

# Managing HPC Software Complexity with Spack

Supercomputing 2016 Half-day Tutorial  
November 13, 2016  
Salt Lake City, UT



LLNL-PRES-80606

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344.  
Lawrence Livermore National Security, LLC.

<http://github.com/LLNL/spack>

 Lawrence Livermore  
National Laboratory



INFORMATIQUE SCIENTIFIQUE & SUPPORT APPROFONDI  
**SCITAS**  
QUINTETTE IT AND APPLICATION SERVICES



Salt Lake City / HPC  
SC16  


# Tutorial Materials

**Materials:** Download the latest version of slides and handouts at:

**<http://spack.readthedocs.io/>**

**Be sure to have the most current slides before we begin!**

## Resources:

- Spack GitHub repository: **<http://github.com/LLNL/spack>**
- Spack Documentation: **<http://spack.readthedocs.io/>**
- Handouts and additional documentation:
  - See “Tutorials” section of documentation page.

# Tutorial Presenters & Contributors

---

## Presenters

- LLNL
  - Todd Gamblin
  - Gregory Becker
  - Greg Lee
  - Matt Legendre
  
- EPFL
  - Massimiliano Culpo

## Contributors

- CERN
  - Benedikt Hegner
  
- NASA
  - Elizabeth Fischer

# Who is this Tutorial for?

**People who want to use or distribute software for HPC!**

## 1. End Users of HPC Software

- Install and run HPC applications and tools

## 2. HPC Application Teams

- Manage third-party dependency libraries

## 3. Package Developers

- People who want to package their own software for distribution

## 4. User support teams at HPC Centers

- People who deploy software for users at large HPC sites

# HPC Software Complexity Prevents Reuse & Reduces Productivity

- Not much standardization in HPC: every machine/app has a different software stack
  - This is done to get the best **performance**
- HPC frequently trades reuse and usability for performance
  - Reusing a piece of software frequently requires you to port it to many new platforms
- Example environment for some LLNL codes:



**Spack lowers barriers to software reuse by automating these builds!**

# What is the “production” environment for HPC?

- Someone's home directory?
- LLNL? LANL? Sandia? ANL? LBL? TACC?
  - Environments at large-scale sites are very different.
- Which MPI implementation?
- Which compiler?
- Which dependencies?
- Which versions of dependencies?
  - Many applications require specific dependency versions.



**Real answer:** there isn't a single production environment or a standard way to build.  
**Reusing someone else's software is HARD.**

# Tutorial Overview

- 
- 1. Welcome & Overview 1:30 - 1:40
  - 2. **For everyone: Basics of Building and Linking** 1:40 - 1:55
  - 3. **For everyone: Spack Basics (hands on)** 1:55 - 2:40
  - 4. **For everyone: Core Spack Concepts** 2:40 - 3:00
  - 5. -- Break -- 3:00 - 3:30
  - 6. **For developers: Making your own Spack Packages (hands on)** 3:30 - 4:15
  - 7. **For HPC centers: Deploying Spack at HPC Facilities (hands on)** 4:15 - 5:00

# Why use a new tool?

---

1. Automate the process of building large software packages
2. Automate testing with different compilers and options
3. Leverage the work of others with shared build recipes
  1. Don't repeat the same builds!
4. Distribute your software to a growing audience of HPC users

# Spack is a flexible package manager for HPC

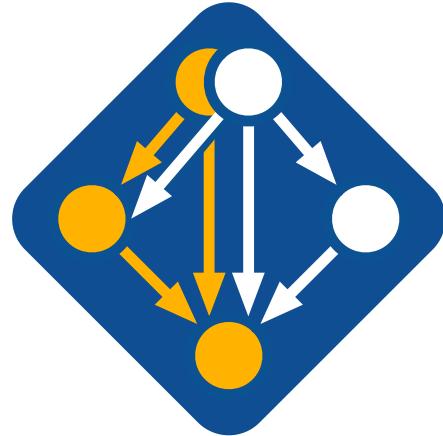
- How to install Spack:

```
$ git clone https://github.com/LLNL/spack.git
$ . spack/share/spack/setup-env.sh
```

- How to install a package:

```
$ spack install hdf5
```

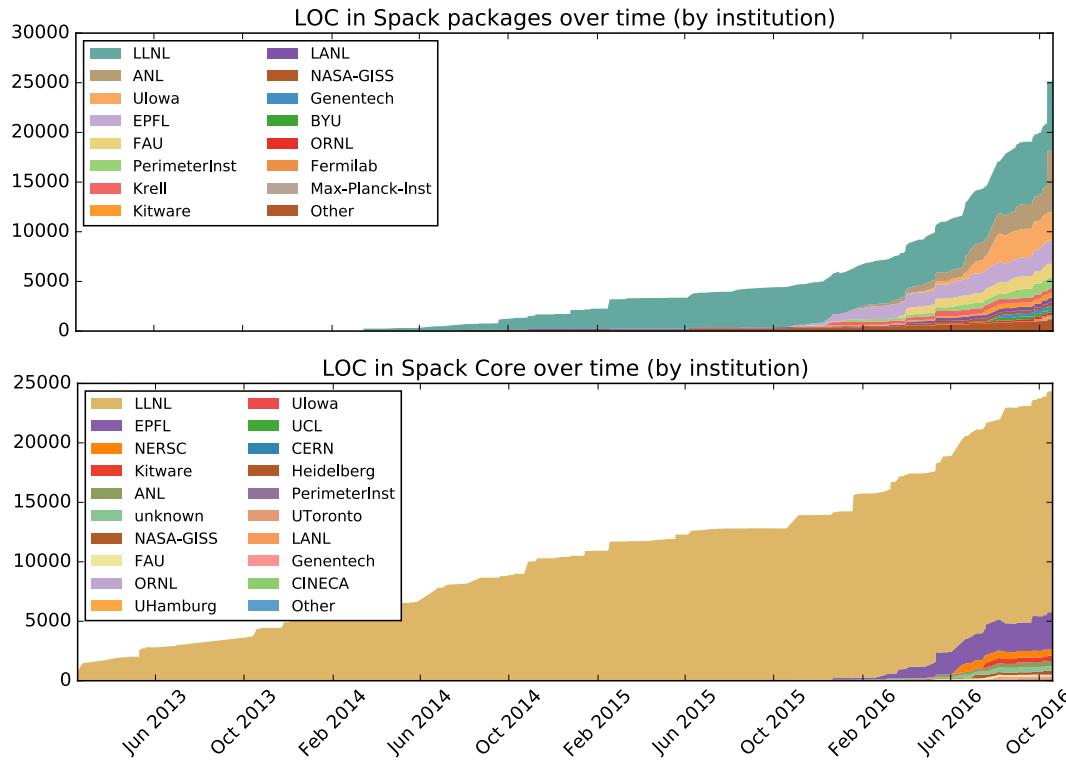
- HDF5 and its dependencies are installed within the Spack directory.
- Unlike typical package managers, Spack can also install many variants of the same build.
  - Different compilers
  - Different MPI implementations
  - Different build options



**Get Spack!**

<http://github.com/LLNL/spack>

# Contributions to Spack have grown rapidly over the past year



- 1 year ago, LLNL provided most of the contributions to Spack
- Since last year, we've gone from 300 to 1,000 packages
- Over 75% of lines of code in packages are from external contributors.
  - Contributors add to the core as well.
- We are committed to sustaining Spack's open source ecosystem!



# Join the Spack community!

- **20+ organizations**  
**90+ contributors**  
Sharing 1,016 packages and growing
- **Spack can be a central repository for tools**
  - Make it easy for others to use them!
  - Build code the same way across HPC centers.
- **Spack is used in production at LLNL**
  - Livermore Computing
  - LLNL production multi-physics codes
- **Other organizations using Spack:**
  - NERSC using Spack on Cori: Cray support.
  - EPFL (Switzerland) contributing core features.
  - Fermi, CERN, BNL: high energy physics ecosystem.
  - ANL using Spack on production Linux clusters.
  - NASA packaging an Ice model code.
  - ORNL working with us on Spack for CORAL.
  - Kitware: core features, ParaView, UV-CDAT support

 Lawrence Livermore  
National Laboratory

 NERSC

 Argonne  
NATIONAL LABORATORY

 OAK  
RIDGE  
National Laboratory

 EPFL  
ÉCOLE POLYTECHNIQUE  
FÉDÉRALE DE LAUSANNE

 Sandia  
National  
Laboratories

 Kitware

 Inria  
INVENTORS FOR THE DIGITAL WORLD

 Los Alamos  
NATIONAL LABORATORY  
EST. 1943

 CERN

 NASA

 Fermilab

 KRELL  
Institute

 PERIMETER  
INSTITUTE

 UCDAVIS  
UNIVERSITY OF CALIFORNIA

 CINECA

 BROOKHAVEN  
NATIONAL LABORATORY

 IBM  
THE UNIVERSITY  
OF IOWA

 University of  
BRISTOL

 FAU  
FRIEDRICH-ALEXANDER  
UNIVERSITÄT  
ERLANGEN-NÜRNBERG

 openHPC

 O  
UNIVERSITY  
OF OREGON

 INFORMATIQUE SCIENTIFIQUE & SUPPORT APPLICATIF  
SCITAS  
SCIENTIFIC IT AND APPLICATION SUPPORT

 EPFL  
ÉCOLE POLYTECHNIQUE  
FÉDÉRALE DE LAUSANNE

 SC16  
SUPERCOMPUTING



# Related Work

## Spack is not the first tool to automate builds

- Inspired by copious prior work

### 1. “Functional” Package Managers

- Nix
- GNU Guix

<https://nixos.org/>

<https://www.gnu.org/s/guix/>

### 2. Build-from-source Package Managers

- Homebrew
- MacPorts

<http://brew.sh>

<https://www.macports.org>

## Other tools in the HPC Space:

### ▪ Easybuild

- An *installation* tool for HPC
- Focused on HPC system administrators – different package model from Spack
- Relies on a fixed software stack – harder to tweak recipes for experimentation

<http://hpcugent.github.io/easybuild/>

### ▪ Hashdist

<https://hashdist.github.io>

# Building & Linking Basics

# What's a package manager?

- Spack is a ***package manager***
    - Does not replace Cmake/Autotools
    - Packages built by Spack can have any build system they want
  - Spack manages *dependencies*
    - Drives package-level build systems
    - Ensures consistent builds
  - Determining magic configure lines takes time
    - Spack is a cache of recipes
- Package Manager**

**High Level Build System**

**Low Level Build System**
- Manages dependencies
  - Drive package-level build systems
  - Cmake, Autotools
  - Handle library abstractions
  - Generate Makefiles, etc.
  - Make, Ninja
  - Handles dependencies among *commands* in a single build

# Static vs. shared libraries

- Static libraries: `libfoo.a`
  - .a files are archives of .o files (object files)
  - Linker includes needed parts of a static library in the output executable
  - No need to find dependencies at runtime – only at build time.
  - Can lead to large executables
  - Often hard to build a completely static executable on modern systems.
- Shared libraries: `libfoo.so` (Linux), `libfoo.dylib` (MacOS)
  - More complex build semantics, typically handled by the build system
  - Must be found by `ld.so` or `dyld` (dynamic linker) and loaded at runtime
    - Can cause lots of headaches with multiple versions
  - 2 main ways:
    - `LD_LIBRARY_PATH`: environment variable configured by user and/or module system
    - `RPATH`: paths embedded in executables and libraries, so that they know where to find their own dependencies.

# API and ABI Compatibility

## ■ API: Application Programming Interface

- Source code functions and symbol names exposed by a library
- If API of a dependency is backward compatible, source code need not be changed to use it
- **May** need to recompile code to use a new version.

## ■ ABI: Application Binary Interface

- Calling conventions, register semantics, exception handling, etc.
- Defined by how the compiler builds a library
  - Binaries generated by different compilers are typically ABI-incompatible.
- May also include things like standard runtime libraries and compiler intrinsic functions
- May also include values of hard-coded symbols/constants in headers.

## ■ HPC code, including MPI, is typically API-compatible but not ABI-compatible.

- Causes many build problems, especially for dynamically loaded libraries
- Often need to rebuild to get around ABI problems
- Leads to combinatorial builds of software at HPC sites.

# 3 major build systems to be aware of

---

## 1. Make (usually GNU Make)

- <https://www.gnu.org/software/make/>

## 2. GNU Autotools

- Automake: <https://www.gnu.org/software/automake/>
- Autoconf: <https://www.gnu.org/software/autoconf/>
- Libtool: <https://www.gnu.org/software/libtool/>

## 3. CMake:

- <https://cmake.org>

# Make and GNU Make

- Many projects opt to write their own Makefiles.
  - Can range from simple to very complicated
- Make declares some standard variables for various compilers
  - Many HPC projects don't respect them
  - No standard install prefix convention
  - Makefiles may not have install target
- Automating builds with Make usually requires editing files
  - Typical to use sed/awk/some other regular expression tool on Makefile
  - Can also use patches

## Typical build incantation

```
<edit Makefile>  
make PREFIX=/path/to/prefix
```

## Configure options

None. Typically must edit Makefiles.

## Environment variables

CC	CFLAGS	LDFLAGS
CXX	CXXFLAGS	LIBS
FC	FFLAGS	CPP
F77	FFLAGS	



# Autotools

- Three parts of autotools:
  - autoconf: generates a portable configure script to inspect build host
  - automake: high-level syntax for generating lower-level Makefiles.
  - libtool: abstraction for shared libraries
- Typical variables are similar to make
- Much more consistency among autotools projects
  - Wide use of standard variables and configure options
  - Standard install target, staging conventions.

## Typical build incantation

```
./configure --prefix=/path/to/install_dir  
make  
make install
```

## Configure options

```
./configure \  
--prefix=/path/to/install_dir \  
--with-package=/path/to/dependency \  
--enable-foo \  
--disable-bar
```

## Environment variables

CC	CFLAGS	LDFLAGS
CXX	CXXFLAGS	LIBS
FC	FFLAGS	CPP
F77	FFLAGS	



# CMake

- Gaining popularity
- Arguably easier to use (for developers) than autotools
- Similar standard options to autotools
  - different variable names
  - More configuration options
  - Abstracts platform-specific details of shared libraries
- Most CMake projects should be built “out of source”
  - Separate build directory from source directory

## Typical build incantation

```
mkdir BUILD && cd BUILD  
cmake -DCMAKE_INSTALL_PREFIX=/path/to/install_dir ..  
make  
make install
```

## Configure options

```
cmake \  
  -D CMAKE_INSTALL_PREFIX=/path/to/install_dir \  
  -D ENABLE_FOO=yes \  
  -D ENABLE_BAR=no \  
  ..
```

## Common -D options

CMAKE_C_COMPILER	CMAKE_C_FLAGS
CMAKE_CXX_COMPILER	CMAKE_CXX_FLAGS
CMAKE_Fortran_COMPILER	CMAKE_Fortran_FLAGS
CMAKE_SHARED_LINKER_FLAGS	CMAKE_EXE_LINKER_FLAGS
CMAKE_STATIC_LINKER_FLAGS	



# Spack Basics

# Spack provides a *spec* syntax to describe customized DAG configurations

```
$ spack install mpileaks                                unconstrained
$ spack install mpileaks@3.3                            @ custom version
$ spack install mpileaks@3.3 %gcc@4.7.3               % custom compiler
$ spack install mpileaks@3.3 %gcc@4.7.3 +threads      +/- build option
$ spack install mpileaks@3.3 cppflags="-O3"            setting compiler flags
$ spack install mpileaks@3.3 os=CNL10 target=haswell    setting target for X-compile
$ spack install mpileaks@3.3 ^mpich@3.2 %gcc@4.9.3     ^ dependency information
```

- Each expression is a *spec* for a particular configuration
  - Each clause adds a constraint to the spec
  - Constraints are optional – specify only what you need.
  - Customize install on the command line!
- Spec syntax is recursive
  - Full control over the combinatorial build space

# `spack list` shows what packages are available

```
$ spack list
==> 303 packages.
activeharmony cgal fish gtkplus libgd mesa openmpi py-coverage qt tcl
adept-utils cgm flex harfbuzz libggp-error metis openspeditshop py-cython py-pyelftools qthreads texinfo
apex cityhash fltk hdf libjpeg-turbo Mitos openssl py-dateutil py-pygments R the_silver_searcher
arpack cleverleaf flux fontconfig hwlloc libjson-c mpc otf py-epydoc py-pylint ravel thrift
asciidoc cloog freetype hypre libmonitor mpfr pango py-funcsigs py-pypar readline tk
atk cmake gasnet icu libNBc mpibash papi py-gnuplot py-pyparsing rose tmux
atlas cmocka libcurl icu4c libpicaaccess mpich paraver py-hspy py-pyside rsync tmuxinator
atop coreutils gcc ImageMagick libpng libpileaks paraview py-ipython py-pytables ruby trilinos
autoconf cppcheck gdk-pixbuf isl libodium mernet parmetis py-libxml2 py-python-daemon SAMRAI uncrustify
automated cram jdk libtiff mumps parpack py-lockfile py-ptz samtools util-linux
automake cscope geos jemalloc libtool munge patchelf py-mako py-rpy2 scalasca valgrind
bear cube gflags jpeg libunwind muster pcre py-matplotlib py-scientificpython scorep vim
bib2xhtml curl ghostscript libuiuid mvapich2 pcre2 py-mock py-scikit-learn scr wget
binutils czmq git judy libxcb nasm pdt py-mpi4py py-scipy silo wx
bison damselfly glib julia libxml2 ncdu petsc py-mx py-setupools snappy wxpropgrid
boost dbus glm launchmon lcms libxshmfence ncurses pidx py-mysqldb1 py-shiboken sparsehash xcb-proto
bowtie2 docbook-xml global leveldb libxslt netcdf pixman py-nose py-sip spindle xerces-c
boxlib doxygen glog libarchive llvm netgauge pkg-config py-numexpr py-six spot xz
bzip2 driproto glpk libcurl libcperf llvm-lld netlib-blas pmgr_collective py-numpy py-sphinx sqlite yasm
cairo dtcmpl gmp libcurl libcircle lmdb netlib-lapack postgresql py-pandas py-sympy stat zeromq
callpath dyninst gmsn libcurl libdrm lmod netlib-scalapack ppl py-pbr py-tappy sundials zlib
cblas eigen gnuplot libdwarf lua nettle protobuf py-periodictable py-twisted swig zsh
cbtf elfutils gnutls libedit lwgrp ninja py-astropy py-pexpect py-urwid szip
cbtf-argonavis elpa gperf libelf lwm2 ompss py-baseemap py-pil py-virtualenv tar
cbtf-krell expat gperf tools libevent matio ompt-openmp py-biopython py-pillow py-yapf task
cbtf-lanl extrae graphlib libevent memax libff libmbedtls opari2 py-blessings py-pmw python
cereal exuberant-ctags graphviz libff libgcrypt memaxes libopenblas py-cffi py-pychecker qhull
cfitsio fftw gsl libgrypt
```

- Spack has over 1000 packages now.

# `spack find` shows what is installed

```
$ spack find
==> 103 installed packages.
-- linux-redhat6-x86_64 / gcc@4.4.7 -----
ImageMagick@6.8.9-10 glib@2.42.1 libtiff@4.0.3 pango@1.36.8 qt@4.8.6
SAMRAI@3.9.1 graphlib@2.0.0 libtool@2.4.2 parmetis@4.0.3 qt@5.4.0
adept-utils@1.0 gtkplus@2.24.25 libxcb@1.11 pixman@0.32.6 ravel@1.0.0
atk@2.14.0 harfbuzz@0.9.37 libxml2@2.9.2 py-dateutil@2.4.0 readline@6.3
boost@1.55.0 hdf5@1.8.13 llvm@3.0 py-ipython@2.3.1 scotch@6.0.3
cairo@1.14.0 icu@54.1 metis@5.1.0 py-nose@1.3.4 starpu@1.1.4
callpath@1.0.2 jpeg@9a mpich@3.0.4 py-numumpy@1.9.1 stat@2.1.0
dyninst@8.1.2 libdwarf@20130729 ncurses@5.9 py-pytz@2014.10 xz@5.2.0
dyninst@8.1.2 libelf@0.8.13 ocr@2015-02-16 py-setuptools@11.3.1 zlib@1.2.8
fontconfig@2.11.1 libffi@3.1 openssl@1.0.1h py-six@1.9.0
freetype@2.5.3 libmng@2.0.2 otf@1.12.5salmon python@2.7.8
gdk-pixbuf@2.31.2 libpng@1.6.16 otf2@1.4 qhull@1.0

-- linux-redhat6-x86_64 / gcc@4.8.2 -----
adept-utils@1.0.1 boost@1.55.0 cmake@5.6-special libdwarf@20130729 mpich@3.0.4
adept-utils@1.0.1 cmake@5.6 dyninst@8.1.2 libelf@0.8.13 openmpi@1.8.2

-- linux-redhat6-x86_64 / intel@14.0.2 -----
hwloc@1.9 mpich@3.0.4 starpu@1.1.4

-- linux-redhat6-x86_64 / intel@15.0.0 -----
adept-utils@1.0.1 boost@1.55.0 libdwarf@20130729 libelf@0.8.13 mpich@3.0.4

-- linux-redhat6-x86_64 / intel@15.0.1 -----
adept-utils@1.0.1 callpath@1.0.2 libdwarf@20130729 mpich@3.0.4
boost@1.55.0 hwloc@1.9 libelf@0.8.13 starpu@1.1.4
```

- All the versions coexist!
  - Multiple versions of same package are ok.
- Packages are installed to automatically find correct dependencies.
- Binaries work *regardless of user's environment*.
- Spack also generates module files.
  - Don't have to use them.

# Users can query the full dependency configuration of installed packages.

```
$ spack find callpath  
==> 2 installed packages.  
-- linux-x86_64 / clang@3.4 -----  
callpath@1.0.2  
-- linux-x86_64 / gcc@4.9.2 -----  
callpath@1.0.2
```



Expand dependencies  
with `spack find -d`

```
$ spack find -dl callpath  
==> 2 installed packages.  
-- linux-x86_64 / clang@3.4 -----  
xv2clz2    callpath@1.0.2  
ckjazss    ^adept-utils@1.0.1  
3ws43m4     ^boost@1.59.0  
ft7znm6    ^mpich@3.1.4  
qqnuet3    ^dyninst@8.2.1  
3ws43m4     ^boost@1.59.0  
g65rdud    ^libdwarf@20130729  
cj5p5fk    ^libelf@0.8.13  
cj5p5fk    ^libelf@0.8.13  
g65rdud    ^libdwarf@20130729  
cj5p5fk    ^libelf@0.8.13  
cj5p5fk    ^libelf@0.8.13  
ft7znm6    ^mpich@3.1.4  
-- linux-x86_64 / gcc@4.9.2 -----  
udltshs    callpath@1.0.2  
rfsu7fb    ^adept-utils@1.0.1  
ybet64y    ^boost@1.55.0  
aa4ar6i    ^mpich@3.1.4  
tmnnge5    ^dyninst@8.2.1  
ybet64y    ^boost@1.55.0  
g2mxrl2    ^libdwarf@20130729  
ynpai3j    ^libelf@0.8.13  
ynpai3j    ^libelf@0.8.13  
g2mxrl2    ^libdwarf@20130729  
ynpai3j    ^libelf@0.8.13  
ynpai3j    ^libelf@0.8.13  
aa4ar6i    ^mpich@3.1.4
```

- Architecture, compiler, versions, and variants may differ between builds.

# Spack manages installed compilers

- Compilers are automatically detected
  - Automatic detection determined by OS
  - Linux: PATH
  - Cray: `module avail`
- Compilers can be manually added
  - Including Spack-built compilers

```
$ spack compilers
==> Available compilers
-- gcc -----
gcc@4.2.1      gcc@4.9.3

-- clang -----
clang@6.0
```

compilers.yaml

```
compilers:
- compiler:
  modules: []
  operating_system: ubuntu14
  paths:
    cc: /usr/bin/gcc/4.9.3/gcc
    cxx: /usr/bin/gcc/4.9.3/g++
    f77: /usr/bin/gcc/4.9.3/gfortran
    fc: /usr/bin/gcc/4.9.3/gfortran
    spec: gcc@4.9.3
- compiler:
  modules: []
  operating_system: ubuntu14
  paths:
    cc: /usr/bin/clang/6.0/clang
    cxx: /usr/bin/clang/6.0/clang++
    f77: null
    fc: null
    spec: clang@6.0
- compiler:
  ...
```

# Hands-on Time

# Extensions and Python Support

- Spack installs packages in own prefix
- Some packages need to be installed within directory structure of other packages
  - i.e., Python modules installed in \$prefix/lib/python-<version>/site-packages
  - Spack supports this via extensions

```
class PyNumpy(Package):
    """NumPy is the fundamental package for scientific computing with Python."""

    homepage = "https://numpy.org"
    url      = "https://pypi.python.org/packages/source/n/numpy/numpy-1.9.1.tar.gz"
    version('1.9.1', '78842b73560ec378142665e712ae4ad9')

    extends("python")

    def install(self, spec, prefix):
        setup_py("install", "--prefix={0}".format(prefix))
```

# Spack extensions

- Examples of extension packages:
  - python libraries are a good example
  - R, Lua, perl
  - Need to maintain combinatorial versioning

```
spack/opt/
linux-redhat6-x86_64/
gcc-4.7.2/
python-2.7.12-6y6vvaw/
python/
py-numpy-1.10.4-oaxix36/
py-numpy/
...
```

```
$ spack activate py-numpy @1.10.4
```

- Symbolic link to Spack install location
- Automatically activate for correct version of dependency
  - Provenance information from DAG
  - Activate all dependencies that are extensions as well

```
spack/opt/
linux-redhat6-x86_64/
gcc-4.7.2/
python-2.7.12-
python/
py-numpy
py-numpy-1.10.4-oaxix36/
py-numpy/
...
```

# Extensions must be activated into extender

```
$ spack extensions python
==> python@2.7.12% gcc@4.9.3 ~tk~ucs4 arch=linux-redhat7-x86_64-i25k4oi
==> 118 extensions:
...
py-autopackage py-docutils py-mako py-numpy py-py2neo
...
==> 3 installed:
-- linux-redhat7-x86_64 / gcc@4.9.3 -----
py-nose@1.3.7 py-numpy@1.11.0 py-setuptools@20.7.0

==> None activated.

$ spack load python
$ python
Python 2.7.12 (default, Aug 26 2016, 15:12:42)
[GCC 4.9.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy
Traceback (most recent call last): File "<stdin>", line 1, in <module>
ImportError: No module named numpy
>>>
```

```
$ spack activate py-numpy
==> Activated extension py-numpy@1.11.0% gcc@4.9.3 +blas+lapack
arch=linux-redhat7-x86_64-77im5ny for python@2.7.12 ~tk~ucs4 % gcc@4.9.3
$ spack extensions python
==> python@2.7.12% gcc@4.9.3 ~tk~ucs4 arch=linux-redhat7-x86_64-i25k4oi
...
==> 1 currently activated:
-- linux-redhat7-x86_64 / gcc@4.9.3 -----

$ python
Python 2.7.12 (default, Aug 26 2016, 15:12:42)
[GCC 4.9.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy
>>> x = numpy.zeros(10)
>>>
```

- Python unaware of numpy installation

- activate symlinks entire numpy prefix into Python installation
- Can alternatively load extension

```
$ spack load python
$ spack load py-numpy
```

# Hands-on Time

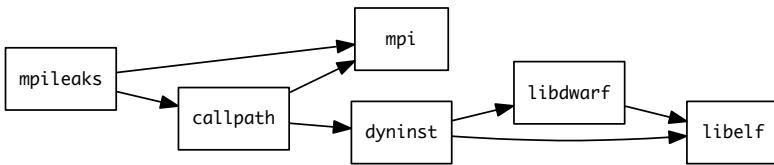
# Core Spack Concepts

# Most existing tools do not support combinatorial versioning

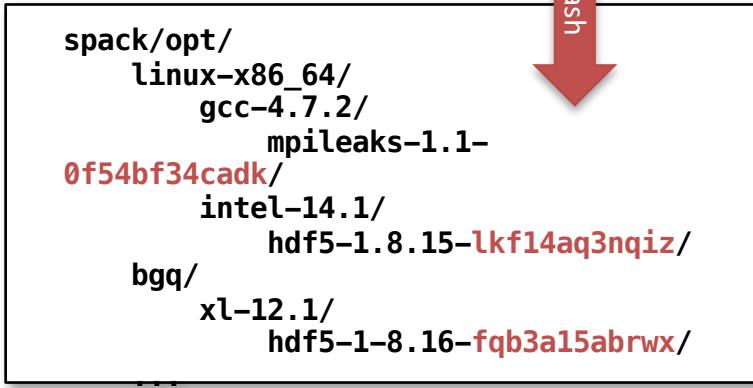
- Traditional binary package managers
  - RPM, yum, APT, yast, etc.
  - Designed to manage a single stack.
  - Install *one* version of each package in a single prefix (/usr).
  - Seamless upgrades to a *stable, well tested* stack
- Port systems
  - BSD Ports, portage, Macports, Homebrew, Gentoo, etc.
  - Minimal support for builds parameterized by compilers, dependency versions.
- Virtual Machines and Linux Containers (Docker)
  - Containers allow users to build environments for different applications.
  - Does not solve the build problem (someone has to build the image)
  - Performance, security, and upgrade issues prevent widespread HPC deployment.

# Spack handles combinatorial software complexity.

## Dependency DAG

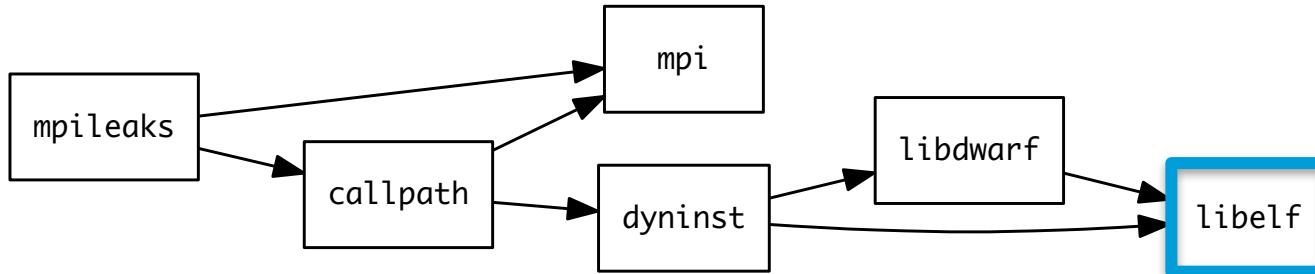


## Installation Layout



- Each unique dependency graph is a unique **configuration**.
- Each configuration installed in a unique directory.
  - Configurations of the same package can coexist.
- **Hash** of entire directed acyclic graph (DAG) is appended to each prefix.
- Installed packages automatically find dependencies
  - Spack embeds RPATHs in binaries.
  - No need to use modules or set LD\_LIBRARY\_PATH
  - Things work *the way you built them*

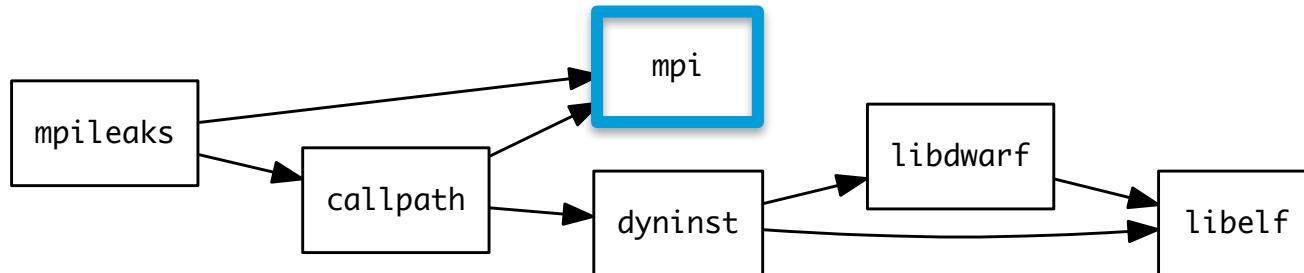
# Spack Specs can constrain versions of dependencies



```
$ spack install mpileaks %intel@12.1 ^libelf@0.8.12
```

- Spack ensures *one* configuration of each library per DAG
  - Ensures ABI consistency.
  - User does not need to know DAG structure; only the dependency *names*.
- Spack can ensure that builds use the same compiler, or you can mix
  - Working on ensuring ABI compatibility when compilers are mixed.

# Spack handles ABI-incompatible, versioned interfaces like MPI



- *mpi* is a *virtual dependency*
- Install the same package built with two different MPI implementations:

```
$ spack install mpileaks ^mvapich@1.9
```

```
$ spack install mpileaks ^openmpi@1.4:
```

- Let Spack choose MPI implementation, as long as it provides MPI 2 interface:

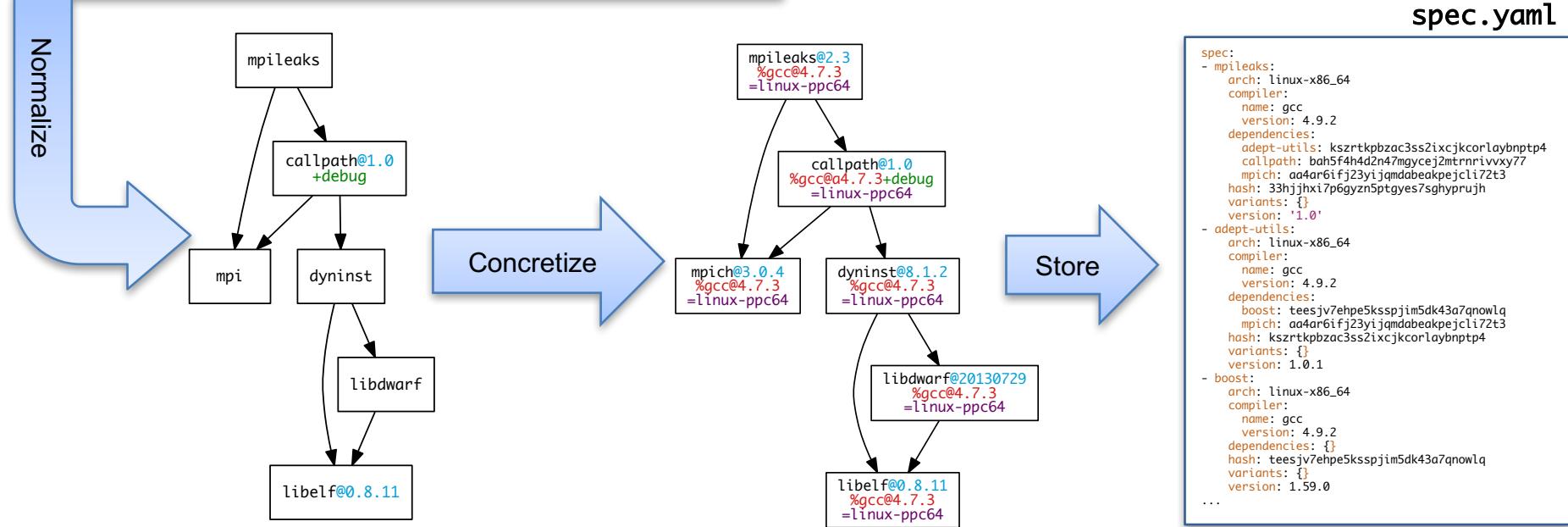
```
$ spack install mpileaks ^mpi@2
```

# Concretization fills in missing configuration details when the user is not explicit.

mpileaks ^callpath@1.0+debug ^libelf@0.8.11

User input: abstract spec with some constraints

spec.yaml



# Use spack spec to see the results of concretization

```
$ spack spec mpileaks
Input spec
-----
mpileaks

Normalized
-----
mpileaks
  ^adept-utils
    ^boost@1.42:
      ^mpi
      ^callpath
        ^dyninst
          ^libdwarf
          ^libelf

Concretized
-----
mpileaks@1.0%gcc@5.3.0 arch=darwin-elcapitan-x86_64
  ^adept-utils@1.0.1%gcc@5.3.0 arch=darwin-elcapitan-x86_64
    ^boost@1.61.0%gcc@5.3.0+atomic+chrono+date_time~debug+filesystem~graph
      ~icu_support+iostreams+locale+log+math-mpi+multithreaded+program_options
      ~python+random +regex+serialization+shared+signals+singlethreaded+system
      +test+thread+timer+wave arch=darwin-elcapitan-x86_64
    ^bzzip2@1.0.6%gcc@5.3.0 arch=darwin-elcapitan-x86_64
    ^zlib@1.2.8%gcc@5.3.0 arch=darwin-elcapitan-x86_64
  ^openmpi@2.0.0%gcc@5.3.0~mxm+pmi+psm2~slurm~sqlite3~thread_multiple~tm~verbs+vt arch=darwin-elcapitan-x86_64
    ^hwloc@1.11.3%gcc@5.3.0 arch=darwin-elcapitan-x86_64
      ^libpciaccess@0.13.4%gcc@5.3.0 arch=darwin-elcapitan-x86_64
      ^libtool@2.4.6%gcc@5.3.0 arch=darwin-elcapitan-x86_64
      ^m4@1.4.17%gcc@5.3.0+sigsegv arch=darwin-elcapitan-x86_64
      ^libsigsegv@2.10%gcc@5.3.0 arch=darwin-elcapitan-x86_64
  ^callpath@1.0.2%gcc@5.3.0 arch=darwin-elcapitan-x86_64
    ^dyninst@0.2.0%gcc@5.3.0~stat_dysect arch=darwin-elcapitan-x86_64
      ^libdwarf@20160507%gcc@5.3.0 arch=darwin-elcapitan-x86_64
      ^libelf@0.8.13%gcc@5.3.0 arch=darwin-elcapitan-x86_64
```

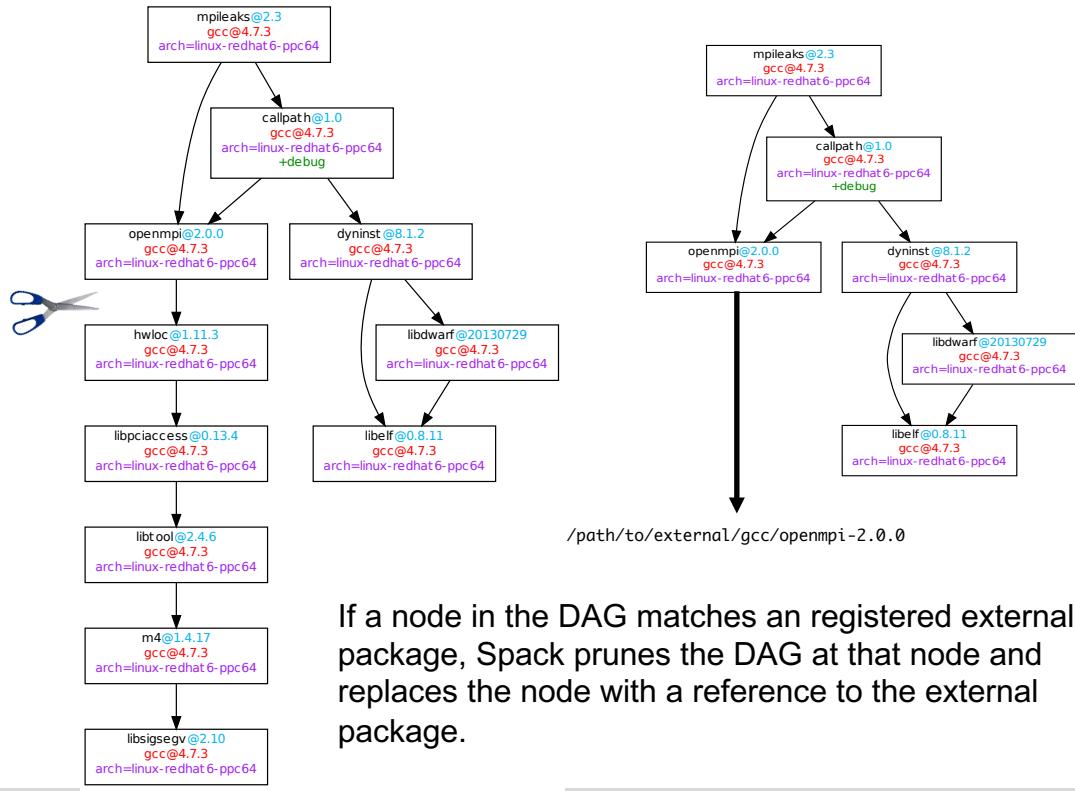
# Building against externally installed software

```
mpileaks ^callpath@1.0+debug  
^openmpi ^libelf@0.8.11
```

packages.yaml

```
packages:  
  mpi:  
    buildable: False  
  openmpi:  
    buildable: False  
    paths:  
      openmpi@2.0.0 %gcc@4.7.3 arch=linux-redhat6-ppc64:  
        /path/to/external/gcc/openmpi-2.0.0  
      openmpi@1.10.3 %gcc@4.7.3 arch=linux-redhat6-ppc64:  
        /path/to/external/gcc/openmpi-1.10.3  
      openmpi@2.0.0 %intel@16.0.0 arch=linux-redhat6-ppc64:  
        /path/to/external/intel/openmpi-2.0.0  
      openmpi@1.10.3 %intel@16.0.0 arch=linux-redhat6-ppc64:  
        /path/to/external/intel/openmpi-1.10.3  
  ...
```

A user registers external packages with Spack.

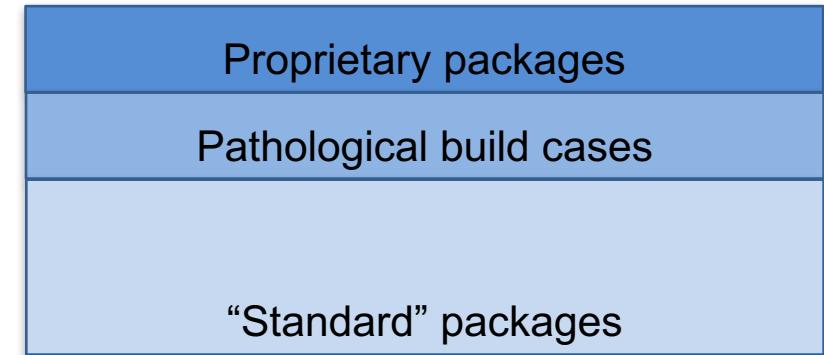


If a node in the DAG matches an registered external package, Spack prunes the DAG at that node and replaces the node with a reference to the external package.

# Spack package repositories

- Some packages can not be released publicly
- Some users have use cases that require bizarre custom builds
- Packaging issues should not prevent users from updating Spack
  - Solution: separate repositories
  - A repository is simply a directory of package files

my\_repo



spack/var/repos/builtin



# Fetching source code for spack

```
spack install mpileaks
```

- User Cache
  - Users create a “mirror” of tar archives for packages
  - Remove internet dependence
- Spack Cache
  - Spack automatically caches tar archives for previously installed software
- The Internet
  - Spack packages can find source files online

Load package from repository

Concretize

Recursively install dependencies

Fetch package source

Build software

User Cache

Spack Cache

The Internet

# Adding compiler flags (for compiler parameter studies)

```
$ spack install ares cflags=\"-O3 -g -fast -fpack-struct\"
```

- This would install ARES with the specified flags
  - Flags are injected via Spack's compiler wrappers.
- Flags are propagated to dependencies automatically
  - Flags are included in the **DAG hash**
  - Each build is considered a **different version**
- This provides an easy harness for doing parameter studies for tuning codes
  - Previously working with large codes was very tedious.
- Supports cflags, cxxflags, fflags, cppflags, ldflags, and ldlibs
  - Added from CLI or config file

**Spack provides hooks that enable tools to work with large codes.**

# Making your own Spack Packages

# Creating your own Spack Packages

- Package is a recipe for building
- Each package is a Python class
  - Download information
  - Versions/Checksums
  - Build options
  - Dependencies
  - Build instructions
- Package collections are repos
  - Spack has a “builtin” repo in  
**\$REPO/var/spack/repos/builtin**

## \$REPO/packages/zlib/package.py

```
from spack import *

class Zlib(Package):
    """A free, general-purpose, legally unencumbered lossless
       data-compression library."""

    homepage = "http://zlib.net"
    url     = "http://zlib.net/zlib-1.2.8.tar.gz"

    version('1.2.8', '44d667c142d7cda120332623eab69f40')

    depends_on('cmake', type='build')

    def install(self, spec, prefix):
        configure('--prefix={0}'.format(prefix))

        make()
        make('install')
```

# Packages are Parameterized

- Each package has one class
  - e.g., zlib for Intel compiler and zlib for GCC compiler are built with the same recipe.
- Can query what's being built.
- Compiler wrappers handle many details automatically.
  - Spack feeds compiler wrappers to (cc, c++, f90, ...) to autoconf, cmake, gmake, ...
  - Wrappers select compilers, dependencies, and options under the hood.

## package.py

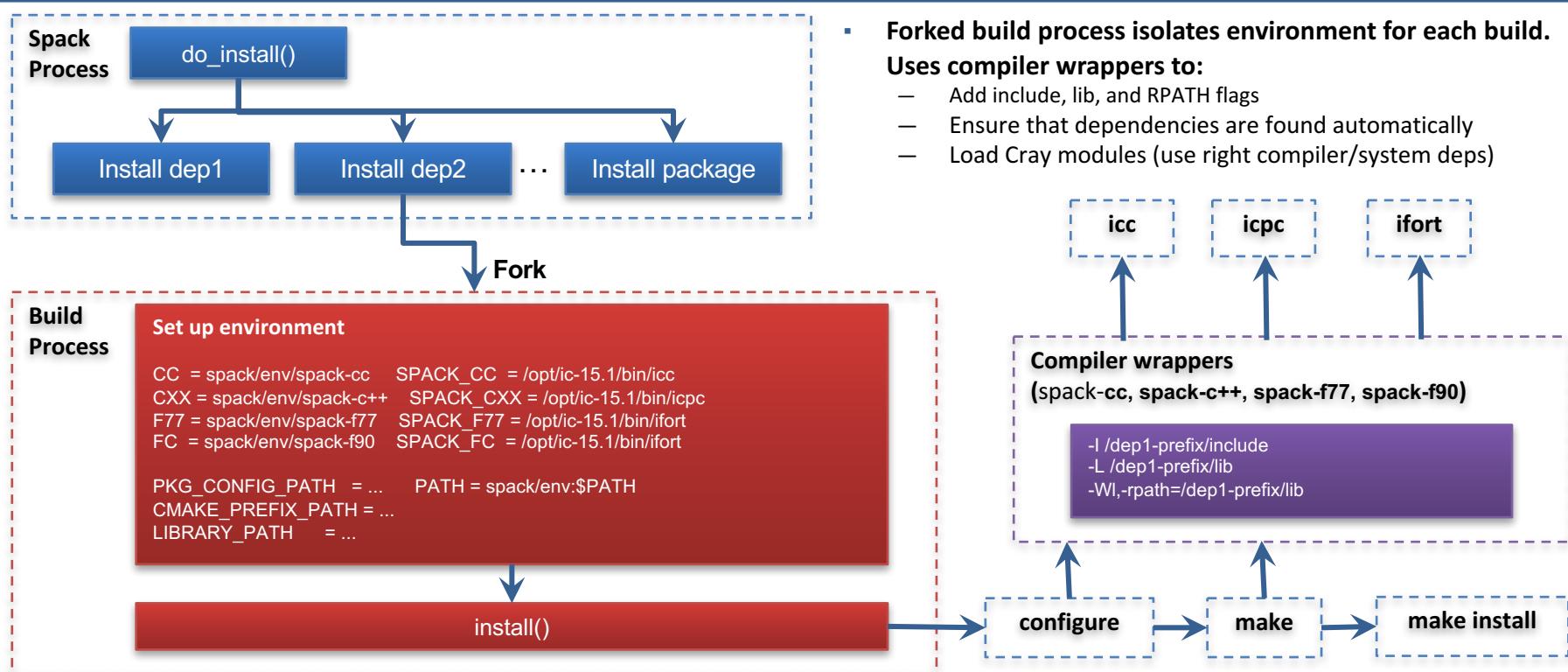
```
def install(self, spec, prefix):
    config_opts=['--prefix=' + prefix]

    if '~shared' in self.spec:
        config_opts.append('--disable-shared')
    else:
        config_opts.append('--enable-shared')

    configure(config_opts)
    make()
    make('install')
```



# Spack builds each package in its own compilation environment



# Writing Packages - Versions and URLs

## \$REPO/packages/mvapich/package.py

```
class Mvapich2(Package):
    homepage = "http://mvapich.cse.ohio-state.edu/"
    url = "http://mvapich.cse.ohio-state.edu/download/mvapich/mv2/mvapich2-2.2rc2.tar.gz"

    version('2.2rc2', 'f9082ffc3b853ad1b908cf7f169aa878')
    version('2.2b', '5651e8b7a72d7c77ca68da48f3a5d108')
    version('2.2a', 'b8ceb4fc5f5a97add9b3ff1b9cbe39d2')
    version('2.1', '0095ceecb19bbb7fb262131cb9c2cdd6')
    version('2.0', '9fb68a4111a8b6338e476dc657388b4')
```

- Package downloads are hashed with md5
  - or SHA-1, SHA-256, SHA-512
- Download URLs can be automatically extrapolated from URL.
  - Extra options can be provided if Spack can't extrapolate URLs
- Options can also be provided to fetch from VCS repositories

# Writing Packages – Variants and Dependencies

\$REPO/packages/petsc/package.py

```
class Petsc(Package):
    variant('mpi', default=True, description='Activates MPI support')
    variant('complex', default=False, description='Build with complex numbers')
    variant('hdf5', default=True, description='Activates support for HDF5 (only parallel)')

    depends_on('blas')
    depends_on('python@2.6:2.7')
    depends_on('mpi', when='+mpi')
```

- Variants are named, have default values and help text
- Other packages can be dependencies
  - **when** clause provides conditional dependencies
  - Can depend on specific versions or other variants

# Writing Packages – Build Recipes

- Functions wrap common ops
  - cmake, configure, patch, make, ...
  - **Executable** and **which** for new wrappers.
- Commands executed in clean environment
- Full Python functionality
  - Patch up source code
  - Make files and directories
  - Calculate flags
  - ...

\$REPO/packages/dyninst/package.py

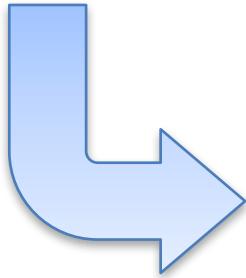
```
def install(self, spec, prefix):
    with working_dir("build", create=True):
        cmake("../", *std_cmake_args)
        make()
        make("install")

@when('@:8.1')
def install(self, spec, prefix):
    configure("--prefix=" + prefix)
    make()
    make("install")
```



# Create New Packages with `spack create`

```
$ spack create http://zlib.net/zlib-1.2.8.tar.gz
```



`$REPO/packages/zlib/package.py`

```
class Zlib(Package):
    # FIXME: Add a proper url for your package's homepage here.
    homepage = "http://www.example.com"
    url     = "http://zlib.net/zlib-1.2.8.tar.gz"
    version('1.2.8', '44d667c142d7cda120332623eab69f40')

    def install(self, spec, prefix):
        # FIXME: Modify the cmake line to suit your build system here.
```

- Spack create <url> will create a skeleton for a package
  - Spack reasons about URL, hash, version, build recipe.
- Spack edit <package> for subsequent changes

# Hands-on Time

# Deploying Spack at HPC Sites with Modules

# What are module files?

- **Purpose of module files**
  - modify dynamically user's env
  - manage multiple versions of same library / application
  - minimal user interface
- **State of the art**
  - different file formats
  - different layouts
  - different tools
- **Spack support for module files**
  - TCL / Non-hierarchical layout
  - Lua / Hierarchical layout

	TCL Non-hierarchical	Lua Hierarchical
Generator name	tcl	lmod
Default folder	share/spack/modules	share/spack/lmod
Compatible tools	Env. modules LMod	LMod

# How do module files look like?

```
$ module av  
  
--- share/spack/modules/linux-Ubuntu14-x86_64 ---  
autoconf-2.69-gcc-4.8-eud4m3w  
ghostscript-9.18-gcc-4.8-5n627qp  
lua-luaposix-33.4.0-gcc-4.8-s7lrtog  
autoconf-2.69-gcc-6.1.0-wo3aeho  
ghostscript-9.18-gcc-6.1.0-fsencrw  
libpciaccess-0.13.4-clang-3.8.0-oqbf3i7  
m4-1.4.17-clang-3.8.0-gstiykt  
pkg-config-0.29.1-gcc-4.8-6zuabcp  
automake-1.15-gcc-4.8-z42qh3r  
gmp-6.1.1-gcc-4.8-xsezhyp  
libpciaccess-0.13.4-gcc-4.8-62psbr2  
m4-1.4.17-gcc-4.8-drmfctm  
...  
...
```

```
##%Module1.0  
## Module file created by spack ...  
##  
## <full-spec>  
##  
module-whatis "autoconf @2.69"  
  
proc ModulesHelp { } {  
    puts stderr " Autoconf -- system  
    configuration part of autotools"  
}  
  
prepend-path PATH "<full-prefix>/autoconf-  
2.69-eud4m3weu6hfeu4o22xiwavnrarxjluo/bin"  
prepend-path MANPATH ...  
prepend-path CMAKE_PREFIX_PATH ...
```



# Life cycle of module files within spack

```
def do_install(self, **kwargs):
    # Check if the package is already installed
    if self.installed:
        return
    # Recurse to install dependencies
    for x in self.dependencies():
        x.do_install(**kwargs)
    # Run pre-install hooks
    self.pre_install_hooks()
    # Real installation happens here
    self.install(self, self_spec, self.prefix)
    # Module files generation happens here
    self.post_install_hooks()
```

```
def do_uninstall(self, **kwargs):
    # Run pre-uninstall hooks
    self.pre_uninstall_hooks()

    # "Uninstall" package
    self.remove_prefix()

    # Deletion of module files
    # happens here
    self.post_install_hooks()
```



# Package file API: package related content customization

```
def setup_dependent_environment(  
    self, spack_env, run_env, extension_spec  
):  
    """Set up the compile and runtime environments  
    for packages that depends on this one.  
    """  
  
    ...  
  
    if extension_spec.package.extends(self.spec):  
        run_env.prepend_path(  
            'LUA_PATH',  
            ';' .join(lua_patterns),  
            separator=';'  
        )
```

```
def setup_environment(  
    self, spack_env, run_env  
):  
    """Set up the compile and runtime environments  
    for a package.  
    """  
  
    ...  
  
    ld_library_path = join_path(  
        self.prefix, 'rlib', 'R', 'lib'  
    )  
  
    run_env.prepend_path(  
        'LIBRARY_PATH', ld_library_path  
    )
```

# “modules.yaml”: site related customizations

```
modules:  
  # Module generation needs to be activated  
  # Every module type will have its own section  
  enable :  
    - tcl  
    - lmod  
  # Configurations common to all module types  
  prefix_inspections:  
    bin:  
      - PATH  
  tcl:  
    # TCL specific section  
  lmod:  
    # LMOD specific section
```

Enable generation of tcl and lmod module files

Options that are common to all types of module files

Options that are specific to tcl and lmod respectively



# Inspection of package's installation prefix

```
modules:  
  prefix_inspections:  
    bin:  
      - PATH  
    man:  
      - MANPATH  
    share/man:  
      - MANPATH  
    lib:  
      - LIBRARY_PATH  
      - LD_LIBRARY_PATH  
    ...
```

```
$ ls -l ${PREFIX_ROOT}/autoconf-2.69-eud4m3weu6h  
total 8  
drwxrwxr-x 2 mculpo mculpo 4096 ago 17 08:35 bin  
drwxrwxr-x 6 mculpo mculpo 4096 ago 17 08:35 share
```

```
prepend-path PATH "${PREFIX_ROOT}/autoconf-2.69-  
eud4m3weu6h/bin"  
prepend-path MANPATH "${PREFIX_ROOT}/autoconf-2.69-  
eud4m3weu6h/share/man"  
prepend-path CMAKE_PREFIX_PATH  
"${PREFIX_ROOT}/autoconf-2.69-eud4m3weu6h"
```

# Apply a rule to a set of packages: “anonymous specs”

```
modules:  
  tcl:  
    all:  
      # Modifications that affect all packages  
      # (always evaluated first)  
    intel:  
      # Modifications that affect all 'intel'  
      # packages  
    '^openmpi':  
      # Modifications that affect all the packages  
      # that depends on 'openmpi'  
    '%gcc@4.4.7':  
      # Modifications that affect all the packages  
      # that are compiled with 'gcc@4.4.7'
```

## anonymous specs

- specs possibly without root package
- used for constraint-matching purposes

## Where are they used as a selection mechanism?

- module file suffixes
- prereq / autoload dependencies
- load external modules
- conflicts
- custom environment modifications



# Refresh or remove module files: `spack module` command

```
$ spack module refresh -m tcl hdf5
==> You are about to regenerate tcl module files for:

-- linux-Ubuntu14-x86_64 / gcc@4.8 --
narq2au hdf5@1.10.0-patch1
2qxlg4m hdf5@1.10.0-patch1

-- linux-Ubuntu14-x86_64 / gcc@6.1.0 --
o52kh5m hdf5@1.10.0-patch1

==> Do you want to proceed ? [y/n]
y
==> Regenerating tcl module files
```

```
spack module rm -m tcl hdf5
==> You are about to remove tcl module files the
following specs:

-- linux-Ubuntu14-x86_64 / gcc@4.8 --
narq2au hdf5@1.10.0-patch1
2qxlg4m hdf5@1.10.0-patch1

-- linux-Ubuntu14-x86_64 / gcc@6.1.0 --
o52kh5m hdf5@1.10.0-patch1

==> Do you want to proceed ? [y/n]
y
```

# Filter unwanted modifications to the environment

```
modules:  
  enable:  
    - tcl  
  
tcl:  
  all:  
    filter:  
      # Exclude changes to any of these variables  
      # in all packages, may be more selective  
    environment_blacklist:  
      - CPATH  
      - LIBRARY_PATH
```

## all:

- selects all packages

## filter:

- reserved to gather filtering directives
- right now supports only what is shown

## environment\_blacklist:

- takes a list of env variables
- that won't be modified in module files



# Prevent module files from being generated

```
modules:  
  enable:  
    - 'tcl'  
  
  tcl:  
    whitelist:  
      - 'gcc'  
      - 'llvm'  
      - 'intel'  
  
    blacklist:  
      - '%gcc@4.4.7'  
      - 'py-numpy'  
      - 'py-scipy'  
      - 'py-matplotlib'  
      - 'py-pip'
```

Whitelisted entries  
take precedence over  
blacklisted ones

Blacklist modules  
using anonymous specs

```
==> WHITELIST :  
  llvm@3.8.0%gcc@6.1.0+all_targets+clang+...  
  arch=linux-Ubuntu14-x86_64-gpwe4lu  
  [matches : llvm ]  
  
==> WRITE :  
  llvm@3.8.0%gcc@6.1.0+all_targets+clang+...  
  arch=linux-Ubuntu14-x86_64-gpwe4lu  
  [.../llvm-3.8.0-gcc-6.1.0-gpwe4lu]  
  
==> BLACKLIST :  
  lmod@6.4.5%gcc@4.8 arch=linux-Ubuntu14-x86_64-  
  sth2wun  
  [matches : %gcc@4.8 ]
```



# Change the naming of module files: control the hash length

```
modules:  
  enable:  
    - tcl  
    - lmod  
  tcl:  
    hash_length : 7 # Default value
```

--- /modules/linux-Ubuntu14-x86\_64 ---  
autoconf-2.69-gcc-4.8-eud4m3w  
ghostscript-9.18-gcc-4.8-5n627qp  
libedit-3.1-gcc-6.1.0-hzefu25  
lua-luaposix-33.4.0-gcc-4.8-s7lrtog  
...

```
modules:  
  enable:  
    - tcl  
    - lmod  
  tcl:  
    hash_length : 0
```

--- /modules/linux-Ubuntu14-x86\_64 ---  
autoconf-2.69-gcc-4.8  
ghostscript-9.18-gcc-4.8  
libedit-3.1-gcc-6.1.0  
lua-luaposix-33.4.0-gcc-4.8  
...

# Change the naming of module files: add custom suffixes

```
modules:  
  tcl:  
    all:  
      prefixes:  
        # Add '-openmp' suffix if satisfies +openmp  
        '+openmp': openmp  
        # ... or if it satisfies ^fftw+openmp  
        '^fftw+openmp': openmp  
        # Add 'openblas' if depends on openblas  
        '^openblas': openblas  
  
  netlib-scalapack:  
    prefixes:  
      '^openmpi': openmpi  
      '^mpich': mpich
```

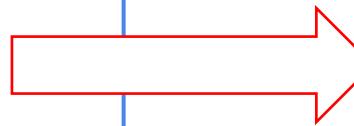
## suffixes:

- key: anonymous spec
- value: suffix to be added (if key matches)
- concatenate multiple matches



# Change the naming of module files: set the naming scheme

```
modules:  
tcl:  
  # In this directive you can use  
  # either literals or tokens that  
  # are understood by `Spec.format`  
  naming_scheme: '${PACKAGE}/${VERSION}-  
${COMPILERNAME}-${COMPILERVER}'  
all:  
  # The same applies to the  
  # List of conflicts  
conflict:  
  - '${PACKAGE}'
```



```
--- /modules/linux-Ubuntu14-x86_64 ---  
autoconf/2.69-gcc-4.8-eud4m3w  
ghostscript/9.18-gcc-4.8-5n627qp  
libedit/3.1-gcc-6.1.0-hzefu25  
...
```

## naming\_scheme:

- sets the layout for module files
- TCL / Non-hierarchical modules only

## conflict:

- sets a conflict directive in the module file
- behavior varies depending on the tool



# Add custom environment modifications

```
modules:  
  tcl:  
    all:  
      environment:  
        set: # Variable pointing to package prefix  
          '${PACKAGE}_ROOT': '${PREFIX}'  
        intel: # Customize intel package  
          environment:  
            set:  
              'CC': icc  
              ...  
          prepend_path:  
            'PATH': '<paths-to-be-prepended>'  
            ...
```

## environment:

- gathers all directives for env modifications
- key-value: *set, prepend\_path, append\_path*
- key only: *unset*
- tokens from `Spec.format` accepted in sub-directives



# Autoloading dependencies in module files

```
modules:  
  tcl:  
    all:  
      # Add code to load dependencies  
      # automatically. Either 'direct'  
      # or 'all' are allowed values.  
      autoload: 'direct'  
      # Load "literal" modules  
      load:  
        - 'intel'
```

```
...  
if ![ is-loaded openmpi-2.0.0-gcc-4.8-r2nqvn ]  
  puts stderr "Autoloading openmpi-2.0.0-gcc-4.8-r2nqvn"  
  module load openmpi-2.0.0-gcc-4.8-r2nqvn  
}  
  
if ![ is-loaded intel ] {  
  puts stderr "Autoloading intel"  
  module load intel  
}  
  
conflict hdf5  
conflict foo
```



# Lua - Hierarchical module files: Core/Compiler/MPI hierarchy

```
modules:  
  # Double colon is intentional:  
  # it overrides the same keyword  
  # when merging configuration files  
  # at different scopes  
  
enable::  
  - lmod  
  
lmod:  
  # Specify which compilers are to be  
  # considered Core compilers in the  
  # hierarchy  
  
core_compilers:  
  - 'gcc@4.8'
```



```
# Unuse the default TCL / Non-hierarchical modules  
$ module unuse ${SPACK_ROOT}/share/spack/modules/<arch>  
  
# Use the Core part of the Hierarchy  
$ module use ${SPACK_ROOT}/share/spack/lmod/<arch>/Core
```

## core\_compilers:

- accepts a list of compilers
- anything '%gcc@4.8' generates a module in the Core part of the hierarchy
- usually only utilities and compilers in Core

# Extend the hierarchy to other virtual providers [experimental]

```
modules:  
enable::  
- lmod  
lmod:  
core_compilers:  
- 'gcc@4.8'  
# Any virtual package in principle  
# can be used. It will be treated  
# as MPI in the usual hierarchy  
hierarchical_scheme:  
- lapack
```

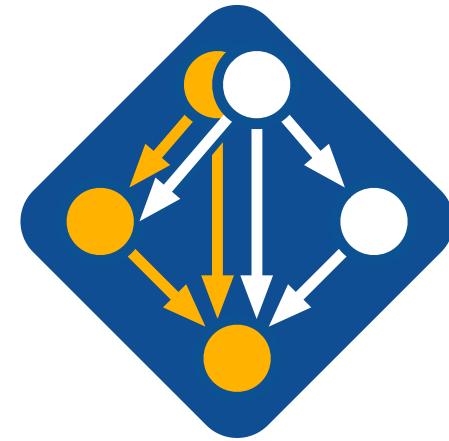
## hierarchical\_scheme:

- accepts a list of virtual packages
- added at the same level as MPI
- handles multiple virtual dependencies



# Join the Spack community!

- Contributing packages to Spack is simple
  - Make packages on your own system
  - Send a pull request to Spack to let others use them
  - GitHub guide to pull requests:  
<https://help.github.com/articles/creating-a-pull-request/>
  - See contributor guide in the Spack repository
  - Spack is licensed under **LGPLv2.1**
- We want more than just packages
  - New features
  - New documentation
  - Any contribution can help the community
- Spack has a helpful online community
  - Typically happy to respond to GitHub issues
  - Active mailing list on Google Groups:  
<https://groups.google.com/d/forum/spack>
- We hope to make distributing & using HPC software easy.



**<http://github.com/LLNL/spack>**



Lawrence Livermore  
National Laboratory