Managing HPC Software Complexity with Spack

HPC Knowledge Meeting ‘17
June 16, 2017
San Sebastián, Spain

http://github.com/LLNL/spack
Tutorial Materials

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

http://spack.readthedocs.io

Slides and hands-on scripts are in the “Tutorial” Section of the docs.

**Other Resources:**

- Spack GitHub repository: http://github.com/LLNL/spack
- Spack Documentation: http://spack.readthedocs.io
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:

  48 third party packages × 3 MPI versions
  - mvapich
  - mvapich2
  - OpenMPI

  Up to 7 compilers
  - Intel
  - GCC
  - XLC
  - Clang
  - PGI
  - Cray
  - Pathscale

  Oh, and 2-3 versions of each package

  3-ish Platforms
  - Linux
  - BlueGene
  - Cray

= ~7,500 combinations

Spack lowers barriers to software reuse by automating these builds!

http://github.com/LLNL/spack
What is the “production” environment for HPC?

- Someone’s home directory?
  - 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  15:00 - 15:10
4. -- Break --  15:30 - 16:00
5. Spack Basics (demo)  16:00 - 16:35
7. Making your own Spack Packages (demo)  16:55 - 17:30
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

http://github.com/LLNL/spack
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

- In November 2015, LLNL provided most of the contributions to Spack
- Since then, we’ve gone from 300 to over 1,500 packages
- Over 75% of lines of code in packages are now from external contributors.
  - Many external orgs now contribute to packages and core
- We are committed to sustaining Spack’s open source ecosystem!

http://github.com/LLNL/spack
The Spack community now spans DOE and beyond

- **30+ organizations**
- **140+ contributors**

Sharing over **1,500 packages** and growing

- **Other use cases:**
  - ARM using for entire compiler regression suite.
  - LIGO collaboration using for deployment
  - Intel using Spack to package ML software
  - NERSC using Spack on Cori: Cray support.
  - EPFL (Switzerland) contributing core features.
  - Fermi, CERN, BNL: high energy physics.
  - 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, Qt, UV-CDAT support

[http://github.com/LLNL/spack](http://github.com/LLNL/spack)
Related Work

Spack is not the first tool to automate builds
— Inspired by copious prior work

1. **“Functional” Package Managers**
   — Nix
   — GNU Guix

2. **Build-from-source Package Managers**
   — Homebrew
   — MacPorts

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

- **Hashdist**

Related Work

http://github.com/LLNL/spack
Building & Linking Basics
What’s a package manager?

- **Spack is a package manager**
  - **Does not** a 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**
- Manages package installation
- Manages dependency relationships
- Drives package-level build systems

**High Level Build System**
- Cmake, Autotools
- Handle library abstractions
- Generate Makefiles, etc.

**Low Level Build System**
- Make, Ninja
- Handles dependencies among commands in a single build

---

http://github.com/LLNL/spack
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/](https://www.gnu.org/software/make/)

2. GNU Autotools
   — Automake: [https://www.gnu.org/software/automake/](https://www.gnu.org/software/automake/)
   — Autoconf: [https://www.gnu.org/software/autoconf/](https://www.gnu.org/software/autoconf/)
   — Libtool: [https://www.gnu.org/software/libtool/](https://www.gnu.org/software/libtool/)

3. CMake:
   — [https://cmake.org](https://cmake.org)

Spack has built-in support for these plus Waf, Perl, Python, and R
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

<table>
<thead>
<tr>
<th>CC</th>
<th>CFLAGS</th>
<th>LDFLAGS</th>
</tr>
</thead>
<tbody>
<tr>
<td>CXX</td>
<td>CXXFLAGS</td>
<td></td>
</tr>
<tr>
<td>FC</td>
<td>FFLAGS</td>
<td>LIBS</td>
</tr>
<tr>
<td>F77</td>
<td>FFLAGS</td>
<td>CPP</td>
</tr>
</tbody>
</table>

http://github.com/LLNL/spack
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

<table>
<thead>
<tr>
<th>CC</th>
<th>CFLAGS</th>
<th>LDFLAGS</th>
</tr>
</thead>
<tbody>
<tr>
<td>CXX</td>
<td>CXXFLAGS</td>
<td>LIBS</td>
</tr>
<tr>
<td>FC</td>
<td>FFLAGS</td>
<td>CPP</td>
</tr>
<tr>
<td>F77</td>
<td>FFLAGS</td>
<td></td>
</tr>
</tbody>
</table>
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

- 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 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 -g3" 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
```
`spack list` shows what packages are available

| Spack has over 1,500 packages now. |  | 
| --- | --- | --- |
|  |  |  |
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.

`$ 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
  SAMRAI@3.9.1  graphlib@2.0.0
  adept-utils@1.0  gtkplus@2.24.25
  atk@2.14.0  libxcb@1.11
  boost@1.55.0  libxml2@2.9.2
  cairo@1.14.0  libtiff@4.0.3
  callpath@1.0.2  pango@1.36.8
  dyninst@8.1.2  qt@4.8.6
  hdf5@1.8.13  qt@5.4.0
  icu@54.1  ravel@1.0.0
  jpeg@5.1  readline@6.3
  icu@2.14.0  scotch@6.0.3
  libdap@6.0.3  sqlite@2.1.0
  libelf@0.8.13  starpu@1.1.4
  libffi@3.1  stat@2.1.0
  libgkg@2.31.2  stat@2.1.0
  libpng@1.6.16  starpu@1.1.4
  libas Cliff@2.1.0  starpu@1.1.4
  libas Cliff@2.1.0
```

---

Browser http://github.com/LLNL/spack
Users can query the full dependency configuration of installed packages.

```bash
$ 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

$ spack find -dl callpath
===> 2 installed packages.
-- linux-x86_64 / clang@3.4 -------
xv2clz2    callpath@1.0.2
ckjazss    ^adept-utils@1.0.1
3w543m4    ^boost@1.59.0
ft7znm6    ^mpich@3.1.4
qgnuet3    ^dyninst@8.2.1
3w543m4    ^boost@1.59.0
g65rdud    ^libdwarf@20130729
cj5p5fk    ^libelf@0.8.13
                                   udltshs  callpath@1.0.2
                                   rsfu7fb  ^adept-utils@1.0.1
                                   ybet64y  ^boost@1.55.0
                                   aa4ar6i  ^mpich@3.1.4
                                   tmnngse5 ^dyninst@8.2.1
                                   ybet64y  ^boost@1.55.0
                                   ^libdwarf@20130729
                                   g2mxrl2  ^libelf@0.8.13
                                   ynpai3j  ^libelf@0.8.13
                                   g2mxrl2  ^libelf@0.8.13
                                   ynpai3j  ^libelf@0.8.13
                                   g2mxrl2  ^libelf@0.8.13
                                   ynpai3j  ^libelf@0.8.13
                                   ynpai3j  ^libelf@0.8.13
                                   g2mxrl2  ^libelf@0.8.13
```

Expand dependencies with `spack find -d`

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

http://github.com/LLNL/spack
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:
- 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:
```

http://github.com/LLNL/spack
Hands-on Time

Follow script at http://spack.readthedocs.io
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.

- 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

http://github.com/LLNL/spack
Spack Specs can constrain versions of dependencies

- 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 install mpi @leaks %intel@12.1 ^libelf@0.8.12
```
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.

User input: abstract spec with some constraints

Abstract, normalized spec with some dependencies.

Concretize

Concrete spec is fully constrained and can be passed to install.

Detailed provenance is stored with the installed package

mpileaks ^callpath@1.0+debug ^libelf@0.8.11

spec.yaml

spec:
  - mpileaks:
    arch: linux-x86_64
    compiler:
      name: gcc
      version: 4.9.2
    dependencies:
      adept-utils: kszrtkpbzac3ss2ixcjkcorlaybnptp4
    callpath: bah5f4h4d2n47mgycej2mtrnrivvxy77
    mpi: aakorfj23yjyjmd6abekpejctj172t3
    hash: 33b2jxwi77yeyj3d55pryes7qhyvnu3jh
    variants: {}
    version: 1.0
  - adept-utils:
    arch: linux-x86_64
    compiler:
      name: gcc
      version: 4.9.2
    dependencies:
      boost: tesejy7fhepe5ksspjin5dkkb7gnow1q
    mpi: aakorfj23yjyjmd6abekpejctj172t3
    hash: kszrtkpbzac3ss2ixcjkcorlaybnptp4
    variants: {}
    version: 1.0.1
  - boost:
    arch: linux-x86_64
    compiler:
      name: gcc
      version: 4.9.2
    dependencies: {}
    hash: tesejy7fhepe5ksspjin5dkkb7gnow1q
    variants: {}
    version: 1.59.0
...
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
 ^boost@1.61.0
 ^mpi
 ^callpath
 ^dyninst
 ^libdwarf

 ^libelf

 ^libtool
 ^m4
 ^libsigsegv
 ^callpath
 ^dyninst
 ^libdwarf
 ^libelf

 ^openmpi
 ^zlib
 ^bzip2
 ^openmp
 ^m4
 ^libsigsegv
 ^callpath
 ^callpath
 ^dyninst
 ^libdwarf
 ^libelf

Concretized
-------------------------------
mpileaks@1.0
 %gcc@5.3.0
 arch=darwin-elcapitan-x86_64

 ^adept-utils@1.0.1
 ^boost@1.61.0
 ^mpi
 ^callpath
 ^dyninst
 ^libdwarf

 ^libelf

 ^libtool
 ^m4
 ^libsigsegv
 ^callpath
 ^callpath
 ^dyninst
 ^libdwarf

 ^libelf
```

http://github.com/LLNL/spack
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

```python
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

- 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 activate py-numpy @1.10.4
```

```
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/opt/
linux-redhat6-x86_64/
gcc-4.7.2/
  python-2.7.12-6y6vvaw/
    python/
      py-numpy
```

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

```bash
$ spack activate py-numpy @1.10.4
```
Extensions must be activated into extendee

- Python unaware of numpy installation

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

$ spack load python
$ spack load py-numpy
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
- Spack supports external repositories that can be layered on top of the built-in packages
- Custom packages can depend on built-in packages (or packages in other repositories)

<table>
<thead>
<tr>
<th>Proprietary packages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pathological build cases</td>
</tr>
<tr>
<td>“Standard” packages</td>
</tr>
</tbody>
</table>

my_repo

var/spack/repos/builtin

http://github.com/LLNL/spack
Fetching source code for spack

- **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

```
spack install mpiruns
Load package from repository
Concretize
Recursively install dependencies
Fetch package source
Build software
```

- User Cache
- Spack Cache
- The Internet

http://github.com/LLNL/spack
Adding custom compiler flags

$ spack install hdf5 cflags='-O3 -g -fast -fpack-struct'

- This installs HDF5 with the specified flags
  - Flags are injected via Spack’s compiler wrappers (discussed later).

- Flags are propagated to dependencies automatically
  - Flags are included in the DAG hash
  - Each build with different flags 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
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 `var/spack/repos/builtin`

```python
$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={}'.format(prefix))
        make()
        make('install')
```

http://github.com/LLNL/spack
Spack packages are *templates* for builds

- Each package has one class
  - zlib for Intel compiler and zlib for GCC compiler are built with the same recipe.

- Can add conditional logic using spec syntax
  - Think of package as *translating* a concrete DAG to build instructions.
  - Dependencies are already built
  - No searching or testing; just do what the DAG says

- 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**

```python
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')
```

---

http://github.com/LLNL/spack
Spack builds each package in its own compilation environment

- Forked build process isolates environment for each build. Uses compiler wrappers to:
  - Add include, lib, and RPATH flags
  - Ensure that dependencies are found automatically
  - Load Cray modules (use right compiler/system deps)

**Set up environment**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CC</td>
<td>spack/env/spack-cc</td>
</tr>
<tr>
<td>CXX</td>
<td>spack/env/spack-c++</td>
</tr>
<tr>
<td>F77</td>
<td>spack/env/spack-f77</td>
</tr>
<tr>
<td>FC</td>
<td>spack/env/spack-f90</td>
</tr>
<tr>
<td>PKG_CONFIG_PATH</td>
<td>...</td>
</tr>
<tr>
<td>CMAKE_PREFIX_PATH</td>
<td>...</td>
</tr>
<tr>
<td>LIBRARY_PATH</td>
<td>...</td>
</tr>
<tr>
<td>PATH</td>
<td>spack/env:$PATH</td>
</tr>
</tbody>
</table>

- **Compiler wrappers** (spack-cc, spack-c++, spack-f77, spack-f90)
  - icc
  - icpc
  - ifort
  - `-I /dep1-prefix/include`
  - `-L /dep1-prefix/lib`
  - `-Wl,-rpath=/dep1-prefix/lib`

**Build Process**

- `do_install()`
- Forked build process isolates environment for each build.
- `install()`
Writing Packages - Versions and URLs

```python
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', 'f9082ffe3b853ad1b908cf7f169aa878')
    version('2.2b', '5651e8b7a72d7c77ca68da4f3a5d108')
    version('2.2a', 'b8ceb4fc5f5a97add9b3ff1b9cbe39d2')
    version('2.1', '0095ceecb19b7b7b262131cb9c2cdd6')
```

- Package downloads are hashed with MD5 by default
  - Also supports SHA-1, SHA-256, SHA-512
  - We’ll be switching to SHA-256 or higher soon.

- 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

```python
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
  - ...

```python
$REPO/packages/dyninst/package.py

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

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

http://github.com/LLNL/spack
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.
  — Generates boilerplate for Cmake, Makefile, autotools, Python, R, Waf, Perl
    — Not intended to completely write the package, but gets you 80% of the way there.

• spack edit <package> for subsequent changes
Hands-on Time
Spack can be configured to automatically generate modules

- **Supported module systems**
  - Lmod
  - TCL Modules
  - Dotkit (LLNL-only, being phased out)

- **Configuration**
  - Specify naming schemes w/YAML files
  - Lmod generator uses compilers & virtual deps (MPI, BLAS, LAPACK) for hierarchy
  - Can customize:
    - Hash length
    - Suffixes based on variants
    - Whitelist/blacklist using spec matchers
    - more...

<table>
<thead>
<tr>
<th></th>
<th>TCL Non-hierarchical</th>
<th>Lua Hierarchical</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Generator name</strong></td>
<td>tcl</td>
<td>lmod</td>
</tr>
<tr>
<td><strong>Default location</strong></td>
<td>share/spack/modules</td>
<td>share/spack/lmod</td>
</tr>
</tbody>
</table>

See spack.readthedocs.io for more!

http://github.com/LLNL/spack
What’s on the Spack road map?

1. **Environments** (like Conda/virtualenv, nix/guix profiles)
   - virtualenv, but not just for Python
   - Allow creation of arbitrarily many prefixes
   - Adjustable consistency:
     - Loose library consistency for run environments
     - Strict library consistency for development environments

2. **Secure binary packaging**
   - Collaboration with CERN, Kitware
   - Cache relocatable, signed, optimized binaries for Spack builds
   - Install in seconds, not hours

3. **Concretization improvements**
   - Faster: Use a real SAT solver with backtracking
   - Build with the packages that are there already (don’t rebuild if possible)
   - Allow more flexible notions of “reproducing” a build
     - e.g.: preserve only versions & variants, but change compiler/architecture/etc.
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

We hope to make distributing & using HPC software easy!