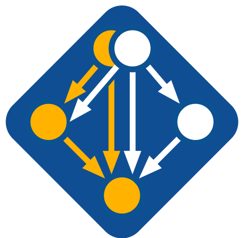


Managing HPC Software Complexity with Spack

Full-day Tutorial
1st Workshop on NSF and DOE High Performance Computing Tools
July 10, 2019
Eugene, Oregon

The most recent version of these slides can be found at:
https://spack.readthedocs.io/en/latest/tutorial.html

LLNL-PRES-806064

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

github.com/spack/spack

Lawrence Livermore National Laboratory

ECP EXASCALE COMPUTING PROJECT

Tutorial Materials

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

bit.ly/spack-tutorial

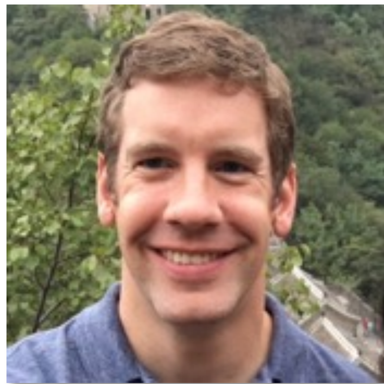
Slides and hands-on scripts are in the "Tutorial" Section of the docs.

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

Tweet at us!

@spackpm

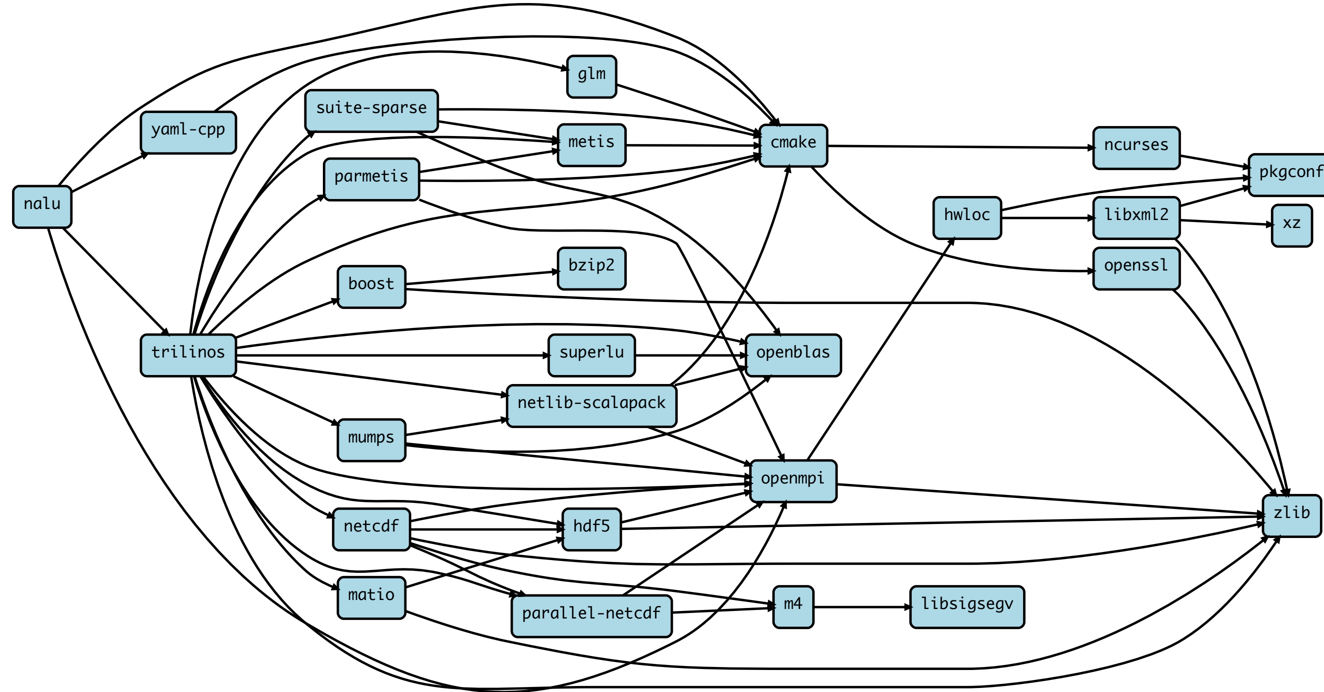
Tutorial Presenters



Todd Gamblin

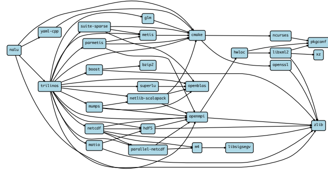
Greg Becker

Software complexity in HPC is growing

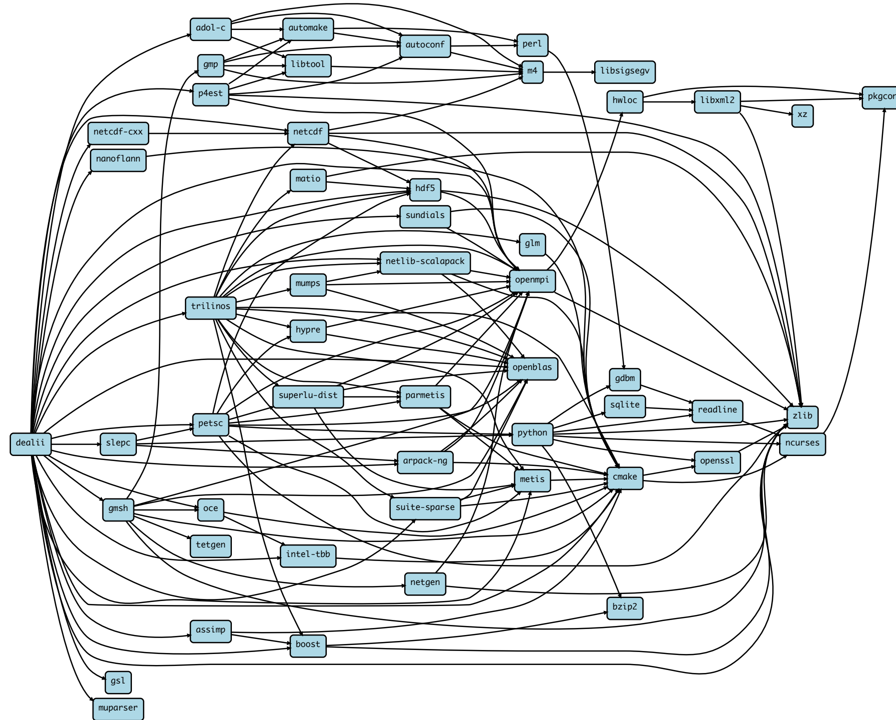


Nalu: Generalized Unstructured Massively Parallel Low Mach Flow

Software complexity in HPC is growing

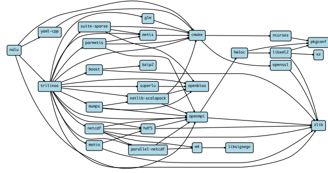


Nalu: Generalized Unstructured Massively Parallel Low Mach Flow

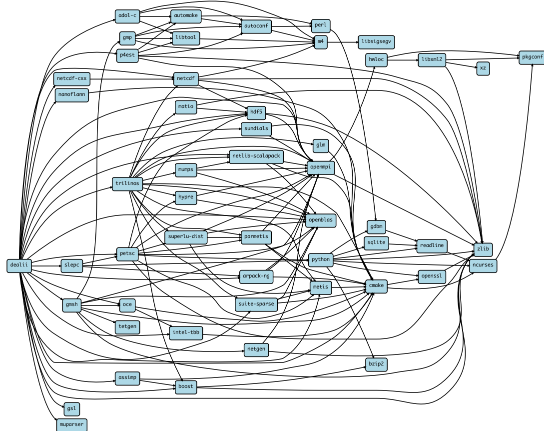


dealii: C++ Finite Element Library

Software complexity in HPC is growing



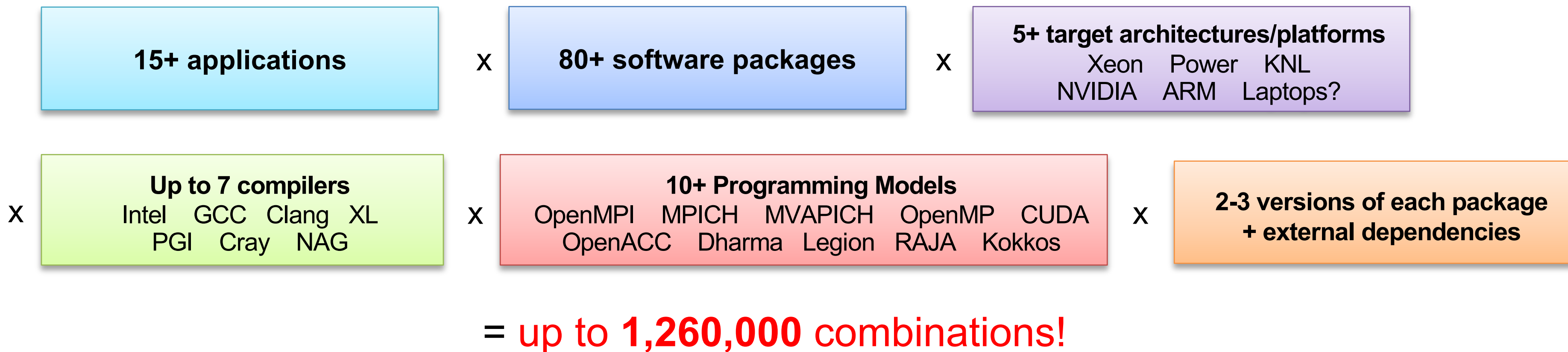
Nalu: Generalized Unstructured Massively Parallel Low Mach Flow



dealii: C++ Finite Element Library

R Miner: R Data Mining Library

The complexity of the exascale ecosystem threatens productivity.



= up to **1,260,000** combinations!

- 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.

We must make it easier to rely on others' software!

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.

Spack is a flexible package manager for HPC

- How to install Spack:

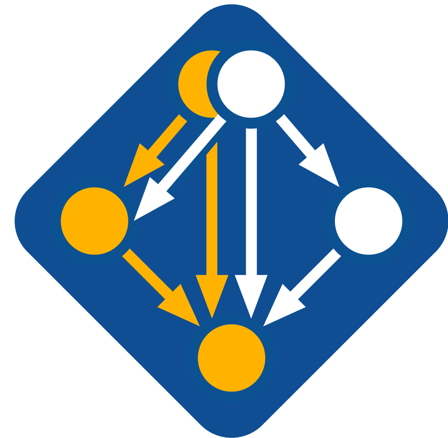
```
$ git clone https://github.com/spack/spack
$ . spack/share/spack/setup-env.sh
```

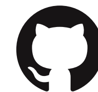
- How to install a package:

```
$ 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

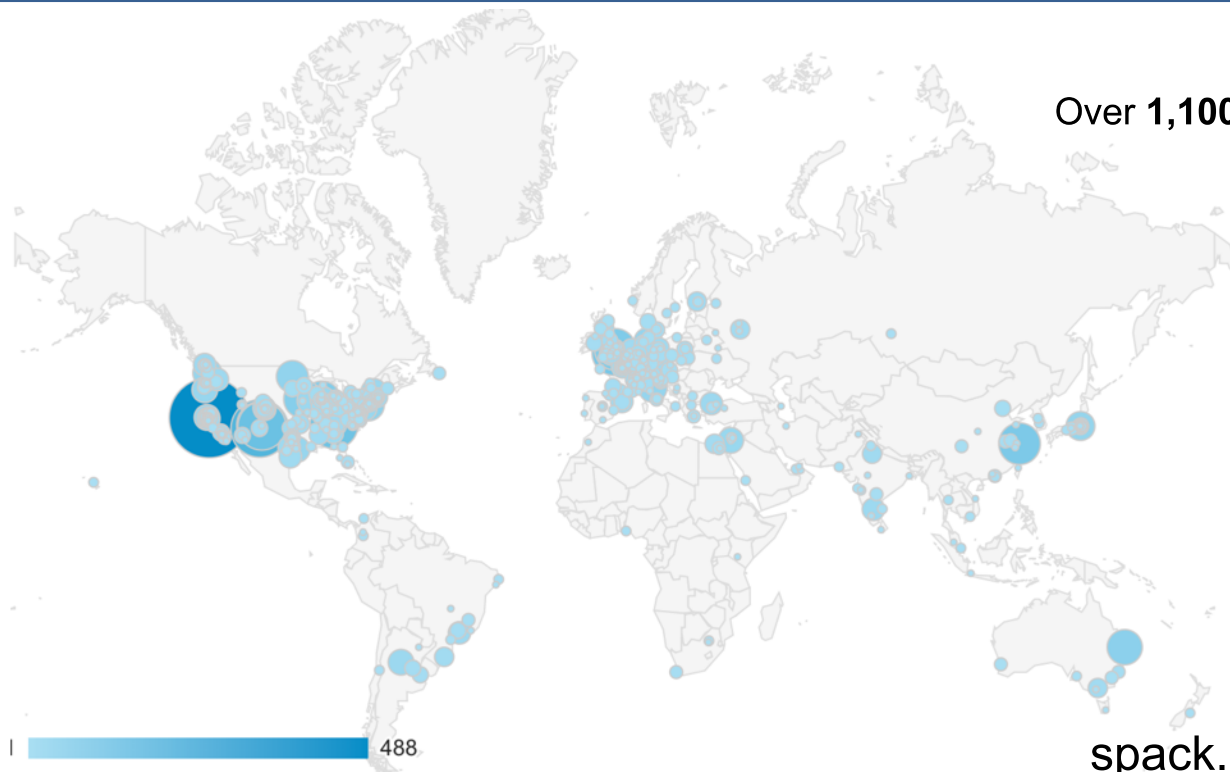
@spackpm

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!

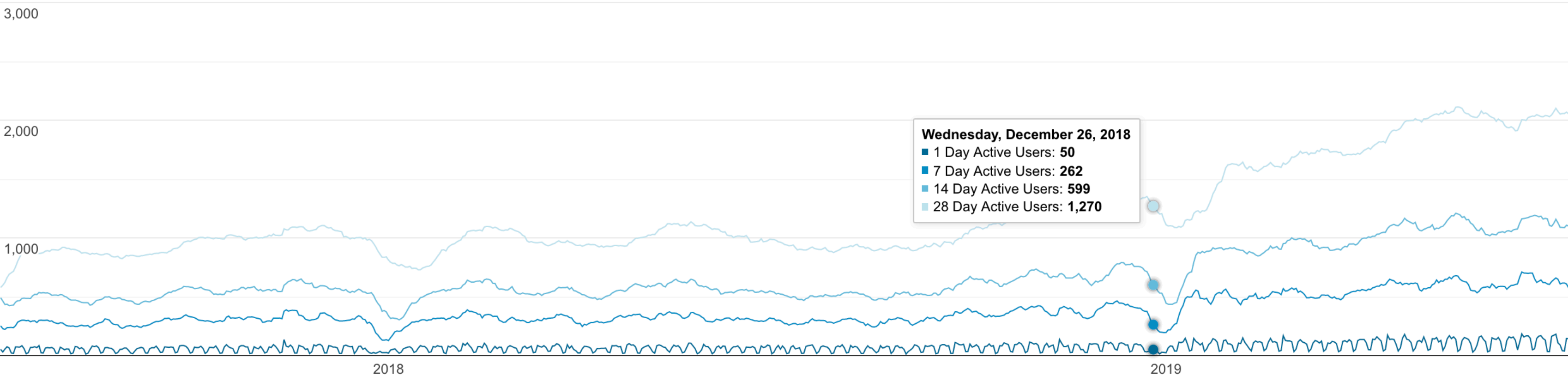


Over **3,200** software packages
Over **1,100** monthly active users (on docs site)

Over **400** contributors
from labs, academia, industry

Plot shows sessions on
spack.readthedocs.io for one month

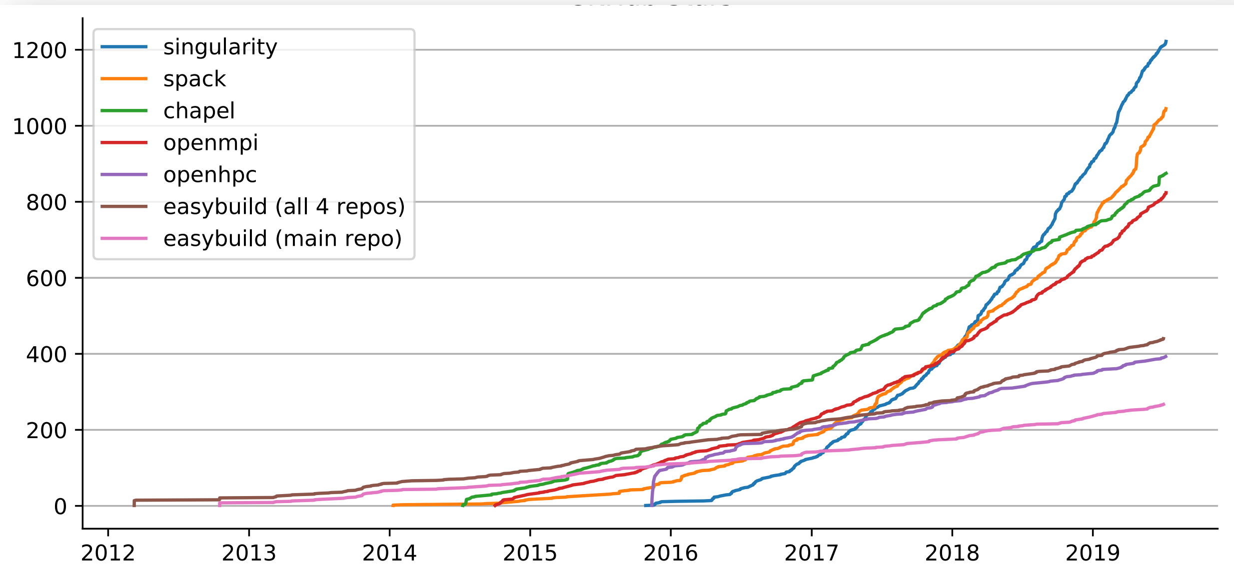
Active Users on the spack.readthedocs.io



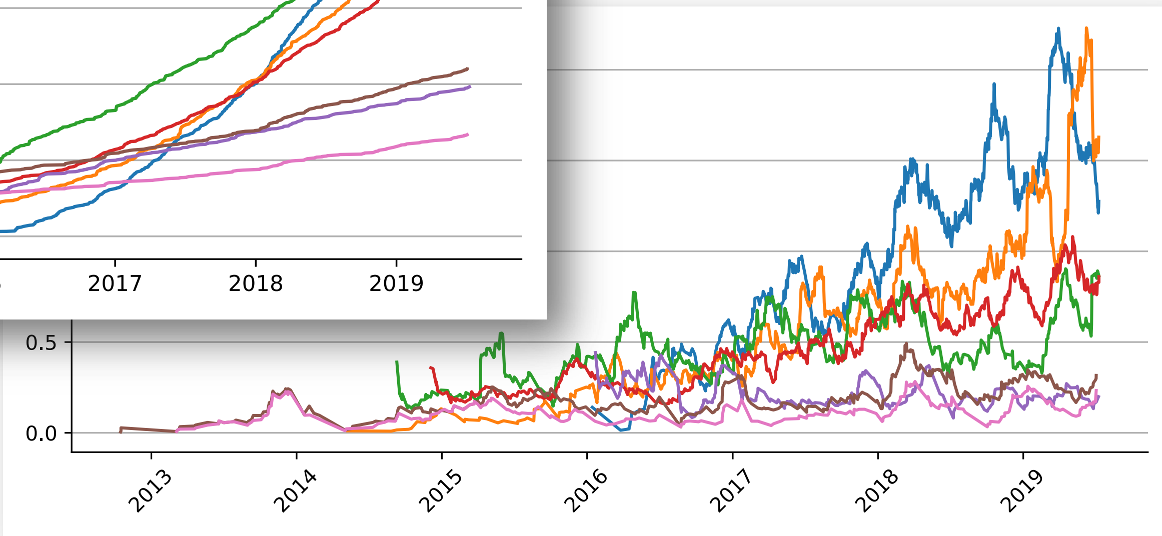
1 Day Active Users	7 Day Active Users	14 Day Active Users	28 Day Active Users
142	574	1,100	2,054
% of Total: 100.00% (142)	% of Total: 100.00% (574)	% of Total: 100.00% (1,100)	% of Total: 100.00% (2,054)

GitHub Stars: Spack and some other popular HPC projects

Stars over time



Stars per day
(same data, 60-day window)



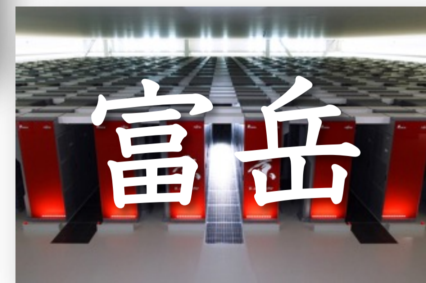
Spack is being used on many of the top HPC systems

- At HPC sites for software stack+ modules
  - Reduced Summit deploy time from 2 weeks to 12 hrs.
  - EPFL deploys its software stack with Jenkins + Spack
  - NERSC, LLNL, ANL, other US DOE sites
  - SJTU in China
- Within ECP as part of their software release process
  - ECP-wide software distribution
  - SDK workflows
- Within High Energy Physics (HEP) community
  - HEP (Fermi, CERN) have contributed many features to support their workflow
- Many others

Summit (ORNL)
Sierra (LLNL)

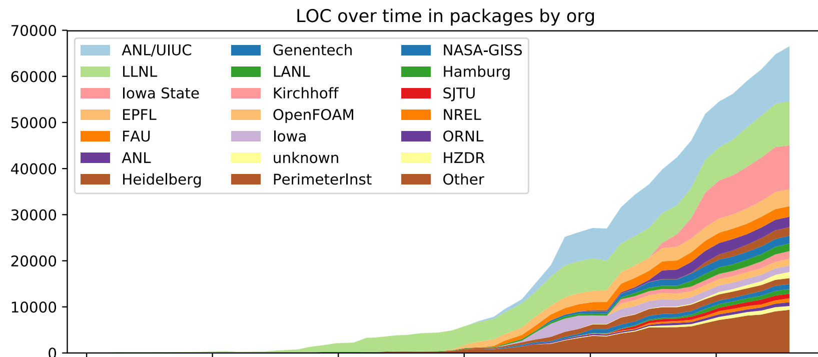
Cori (NERSC)

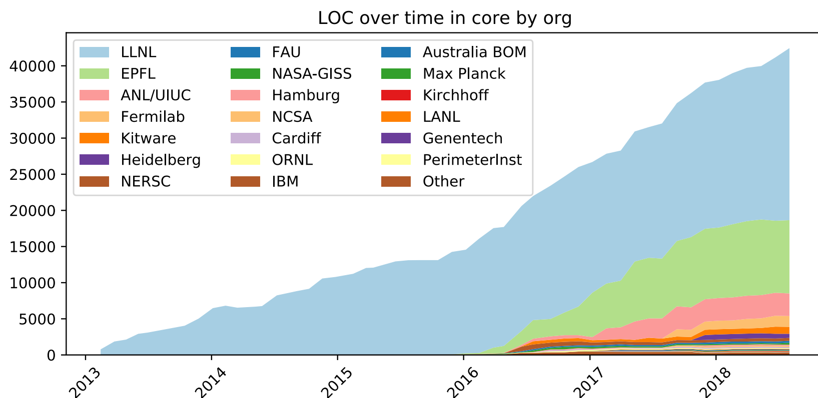
SuperMUC-NG (LRZ)



Fugaku, formerly Post-K (RIKEN)

Contributions to Spack continue to grow!





- In November 2015, LLNL provided most of the contributions to Spack
- Since then, we've gone from 300 to over 3,200 packages
- Most packages are from external contributors!
- Many contributions in core, as well.
- We are committed to sustaining Spack's open source ecosystem!

Spack v0.12.1 is the latest release

- **New features:**
  1. Spack environments (covered today)
  2. `spack.yaml` and `spack.lock` files for tracking dependencies (covered today)
  3. Custom configurations via command line (covered today)
  4. Better support for linking Python packages into view directories (pip in views)
  5. Support for uploading build logs to CDash
  6. Packages have more control over compiler flags via flag handlers
  7. Better support for module file generation
  8. Better support for Intel compilers, Intel MPI, etc.
  9. Many performance improvements, improved startup time

- Spack is now permissively licensed under **Apache-2.0 or MIT**
  — previously LGPL

- Over 2,900 packages (800 added since last year)
  — This is from November; over 3,200 in latest develop branch

Related Work

Spack is not the first tool to automate builds

— Inspired by copious prior work

1. **"Functional" Package Managers**
   — Nix https://nixos.org/
   — GNU Guix https://www.gnu.org/s/guix/

2. **Build-from-source Package Managers**
   — Homebrew http://brew.sh
   — MacPorts https://www.macports.org

Other tools in the HPC Space:

- **Easybuild** http://hpcugent.github.io/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

- **Conda** https://conda.io
  — Very popular binary package manager for data science
  — Not targeted at HPC; generally unoptimized binaries

What's on the road map?

- **Spack Stacks:** better support for facility deployment
  - Extension of environments to better support facility workflows
  - Ability to easily specify full facility software stack and module configuration
  - Build all of it at once with a single command!

- **Better CI Support**
  - GitLab Integration with environments

- **Architecture-specific binaries**
  - Better provenance for builds
  - Better support for matching optimized binary packages to machines

- **Better concretization**
  - Spack's dependency resolution can hit snags
  - We need better support to enable features above

More on the Spack Road Map tomorrow Morning!

Tutorial Overview (times are estimates)

1.	For everyone:	**Welcome & Overview**	9:15 - 9:30
2.	For everyone:	**Basics of Building and Linking**	9:30 - 9:45
3.	For everyone:	**Spack Basics (hands on)**	9:45 - 10:30
		-- Break --	
4.	For everyone:	**Core Spack Concepts**	11:00 - 11:15
5.	For everyone:	**Configuration (hands-on)**	11:15 - 11:45
6.	For packagers:	**Making your own Packages (hands on)**	11:45 - 12:30
		-- Lunch --	
7.	For HPC centers:	**Generating Module Files with Spack (hands-on)**	2:00 - 2:45
8.	For everyone:	**Spack environments, spack.yaml/spack.lock**	2:45 - 3:30
		-- Break --	
9.	For packagers:	**Advanced Packaging: Build systems**	4:00 - 4:20
10.	For packagers:	**Advanced Packaging: Virtual dependencies: BLAS/LAPACK/MPI**	4:20 - 5:00
11.	For everyone:	**Build your own packages with Spack / Q&A (hands-on)**	5:00 - 5:30

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

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

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

```
<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
```

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
```

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 –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
```

- 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        py-pycparser         qt           tcl
adept-utils     cgm              flex         harfbuzz     libgpg-error   metis             openspeedshop     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         hdf5         libjson-c      mpc               otf               py-epydoc          py-pylint            ravel        thrift
asciidoc        cloog            fontconfig   hwloc        libmng         mpe2              otf2              py-funcsigs        py-pypar             readline     tk
atk             cmake            freetype     hypre        libmonitor     mpfr              pango             py-genders         py-pyparsing         rose         tmux
atlas           cmocka           gasnet       icu          libNBC         mpibash           papi              py-gnuplot         py-pyqt              rsync        tmuxinator
atop            coreutils        gcc          icu4c        libpciaccess   mpich             paraver           py-h5py            py-pyside            ruby         trilinos
autoconf        cppcheck         gdb          ImageMagick  libpng         mpileaks          paraview          py-ipython         py-pytables          SAMRAI       uncrustify
automaded       cram             gdk-pixbuf   isl          libsodium      mrnet             parmetis          py-libxml2         py-python-daemon     samtools     util-linux
automake        cscope           geos         jdk          libtiff        mumps             parpack           py-lockfile        py-pytz              scalasca     valgrind
bear            cube             gflags       jemalloc     libtool        munge             patchelf          py-mako            py-rpy2              scorep       vim
bib2xhtml       curl             ghostscript  jpeg         libunwind      muster            pcre              py-matplotlib      py-scientificpython  scotch       vtk
binutils        czmq             git          judy         libuuid        mvapich2          pcre2             py-mock            py-scikit-learn      scr          wget
bison           damselfly        glib         julia        libxcb         nasm              pdt               py-mpi4py          py-scipy             silo         wx
boost           dbus             glm          launchmon    libxml2        ncdu              petsc             py-mx              py-setuptools        snappy       wxpropgrid
bowtie2         docbook-xml      global       lcms         libxshmfence   ncurses           pidx              py-mysqldb1        py-shiboken          sparsehash   xcb-proto
boxlib          doxygen          glog         leveldb      libxslt        netcdf            pixman            py-nose            py-sip               spindle      xerces-c
bzip2           dri2proto        glpk         libarchive   llvm           netgauge          pkg-config        py-numexpr         py-six               spot         xz
cairo           dtcmp            gmp          libcerf      llvm-lld       netlib-blas       pmgr_collective   py-numpy           py-sphinx            sqlite       yasm
callpath        dyninst          gmsh         libcircle    lmdb           netlib-lapack     postgresql        py-pandas          py-sympy             stat         zeromq
cblas           eigen            gnuplot      libdrm       lmod           netlib-scalapack  ppl               py-pbr             py-tappy             sundials     zlib
cbtf            elfutils         gnutls       libdwarf     lua            nettle            protobuf          py-periodictable   py-twisted           swig         zsh
cbtf-argonavis  elpa             gperf        libedit      lwgrp          ninja             py-astropy        py-pexpect         py-urwid             szip
cbtf-krell      expat            gperftools   libelf       lwm2           ompss             py-basemap        py-pil             py-virtualenv        tar
cbtf-lanl       extrae           graphlib     libevent     matio          ompt-openmp       py-biopython      py-pillow          py-yapf              task
cereal          exuberant-ctags  graphviz     libffi       mbedtls        opari2            py-blessings      py-pmw             python               taskd
cfitsio         fftw             gsl          libgcrypt    memaxes        openblas          py-cffi           py-pychecker       qhull                tau
```

- Spack has over 2,900 packages now.

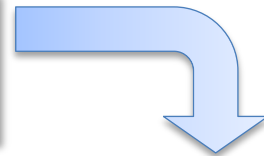
`spack find` shows what is installed

```
$ spack find
==> 103 installed packages.
-- linux-rhel6-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-numpy@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-rhel6-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-rhel6-x86_64 / intel@14.0.2 -----------------------------
hwloc@1.9  mpich@3.0.4  starpu@1.1.4

-- linux-rhel6-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-rhel6-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-rhel6-x86_64 / clang@3.4 --      -- linux-rhel6-x86_64 / gcc@4.9.2 -------
callpath@1.0.2                            callpath@1.0.2
```



```
$ spack find -dl callpath
==> 2 installed packages.
-- linux-rhel6-x86_64 / clang@3.4 -----      -- linux-rhel6-x86_64 / gcc@4.9.2 -----
xv2clz2     callpath@1.0.2                   udltshs     callpath@1.0.2
ckjazss         ^adept-utils@1.0.1           rfsu7fb         ^adept-utils@1.0.1
3ws43m4             ^boost@1.59.0            ybet64y             ^boost@1.55.0
ft7znm6             ^mpich@3.1.4             aa4ar6i             ^mpich@3.1.4
qqnuet3         ^dyninst@8.2.1               tmnnge5         ^dyninst@8.2.1
3ws43m4             ^boost@1.59.0            ybet64y             ^boost@1.55.0
g65rdud             ^libdwarf@20130729       g2mxrl2             ^libdwarf@20130729
cj5p5fk                 ^libelf@0.8.13       ynpai3j                 ^libelf@0.8.13
cj5p5fk             ^libelf@0.8.13           ynpai3j             ^libelf@0.8.13
g65rdud         ^libdwarf@20130729           g2mxrl2         ^libdwarf@20130729
cj5p5fk             ^libelf@0.8.13           ynpai3j             ^libelf@0.8.13
cj5p5fk         ^libelf@0.8.13               ynpai3j         ^libelf@0.8.13
ft7znm6         ^mpich@3.1.4                 aa4ar6i         ^mpich@3.1.4
```

Expand dependencies with `spack find -d`

- 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

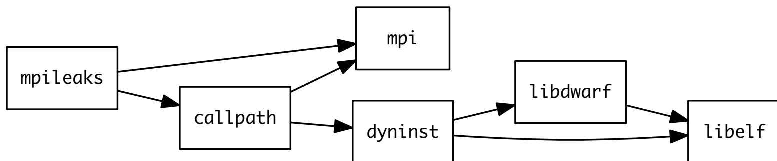
- Traditional binary package managers
  - RPM, yum, APT, yast, etc.
  - Designed to manage a single stack.
  - Install *one* version of each package in a single prefix (`/usr`).
  - Seamless upgrades to a *stable, well tested* stack

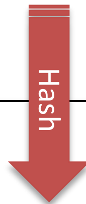
- Port systems
  - BSD Ports, portage, Macports, Homebrew, Gentoo, etc.
  - Minimal support for builds parameterized by compilers, dependency versions.

- 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.

Dependency DAG



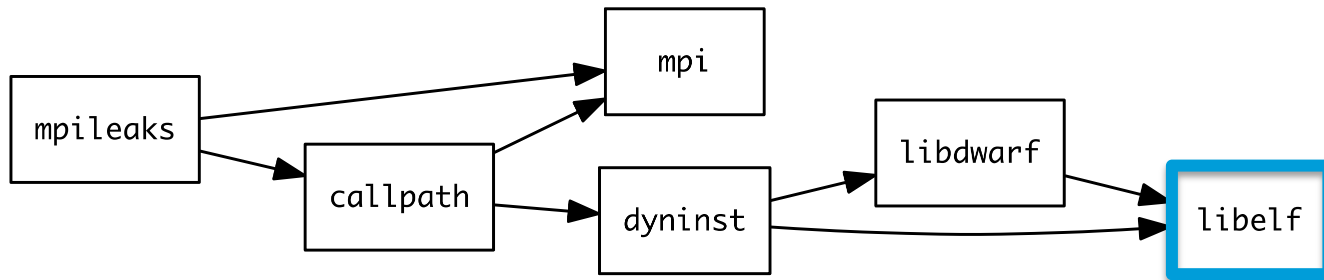


Installation Layout

```
spack/opt/
  linux-x86_64/
    gcc-4.7.2/
      mpileaks-1.1-0f54bf34cadk/
    intel-14.1/
      hdf5-1.8.15-lkf14aq3nqiz/
  bgq/
    xl-12.1/
      hdf5-1-8.16-fqb3a15abrwx/
  ...
```

- 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

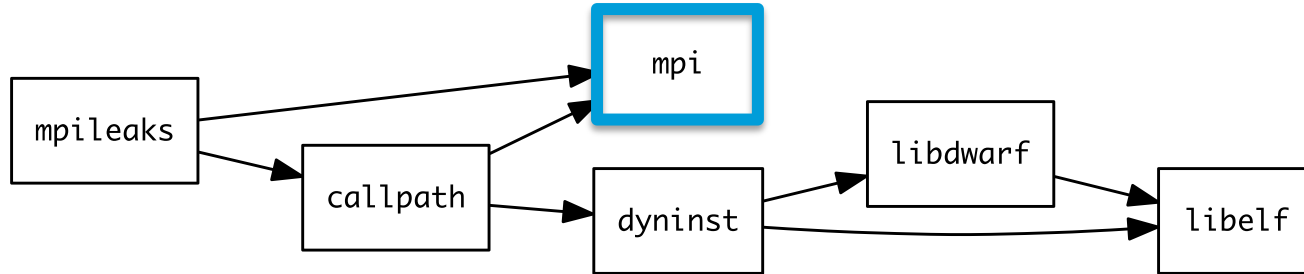



```
$ spack install mpileaks %intel@12.1 ^libelf@0.8.12
```

- 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:

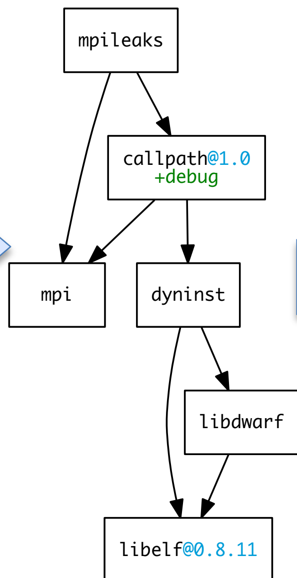
```
$ spack install mpileaks ^mpi@2
```

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

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

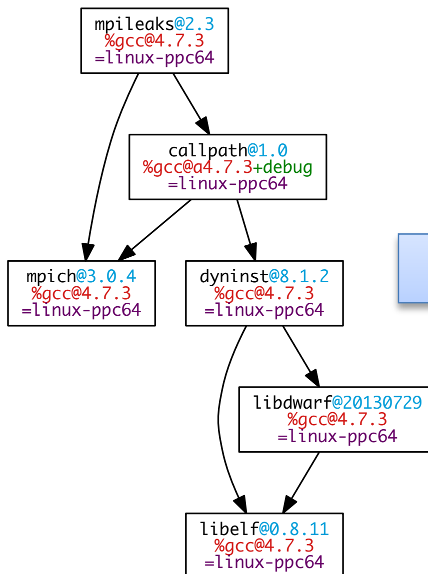
User input: *abstract* spec with some constraints

Normalize



Abstract, normalized spec with some dependencies.

Concretize



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

Store

spec.yaml


```yaml
spec:
- mpileaks:
    arch: linux-x86_64
    compiler:
      name: gcc
      version: 4.9.2
    dependencies:
      adept-utils: kszrtkpbzac3ss2ixcjkcorlaybnptp4
      callpath: bah5f4h4d2n47mgycej2mtrnrivvxy77
      mpich: aa4ar6ifj23yijqmdabeakpejcli72t3
    hash: 33hjjhxi7p6gyzn5ptgyes7sghyprujh
    variants: {}
    version: '1.0'
- adept-utils:
    arch: linux-x86_64
    compiler:
      name: gcc
      version: 4.9.2
    dependencies:
      boost: teesjv7ehpe5ksspjim5dk43a7qnowlq
      mpich: aa4ar6ifj23yijqmdabeakpejcli72t3
    hash: kszrtkpbzac3ss2ixcjkcorlaybnptp4
    variants: {}
    version: 1.0.1
- boost:
    arch: linux-x86_64
    compiler:
      name: gcc
      version: 4.9.2
    dependencies: {}
    hash: teesjv7ehpe5ksspjim5dk43a7qnowlq
    variants: {}
    version: 1.59.0
...
```


Detailed provenance is stored with the installed package

Use `spack spec` to see the results of concretization

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

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
              ^bzip2@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~psm~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@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 each package in its 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))
```

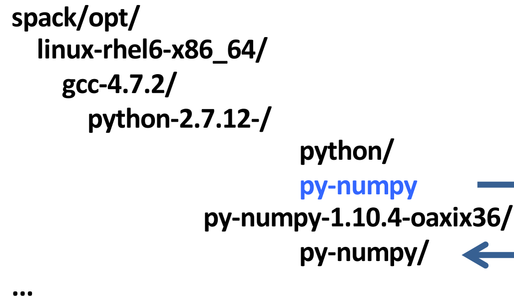
Spack extensions

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

```
spack/opt/
  linux-rhel6-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



Extensions must be activated into extendee

```
$ spack extensions python
==> python@2.7.12% gcc@4.9.3 ~tk~ucs4 arch=linux-rhel7-x86_64-i25k4oi
==> 118 extensions:
…
py-autopep8     py-docutils        py-mako           py-numpy       py-py2neo
…

==> 3 installed:
-- linux-rhel7-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-rhel7-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-rhel7-x86_64-i25k4oi
…
==> 1 currently activated:
-- linux-rhel7-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
```

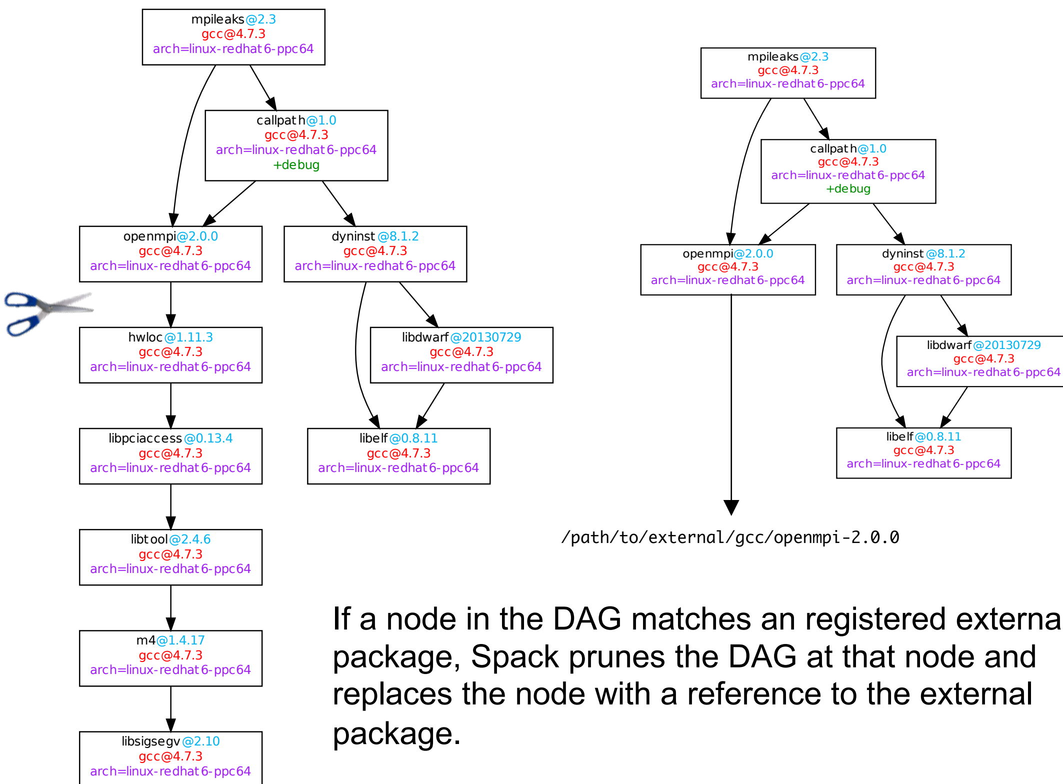
Building against externally installed software

```
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-rhel6-ppc64:
        /path/to/external/gcc/openmpi-2.0.0
      openmpi@1.10.3 %gcc@4.7.3 arch=linux-rhel6-ppc64:
        /path/to/external/gcc/openmpi-1.10.3
      openmpi@2.0.0 %intel@16.0.0 arch=linux-rhel6-ppc64:
        /path/to/external/intel/openmpi-2.0.0
      openmpi@1.10.3 %intel@16.0.0 arch=linux-rhel6-ppc64:
        /path/to/external/intel/openmpi-1.10.3
...
```

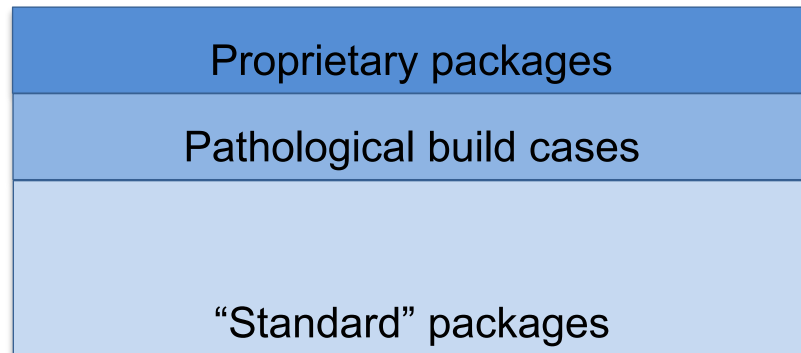
A user registers external packages with Spack.



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

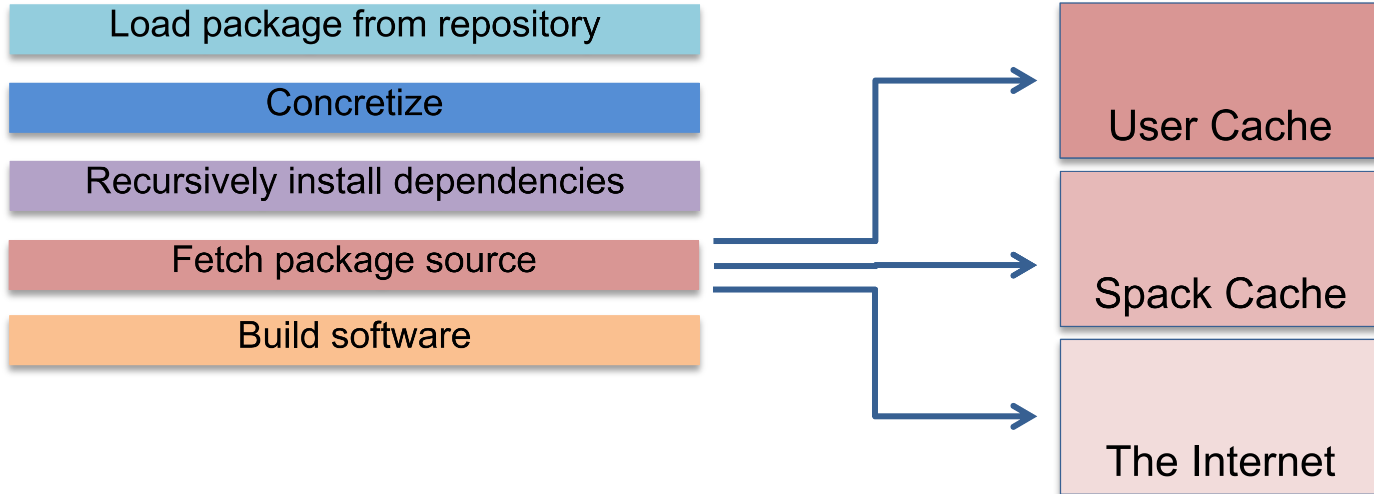
- Some packages can not be released publicly
- Some users have use cases that require ~~bizarre~~ custom builds
- Packaging issues should not prevent users from updating Spack
  - Solution: separate repositories
  - A repository is simply a directory of package files
- Spack supports external repositories that can be layered on top of the built-in packages
- Custom packages can depend on built-in packages (or packages in other repositories)



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

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**

$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')
```

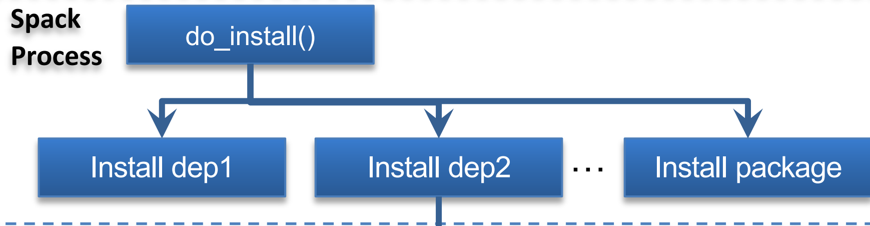
Spack packages are *templates* for builds

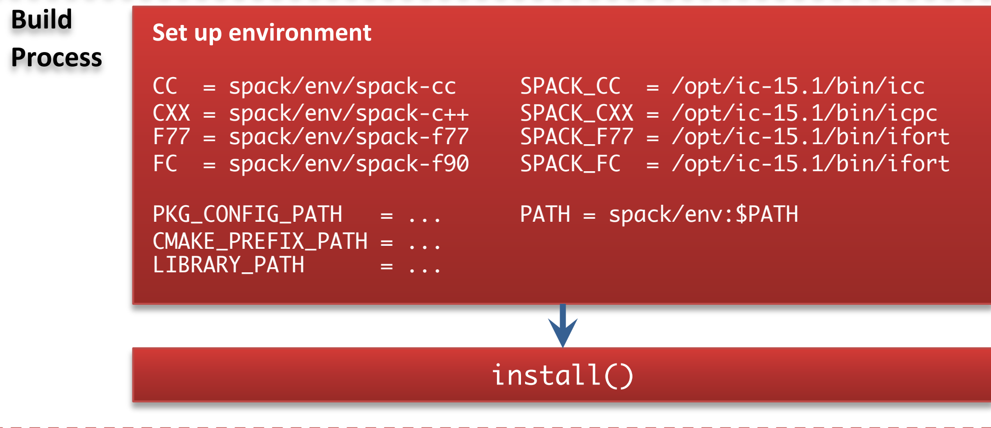
- Each package has one class
  - zlib for Intel compiler and zlib for GCC compiler are built with the same recipe.
- Can add conditional logic using spec syntax
  - Think of package as *translating* a concrete DAG to build instructions.
  - Dependencies are already built
  - No searching or testing; 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

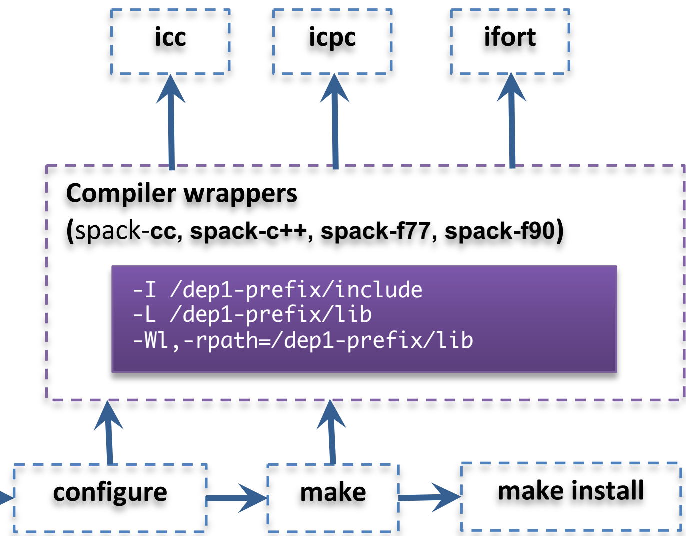
```
def install(self, spec, prefix):
    config_opts=['--prefix='+prefix]

    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





- **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)



Writing Packages - Versions and URLs

$REPO/packages/mvapich2/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')
```

- Package downloads are hashed with SHA-256 by default
  - Also supports SHA-512
  - Many packages still have legacy md5 checksums
- 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

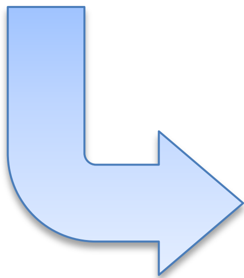
- Functions wrap common ops
  - cmake, configure, patch, make, …
  - **Executable** and **which** for new wrappers.
- Commands executed in clean environment
- Full Python functionality
  - Patch up source code
  - Make files and directories
  - Calculate flags
  - …

$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

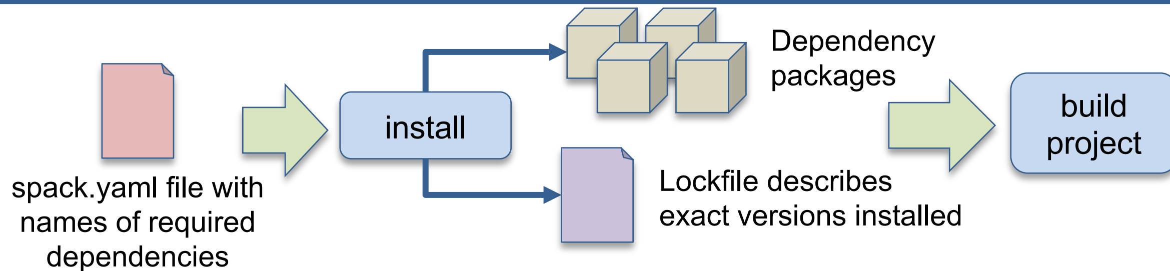
```
class Zlib(Package):
    # FIXME: Add a proper url for your package's homepage here.
    homepage = "http://www.example.com"
    url      = "http://zlib.net/zlib-1.2.8.tar.gz"
    version('1.2.8', '44d667c142d7cda120332623eab69f40')

    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.
  - 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
Under “Tutorial: Spack 101"

Spack has added environments and spack.yaml / spack.lock



- Allows developers to bundle Spack configuration with their repository
- Can also be used to maintain configuration together with Spack packages.
  - E.g., versioning your own local software stack with consistent compilers/MPI implementations
- Manifest / Lockfile model pioneered by Bundler is becoming standard
  - spack.yaml describes project requirements
  - spack.lock describes exactly what versions/configurations were installed, allows them to be reproduced.

Simple spack.yaml file

```yaml
spack:
  # include external configuration
  include:
  - ../special-config-directory/
  - ./config-file.yaml

  # add package specs to the `specs` list
  specs:
  - hdf5
  - libelf
  - openmpi
```

Concrete spack.lock file (generated)


```json
{
 "concrete_specs": {
  "6s63so2kstp3zyvjezglndmavy613nul": {
   "hdf5": {
    "version": "1.10.5",
    "arch": {
     "platform": "darwin",
     "platform_os": "mojave",
     "target": "x86_64"
    },
    "compiler": {
     "name": "clang",
     "version": "10.0.0-apple"
    },
    "namespace": "builtin",
    "parameters": {
     "cxx": false,
     "debug": false,
     "fortran": false,
     "hl": false,
     "mpi": true,
     "pic": true,
     "shared": true,
     "szip": false,
     "threadsafe": false,
     "cflags": [],
```

Spack environments help with building containers

- We recently started providing base images on DockerHub with Spack preinstalled.
- **Very** easy to build a container with some Spack packages in it:

```
spack-docker-demo/
    Dockerfile
    spack.yaml
```

```
FROM spack/centos:7

WORKDIR /build
COPY spack.yaml .
RUN spack install
```

Base image with Spack in PATH

Copy in spack.yaml
Then run `spack install`

```
spack:
  specs:
    - hdf5 @1.8.16
    - openmpi fabrics=libfabric
    - nalu
```

List of packages to install, with constraints

Build with **`docker build .`**

Run with Singularity
(or some other tool)

Hands-on Time: Environments

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"

Hands-on Time: Build Systems

Follow script at http://spack.rtfd.io
Under “Tutorial: Spack 101"

Build Systems

- Repeated patterns in packages
- Build System Subclasses
  - Provide convenience methods for ease of writing package files
  - Better control over the install phases
  - Sensible defaults
- Build Systems

• Autotools	• QMake	• Intel	• Makefile
• Cmake	• Waf	• Perl	
• Python	• Scons	• Aspell Dict	
• R	• Octave	• Cuda	

Optimize for the Common Case

CP2K

```
def install(self, spec, prefix):
      # Construct a proper filename for the architecture file
      cp2k_arch= '{0}-{1}'.format(spec.arch, spec.compiler)
      cp2k_ver = 'sopt' if '~mpi' in spec else 'popt'
      makefile_basename = '.'.join([cp2k_arch, cp2k_ver])
      makefile = join_path('arch', makefile_basename)

      # Write the custom makefile
      with open(makefile, 'w') as mkf:

          # Special Optimization flags
          optflags = {
               'gcc': [
                  '-O2',
                  '-mtune=native',
                  '-funroll-loops',
                  '-ffast-math',
                  '-ftree-vectorize',
                      ], 'intel': [
                        '-O2',
                         '-pc64',

      ...
```

mpileaks

```
def install(self, spec, prefix):
    configure("--prefix=" + prefix,
              "--with-adept-utils=" + spec['adeptutils'].prefix,
              "--with-callpath=" + spec['callpath'].prefix)
    make()
    make("install")
```

cups

```
def install(self, spec, prefix):
    configure("—prefix=" + prefix,
              "—enable-gnutls",
              "—with-components=core")
    make()
    make("install")
```

Common Build Patterns

gmp (Autotools)

```
./configure
make
make install
```

openJPEG (CMake)

```
cmake ..
make
make install
```

BSGenome (R)

```
R CMD INSTALL –l lib pkgs
```

pandas (setuptools)

```
python setup.py build
python setup.py install
```

Spack Build Systems Implement Common Patterns

```
class AutotoolsPackage(Package):

    def configure(self, spec, prefix):
        options = get_configure_options()
        with working_dir(self.build_directory):
            inspect.getmodule(self).configure(*options)

    def build(self, spec, prefix):
        with working_dir(self.build_directory):
            inspect.getmodule(self).make(*self.build_targets)

    def install(self, spec, prefix):
        with working_dir(self.build_directory):
            inspect.getmodule(self).make(*self.install_targets)
```

```
./configure [options]

make

make install
```

Spack Build Systems Implement Common Patterns

```
class CmakePackage(Package):

    def cmake(self, spec, prefix):
        options = get_cmake_args()
        with working_dir(self.build_directory):
            inspect.getmodule(self).cmake(*options)

    def build(self, spec, prefix):
        with working_dir(self.build_directory):
            inspect.getmodule(self).make(*self.build_targets)

    def install(self, spec, prefix):
        with working_dir(self.build_directory, create=True):
            inspect.getmodule(self).make(*self.install_targets)
```

```
cmake [options]

make

make install
```

Spack Build Systems Implement Common Patterns

```
class PythonPackage(Package):

    def build(self, spec, prefix):
        args = get_build_args()
        self.setup_py('build', *args)


    def install(self, spec, prefix):
       args = get_install_args()
       self.setup_py('install', *args)
```

```
python setup.py build
```

```
python setup.py install
```

Packagers Provide Minimal Details

- Spack provides sensible defaults to the build system and takes care of common patterns.
- Packagers only need to provide arguments (if any).

cups/package.py

```
class Cups(AutotoolsPackage):

    def configure_args(self):
        return ['--enable-gnutls', '--with-components=core']
```

hpx/package.py

```
class Hpx(CMakePackage):

def cmake_args(self):
    return ['-DHPX_BUILD_EXAMPLES=OFF', '-DHPX_MALLOC=system']
```

Example: AutotoolsPackage

```
from spack import *

class Gmp(Package):
    """GMP is a free library for arbitrary precision arithmetic"""

    homepage = "https://gmplib.org"
    url      = "https://gmplib.org/download/gmp/gmp-6.0.0a.tar.bz"

    version("6.1.1", '4c175f86e11eb32d8bf9872ca3a8e11d' )
    version("6.1.0", ''86ee6e54ebfc4a90b643a65e402c4048')
    version("6.0.0a", 'b7ff2d88cae7f8085bd5006096eed470')

    # dependencies ...

    def install(self, spec, prefix):
        configure("--prefix={0}".format(prefix),
                  "—enable-cxx")
        make()
        make("install")
```

```
from spack import *

class Gmp(AutotoolsPackage):
    """GMP is a free library for arbitrary precision arithmetic"""

    homepage = "https://gmplib.org"
    url      = "https://ftp.gnu.org/gnu/gmp/gmp-6.1.2.tar.bz2"

    version('6.1.1',  '4c175f86e11eb32d8bf9872ca3a8e11d')
    version('6.1.0',  '86ee6e54ebfc4a90b643a65e402c4048')
    version('6.0.0a', 'b7ff2d88cae7f8085bd5006096eed470')

    # dependencies ...

    def configure_args(self):
        return ['--enable-cxx']
```

Example: CMakePackage

```
from spack import *

class Openjpeg(Package):
    """OpenJPEG is an open-source JPEG 2000 codec."""

    homepage = "https://github.com/uclouvain/openjpeg"
    url = "https://github.com/uclouvain/version.2.1.tar.gz"

    version("2.1.0", '3e1c451c087f8462955426da38aa3b3d')
    version("2.0.1", '105876ed43ff7dbb2f90b41b5a43cfa5')
    version("2.0.0", 'cdf266530fee8af87454f15feb619609')

    def install(self, spec, prefix):
        cmake(".", "-DCMAKE_BUILD_TYPE=RelWithDebInfo")
        make()
        make("install")
```

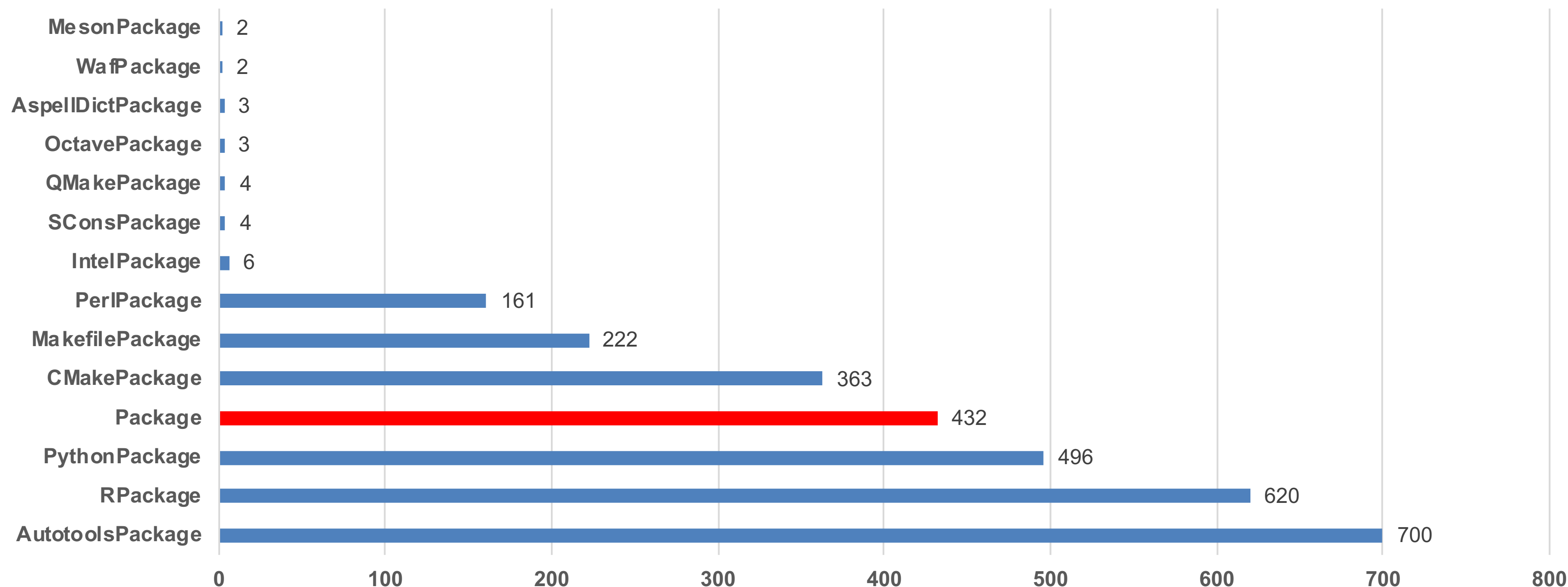
```
from spack import *

class Openjpeg(CMakePackage):
    """OpenJPEG is an open-source JPEG 2000 codec."""

    homepage = "https://github.com/uclouvain/openjpeg"
    url = "https://github.com/uclouvain/version.2.1.tar.gz

    version('2.1.0', '3e1c451c087f8462955426da38aa3b3d')
    version('2.0.1', '105876ed43ff7dbb2f90b41b5a43cfa5')
    version('2.0.0', ''cdf266530fee8af87454f15feb619609')

    def cmake_args(self):
        return ["-DCMAKE_BUILD_TYPE=RelWithDebInfo"]
```

Example: PythonPackage

```
from spack import *

class PyPandas(Package):
    """pandas is a Python package providing fast, flexible, and
      expressive data structures."""

    homepage = "http://pandas.pydata.org"
    url = "https://pypi.python.org/pandas-0.16.0.tar.gz"

    version('0.16.0', 'bfe311f05dc0c351f8955fbd1e296e73')
    version('0.16.1', 'fac4f25748f9610a3e00e765474bdea8')

    depends_on("py-dateutil", type=("build", "run")
    depends_on("py-numpy", type=("build", "run")
    depends_on("py-setuptools", type=("build", "run"))
    depends_on("py-cython", type=("build", "run"))
    depends_on("py-pytz", type=("build", "run"))
    depends_on("py-numexpr", type=("build", "run"))
    depends_on("py-bottleneck", type=("build", "run"))

    def install(self, spec, prefix):
        python("setup.py", "install", "--prefix=%s" % prefix)
```

```
from spack import *

class PyPandas(PythonPackage):
  """pandas is a Python package providing fast, flexible, and
    expressive data structures."""

    homepage = "http://pandas.pydata.org"
    url = "https://pypi.python.org/pandas-0.16.0.tar.gz"

    version('0.16.1', 'fac4f25748f9610a3e00e765474bdea8')
    version('0.16.0', 'bfe311f05dc0c351f8955fbd1e296e73')

    depends_on('py-dateutil', type=('build', 'run'))
    depends_on('py-numpy', type=('build', 'run'))
    depends_on('py-setuptools', type='build')
    depends_on('py-cython', type='build')
    depends_on('py-pytz', type=('build', 'run'))
    depends_on('py-numexpr', type=('build', 'run'))
    depends_on('py-bottleneck', type=('build', 'run'))
```

Total Packages: 3,018



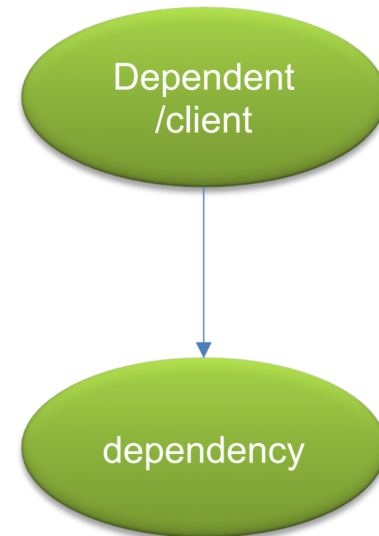
Simplified Package Writing

- Only need to provide arguments to configure, cmake, etc
- Reduce boilerplate code
- Can easily extend classes and add more behavior
- Can control install phases with finer granularity
  - spack configure
  - spack build
  - spack install

Advanced Packaging

Advanced Topics in Packaging

- Spack tries to automatically configure packages with information from dependencies
  - But there are many special cases. Often you need to retrieve details about dependencies to configure properly
- The goal is to answer the following questions that come up when writing package files:
  - How do I retrieve dependency libraries/headers when configuring my package?
  - How does spack help me configure my build-time environment?
- We'll start with a client view and then look at how we add functionality to packages to make it easier for dependents



Accessing Dependency Libraries

- Although Spack performs some work to help a build find libraries, you may need to explicitly specify dependency libraries during configuration
- Specs provide a ".libs" property which retrieves the individual library files provided by the package
- Accessing ".libs" for a virtual package will retrieve the libraries provided by the chosen implementation

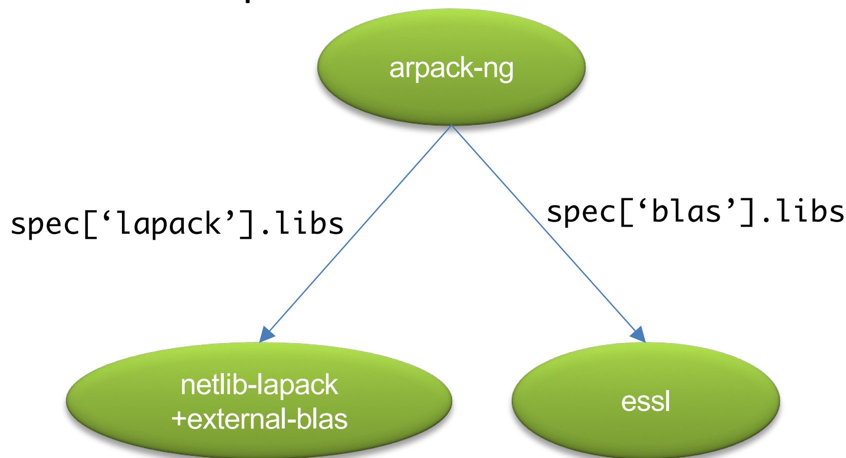
```
class ArpackNg(Package):
    depends_on('blas')
    depends_on('lapack')

    def install(self, spec, prefix):
        lapack_libs = spec['lapack'].libs.joined(';')
        blas_libs = spec['blas'].libs.joined(';')

        cmake(*[
            '-DLAPACK_LIBRARIES={0}'.format(lapack_libs),
            '-DBLAS_LIBRARIES={0}'.format(blas_libs)
        ], '.')
```

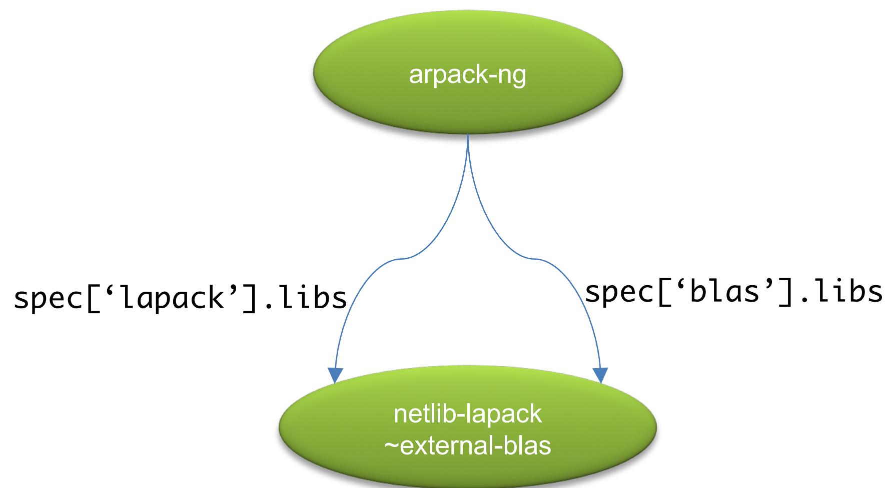
.libs.joined() expresses the list of libraries as a single string like "/…/lib1.so;/…/lib2.so" (e.g. for cmake)

.libs.search_flags expresses the libraries as linker arguments like "-L/…/libdir1/ -L/…/libdir2/" (e.g. as an argument to the compiler)

Accessing Dependency Libraries: Virtuals

- The client side code for accessing ".libs" is the same regardless of which implementation of blas is used
- As a client, you don't have to care whether 'blas' and 'lapack' are provided by the same implementation





Accessing Dependency Libraries: Subsets

- HDF5 builds many libraries, what if you just want the libraries for the high-level interface?
- You can qualify spec queries with additional parameters to specify a subset of libraries from a package

```
class Netcdf(AutotoolsPackage):
    depends_on('hdf5@1.8.9:+hl')

    def configure_args(self):
        LDFLAGS = []

        # Starting version 4.1.3, --with-hdf5= and other such configure options
        # are removed. Variables CPPFLAGS, LDFLAGS, and LD_LIBRARY_PATH must be
        # used instead.
        hdf5_hl = self.spec['hdf5:hl']
        LDFLAGS.append(hdf5_hl.libs.search_flags)
```

Since the hdf5 query was qualified with “hl”, only the libraries for hdf5’s high level interface will be retrieved

Accessing Dependency Headers

- Just like Spack tries to help build systems find libraries, it also tries to automate finding headers

- When that doesn't work and you need to explicitly configure dependency headers, the ".headers" property provides them

```
class Netcdf(AutotoolsPackage):
    depends_on('hdf5@1.8.9:+hl')

    def configure_args(self):
        LDFLAGS = []
        CPPFLAGS = []

        # Starting version 4.1.3, --with-hdf5= and other such configure options
        # are removed. Variables CPPFLAGS, LDFLAGS, and LD_LIBRARY_PATH must be
        # used instead.
        hdf5_hl = self.spec['hdf5:hl']
        CPPFLAGS.append(hdf5_hl.headers.cpp_flags)
        LDFLAGS.append(hdf5_hl.libs.search_flags)
```

headers.cpp_flags gives
`'-I/dir1 -I/dir2 -DMACRO_DEF_EXAMPLE'`

headers.include_flags gives
`'-I/dir1 -I/dir2'` (e.g. for CFLAGS)

Accessing Dependency Command

```python
class Openbabel(CMakePackage):
    variant('python', default=True, description='Build Python bindings')
    extends('python', when='+python')
    depends_on('python', type=('build', 'run'), when='+python')

    def cmake_args(self):
        spec = self.spec
        args = []

        if '+python' in spec:
            args.extend([
                '-DPYTHON_BINDINGS=ON',
                '-DPYTHON_EXECUTABLE={0}'.format(spec['python'].command.path),
            ])
        else:
            args.append('-DPYTHON_BINDINGS=OFF')

        return args
```

- Some packages have a single well-known binary to run
- The ".command" spec property can retrieve it

Convenience Methods/Attributes from Dependencies

- Dependencies may provide shortcuts for invoking binaries

- For example: the Python package provides a globally-defined “python” function to run the python exe:

```
class PyScipy(PythonPackage):
    def install_test(self):
        python('-c', 'import scipy; scipy.test("full", verbose=2)')
```

- For example: all mpi implementations set the .mpicc attribute on their associated spec

```
class Elemental(CMakePackage):
    depends_on('mpi')

    def cmake_args(self):
        spec = self.spec
        args = [
            '-DCMAKE_C_COMPILER=%s' % spec['mpi'].mpicc
        ]
```

Environment Setup from Dependencies

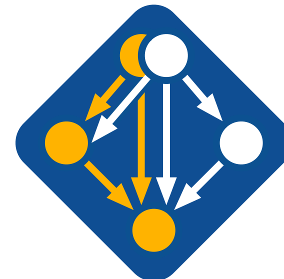
- If your package uses a python extension to build (e.g. setuptools), the Python package makes it available at build time with PYTHONPATH
- All MPI implementation packages make MPICC available in the environment at build for their dependents

Hands-on Time: Advanced Packaging

Follow script at http://spack.rtfd.io
Under "Tutorial: Spack 101"

Join the Spack community!

- There are lots of ways to get involved!
  - Contribute packages, documentation, or features at **github.com/spack/spack**
  - Contribute your configurations to **github.com/spack/spack-configs**



- Talk to us!
  - Join our **Google Group** (see GitHub repo for info)
  - Join our **Slack channel** (see GitHub repo for info)
  - Submit GitHub issues and talk to us!

Star us on GitHub!
github.com/spack/spack

Follow us on Twitter!
@spackpm

We hope to make distributing & using HPC software easy!

Lawrence Livermore
National Laboratory