc                          = "G",
   phdf5                        = "H5"
   hdf5                         = "H5"
}

TargPathLoc = "first"

HostnameTbl = { 2}
```

ModuleTbl connects module names with a category. It is also used to define "build_scenario" which is just words to declare a build state. In other words, in the above table "dbg" and "opt" could be anything. The only hard-wired name is "empty". The category "build_scenario" is also hard-wired. The names of all other categories are not fixed and you are free to add other categories.

This table is also how settarg knows what the names of the compiler and mpi stacks are. If your site uses the name "ompi" for openmpi then the above table will have to be modified to match.

TargetList defines how TARG_SUMMARY is assembled. It is an array of categories. The category "mach" is special it is always defined to be *uname -m* plus on Linux systems it contains the cpu family and model from /proc/cpuinfo. Each piece is concatenated together with "_". If an item is undefined then the extra "_" is removed.

Settarg ships with the order given above, but sites and users can change the order to be anything they like. Also notice that there are many more categories then are listed in TargetList. More on this aspect in the "Custom Configuration" section.

SettargDirTemplate specifies how TARG is assembled. In the case shown above then env. var SETTARG_TAG1 is combined with "/" and SETTARG_TAG2 followed by TARG_SUMMARY. Both "TAG" variables have to be set in the environment. Here we have assumed that SETTARG_TAG1 is "OBJ" and SETTARG_TAG2 is "_". This leads to TARG being:

> TARG=OBJ/_x86_64_06_2d_dbg_gcc-7.1_openmpi-2.2

The NoFamilyList is an array of categories that do not get the FAMILY version. All categories do. For example, if TARG_COMPILER is "gcc/7.1" then TARG_COMPILER_FAMILY is "gcc".

The TARG_TITLE_BAR and TARG_TITLE_BAR_PAREN are strings that could be used in a terminal title bar. Every item in the TARG_SUMMARY is in the TITLE bar variables (except for TARG_MACH). Because the title bar space is limited, TitleTbl is a way to map each item into an abbreviation. The order in which categories appear on the title bar is the same as TargetList. So a title bar with "O G/7.1 O/2.2" would mean that you are in "opt" mode with gcc/7.1 and openmpi/2.2 loaded.

TargPathLoc controls where (or if) $TARG. Note that the environment variable LMOD_SETTARG_TARG_PATH_LOCATION is use to control TargPathLoc. Normally the value of TARG is placed in the PATH at the beginning of your PATH. You can place it at the end of your PATH when TargPathLoc = "last". If TargPathLoc is "empty" then TARG is removed from your path. Actually the rules controlling where TARG goes in your path are slightly more complicated. TargPathLoc controls where $TARG is placed in your path when TARG was not there before. After the first time TARG is added to your path, TARG maintains its relative location.

Finally, HostnameTbl tells settarg how to extract an entry from the full hostname to be used as TARG_HOST. If your host has multiple components then a "2" would say to use the second component as TARG_HOST. So if your hostname is "login1.stampede.tacc.utexas.edu" then TARG_HOST would be "stampede". If HostnameTbl was "{ 3,2}" then TARG_HOST would be "tacc.stampede". If your hostname has a single component then that is used for TARG_HOST.

Custom configuration

Settarg will read up to three separate copies of settarg configuration files. The first one is in the same directory as the settarg command is and is called settarg_rc.lua. The second place is in the user's home directory (if `~/.settarg.lua` exists). Then from the current directory up to "/" it looks for another .settarg.lua (if it exists). It will not re-read the `~/.settarg.lua`. Typically a user should copy the system settarg_rc.lua to their home directory (as `~/.settarg.lua`) and specify the generally desired behavior. Then in top directory of a project place a simple .settarg.lua that specifies how the target list should be put together for that project:

Suppose that TargetList `~/.settarg.lua` is:

```
TargetList  = { "mach", "build_scenario", "compiler", "mpi",}
```

Then in `~/project/a` there is another `.settarg.lua` that just has:

```
TargetList  = { "mach", "build_scenario", "compiler", "mpi", "file_io"}
```

Normally in any directory your TARG will be the default, but in any directory below `~/project/a` TARG will have hdf5 or netcdf if either are loaded.

To see the state of the configuration execute:

```
$ settarg --report
```

Makefile integration

See the `contrib/settarg/make_example` directory and the README.txt inside. That directory contains a simple Makefile and a more complicated one to a way to use $TARG in a Makefile so that all generated files (`*.o` and the executable) are in the $TARG directory.

There are four main points to converting a Makefile to know about settarg. The first is to set the compiler based on `TARG_COMPILER_FAMILY`:

```
CC := gcc
########################################################################
#  Use TARG_COMPILER_FAMILY to set the C compiler name

ifeq ($(TARG_COMPILER_FAMILY),gcc)
   CC := gcc
endif

ifeq ($(TARG_COMPILER_FAMILY),intel)
   CC := icc
endif
```

The second is to set the optimization based on `TARG_BUILD_SCENARIO`:

```
CF := -O2
########################################################################
#  Use TARG_BUILD_SCENARIO to set the compiler options for either
#  debug or optimize.

ifeq ($(TARG_BUILD_SCENARIO),dbg)
  CF := -g -O0
```

(continues on next page)

(continued from previous page)

```
endif

ifeq ($(TARG_BUILD_SCENARIO),opt)
  CF := -O3
endif
override CFLAGS   := $(CFLAGS) $(CF)
```

The third point is to force the make file to use the $TARG directory if defined and change the compilation rules:

```
########################################################################
#  Use O_DIR as equal to $(TARG)/ so that if TARG is empty then O_DIR
#  will be empty.  But if $(TARG) as a value then O_DIR will have a
#  trailing slash.

ifneq ($(TARG),)
  override O_DIR := $(TARG)/
endif


######################## compilation rules ###############################

$(O_DIR)%.o : %.c
        $(COMPILE.c) -o $@ -c $<
```

The four point is that the dependencies have to change to use $(O_DIR):

```
######################## Dependencies ####################################

$(O_DIR)main.o : main.c hello.h

$(O_DIR)hello.o: hello.c hello.h
```

For small projects, generating the dependencies by hand is manageable. But for larger projects it can get unwieldy. The `Makefile` shows how to generate the dependencies automatically.

1.6.30 Improving performance of Lmod

Lmod is written two scripting languages: Lua and TCL. Your site can improve the loading of modules by doing the following steps:

1. Create a system cache file (described at *System Spider Cache*)
2. **ALWAYS** keep the cache file up-to-date.
3. Configure Lmod to use LMOD_CACHED_LOADS=yes

Theses are the most important steps. As long as your site keeps the cache file up-to-day, the above steps will improve performance the most.

If your site has many .version or .modulerc files, your site should consider over time converting them to the .modulerc.lua equivalent. Parsing Lua files always is faster than TCL. Lmod uses a TCL program to convert the TCL into Lua. This means that any TCL file is interpreted twice once by TCL and then by Lua.

Lmod used to make a system call to run the TCL interpreter and capture the output. Lmod 8+ now integrates in a TCL interpreter as part of the Lmod program to improve performance. This improves the time to process the TCL .version or modulefile. But a Lua .modulerc.lua or a lua modulefile is faster because the double interpretation is avoided.

1.6.31 Module Extensions

Lmod provides a way to let users know that a package provides one or more extension. So for example a python package might provide the numpy and scipy software. But how would a user know that a it is accessible from the python module. Sites can create modulefiles that support the **extensions** () function:

```
extensions("numpy/1.12, scipy/1.1")
```

In tcl, this can be written as:

```
extensions numpy/1.12 scipy/1.1
```

The names in the string need to be of the form "name/version" only. The case of the name doesn't matter. Category/Name/Version is not supported. Neither is "Name/Version/Version".

Users can use module spider to learn about what is extensions are available:

```
$ module spider

  ...
  numpy: numpy/1.12 (E), numpy/1.16.4 (E)
  ...
  scipy: scipy/1.1 (E), scipy/1.2.2 (E)
  ...
```

The trailing (E) lets users know that this software is an extension provided by another module and is not a module itself. Also if color is turned on extensions are displayed in blue.

To find out about what extensions exists of a particular software extension users can do:

```
$ module spider numpy

 numpy:
   Versions:
      numpy/1.12 (E)
      numpy/1.16.4 (E)
```

To find out which modules provide the extension a user can do:

```
$ module spider numpy/1.16.4

  numpy: numpy/1.16.4
  This extension is provided by the following modules. To access the extension you must␣
↪load one. Note that any module names in parentheses show the module location in the␣
↪software hierarchy.
    python2/2.7.14
    python2/2.7 (intel/11.2)
    python2/2.7 (intel/11.0)
    python2/2.7 (gcc/4.2.5)
    python2/2.7 (gcc/4.2.3)
```

Users can find out what extensions exist by using the **module avail** command:

```
$ module avail

...
This is a list of module extensions
  numpy (E)    scipy (E)
```

Note that the name of the extension is shown without any version information. The command **module avail** shows modules that can be loaded. A user *cannot* load an extension directly. If a user tries to load an extension directly, Lmod knows that it is an extension and directs the user to use **module spider** to find where the extension is. Again if color is turned on then the names of the extensions are in blue. Sites or users can set the environment variable LMOD_COLORIZE to "no" to turn off color.

1.6.32 A Personal Hierarchy Mirroring the System Hierarchy

Suppose you as a user wants to have personal modules that work as part of the software hierarchy. You might want to do that if you are a library or parallel application developer. You might want to test libraries or parallel applications before making them publically available. These are two possible of many that you might want to mirror the software hierarchy. So how to go about it.

The simplest to understand (but not implement) is to just copy over the entire compiler-mpi tree to your account and then tweek the compiler and mpi modules to point inside your directory. Once you have done that, you can add the modules that you need.

But that it work. Is there something easier to do? Yes there is. There is much easier method (but do not implement this one either). Just copy over the compiler and mpi modules you wish to have in your tree. Then add an additional directory for your packages to prepend to **$MODULEPATH**. However this is still problematical as the site might change the system compiler modules tomorrow but your copy this compiler module will be out of date.

So Lmod provides a way to get the contents of the system compiler which then you can add on to. You can use the *inherit()* function to "import" the same named module in the module tree.

To make things concrete, let's assume that you are a boost developer and the system has a boost library as well. The system boost version is 1.8 and you are working on 1.9. There is a system gcc 9.1 module. You create the following directory: $HOME/my_modules and under there you create Core, Compiler and MPI directories:

```
$ mkdir -p ~/my_modules/{Core,Compiler,MPI}
```

You also set the following environment variable:

```
$ export MY_MODULEPATH_ROOT=$HOME/my_modules
```

When this is set up you will do:

```
$ module use ~/my_modules/Core
```

Then in the file ~/my_modules/Core/gcc/9.1.lua you have:

```
inherit()
local compiler = "gcc"
local MP_ROOT  = os.getenv("MY_MODULEPATH_ROOT")
local version  = "9"

prepend_path("MODULEPATH", pathJoin(MP_ROOT, "Compiler",compiler,version))
```

Suppose you also have the system intel/19.0.5 module. Then you would need at ~/my_modules/Core/intel/19.0.5.lua you have:

```
inherit()
local compiler = "intel"
local MP_ROOT  = os.getenv("MY_MODULEPATH_ROOT")
local version  = "19"

prepend_path("MODULEPATH", pathJoin(MP_ROOT, "Compiler",compiler,version))
```

Inheriting from a marked default

It would be best to mark your versions of the compilers (and mpi modules if they exist) as defaults. Module that are marked defaults are chosen over the highest version. The easiest way to mark module as the default is with a symbolic link:

```
$ cd ~/my_modules/Core/intel; ln -s 19.0.5.lua default
$ cd ~/my_modules/Core/gcc;   ln -s 9.1.lua    default
```

Then to support your personal boost 1.9 module for gcc goes at **~/my_modules/Compiler/gcc/9/boost/1.9.lua**. And the boost 1.9 module for the intel 19.0.5 goes at **~/my_modules/Compiler/intel/19/boost/1.9.lua**

Note that here I have used the major version number for the compiler modules. Here I have assumed that all gcc 9.* version work the same. You can do this or use the full version whichever works for you. For each compiler version you will have to have your own personal version as well.

MPI versions

If you wish to have a full personal compiler-MPI hierarchy mirroring the system one, you need to add mpi stack modulefiles as well. Assuming that you have an impi 19.0.5, you will place at **~/my_modules/Compiler/intel/19/impi/19.0.5.lua**:

```
inherit()
local compiler   = "intel"
local MP_ROOT    = os.getenv("MY_MODULEPATH_ROOT")
local c_version  = "19"
local mpiNm      = impi
local m_version  = "19"

prepend_path("MODULEPATH", pathJoin(MP_ROOT, "MPI",compiler,c_version,mpiNm,m_version))
```

Then your parallel application ACME version 1.3 will have a modulefile found at **~/my_modules/MPI/intel/19/impi/19/acme/1.3.lua**

An example of setting up a user can be found in the source in the rt/user_inherit directory. In the **mf** directory is the “system” directory and the **user_mf** directory is a user supplied tree.

1.6.33 Support Community Modules Collections Safely

Sites may wish to include community based packages use the module system. However not all users at a site want to know about a specialized group of modules. Also the community may provide bugg modules. There is a way to protect your regular users from the different collections.

The goals of this method are the following:

1. All users just see a gateway module for each collection.
2. The modules in the collection do not show up in avail or spider for regular users.
3. To see the collection a user must load the gateway module.
4. After loading the gateway module then will “module avail” and “module spider” show the collection.
5. Optionally sites can provide a spider cache for each separate collection if it is large.

Suppose you have a group that provides a collection of biology packages. Lets call the gateway module “biology”. Here is a prototype module where you are going provide a spider cache for the collection:

```
if ( mode() ~= "spider" ) then
   prepend_path("MODULEPATH",    "/path/to/biology/modulefiles")
end

-- Only do the following if you are providing a spider cache for
-- this collection
prepend_path("LMOD_RC",         "/path/to/biology/lmodrc.lua")
```

If you are providing a spider cache for the collection then your site will need a lmodrc.lua file for the collection:

```
scDescriptT = {
  {
    ["dir"]       =  "/path/to/biology/spider_cache",
    ["timestamp"] = "/path/to/biology/timestamp",
  },
}
```

Finally you need to build the spider cache. This can be done with **update_lmod_system_cache_files**:

```
% $LMOD_DIR/update_lmod_system_cache_files -d /path/to/biology/spider_cache -t /path/to/
↪biology/timestamp /path/to/modulefiles
```

This command to update the cache files must be done every time there are new modulefiles added to the collection and typically has to be run with a privileged account or one that can write to the directories. It may be best if the cache file is regenerated automatically (say every 30 minutes) to keep it up-to-date.

Also it is important that all the paths match between the gateway modulefile, the lmodrc.lua file and command line above.

1.6.34 Checking Syntax in Your Sites Modulefiles

As of Lmod (8.4.3+), it is possible to check the syntax of your sites module tree(s). The command **check_module_tree_syntax** will walk your module directories just like the spider command. But the spider command does not report warning and errors, the **check_module_tree_syntax** command does. This command is meant to be run by a Site's staff and not users as it is designed to catch modulefile errors *before* your users do.

To run do:

```
$LMOD_DIR/check_module_tree_syntax $MODULEPATH
```

This command will report all the modulefiles that have syntax errors.

Check for multiple ways set a marked default

Lmod has multiple ways to mark a default (See *Marking a Version as Default* for more details). With a large module tree it can be difficult to know that a particular a directory that contains multiple modulefiles has more than one file to mark a default.

When this command walks the module tree reading, it also checks for multiple ways to mark a default. For example if a directory has:

> .version 1.0.lua 2.0.lua 3.0.lua default@

Where symlink default points to the file 2.0.lua. The command will report that this directory has multiple files marking a default.

Using the –checkSyntax option to check the syntax of a module

Tools that generate modulefiles may wish to check the syntax of a modulefile before installing it. For example, TACC builds RPMs that include the package and the modulefile. They use Lmod to check the syntax of a modulefile *without* having to have the tree installed.

In other words, this options ignores the Lmod commands like **execute{}**, **load()**, **depends_on()** and other Lmod commands that would require a whole module tree be present.

To use put the modulefile in a sub-directory with just the modulefile in it. Then set $MODULEPATH to point to that sub-directory. Then load the module as follows:

```
module --checkSyntax load <modulefile>
```

Where <modulefile> is modulefile you wish to test.

1.7 Internal Structure of Lmod

1.8 Topics yet to be written

1. Optional Software layout, two digit rule
2. Advanced Topics: priority path,

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search