Managing HPC Software Complexity with Spack

Supercomputing 2017 Full-day Tutorial
November, 2017
Denver, Colorado
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.

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

Tweet at us!

@spackpm #SC17
Tutorial Presenters

Todd Gamblin
Gregory Becker
Greg Lee
Matt Legendre

Massimiliano Culpo
Adam Stewart
Mario Melara
The complexity of the exascale ecosystem threatens productivity.

- Every application has its own stack of dependencies.
- Developers, users, and facilities dedicate (many) FTEs to building & porting.
- Often trade reuse and usability for performance.

<table>
<thead>
<tr>
<th>15+ applications</th>
<th>80+ software packages</th>
<th>5+ target architectures/platforms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Up to 7 compilers</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Intel GCC Clang XL</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>PGI Cray NAG</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10+ Programming Models</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OpenMPI MPICH MVAPICH OpenMP CUDA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OpenACC Dharma Legion RAJA Kokkos</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2-3 versions of each package + external dependencies</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

= up to 1,260,000 combinations!

We must make it easier to rely on others’ software!
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.
Spack is a flexible package manager for HPC

- How to install Spack:
  
  $ git clone https://github.com/spack/spack
  $ . 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/spack/spack
@spackpm
Who can use Spack?

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
Spack is used worldwide

Sessions at spack.readthedocs.io by location, since April 2017
Contributions to Spack are growing!

- In November 2015, LLNL provided most of the contributions to Spack
- Since then, we’ve gone from 300 to over 2,100 packages
- Most packages are from external contributors!
- Many contributions in core, as well.
- We are committed to sustaining Spack’s open source ecosystem!
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

github.com/spack/spack
Spack v0.11.0 was just released

- Major new features:
  1. Binary packaging support (we’ll use this today)
  2. An interface for passing build info between packages (demoed today)
  3. Test dependencies
  4. Multi-valued variants
  5. More control over compiler flags
  6. Better help output for Spack’s many commands
  7. Better build output handling and error display
  8. Numerous other bug fixes and speed improvements

- Over 2,100 packages

- Spack now has its own GitHub organization ([https://github.com/spack](https://github.com/spack))
What is on the Spack road map?

- **Features:**
  - Build/test infrastructure to build and validate package builds regularly
  - Hosting infrastructure to make select package configurations available as binaries
  - Support for creating custom isolated “environments” (cf. Conda, Nix, Virtualenv)
  - Better dependency resolution (concretization) and compiler support
  - Distributed-parallel builds

- **Collaborations:**
  - Work with US Exascale Program code teams to coordinate software releases
  - Work with DOE labs to build and test software on large supercomputers
  - Continue working with broader community to integrate contributions

[github.com/spack/spack]
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

- Hashdist

http://hpcugent.github.io/easybuild/
https://hashdist.github.io

github.com/spack/spack
# Tutorial Overview (times are estimates)

<table>
<thead>
<tr>
<th></th>
<th>For everyone:</th>
<th></th>
<th></th>
<th>For packagers:</th>
<th></th>
<th></th>
<th>For HPC centers:</th>
<th></th>
<th></th>
<th>For everyone:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Welcome &amp; Overview</td>
<td>8:30 - 8:50</td>
<td></td>
<td>Basics of Building and Linking</td>
<td>8:50 - 9:00</td>
<td></td>
<td>Deploying Spack with Lmod at HPC Facilities (hands-on)</td>
<td>3:30 - 4:15</td>
<td></td>
<td>Build your own packages with help from the Spack team!</td>
<td>4:15 – 5:00</td>
</tr>
<tr>
<td>2.</td>
<td>Basics of Building and Linking</td>
<td>9:00 - 9:45</td>
<td></td>
<td>Core Spack Concepts</td>
<td>9:45 - 10:00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.</td>
<td>Spack Basics (hands on)</td>
<td>10:00 - 10:30</td>
<td></td>
<td>Configuration (hands-on)</td>
<td>10:30 - 11:00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.</td>
<td>Core Spack Concepts</td>
<td>11:00 – 12:00</td>
<td></td>
<td>Making your own Packages (hands on)</td>
<td>12:00 - 1:30</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.</td>
<td>Break</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.</td>
<td>Tutorials Lunch (Four Seasons Ballroom)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.</td>
<td>Advanced Packaging (build systems)</td>
<td>1:30 – 2:15</td>
<td></td>
<td>Coffee Break</td>
<td>3:30 - 3:30</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8.</td>
<td>Advanced Packaging (virtual dependencies: BLAS/LAPACK/MPI)</td>
<td>2:15-3:00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

github.com/spack/spack
Tutorial reviews are online this year

Please submit your feedback on this tutorial at:


Your reviews help us keep the tutorial going!
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**
- 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
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
### 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

```make
<edit Makefile>
made 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>
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 \n  --disable-bar
```

**Environment variables**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CC</td>
<td>C compiler</td>
</tr>
<tr>
<td>CXX</td>
<td>C++ compiler</td>
</tr>
<tr>
<td>FC</td>
<td>Fortran 77</td>
</tr>
<tr>
<td>F77</td>
<td>Fortran 77</td>
</tr>
<tr>
<td>CFLAGS</td>
<td>Compiler flags for C</td>
</tr>
<tr>
<td>CXXFLAGS</td>
<td>Compiler flags for C++</td>
</tr>
<tr>
<td>LIBS</td>
<td>Library flags for C/C++</td>
</tr>
<tr>
<td>CPP</td>
<td>Preprocessor flags</td>
</tr>
<tr>
<td>FFLAGS</td>
<td>Fortran 77 flags</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

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

#### Configure options

```bash
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
$ spack install mpileaks@3.3
$ spack install mpileaks@3.3 %gcc@4.7.3
$ spack install mpileaks@3.3 %gcc@4.7.3 +threads
$ spack install mpileaks@3.3 cppflags="-O3 -g3"
$ spack install mpileaks@3.3 os=CNL10 target=haswell
$ spack install mpileaks@3.3 ^mpich@3.2 %gcc@4.9.3

unconstrained
@ custom version
% custom compiler
 +/- build option
setting compiler flags
setting target for X-compile
^ dependency information
Spack has over 2,100 packages now.
$ spack find

`spack find` shows what is installed

<table>
<thead>
<tr>
<th>All the versions coexist!</th>
</tr>
</thead>
<tbody>
<tr>
<td>– Multiple versions of same package are ok.</td>
</tr>
</tbody>
</table>

| 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. |

---

<table>
<thead>
<tr>
<th>$ spack find</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>spack find</code> shows what is installed</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>All the versions coexist!</th>
</tr>
</thead>
<tbody>
<tr>
<td>– Multiple versions of same package are ok.</td>
</tr>
</tbody>
</table>

| 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. |

---

<table>
<thead>
<tr>
<th>$ spack find</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>spack find</code> shows what is installed</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>All the versions coexist!</th>
</tr>
</thead>
<tbody>
<tr>
<td>– Multiple versions of same package are ok.</td>
</tr>
</tbody>
</table>

| 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 ---------  -- 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 ---------  -- linux-x86_64 / gcc@4.9.2 ---------
xv2clz2  callpath@1.0.2
ckjazss  ^adept-utils@1.0.1
3ws43m4  ^boost@1.59.0
ft7znm6  ^mpich@3.1.4
qgnet3   ^dyninst@8.2.1
3ws43m4  ^boost@1.59.0
3ws43m4  ^boost@1.59.0
g65rdud  ^libdwarf@20130729
ckjazss  ^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
```

- 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:
  ...```
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*
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.

```bash
$ spack install mpiLeaks %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:

  ```bash
  $ 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:

  ```bash
  $ spack install mpileaks ^mpi@2
  ```
Concretization fills in missing configuration details when the user is not explicit.

User input: abstract spec with some constraints

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

detailed provenance is stored with the installed package
Use `spack spec` to see the results of concretization

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

Normalized
-------------------------------
mpileaks
  ^adep-util
  ^boost@1.42:
    ^mpi
  ^callpath
  ^dyninst
    ^libdwarf
    ^libelf

Concretized
-------------------------------
mpileaks@1.0%gcc@5.3.0 arch=darwin-elcapitan-x86_64
  ^adep-util@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=iostream+locale+log=math=mpi+multithreaded+program_options
    -python+random +regex+serialization=shared+signals+singlethreaded+system
    +test=thread_timer=wave arch=darwin-elcapitan-x86_64
  ^bzip2@1.0.6%gcc@5.3.0 arch=darwin-elcapitan-x86_64
    ^libelf@1.2.8%gcc@5.3.0 arch=darwin-elcapitan-x86_64
  ^openmpi@2.0.0%gcc@5.3.0-mpi-mpi-omm-oms2-slxrm-sqlite3-thread_multipart+verbs+vt arch=darwin-elcapitan-x86_64
    ^mpicc@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@9.2.0%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
```
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

```
$ 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
Extensions must be activated into extendee

$ spack extensions python
  ==> python@2.7.12% gcc@4.9.3 ~tk-ucs4 arch=linux-redhat7-x86_64-i25k4oi
  ==> 118 extensions:
  -
    py-autopep8   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

- Python unaware of numpy installation

$ 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)

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

$ spack load python
$ spack load py-numpy
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 cannot 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)

```
$ spack repo create /path/to/my_repo
$ spack repo add my_repo
$ spack repo list
=> 2 package repositories.
```

Proprietary packages
Pathological build cases
“Standard” packages

my_repo

var/spack/repos/builtin
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

---

github.com/spack/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
Hands-on Time: Configuration

Follow script at http://spack.rtfd.io
Under “Tutorial: Spack 101"
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={0}'.format(prefix))

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

- Package collections are repos
  - Spack has a “builtin” repo in `var/spack/repos/builtin`
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.

```python
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')
```

- Dependencies are already built — No searching or testing; just do what the DAG says
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

CC = spack/env/spack-cc    SPACK_CC = /opt/ic-15.1/bin/icc
CXX = spack/env/spack-c++  SPACK_CXX = /opt/ic-15.1/bin/icpc
F77 = spack/env/spack-f77  SPACK_F77 = /opt/ic-15.1/bin/ifort
FC = spack/env/spack-f90   SPACK_FC = /opt/ic-15.1/bin/ifort
PKG_CONFIG_PATH = ...      PATH = spack/env:$PATH
CMAKE_PREFIX_PATH = ...
LIBRARY_PATH = ...
```

```
install()
```

```
Fork
```

```
do_install()
```

```
Install dep1  Install dep2  ...  Install package
```

```
Build Process
```

```
configure  make  make install
```

```
Compiler wrappers (spack-cc, spack-c++, spack-f77, spack-f90)
```

```
-cc  icpc  ifort
```

```
-1 /dep1-prefix/include
-L /dep1-prefix/lib
-Wl,-rpath=/dep1-prefix/lib
```
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', 'f9082fffc3b853ad1b908cf7f169aa878')
    version('2.2b', '5651e8b7a72d7c77ca68da48f3a5d108')
    version('2.2a', 'b8ceb4fc5f5a97add9b3ff1b9cbe39d2')
    version('2.1', '0095ceecb19bbb7fb262131cb9c2cdd6')
```

- 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

github.com/spack/spack
Writing Packages – Variants and Dependencies

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

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

```python
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")
```

▪ 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

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

```python
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")
```

▪ 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

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

```python
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

```python
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: Creating Packages

Follow script at [http://spack.rtfd.io](http://spack.rtfd.io)
Under “Tutorial: Spack 101"
Hands-on Time: Advanced Packaging

Follow script at http://spack.rtfd.io
Under “Tutorial: Spack 101"
Hands-on Time: Module Files

Follow script at http://spack.rtfd.io
Under “Tutorial: Spack 101”
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/](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!