Lmod Tutorial

Robert McLay

The Texas Advanced Computing Center

April 9, 2011
Introduction

• Fundamental Issues
• Environment Modules
• Software Hierarchy
• Lmod
• Conclusions
Fundamental Issues

- Software Packages are created and updated all the time.
- Some Users need new versions for new features and bug fixes.
- Other Users need older versions for stability and continuity.
- No system can support all versions of all packages.
- User programs using pre-built C++ & Fortran libraries must link with the same compiler.
- Similarly, MPI Applications must build and link with same MPI/Compiler pairing when using prebuilt MPI libraries.
Example of Lmod: Environment Modules (I)

$ module avail

------------------ /opt/apps/modulefiles/MPI/intel/12.0/mpich2/1.4 ------------------
petsc/3.1 (default) petsc/3.1-debug pmetis/4.0 tau/2.20.3

------------------- /opt/apps/modulefiles/Compiler/intel/12.0 -----------------------
boost/1.45.0 gotoblas2/1.13 openmpi/1.4.3
boost/1.46.0 mpich2/1.3.2 openmpi/1.5.1
boost/1.46.1 (default) mpich2/1.4 (default) openmpi/1.5.3 (default)

-------------------------- /opt/apps/modulefiles/Core -------------------------------

PrgEnv intel/11.1 papi/4.1.4
admin/admin-1.0 intel/12.0 (default) scite/2.28
ddt/ddt lmod/lmod tex/2010
dmalloc/dmalloc local/local (default) unix/unix (default)
fdepend/1.2 mk1/mk1 visit/visit
gcc/4.4 noweb/2.11b
gcc/4.5 (default)
Example of Lmod: Environment Modules (II)

$ module list
Currently Loaded Modules:
   1) PrgEnv  2) gcc/4.5  3) mpich2/1.4  4) petsc/3.1
$ module unload gcc
Inactive Modules:
   1) mpich2  2) petsc
$ module list
Currently Loaded Modules:
   1) PrgEnv
Inactive Modules:
   1) mpich2  2) petsc
$ module load intel
Activating Modules:
   1) mpich2  2) petsc
$ module swap intel gcc
Due to MODULEPATH changes the follow modules have been reloaded:
   1) mpich2  2) petsc
Benefits of Modules

- Users do not need to know where software is installed
- Environment Variables to interface package can be set:
  - TACC\_PACKAGE\_LIB
  - TACC\_PACKAGE\_INC
    - User do not need to set hardcoded paths.
- Package required variables such as LICENSE\_PATH can be set automatically.
Benefits of Lmod vs. TCL/C modules

- Lmod provides all the functionality that TCL/C Modules does.
- Directly support for Software Hierarchy.
- It reads both TCL and Lua based modulefiles.
  - Lua module files have “.lua” extension.
  - TCL module files don’t.
- Users can find all modules available via module spider.
- Users can set their default set of modules via module save.
Environment Modules

- The “module” command “loads” a package.
- It adds directories to PATH or LD_LIBRARY_PATH and sets other Variables.
- “module unload package” removes all packages changes: PATH, LD_LIBRARY_PATH, unset vars.
- A module file is a text file containing shell-independent commands:
  - prepend_path("PATH","/opt/apps/git/1.8/bin")
  - setenv("TACC_GIT_DIR","/opt/apps/git/1.8")
Modules and Versions

• Modules are typically named: *package*/version
• For example: git/1.8
• There is a default version so:
  – “module load git” load the default version.
  – “module load git/1.8” load the 1.8 version.
• Typically a module name is a directory name and the version is a file.
Sys-admin Vs. User Control of software

- Sys-admins control what versions are default.
- Users can load newer/older versions instead of the default.
- When compilers are updated, users can switch between versions, allowing for testing.
- This is the key to a flexible system.
- Users can create their own modules for personal software.
Environment Modules History

- Paper described modules in 1991 (Furlani).
- Cray used modules on Unicos mid-1990’s.
- TACC has been using modules since our 1st Cray T3E in the late 90’s.
Environment Modules History (II)

• At some point Environment Modules was rewritten in a TCL/C combination.
• Another module system called CMOD:
• www.modules.org: TCL/C Module files written in TCL (Late ’90s - now)
• A pure TCL based module system: www.modules.org (? - now)
• Lmod: Lua Based Environment Module System (2008 - now) (lmod.sf.net)
How could modules possibly work?

- Child processes inherit the parents environment.
- Not the other way around.
- So how does this work?
The Trick

- The module command `$LMOD_CMD` reads module files.
- The program outputs shell dependent text.
- Second step: evaluate shell dependent text.
- In bash:
  - `module () { eval $(LMOD_CMD bash "$@")}
- In csh module is an alias:
  - `alias module eval `'$LMOD_CMD csh \!*'`
The Trick (II)

Text output of the module command:

- modulefile "foo/1.0.lua":
  ```lua
  setenv("FOO_VERSION","1.0")
  ```

- Output for bash:
  ```bash
  export FOO_VERSION="1.0"
  ```

- Output for csh:
  ```csh
  setenv FOO_VERSION "1.0"
  ```
Interactive Playtime: Shell Startup Debug

- This is to test your VM installs
- Use VM lmod-test, login `mclay`, password: `mclay`
  
  ```bash
  $ cat "export SHELL_STARTUP_DEBUG=1" > $HOME/.init.sh
  $ bash -l
  ```

- This is a great help in installing Lmod and tracking startup bugs.
Software Hierarchy

- TACC used modules from www.modules.org (TCL/C) modules
- Life was good at TACC until ...
- Multiple Compilers and Multiple MPI implementations.
- Pre-built C++ & Fortran libraries must link with the same compiler
- Similarly MPI Applications must build and link with same MPI/Compiler pairing when using prebuilt MPI libraries.
Modulefile Choices

- Flat Naming Scheme
- Hierarchical Naming Scheme
Flat Naming Scheme: PETSc

PETSc is a parallel iterative solver package:

- Compilers: GCC 4.5, Intel 11.1
- MPI Implementations: MVAPICH 1.2, Openmpi 1.5
- MPI Solver package: PETSc 4.1
- 4 versions of PETSc: 2 Compilers $\times$ 2 MPI
Flat: PETSc

1. PETSc-4.1-mvapich-1.2-gcc-4.5
2. PETSc-4.1-mvapich-1.2-intel-11.1
3. PETSc-4.1-openmpi-1.5-gcc-4.5
4. PETSc-4.1-openmpi-1.5-intel-11.1
Problems w/ Flat naming scheme

- Users have to load modules:
  - “intel/11.1”
  - “mvapich/1.2-intel-11.1”
  - “PETSc/4.1-mvapich-1.2-intel-11.1”
  - Changing compilers means unloading all three modules
  - Reloading new compiler, MPI, PETSc modules.
  - Not loading correct modules ⇒ Mysterious Failures!
  - Onus of package compatibility on users!
Hierarchical Naming Schemes

- Store modules under one tree: /opt/apps/modulefiles
- One strategy is to use sub-directories:
  - Core: Regular packages: apps, compilers, git
  - Compiler: Packages that depend on compiler: boost, MPI
  - MPI: Packages that depend on MPI/Compiler: PETSc, TAU
MODULEPATH

- MODULEPATH is a colon separated list of directories containing directories and module files.
- No modulefiles loaded ⇒ users can only load core modules.
- Loading a compiler module adds to the MODULEPATH:
  - Users can load compiler dependent modules.
  - This includes MPI implementations modules.
- Loading an MPI module adds to the MODULEPATH:
  - Users can load MPI libraries that match the MPI/compiler pairing.
Hierarchical Examples: Core

• Generic:
  – Package: /opt/apps/package/version
  – M: /opt/apps/modulefiles
  – Modulefile: $M/Core/package/version

• Git 1.8
  – Package: /opt/apps/git/1.8
  – Modulefile: $M/Core/git/1.8

• Intel compilers 11.1
  – Package: /opt/apps/intel/11.1
  – Modulefile: $M/Core/intel/11.1
  – Modulefile adds $M/Compiler/intel/11.1 to MODULEPATH
Hierarchical Examples: Compiler Dependent

- **Generic:**
  - Package: /opt/apps/compiler-version/package/version
  - M: /opt/apps/modulefiles
  - Modulefile: $M/Compiler/compiler/version/package/version

- **Openmpi 1.5 with gcc 4.5**
  - Package: /opt/apps/gcc-4.5/openmpi/1.5
  - Modulefile: $M/Compiler/gcc/4.5/openmpi/1.5
  - Modulefile adds $M/MPI/gcc/4.5/openmpi/1.5 to MODULEPATH

- **Openmpi 1.5 with intel 11.1**
  - Package: /opt/apps/intel-11.1/openmpi/1.5
  - Modulefile: $M/Compiler/intel/11.1/openmpi/1.5
  - Modulefile adds $M/MPI/intel/11.1/openmpi/1.5 to MODULEPATH
Hierarchical Examples: MPI/Compiler Dependent

• PETSc 4.1 (1)
  – Package: /opt/apps/intel-11.1/openmpi-1.5/petsc/4.1
  – Modulefile: $M$/MPI/intel/11.1/openmpi/1.5/petsc/4.1

• PETSc 4.1 (2)
  – Package: /opt/apps/gcc-4.5/mvapich-1.2/petsc/4.1
  – Modulefile: $M$/MPI/gcc/4.5/mvapich/1.2/petsc/4.1
Loading the correct module

- User loads “intel/11.1” module
- Can only see/load compiler dependent packages that are built with intel 11.1 compiler.
- Can not see/load package built with other versions or other compilers.
- Similar loading “openmpi/1.5” module.
- User can only load package that are built w/ intel 11.1 and openmpi 1.5 and no others.
Better but ...

• Using TCL/C modules, Users can load correct modules by using the Software Hierarchy.

• But swapping compilers or MPI stack ⇒ complicated!

• For Parallel libraries like PETSc:
  – Users must unload PETSc, MPI, compiler
  – Reload compiler, MPI, PETSc
  – Nobody got this right!

• Solution: Yet another Environment Module System: Lmod
Lmod

- Complete Rewrite of the Environment Modules System.
- Reads TCL or Lua modulefiles.
- Based on the Lua scripting language.
- Simple yet powerful with:
  - Functions are first class objects.
  - Simplifies loading/unloading of modules.
  - Tables combine array and hash seamlessly.
Key Insight (I): MODULEPATH

- Lmod remembers the current state of MODULEPATH.
- If it changes then it unloads any modules not in current search path ⇒ inactive.
- It tries to activate any inactive modules.
- It remembers inactive modules.
- It continues to attempt to activate any inactive modules on future invocations.
Key Insight (II): MODULEPATH

- Loading gcc/4.5 and boost/1.47.1
- M=/opt/apps/modulefiles
  - Adds $M/Compiler/gcc/4.5 to MODULEPATH.
  - Boost: $M/Compiler/gcc/4.5/boost/1.47.1
- Unloading gcc/4.5
  - Removes $M/Compiler/gcc/4.5 from MODULEPATH.
  - Inactivates boost/1.47.1
- Loading intel/11.1
  - Adds $M/Compiler/intel/11.1 to MODULEPATH.
  - Activates Boost: $M/Compiler/intel/11.1/boost/1.47.1
Other Safety Features of Lmod (I)

- Users can only load one version of a package.
- "module load xyz/2.1" loads xyz version 2.1
- "module load xyz/2.2" unloads 2.1, loads 2.2
Other Safety Features of Lmod (II)

- Lmod adds a new command in modulefiles: `family("name")`
- All of our compiler modules have `family("compiler")`
- All of our MPI modules have `family("MPI")`
- Users can only load one compiler or MPI at a time
- Powers users can get around this restriction.
Save/Restore

- User can setup their own initially loaded modules.
  - Users simply load, unload and/or swap until happy.
  - `module save` saves state in “default”
  - Our login scripts do: `module restore` which loads the user’s default.

- Users can create other collections by:
  - `module save name` to save it.
  - `module restore name` to retrieve it.

- This used to known as setdefault/getdefault.
Lmod Module Layout

• Supports Flat layouts
• Supports Hierarchical layouts
• Naming Schemes:
  – Name/Version (e.g. bowtie/2.3)
  – Category/Name/version (e.g. bio/bowtie/2.3)
  – Category/Sub/Name/version (e.g. bio/genomics/bowtie/2.3)
For those who can’t type: “ml”

- `ml` is a wrapper:
  - With no argument: `ml` means `module list`
  - With a module name: `ml foo` means `module load foo`
  - With a module command: `ml spider` means `module spider`

- See `ml --help` for more documentation.
Interactive Playtime: Lmod example

- Use VM lmod-test, login *mclay*, password: *mclay*
  
  $ module load gcc mpich parmetis
  $ module list
  $ module avail
  $ module swap gcc clang
  $ module spider
  $ module spider parmetis/4.0.3
Interactive Playtime: Lmod example (II)

$ ml purge
$ ml gcc mpich parmetis
$ ml
$ ml -gcc clang
$ ml spider
$ ml spider parmetis/4.0.3
Searching for Modules

• Three ways to search for modules:
  - “module avail”
  - “module spider”
  - “module keyword”

• The “avail” command reports all “loadable” modules.
• The “spider” command reports all “possible” modules.
• The “keyword” command reports all “possible” modules that match keywords.
• What is the difference?
Module avail

- “avail” only reports modules that are loadable with current MPI/Compiler pairing.
- A parallel library may not be built for all possible pairings.
- Won’t always show with avail.
- Not all package can be build or work with all compiler/MPI suites.
Module spider

• Reports all modules for given MODULEPATH.
• It recursively searches the tree to find all branches.
• Large systems of modules save a cache file that is saved for a day.
• Three modes:
  – module spider - all possible modules no detail.
  – module spider petsc - all versions of petsc, no detail.
  – module spider petsc/3.1 - details.
Module keyword key1 key2 ...

- Modules can have a "whatis" description:
  
  ```
  whatis("Name: Abyss")
  whatis("Version: 1.2.7")
  whatis("Category: computational biology, genomics")
  whatis("Keywords: compbio, genomics")
  whatis("URL:http://www.bcgsc.ca/platform/bioinfo/software/abyss")
  whatis("Description: Assembly By Short Sequences.")
  ```

- module keyword key1 ... will report all modules with any of the key's.
Generic Modulefile support

- Lmod support several functions to help with generic modulefiles:
  - myModuleName() ⇒ boost, bio/bowtie
  - myModuleVersion() ⇒ 1.47.0
  - myModuleFullName() ⇒ boost/1.47.0
Generic Modulefile support: hierarchyA()

- Extracting the hierarchy based on location: mpich/3.1.lua

  ```lua
  local pkgName = myModuleFullName()
  local hierA = hierarchyA(pkgName,1)
  local comp = hierA[1]
  local compDir = comp:gsub("/","-"):gsub("%.","_")
  local base = pathJoin("/opt/apps", compDir, pkgName)
  prepend_path("PATH", pathJoin(base,"bin"))
  ```

- If you use a different layout, you can provide similar functions in “SitePackage.lua”
Interactive Playtime: Generic Modulefiles

• Lets look at one way to support Generic Modulefiles:

```bash
$ cd /opt/apps/modulefiles/Core; more gcc/4.8.lua
$ cd ../Compiler/
$ ls -R *
$ cd .base/mpich/
$ look at 3.1.lua
$ cd ../../MPI/.base/parmetis
$ look at 4.0.3.lua
```
Installation Overview

- Chose a place for module tree: "/opt/apps/modulefiles"
- Use configure to override default.
- Install lua and Lmod applications.
- Place module command in system shell startup.
- Possibly modify bash’s startup procedure.
- Design a default set of modules for your users.
Installing Lmod (I)

- Download files from lmod.sf.net
  - lua-x.y.z.tar.gz
  - lmod-x.y.z.tar.gz
- Assume all “optional” packages are in “/opt/apps/”
- Install lua in “/opt/apps/lua/x.y.z”
- “$ ln -s x.y.z /opt/apps/lua/lua”
- This way “/opt/apps/lua/lua/bin/lua” always points to lua independent of version.
Installing Lmod (II)

- Install Lmod in “/opt/apps/lmod/x.y.z”
- “make install” creates a symlink from “x.y.z” to lmod.
- This way “/opt/apps/lmod/lmod/...” always points to the latest lmod.
Integrating Lmod into User’s Shell (I)

We must add the module alias for all user’s shells by either link or copy:

- `$ 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`
- You may have to create “/etc/profile.d” first.
## Bash Shell startup files

<table>
<thead>
<tr>
<th></th>
<th>System</th>
<th>User</th>
</tr>
</thead>
<tbody>
<tr>
<td>login</td>
<td>/etc/profile</td>
<td>~/.bash_profile</td>
</tr>
<tr>
<td>interactive</td>
<td>/etc/bashrc*</td>
<td>~/.bashrc</td>
</tr>
<tr>
<td>non-interactive</td>
<td>$BASH_ENV</td>
<td></td>
</tr>
</tbody>
</table>

* Not always built-in!

$BASH_ENV points to a file which is run on non-interactive shells.
# Csh Shell startup files

<table>
<thead>
<tr>
<th>System</th>
<th>User</th>
</tr>
</thead>
<tbody>
<tr>
<td>login</td>
<td>~/.cshrc &amp; ~/.login</td>
</tr>
<tr>
<td>interactive</td>
<td>~/.cshrc</td>
</tr>
<tr>
<td>non-interactive</td>
<td>/etc/csh.cshrc</td>
</tr>
</tbody>
</table>
if [ -d /etc/profile.d ]; then
  for i in /etc/profile.d/*; do
    if [ -r $i ]; then
      . $i
    fi
  done
fi
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
fi
Linux /etc/csh.cshrc

if ( -d /etc/profile.d ) then
  set nonomatch
  foreach i ( /etc/profile.d/*.csh )
    if ( -r $i ) then
      if ($?prompt) then
        source $i
      else
        source $i >& /dev/null
      endif
    endif
  endif
end
endif
endif
endif
end
endif
Rebuilding Bash to use /etc/bashrc on non-login shells

• By default bash does not read /etc/bashrc (or similar)
• Debian and Ubuntu read /etc/bash.bashrc on interactive non-login shells.
• Red Hat, Centos, Mac OS X, don’t read /etc/bashrc
• At TACC, we rebuilt bash so that it does read /etc/bashrc
• You must patch config-top.h to change bash’s behavior.
Rebuilding Bash Benefits

• We want bash interactive non-login shells to behave the same as login shells.
• Fortran 90 program typically need “ulimit -s unlimited”
• MPI invokes an interactive non-login, non-prompt shell on every node.
• bash users were not getting “ulimit -s unlimited” from system /etc/profile.d/*.sh
• By rebuilding Bash we guarantee that MPI jobs source /etc/profile.d/*.sh
• We control what goes into /etc/tacc/{profile,bashrc}
Loading Default Modules

- At TACC, all users get a default set of module loaded at startup.
- In "/etc/profile.d/z96_login_modules.sh:"
  
  ```bash
  export LMOD_SYSTEM_DEFAULT_MODULES=TACC
  module --initial_load load restore
  ```
- Most users get the “TACC” set of modules
- We load standard tools, a compiler, and an MPI implementation.
- Users can replace the default via "module save"
Modules and Package Management at TACC

- All of the “optional” software: compilers, MPI Stacks, Libraries, and Applications are installed via the RPM package manager.
- We create our optional RPMs with both the software and the module files.
- Uninstalling a package removes both the software and the module that access it.
- We use a single parameterized RPM spec file to build all compiler/MPI pairings.
Management of Modules

- Encourage users to use modules with their own software.
- Check modulefile syntax errors by running: `module spider`
- Use `prereq("foo","bar")` instead of `load("foo","bar")`
- Use `family("compiler")` and similar in your compiler and MPI module files.
Deploying Lmod

- Lmod can be tested even though your site runs TCL/C modules.
- Make sure that Lua and lua-posix and lua-filesystem are installed.
- Otherwise install lua-5.1.4.8.tar.gz from lmod.sf.net.
- Install Lua, Install Lmod in your account.
Deploying Lmod: Personal startup

```bash
if [ -z "$LMOD_CMD" ]; then
    CURRENT_MPATH=$MODULEPATH
    module purge 2> /dev/null # purge using old cmd

    LMOD_PKG=$HOME/pkg/lmod/lmod
    LMOD_CMD=$LMOD_PKG/libexec/lmod
    export BASH_ENV=$LMOD_PKG/init/bash
    . $BASH_ENV # redefine module cmd
    export LMOD_SYSTEM_DEFAULT_MODULES=...
    MODULEPATH=$CURRENT_MPATH
    module --initial_load restore
fi
```
Opt-in Testing

- You and others can then opt-in

```bash
if [ -d $HOME/.lmod.d ]; then
  # Use Lmod for Modules
  ...
else
  # Use TCL/C modules
  ...
fi
```
Opt-out deployment

- When read to deploy to users do:

  ```bash
  if [ ! -f $HOME/.no.lmod ]; then
    # Use Lmod for Modules
    ...
  else
    # Use TCL/C modules
    ...
  fi
  ```
Module usage

- It is possible to record module usage:
  - Record modules loaded at logout time and/or job submission.
  - Find out what modules are not or under used. \( \Rightarrow \) removal or upgrade.
  - Suggest to users of one module to consider using another module.

- Use “load” hook to record every loaded module via SitePackage.lua
HPC issues: Root

- Root should not define the module command or load a default set of modules during shell startup.
- No non-local path should automatically ever be in root’s path.
- What if /opt/apps/ is unavailable?
HPC issues: Startup Files and Compute nodes

- At TACC: the environment variable: ENVIRONMENT is “BATCH” when on a compute node.
- When in BATCH mode:
  - The module command is defined.
  - The TACC module is NOT loaded.
  - The user’s environment is passed to all processes.
  - Mvapich2: mpirun_rsh ... Var1=V1 Var2=V2 ...
  - Where Var1 ... are env. vars such as $HOME, $PATH ...
- This prevents every node from a user’s parallel job from doing “opendir(”/opt/apps/modulefile/Core”)”
HPC issues: Users

• Users can overwork the parallel file system.
• Encourage your users to use module save
• Or users should load their modules via ~/.bashrc or ~/.cshrc but wrap them:

```bash
if [ -z $_READ -a -z $ENVIRONMENT ]; then
    export _READ=1 # Put any module commands here:
    module load git
fi
```

• Otherwise each node of a user’s 8192 process job will try to load modules ⇒ Parallel file system metadata servers can hang.
**Version Sorting**

- TCL/C module and older Lmod sorted alphabetically:
- intel/9.0 “was” newer than intel/10.0 ⇒ Yuck!
- Python has a clever scheme. I’ve re-implemented it in Lua.
- Lmod 4.1+ uses version sorting:
  - 2.4dev1: 00002.00004.*@.00001.*zfinal
  - 2.4a1: 00002.00004.*a.00001.*zfinal
  - 2.4rc1: 00002.00004.*c.00001.*zfinal
  - 2.4: 00002.00004.*zfinal
  - 2.4-1: 00002.00004.*zfinal-.00001.*zfinal
  - 2.4.1: 00002.00004.00001.*zfinal
  - 3.2-static: 00003.00002.*static.*zfinal
  - 3.2: 00003.00002.*zfinal
Load/Prereq modify functions

- In Lua modulefiles you can now do:
  
  ```lua
  load(atleast("FOO","2.3"))
  load(between("BAR","7.1","10.1"))
  load(latest("BAZ"))
  prereq(atleast("boost","1.47.0"))
  ```

- I’m thinking about similar modify functions for conflict()
Module Properties (I)

• Modules can have properties
• At TACC, we have MIC, and GPU accelerators.
• Some libraries are MIC aware.
  
  add_property("arch","mic")

• This is controlled by the table in .lmodrc.lua
Module Properties (II)

- Some modules will be “MIC” aware: mkl, fftw3, phdf5, ...
- Lmod will decorate these modules:

  1) unix/unix  3) ddt/ddt  5) mpich2/1.5  7) phdf5/1.8.9 (m)
  2) intel/13.0  4) mkl/mkl (*)  6) petsc/3.2  8) PrgEnv

Where:
(m): module is build natively for MIC
(*) : module is build natively for MIC and offload to the MIC

------
add_property("arch","mic") -- > phdf5
add_property("arch","mic:offload") -- > mkl

- What properties would you like to support?
Module Properties (III): Sticky

- A module can be sticky.
- It requires "--force" to unload or purge.
  
  add_property("lmod","sticky")
pushenv

- Suppose you’d like to set CC in the environment.
- setenv won’t work.
- pushenv will!

```bash
$ module load gcc;  # -> CC=gcc  CC=gcc
$ module load mpich; # -> CC=mpicc  CC=mpicc
$ module unload mpich; # -> CC is unset  CC=gcc
$ module unload gcc;  # -> CC is unset  CC is unset
```

- pushenv keeps an private env vars: __LMOD_STACK_NAME
always_load and always_unload

- The load() function is reversed on unload.
- The always_load() function is a no-op on unload.
- Still you may want:
  ```lua
  if (not isloaded("FOO")) then always_load("FOO") end
  ```
- Which is only slightly better than:
  ```lua
  if (not isloaded("FOO") and mode()== "load") then
    load("FOO")
  end
  ```
prepend_path() takes priorities

- When you need to push a path to the front of the line do:
  
  ```
  prepend_path{"PATH","/usr/local/first",
  priority=1000} -- Lua
  prepend-path PATH /usr/local/first 1000 # tcl
  ```

- This is great for wrapper scripts:
- “Do not use mpirun on login nodes”
- Works for append_path() as well.
- Drives paths to be last.
SitePackage.lua: Customize Behavior for your Site

- Use StandardPackage.lua as a guide for your SitePackage.lua
- Many examples of SitePackage.lua in contrib/* directories
local hook = require("Hook")

function load_hook(t)
  if (mode() ~= "load") then return end
  local user = os.getenv("USER")
  local jobid = os.getenv("PBS_JOBID") or "unknown"
  local msg = string.format("user=%s,module=%s,job=%s",
                            user, t.modFullName, jobid)
  os.execute("logger -t lmod -p local0.info " .. msg)
  dbg.fini()
end

hook.register("load",load_hook)
settarg

- Provides safety, flexibility and repeatability in a dynamic environment.
- Dynamically updates the state when modules change:

```
$ env | grep '^TARG'
TARG_BUILD_SCENARIO=dbg
TARG=OBJ/_x86_64_dbg_gcc-4.6_mpich-3.0
TARG_MPI_FAMILY=mpich
TARG_MPI=mpich-3.0
$ module swap mpich openmpi; opt; env | grep '^TARG'
TARG_BUILD_SCENARIO=opt
TARG=OBJ/_x86_64_opt_gcc-4.6_openmpi-1.6
TARG_MPI=openmpi-1.6
TARG_MPI_FAMILY=openmpi
```
settarg (II)

- Typically TARG is OBJ/$ARCH$_$SCENARIO$_$CMPLR$_$MPI
- TARG=OBJ/_x86_64_dbg_gcc-4.6_mpich-3.0
- User can extend this with user level or directory level specialization.
- OBJ/_x86_64_dbg_intel-14.0_mpich-3.0_petsc-3.4
- A makefile can modified to write generated file into $TARG.
- Never need to “make clobber” when switching scenario, compiler, etc.
Interactive Playtime: settarg

$ ml
$ cd ~/w/hello
$ ml clang mpich
$ make; mpirun -n 2 hello
$ ml -clang gcc
$ make; mpirun -n 2 hello
$ dbg
$ make; mpirun -n 2 hello
$ opt
$ make; mpirun -n 2 hello
$ cd OBJ
Spider Cache Advantages

- The spider cache speeds up avail and spider greatly.
- All system modulefiles have been read, properties determined.
- Lua is quite fast and reading and interpreting a single file.
- This is preferable to walking the directory tree and reading every module.
- Why is every module file read: properties.
Spider Cache Disadvantages

- There is only one: Keeping it up-to-date.
- If Lmod sees a valid cache file it assumes it is correct.
- Otherwise what’s the point.
- Currently loads bypass cache but avail and spider depend on it.
- Personal modules are not effected by system cache foo.
Spider Cache Implementation

- There is a cache directory
- A time stamp file that marks the last update date.
- You can have 1 or more of these pairs.
Spider Cache Building Strategies

- Common tool to install (like losf)
  - Use it to rebuild the cache when necessary
- Using rpm or dpkg or similar to install software:
  - Wrap rpm or dpkg to update the time-stamp file
  - A cron job to rebuild the cache when necessary
- Others?
Imagine you are developer of a parallel library
You are on a system deploying a software hierarchy
How to take advantage of the system supplied layout?
Inherit (II)

• Create a personal hierarchy for each compiler and MPI stack you want to test against.
• Then copy each system compiler and MPI modulefile into your personal hierarchy.
• Then add a `prepend_path("MODULEPATH",...)` at the end.
Inherit Compiler Module

- Or you can “inherit” the same named module in the hierarchy.
- For a compiler modulefile it can be:

  ```lua
  inherit()
  local MyMRoot = os.getenv("MY_MODULE_ROOT")
  local compN = myModuleName()
  local compV = myModuleVersion():match("(%d+%.%d+)%.?")
  prepend_path("MODULEPATH", pathJoin(myMRoot,
                       "Compiler",compN,compV))
  ```

- Using `$MY_MODULE_ROOT/Compiler/C/CV`
Inherit MPI Module

- Or you can “inherit” the same named module in the hierarchy.
- For a compiler modulefile it can be:

  ```lua
  inherit()
  local MyMRoot = os.getenv("MY_MODULE_ROOT")
  local pkgName = myModuleName()
  local pkgV = myModuleVersion():match("(%d+\.%d+).?.")
  local hierA = hierarchyA(pkgName,1)
  local comp = hierA[1]
  prepend_path("MODULEPATH", pathJoin(myMRoot,
                                          "Compiler",comp, pkgName, pkgV))
  ```

- Using $MY_MODULE_ROOT/Compiler/C/CV/M/MV
Lessons Learned

- A very small percentage of Lmod users join the mailing list.
- Able support many requests for features but not all.
- Hardening:
  - sandbox for evaluating modulefiles
  - Checking argument types
  - checkModuleSyntax script
- Users sometimes ask the right question.
- Lmod is much better product because of its wider use.
Lessons Learned (II)

• It is hard work developing a users’ communities trust.
• Some site will be managed in a way that I have never dreamed of.
• I struggle with mistakes in design that I have to live with:
  – is_spyder ⇒ mode()
  – set_default, get_default ⇒ save, restore
  – duplicate paths
• Deprecating features is difficult with Lmod.
• Users would get deprecated feature reports that sys-admins must fix.
Regression Testing of Lmod

• A suite of 60+ tests each with many steps.
• No release without passing all those tests.
• These tests make Lmod re-factoring much easier.
• The github repo is generally safe.
Remote Debugging

- No software over ten lines is bug free.
- Lmod is no exception.
- Bug reports are as easy as:
  - `module --config 2> config.log`
  - `module -D avail 2> avail.log`
- Sometimes I’ll create test versions with more debugging for you to test.
Recommendations for Good Site Management

- A module purge should not break things.
- Set `LMOD_SYSTEM_DEFAULT_MODULES` so that module restore works.
- Encourage the use of ”module save” so that users can have a default set of modules.
- Use the family directive to protect users.
- Consider the use of ”checkModuleSyntax” before installing new modules.
Follow on projects from Lmod: Lariat

- At TACC, we want to know what packages and libraries are user use.
- All parallel jobs at TACC use the script “ibrun”.
- Lariat adds two tools to ibrun.
  - checkExec
  - parseLDD
CheckExec

- At TACC, your build environment and your submit environment must match.
- checkExec does an ldd of your executable looking for particular MPI libraries.
- It then uses the “reverse map” to map MPI library to MPI module and compiler module.
- It also extract your current environment to see if they match.
- It generates a warning if they don’t.
ParseLDD

• The program parseLDD also does an `ldd` of your executable.
• It checks to see if the executable or the any of the shared libraries are TACC built modules.
• It records information for later analysis.
Lariat $+$ ALTD $\Rightarrow$ XALT

- Mark Fahey developed ALTD which records similar information.
- Mark and I won an NSF Grant to create XALT.
- Alpha users soon, Beta users summer of 2014.
Future Plans

- Make Lmod available as rpm and debian packages.
- How to deal with libraries and applications that do not fit neatly into the hierarchy.
  - is a framework for solving 1D, 2D, 3D grid in parallel with support for AMR.
  - It depends on boost, petsc, trilinos, grvy, ...
  - If you are a developer how do you test it against multiple version of boost, petsc, etc.
  - I’m still thinking about how to handle this “Matrix” dependency.
  
-
Conclusions

- Download source from: lmod.sourceforge.net (lmod.sf.net)
- Github repo: https://github.com/TACC/lmod.git
- Documentation: www.tacc.utexas.edu/tacc-projects/lmod
- Mailing list: lmod-users@lists.sourceforge.net
Projects to work on here

- Install lua-5.1.4.8.tgz in the mclay account.
- Install Lmod in mclay account
- Use your personal version of Lmod.
- Create Personal modules.
- Generic Module file discussion.
- Talk about Software install issues at your site.