Lmod or how to protect your sanity from dependency hell

Steffen Müthing

Interdisciplinary Center for Scientific Computing
Heidelberg University

Dune User Meeting
Aachen
September 26, 2013

The Issue

- DUNE has a number of non-packaged dependencies
- Some of those contain libraries that are bound to a compiler / MPI version
- You have to support multiple compilers / MPIs
  - Library developer (we *try* to be good about this...)
  - Clusters with different compiler / MPI combinations
- Easily switch between release / debug version of MPI (only with dynamic linking)
- Use GCC in general, but switch to clang during the “fix compilation errors” stage

The Issue

- DUNE has a number of non-packaged dependencies
- Some of those contain libraries that are bound to a compiler / MPI version
- You have to support multiple compilers / MPIs
  - Library developer (we *try* to be good about this...)
  - Clusters with different compiler / MPI combinations
- Easily switch between release / debug version of MPI (only with dynamic linking)
- Use GCC in general, but switch to clang during the “fix compilation errors” stage

⇒ Have to keep around multiple versions of MPI, BLAS, ParMetis, ALUGrid, UGGrid, . . .

Problems

- Do I already have ALUGrid for MPICH?
- If yes, where on earth did I put it?
- Did I really build it with the correct dependencies?
- Why does my build fail? Do all the libraries in my nice `--with=` actually work together?

Solutions

- Roll your own package management
    - Invent a custom directory structure / naming scheme for packages
    - Stick to it!

Solutions

- Roll your own package management
    - Invent a custom directory structure / naming scheme for packages
    - Stick to it!
- Look around for something that's already there
    - Distribution package managers (APT, rpm, Portage, MacPorts, homebrew,. . .)
        - Too heavywheight
        - Little support for installing multiple versions of a (development) package

Solutions

- Roll your own package management
  - Invent a custom directory structure / naming scheme for packages
  - Stick to it!
- Look around for something that's already there
  - Distribution package managers (APT, rpm, Portage, MacPorts, homebrew,. . .)
    - Too heavywheight
    - Little support for installing multiple versions of a (development) package
  - Environment Modules
    - Typically used on compute servers + clusters
    - Built to solve exactly our problem
    - Small + simple

Outline

1. Overview
2. Lmod
3. Writing Modules
4. Conclusion

Working principle

- Install every package with a different `--prefix`
  - Typically easy to do with autotools / CMake
  - Exotic build systems (e.g. SuperLU) require extra work
- Update environment variables to make sure headers / libraries / manpages will be found
- Write one *module file* per supported version of a package:

```
$MODFILE_ROOT/gcc/4.7.3.lua
$MODFILE_ROOT/gcc/4.8.1.lua
$MODFILE_ROOT/git/1.8.3.lua
$MODFILE_ROOT/git/1.8.4.lua
```

Implementations

- modules
    - started in ’91
    - Tcl + C
    - Module files written in TCL
    - `http://modules.sourceforge.net`

Implementations

- modules
  - started in ’91
  - Tcl + C
  - Module files written in TCL
  - `http://modules.sourceforge.net`
- modules
  - Started in ’04
  - Bash / tcsh
  - Module files written as shell scripts with custom functions
  - `https://computing.llnl.gov/?set=jobs&page=dotkit`

Implementations

- modules
    - started in ’91
    - Tcl + C
    - Module files written in TCL
    - `http://modules.sourceforge.net`
- modules
    - Started in ’04
    - Bash / tcsh
    - Module files written as shell scripts with custom functions
    - `https://computing.llnl.gov/?set=jobs&page=dotkit`
- Lmod
    - Reimplementation of modules in Lua
    - Oldest public version: ’08
    - Module files written in Tcl or Lua
    - `http://www.tacc.utexas.edu/tacc-projects/mclay/lmod`

Lmod - Basic facts

- developed by Robert McLay (Texas Advanced Computing Center)
- minimal dependencies (Lua + 2 extension modules)
- Goals
    - Clean up implementation
    - Performance
    - Support module hierarchies
    - Module collections
- Easy installation
    - `./configure --prefix=...` and `make install`
    - Source shell-specific startup script in `.bashrc` or your equivalent

Usage

Central command: `module` or (shorter) `ml`

- Show list of loaded modules: `ml`

```
$ ml
Currently Loaded Modules:
  1) macports/default   3) tbb/4.1_4-cpf
  2) gcc/4.8.1_3-mp     4) psurface/1.3.1
```

- Load a module: `ml mpich`
- Load a specific version: `ml mpich/3.0.4`
- Unload a module: `ml -mpich`
- Show currently loadable modules: `ml avail`
- Show all modules: `ml spider`
- Short info about module: `ml whatis mpich`
- Save current set of modules: `ml save mymodules`
- Load set of modules: `ml restore mymodules`

Writing module files

- Module files are Lua scripts
- Restricted by sandbox – can only call registered functions
- Set of module-specific functions to create a kind of DSL
- Full power of Lua available (conditions, loops, data types, . . .)

Environment setup

- Extend `PATH`-like variables
  - `PATH` - For executables
  - `CPATH` - For C include files
  - `LIBRARY_PATH` - For build-time linking
  - `LD_LIBRARY_PATH` - For run-time linking
  - `PKG_CONFIG_PATH` - search path for pkconfig files
  - `MANPATH` - man pages
  - `INFOPATH` - info pages
  - `PYTHONPATH` - python packages
  - …
- Set scalar variables
  - Package root path
  - License file location
  - Flags controlling package operation
  - …

Example

```
...
local install_path = ...

setenv("UG_DIR",install_path)

prepend_path("PATH", pathJoin(install_path,"bin"))
prepend_path("CPATH", pathJoin(install_path,"include"))
prepend_path("LIBRARY_PATH", pathJoin(install_path,"lib"))
prepend_path("DYLD_LIBRARY_PATH", pathJoin(install_path,"lib"))
prepend_path("PKG_CONFIG_PATH", pathJoin(install_path,"lib/pkgconfig"))
prepend_path("CMAKE_MODULE_PATH", pathJoin(install_path,"lib/cmake"))

...
```

Dependency management

Simple features

- A depends on B
- A depends on one of (B1,B2,B3)
- A requires versions x.y of B
- A conflicts with B
- A belongs to *family* F (e.g. compiler, MPI)
- Supported by all module managers

Dependency hierarchies

- Disambiguate multiple installations of the *same package version* due to different dependencies
- Lmod-specific

Example

```
-- This module loads a compiler
family("compiler")

-- Require packA and packB
prereq("packA","packB")

-- Require ParMetis 4.0.3
prereq("parmetis/4.0.3")

-- Require at leat packC or packD
prereq_any("packC","packD")

-- Don't load this with ICC
conflict("icc")
```

Dependency hierarchies

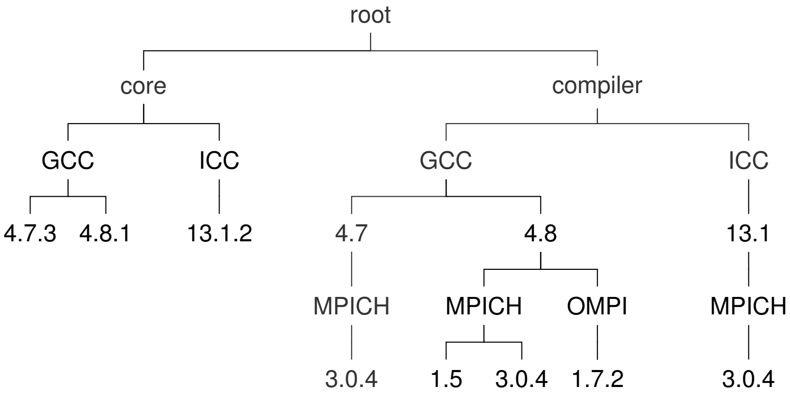
Problem

- ParMetis depends on MPI, OpenBLAS depends on compiler (OpenMP), . . .
- Multiple vendors / versions of base packages:
  - Compiler: GCC, clang, ICC
  - MPI: OpenMPI, MPICH, MVAMPICH
- How to make sure correct version of dependent package gets loaded?

Dependency hierarchies

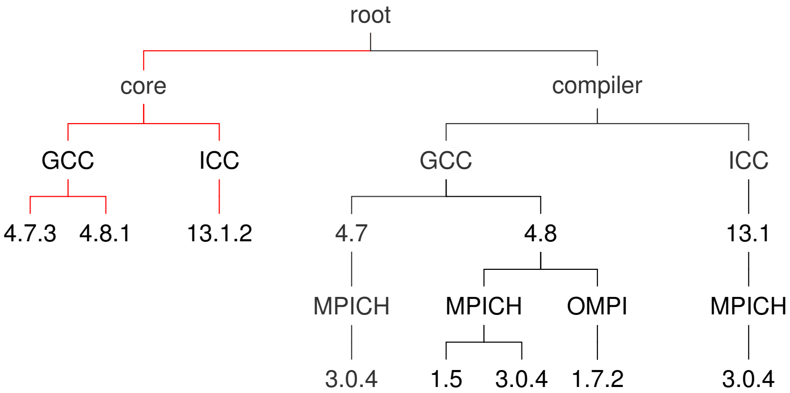
- Multiple trees of module files
- Only subset of module files active
- Modules activate additional trees when loading

Example



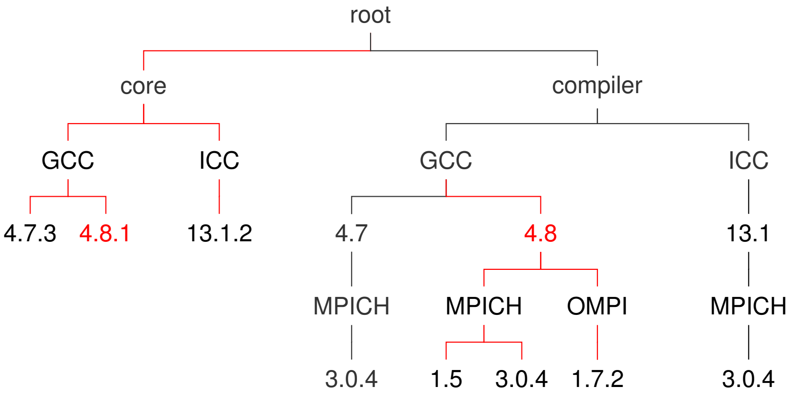
- Directory structure

Example



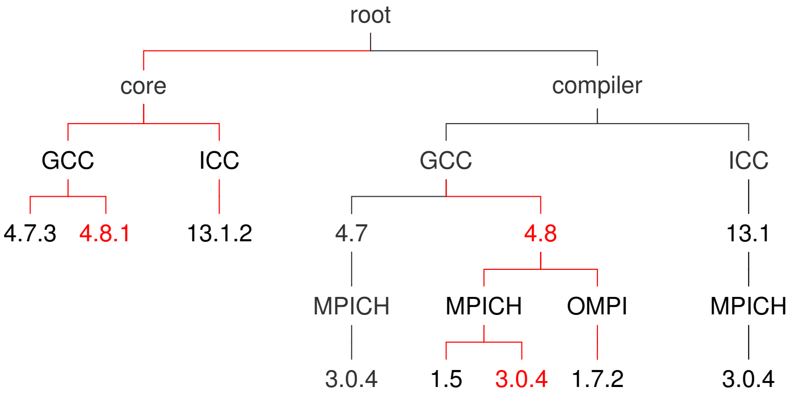
- Directory structure – core modules always active

Example



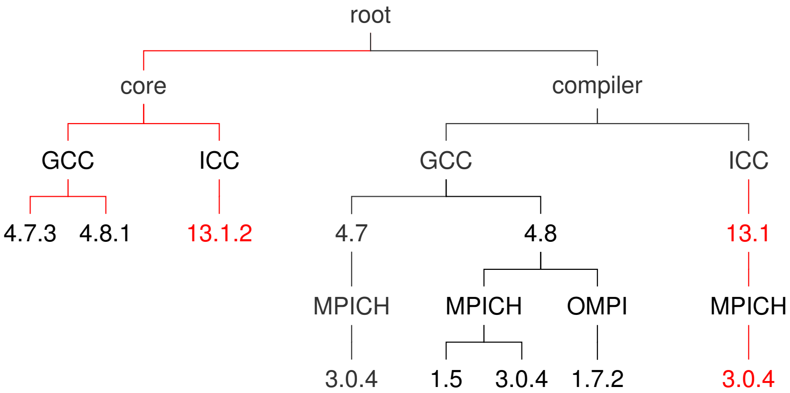
- Directory structure – core modules always active
- Load GCC 4.8.1

Example



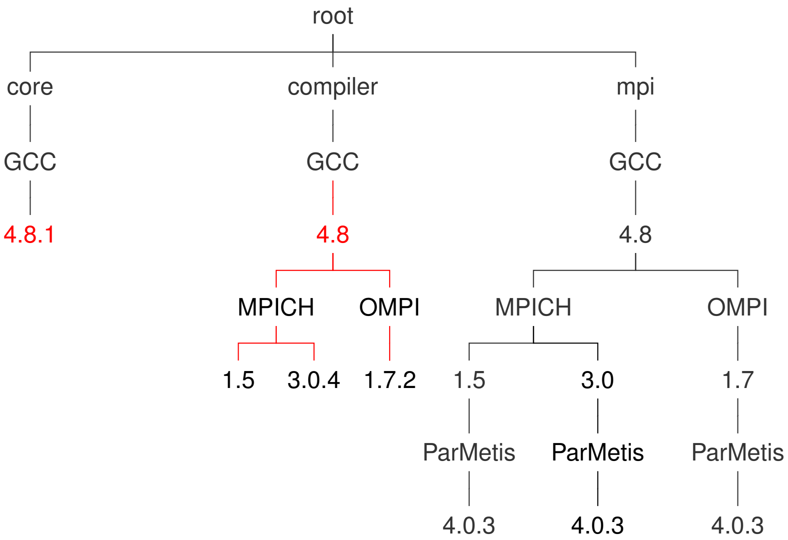
- Directory structure – core modules always active
- Load GCC 4.8.1
- Load MPICH 3.0.4

Example



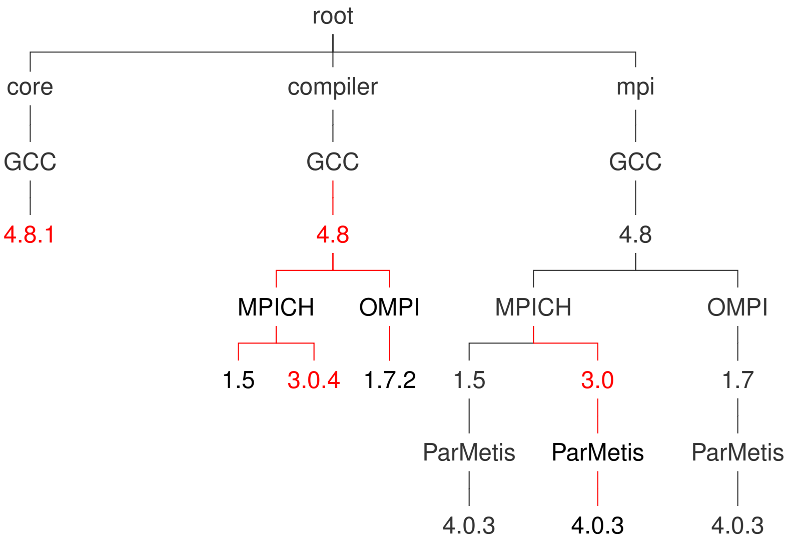
- Directory structure – core modules always active
- Load GCC 4.8.1
- Load MPICH 3.0.4
- Switch to ICC 13.1.2 ⇒ automatic reload of MPICH

Nested hierarchies



Separate hierarchies for every combination of compiler + MPI

Nested hierarchies



Separate hierarchies for every combination of compiler + MPI

Providing Information to module users

Make sure you know what you are loading in half a year:

```
help([[
This module loads the MPICH MPI library.
The package is built with shared libraries
]])

whatis("Name: MPICH")
whatis("Description: MPICH MPI Library")
whatis("Version: 3.0.4")
```

Problems

- Module files very similar for different versions / dependencies
- Hard to maintain, small changes go to lots of files

Problems

- Module files very similar for different versions / dependencies
- Hard to maintain, small changes go to lots of files

Idea: Reuse information encoded in directory structure / file names

- Extend Lmod with custom site package and helper functions
- Single module file for every package, directory structure only contains symlinks
- Export canonical prefix path for every package – greatly simplifies package compilation

Example - MPICH (I)

```
local pkg = declarePkg{
  family = "MPI"
}

local pkgs = loadedPkgs()
local path_deps = extractPathDependencies()
local compiler = pkgs[path_deps[1].name]
local deps = {
  {
    title = "Compiler",
    pkg   = compiler
  }
}

dependenciesPkg(pkg,deps)
```

Example - MPICH (II)

```
local help_string = [[
MPICH MPI library.
]]
help(help_string)

whatisPkg{
  pkg         = pkg,
  description = "MPICH MPI Library.",
  deps = deps
}
```

Example - MPICH (III)

```
local install_path = pkgDir{
  pkg  = pkg,
  deps = deps
}
setenv("MODULE_MPI","MPICH")
setenv("MPI_DIR",install_path)

prepend_path("PATH", pathJoin(install_path,"bin"))
prepend_path("CPATH", pathJoin(install_path,"include"))
prepend_path("LIBRARY_PATH", pathJoin(install_path,"lib"))
prepend_path("DYLD_LIBRARY_PATH", pathJoin(install_path,"lib"))
-- Don't modify MANPATH on OS X, it screws with its additional MANPATH logi
-- prepend_path("MANPATH", pathJoin(install_path,"share/man"))
prepend_path("PKG_CONFIG_PATH", pathJoin(install_path,"lib/pkgconfig"))

-- Setup Modulepath for packages built with this MPI library
local module_dir = pathJoin("mpi",compiler.compat_name,pkg.compat_name)
prependModulePath(module_dir)

registerDependency(pkg)
```

Conclusion

- For certain applications, DUNE development requires carrying around multiple versions of depenencies
- Managing those multiple versions manually is a bad idea
- Environment modules provide a good solution
- Some custom extensions greatly reduce the maintenance burden
- If you want my extensions + module files, I'm happy to share!

Thank you for your attention