Using Spack to Accelerate Developer Workflows

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

ECP Annual Meeting Full-day Tutorial
April 15, 2021

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

Tutorial Materials

Find these slides and associated scripts here:

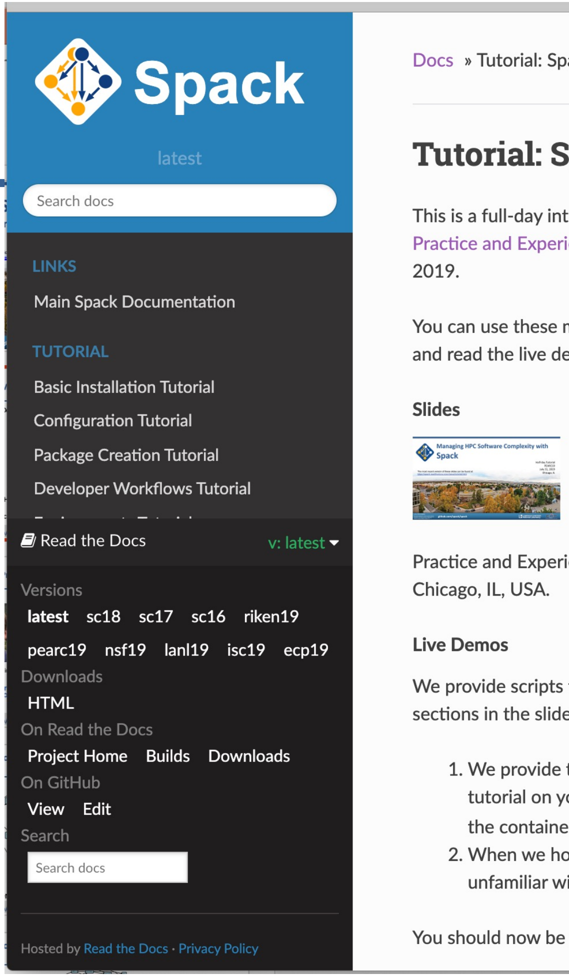
spack-tutorial.readthedocs.io

We will also have a chat room on Spack slack. Get an invite here:

spackpm.herokuapp.com

Join the "tutorial" channel!

We will give you login credentials for the hands-on exercises on Slack.



Tutorial Presenters

Todd Gamblin

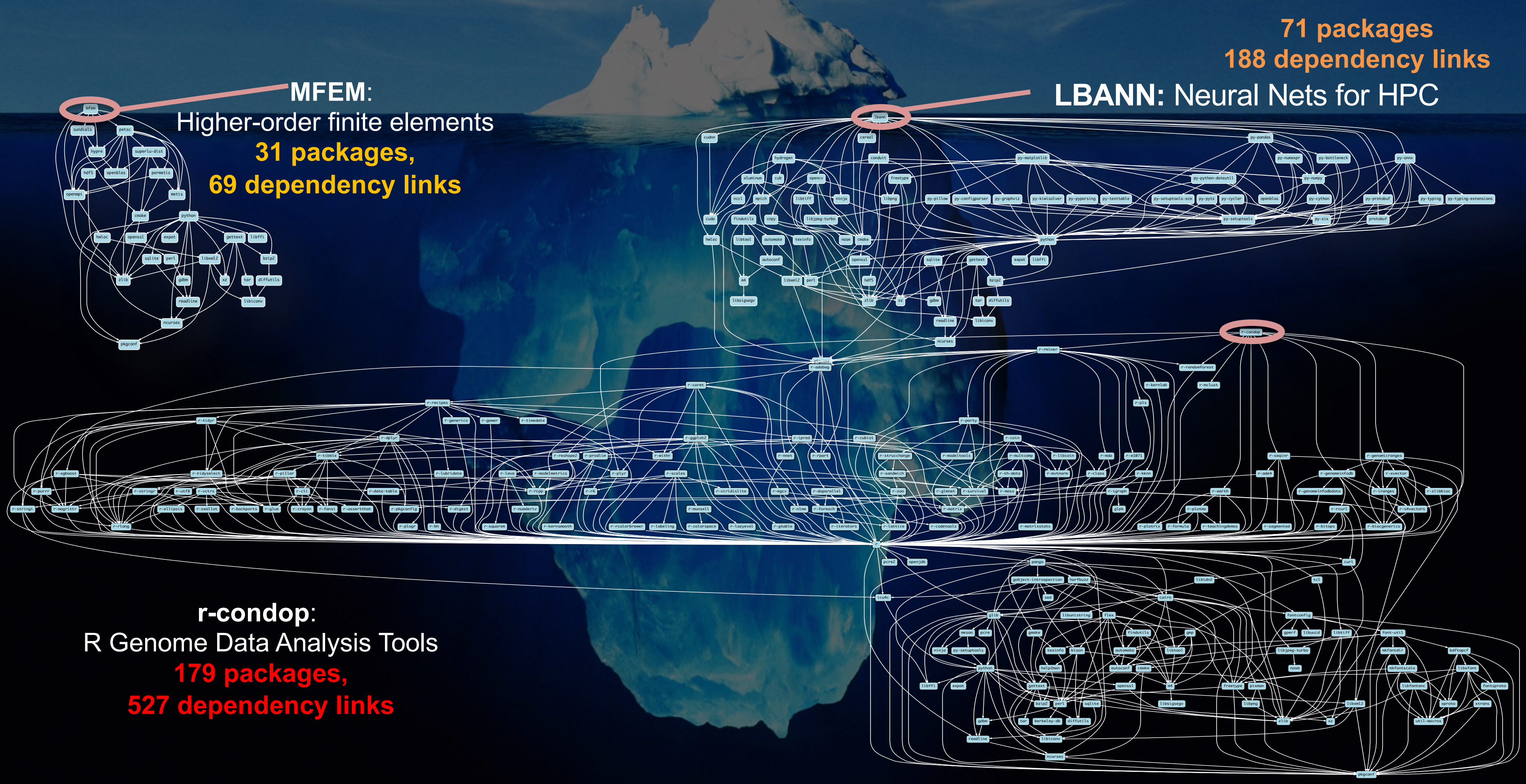
Greg Becker

Peter Scheibel

Tammy Dahlgren

Robert Blake

Modern scientific codes rely on icebergs of dependency libraries



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.

The complexity of the exascale ecosystem threatens productivity.

15+ applications x **80+ software packages** x **5+ target architectures/platforms** Xeon Power KNL NVIDIA ARM Laptops?

x **Up to 7 compilers** Intel GCC Clang XL PGI Cray NAG x **10+ Programming Models** OpenMPI MPICH MVAPICH OpenMP CUDA OpenACC Dharma Legion RAJA Kokkos x **2-3 versions of each package + external dependencies**

= 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 about containers?

- **Containers provide a great way to reproduce and distribute an already-built software stack**

- **Someone needs to build the container!**
  - This isn’t trivial
  - Containerized applications still have hundreds of dependencies

- **Using the OS package manager inside a container is insufficient**
  - Most binaries are built unoptimized
  - Generic binaries, not optimized for specific architectures

- **HPC containers may need to be *rebuilt* to support many different hosts, anyway.**
  - Not clear that we can ever build one container for all facilities
  - Containers likely won’t solve the N-platforms problem in HPC

We need something more flexible to **build** the containers

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

github.com/spack/spack

@spackpm

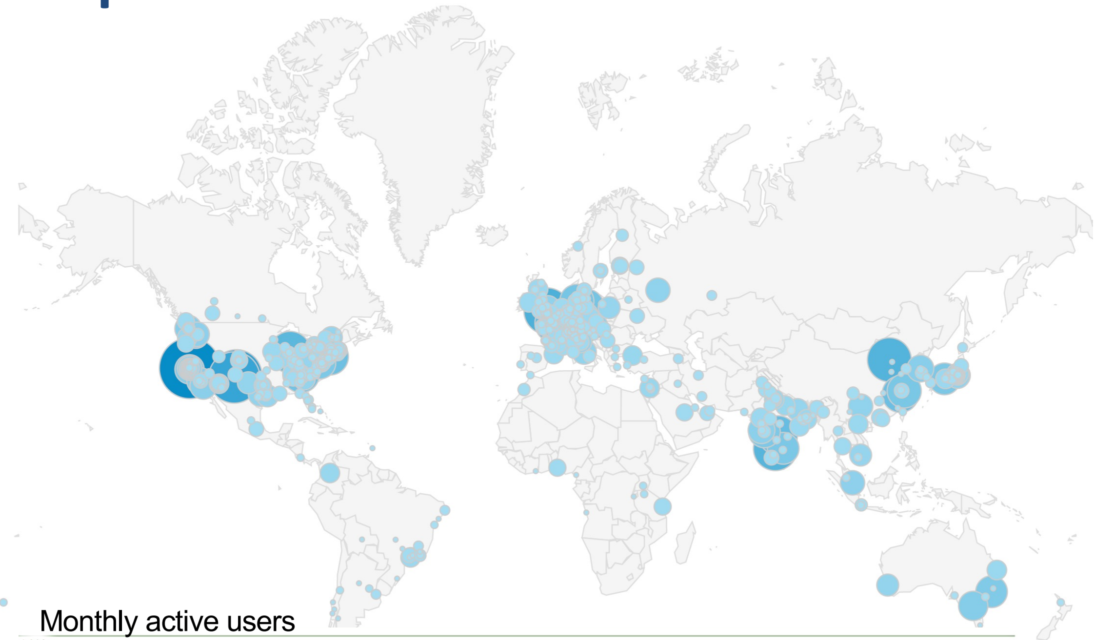
Who can use Spack?

People who want to use or distribute software for HPC!

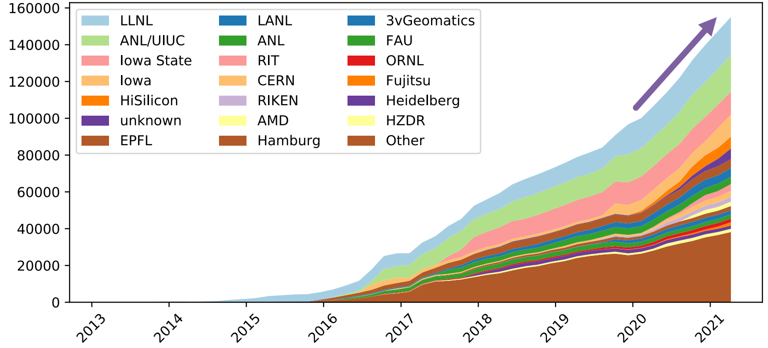
1. **End Users of HPC Software**
   - Install and run HPC applications and tools
2. **HPC Application Teams**
   - Manage third-party dependency libraries
3. **Package Developers**
   - People who want to package their own software for distribution
4. **User support teams at HPC Centers**
   - People who deploy software for users at large HPC sites

Spack is used worldwide!

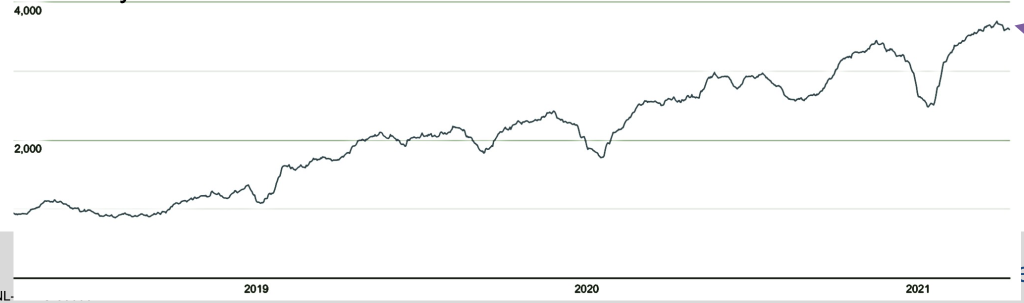
5,400+ software packages
780+ contributors



Package contribution rate increased in 2020

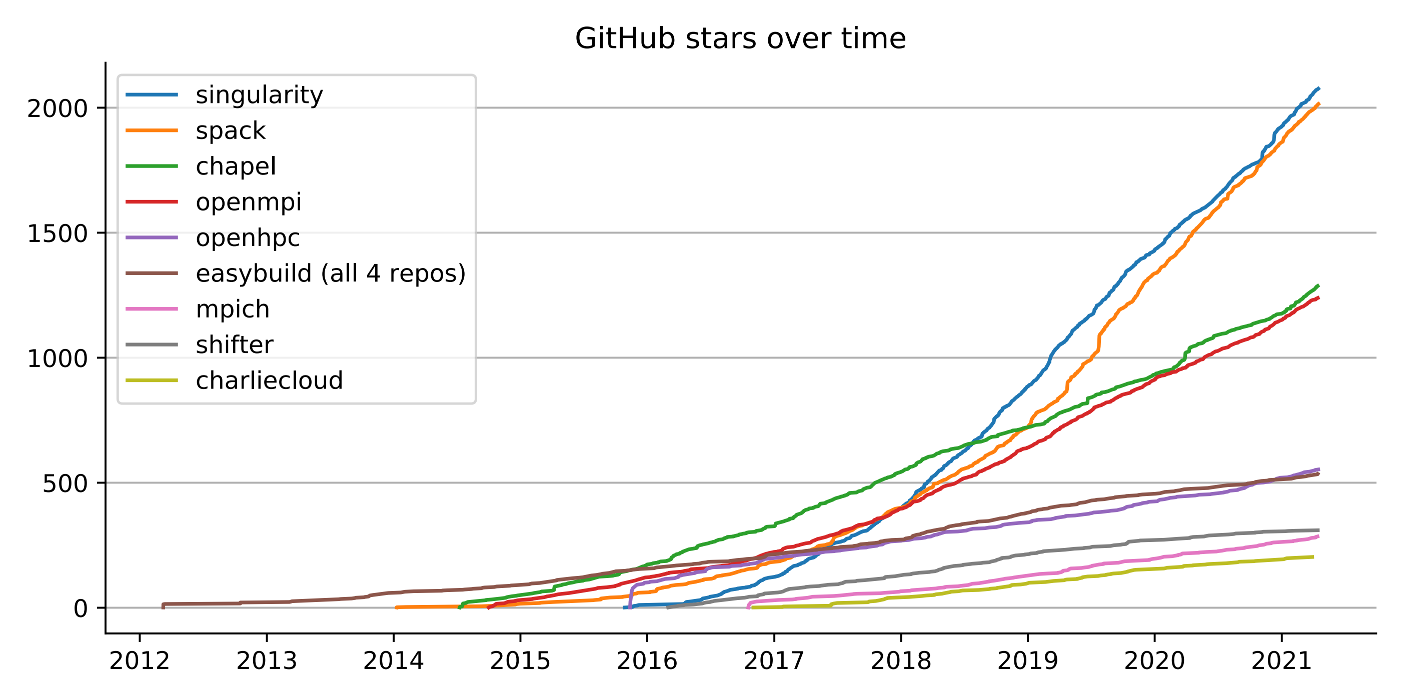


Monthly active users



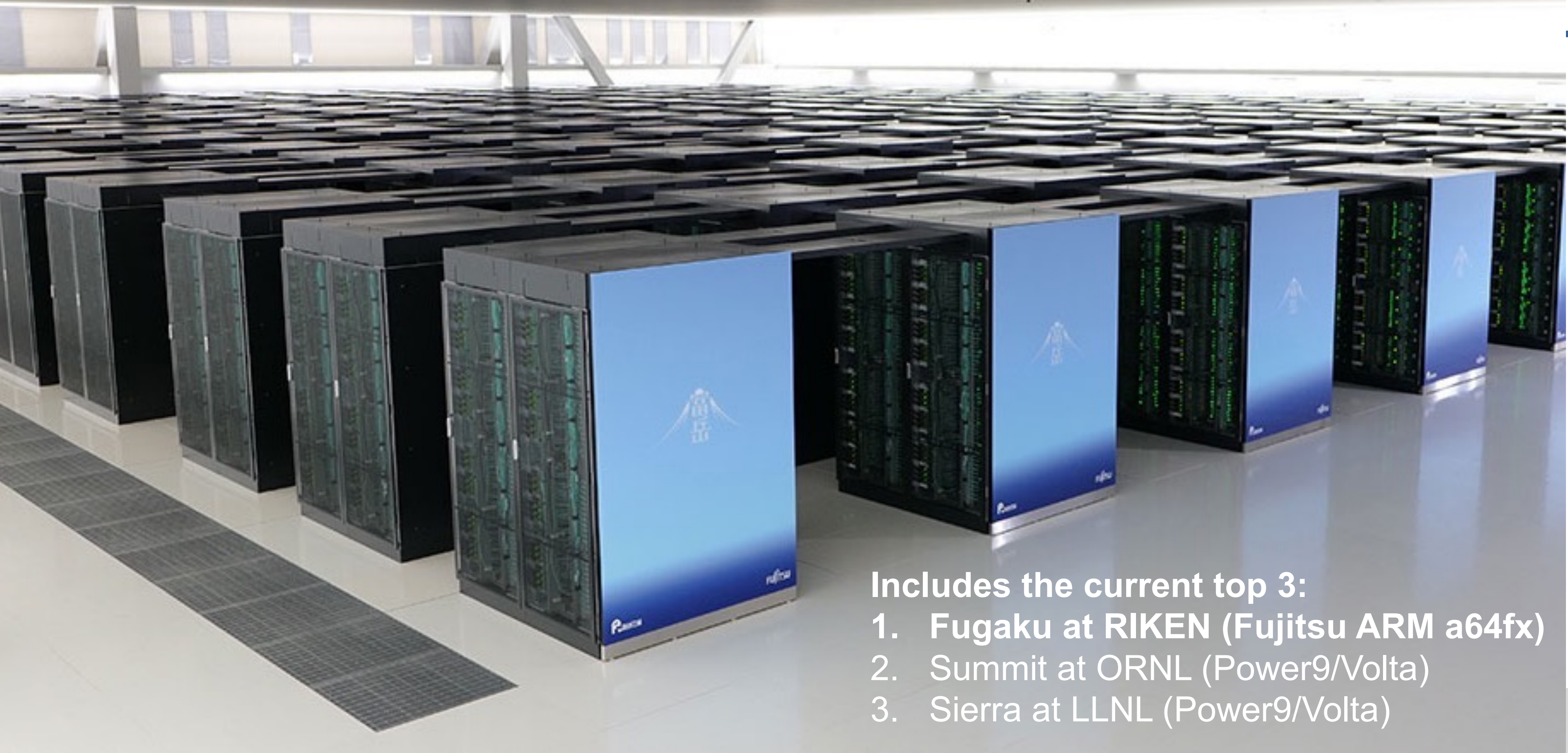
All time high of 3,700 monthly active users this March

Spack has been gaining adoption rapidly (if stars are an indicator)



★ **Star Spack at github.com/spack/spack if you like the tutorial!**

Spack is used on the fastest supercomputers in the world



Includes the current top 3:

1. **Fugaku at RIKEN (Fujitsu ARM a64fx)**
2. Summit at ORNL (Power9/Volta)
3. Sierra at LLNL (Power9/Volta)

Spack is the deployment tool for the U.S. Exascale Computing Project

EXASCALE COMPUTING PROJECT

- Spack will be used to build software for the US's three upcoming exascale systems
- ECP has built the Extreme Scale Scientific Software Stack (E4S) with Spack – more at https://e4s.io
- We are helping ECP fulfill its mission – to create a robust and capable exascale software ecosystem

https://e4s.io

One month of Spack development is pretty busy!

March 16, 2021 – April 16, 2021

Period: 1 month

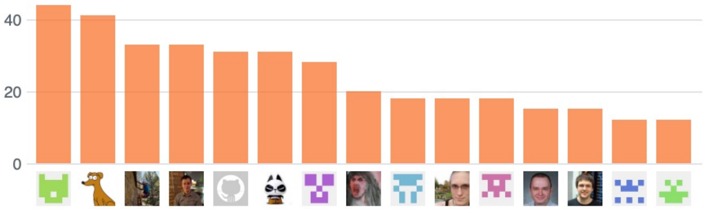
Overview

623 Active Pull Requests

167 Active Issues

527 Merged Pull Requests	96 Open Pull Requests	71 Closed Issues	96 New Issues

Excluding merges, **159 authors** have pushed **524 commits** to develop and **605 commits** to all branches. On develop, **147 files** have changed and there have been **3,688 additions** and **632 deletions**.



527 Pull requests merged by **138** people

We have seen an increase in industry contributions to Spack

- **Fujitsu and RIKEN** have contributed a **huge** number of packages for ARM/a64fx support on Fugaku
- **AMD** has contributed ROCm packages and compiler support
  - 55+ PRs mostly from AMD, also others
  - ROCm, HIP, aocc packages are all in Spack now
- **Intel** contributing oneapi support and compiler licenses for our build farm
- **NVIDIA** contributing NVHPC compiler support and other features
- **ARM** and **Linaro** members contributing ARM support
  - 400+ pull requests for ARM support from various companies
- **AWS** is collaborating with us on our build farm, making optimized binaries for ParallelCluster
  - Joint Spack tutorial in July with AWS had 125+ participants

Spack v0.16.0 was released in November, v0.16.1 in February

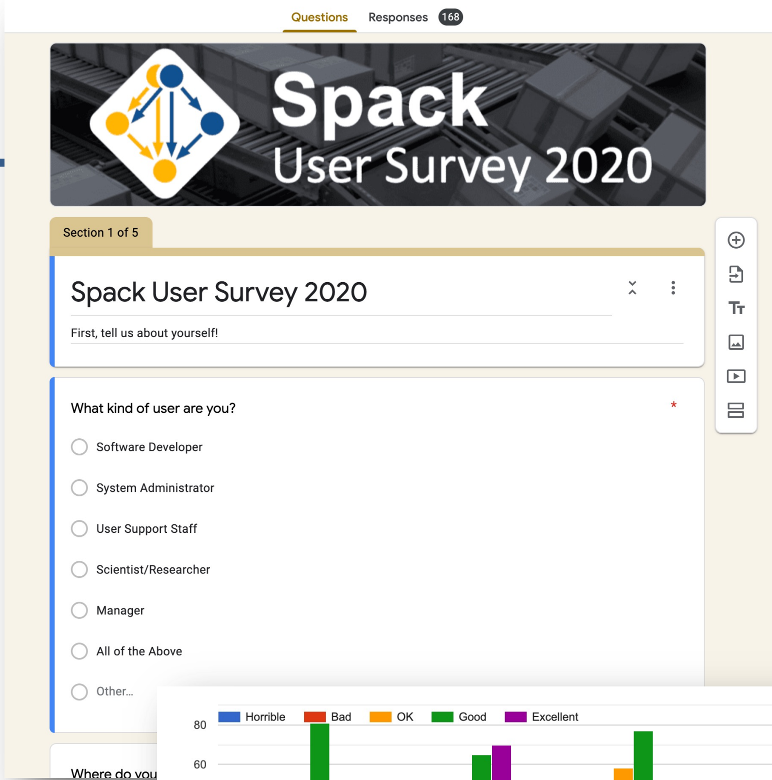
Major new features:

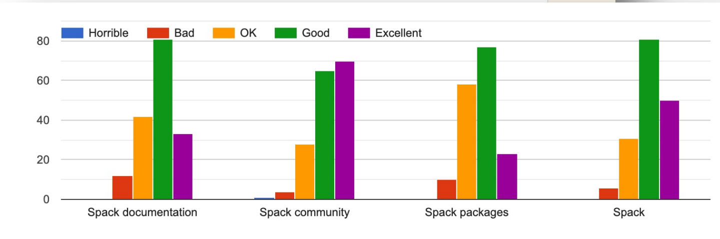
1. New Concretizer (experimental)
2. `spack test` (experimental)
3. `spack develop`
4. Parallel environment builds
5. Custom base images for `spack containerize`
6. `spack external find` support
   - now finds 15 common packages (including perl, MPI, others)
7. Support for aocc, nvhpc, and oneapi compilers

- **5,050** packages (Over **1,500** added since 0.13.1 a year before)
- **Full release notes:** https://github.com/spack/spack/releases/

Spack User Survey 2020

- First widely distributed Spack Survey
  - Sent to all of Slack (900+ users)
  - All of Spack mailing list, ECP mailing list
- Got **169 responses!**
- **Takeaways:**
  - People like Spack and its community!
  - Docs and package stability need the most work
  - Concretizer features and dev features are the most wanted improvements





Results writeup and full survey data at:

https://spack.io/spack-user-survey-2020

Spack is not the only tool that automates builds

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

2. **Build-from-source Package Managers**
   — Homebrew, LinuxBrew — http://brew.sh
   — MacPorts — https://www.macports.org
   — Gentoo — https://gentoo.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 has unoptimized binaries

Agenda

- Part 1 (Morning)

1.	Building & Linking Basics	Slides
2.	Spack Basics	Slides
3.	Basic Spack Usage	Hands-on
4.	Core Spack concepts	Slides
5.	Environments	Hands-on
6.	Configuration	Hands-on

1. Part 2 (Afternoon)

1.	Creating your own Packages	Hands-on
2.	Developer Workflows	Hands-on **(new!)**
3.	Binary Caches and Mirrors	Hands-on **(new!)**
4.	Spack Stacks for facilities	Hands-on
5.	Scripting with Spack	Hands-on (if time)
6.	Future directions and roadmap	Slides

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.

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"             set compiler flags
$ spack install mpileaks@3.3 target=skylake                 set target microarchitecture
$ 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 5,000 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:
     ...
```

Hands-on Time: Spack Basics

Follow script at spack-tutorial.readthedocs.io

Core Spack Concepts

Most existing tools do not support combinatorial versioning

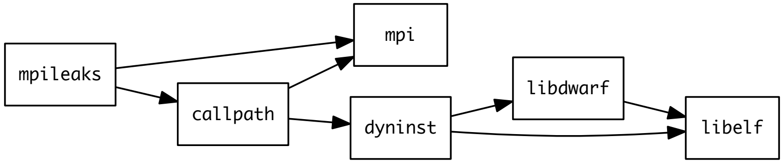
- Traditional binary package managers
  - RPM, yum, APT, yast, etc.
  - Designed to manage a single stack.
  - Install *one* version of each package in a single prefix (`/usr`).
  - Seamless upgrades to a *stable, well tested* stack

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

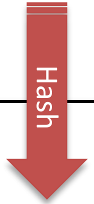
- Virtual Machines and Linux Containers (Docker)
  - Containers allow users to build environments for different applications.
  - Does not solve the build problem (someone has to build the image)
  - Performance, security, and upgrade issues prevent widespread HPC deployment.

Spack handles combinatorial software complexity

Dependency DAG



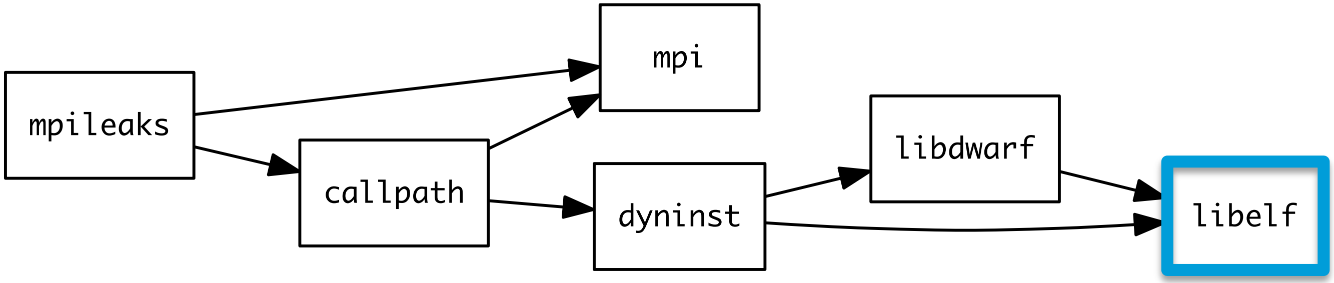
Installation Layout




```
opt
└── spack
    ├── darwin-mojave-skylake
    │   └── clang-10.0.0-apple
    │       ├── bzip2-1.0.8-hc4sm4vuzpm4znmvrfzri4ow2mkphe2e
    │       ├── python-3.7.6-daqqpssxb6qbfrztsezkmhus3xoflbsy
    │       ├── sqlite-3.30.1-u64v26igxvxyn23hysmklfums6tgjv5r
    │       ├── xz-5.2.4-u5eawkvaoc7vonabe6nndkcfwuv233cj
    │       └── zlib-1.2.11-x46q4wm46ay4pltriijbgizxjrhbaka6
    ├── darwin-mojave-x86_64
    │   └── clang-10.0.0-apple
    │       └── coreutils-8.29-pl2kcytejqcys5dzecfrtjqxfdssvnob
```

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

Spack Specs can constrain versions of dependencies

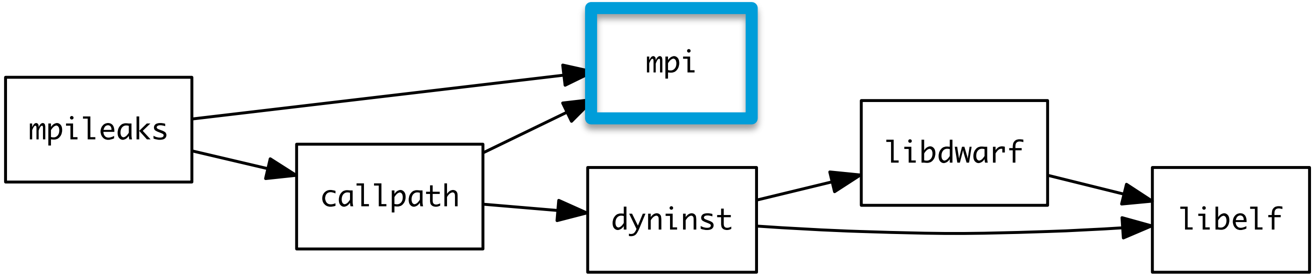



```
$ spack install mpileaks %intel@12.1 ^libelf@0.8.12
```

- Spack ensures *one* configuration of each library per DAG
  - Ensures ABI consistency.
  - User does not need to know DAG structure; only the dependency *names.*

- Spack can ensure that builds use the same compiler, or you can mix
  - Working on ensuring ABI compatibility when compilers are mixed.

Spack handles ABI-incompatible, versioned interfaces like MPI



- mpi is a *virtual dependency*

- Install the same package built with two different MPI implementations:

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

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

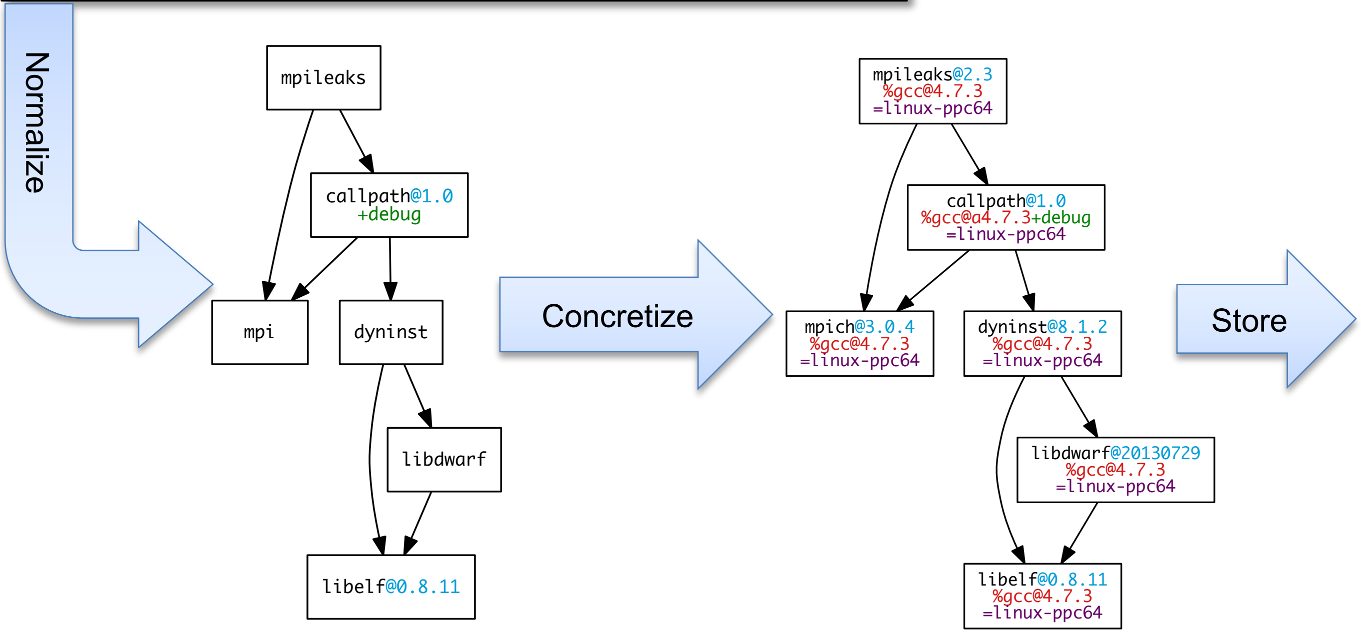
- Let Spack choose MPI implementation, as long as it provides MPI 2 interface:

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

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

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

User input: *abstract* spec with some constraints



spec.yaml


```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
...
```


Abstract, normalized spec with some dependencies.

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

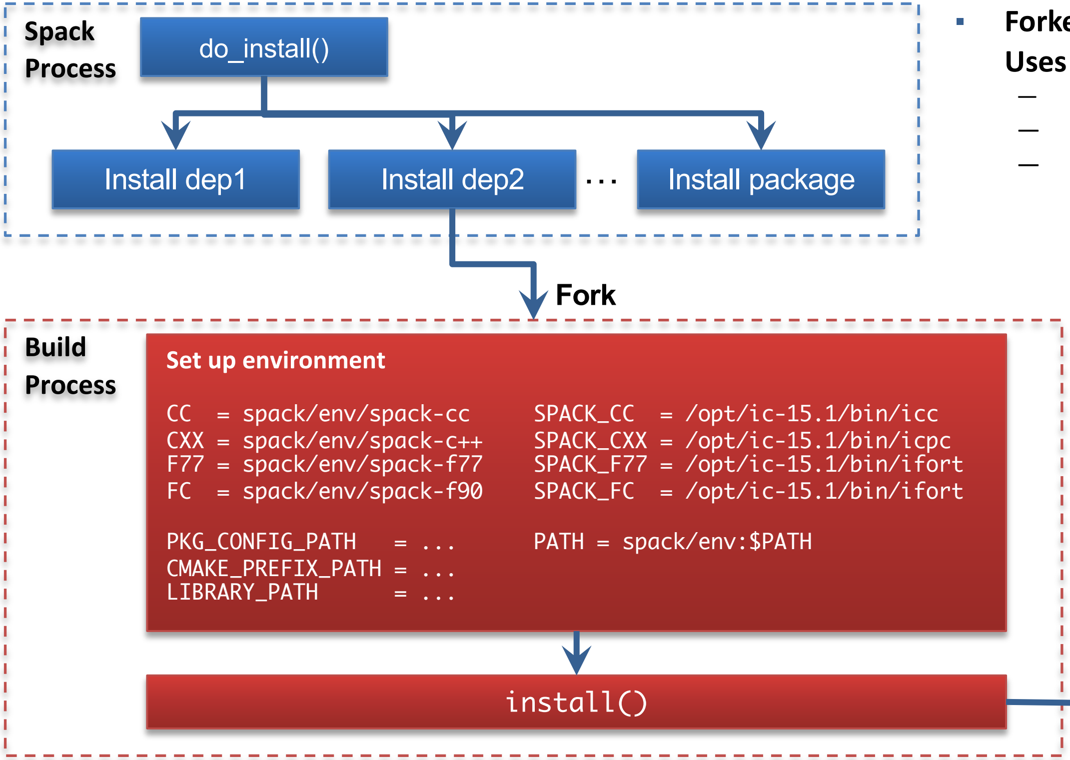
Detailed provenance is stored with the installed package

Use `spack spec` to see the results of concretization

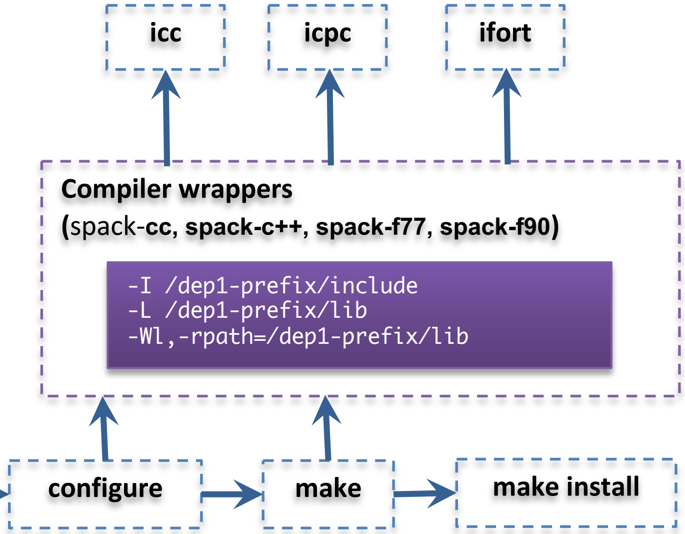
```
$ spack spec mpileaks
Input spec
--------------------------------
  mpileaks

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
```

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)



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

- Some packages need to be installed within directory structure of other packages
- 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
- This is an older feature – we are encouraging users to use **spack environments** instead
  - More on this later!

```
spack/opt/
  linux-rhel6-x86_64/
    gcc-4.7.2/
      python-2.7.12-6y6vvaw/
        lib/python2.7/site-packages/
          ..
      py-numpy-1.10.4-oaxix36/
        lib/python2.7/site-packages/
          numpy/
...
```

```
spack/opt/
  linux-rhel6-x86_64/
    gcc-4.7.2/
      python-2.7.12-6y6vvaw/
        lib/python2.7/site-packages/
          numpy@
      py-numpy-1.10.4-oaxix36/
        lib/python2.7/site-packages/
          numpy/
...
```

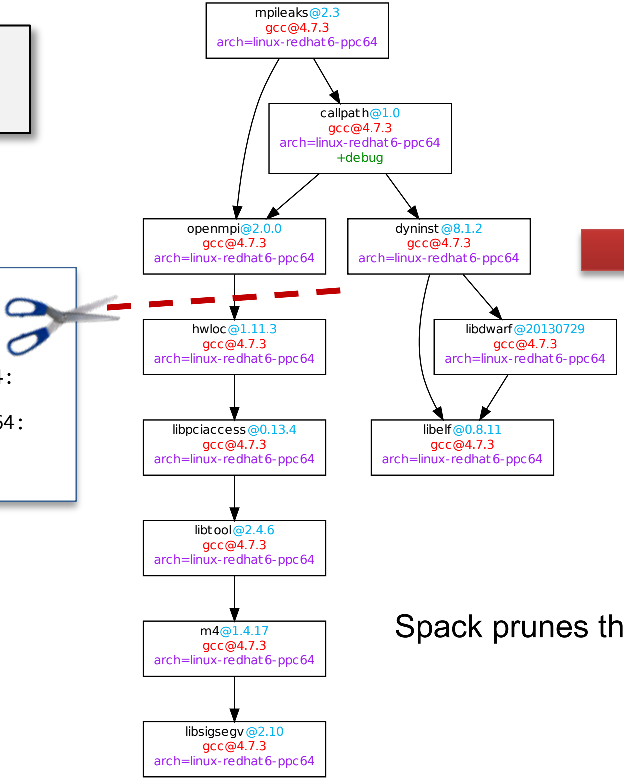
Building against externally installed software

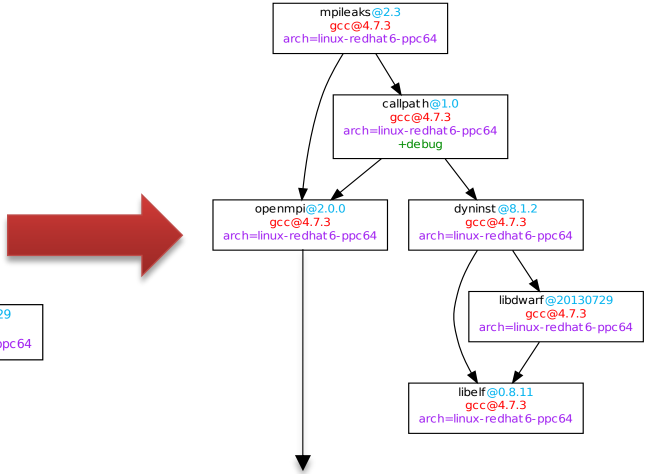
```
mpileaks ^callpath@1.0+debug
    ^openmpi ^libelf@0.8.11
```

packages.yaml

```yaml
packages:
  mpi:
    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
      ...
```

Users register external packages in a configuration file (more on these later).





/path/to/external/gcc/openmpi-2.0.0

Spack prunes the DAG when adding external packages.

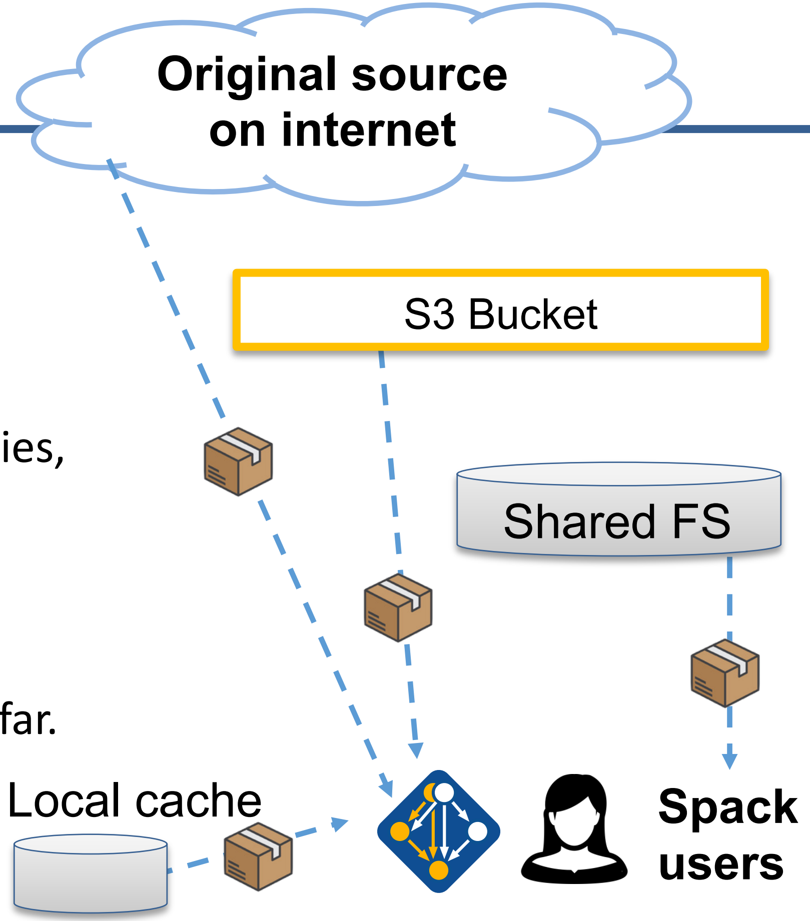
Spack package repositories

- Spack supports external package repositories
  - Separate directories of package recipes
- Many reasons to use this:
  - Some packages can't be released publicly
  - Some sites require ~~bizarre~~ custom builds
  - Override default packages with site-specific versions
- Packages are composable:
  - External repositories can be layered on top of the built-in packages
  - Custom packages can depend on built-in packages (or packages in other repos)

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

Spack mirrors

- Spack allows you to define *mirrors:*
  - Directories in the filesystem
  - On a web server
  - In an S3 bucket
- Mirrors are archives of fetched tarballs, repositories, and other resources needed to build
  - Can also contain binary packages
- By default, Spack maintains a mirror in var/spack/cache of everything you've fetched so far.
- You can host mirrors internal to your site
  - See the documentation for more details



Environments, `spack.yaml` and `spack.lock`

Follow script at spack-tutorial.readthedocs.io

Hands-on Time: Configuration

Follow script at spack-tutorial.readthedocs.io

Day 2
Spack Review

Tutorial Materials

Find these slides and associated scripts here:

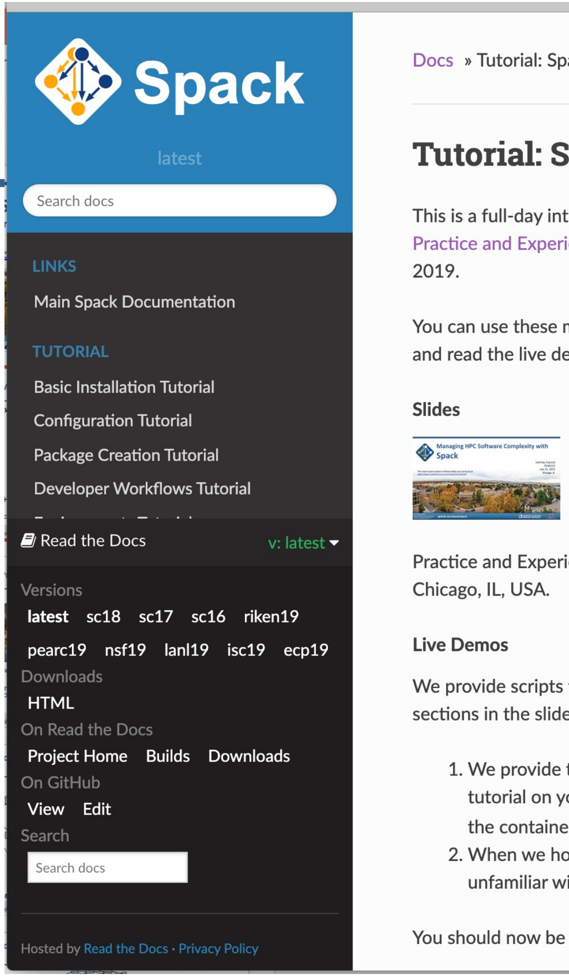
spack-tutorial.readthedocs.io

We will also have a chat room on Spack slack. Get an invite here:

spackpm.herokuapp.com

Join the “tutorial” channel!

We will give you login credentials for the hands-on exercises on Slack!



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"            set compiler flags
$ spack install mpileaks@3.3 target=skylake                set target microarchitecture
$ 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

Not shown: patches, resources, conflicts, other directives.

Spack packages are *templates*
They use a simple Python DSL to define how to build

```
from spack import *

class Kripke(CMakePackage):
    """Kripke is a simple, scalable, 3D Sn deterministic particle
       transport proxy/mini app.
    """

    homepage = "https://computation.llnl.gov/projects/co-design/kripke"
    url      = "https://computation.llnl.gov/projects/co-design/download/kripke-openmp-1.1.tar.gz"

    version('1.2.3', sha256='3f7f2eef0d1ba5825780d626741eb0b3f026a096048d7ec4794d2a7dfbe2b8a6')
    version('1.2.2', sha256='eaf9ddf562416974157b34d00c3a1c880fc5296fce2aa2efa039a86e0976f3a3')
    version('1.1', sha256='232d74072fc7b848fa2adc8a1bc839ae8fb5f96d50224186601f55554a25f64a')

    variant('mpi',    default=True, description='Build with MPI.')
    variant('openmp', default=True, description='Build with OpenMP enabled.')

    depends_on('mpi', when='+mpi')
    depends_on('cmake@3.0:', type='build')

    def cmake_args(self):
        return [
            '-DENABLE_OPENMP=%s' % ('+openmp' in self.spec),
            '-DENABLE_MPI=%s' % ('+mpi' in self.spec),
        ]

    def install(self, spec, prefix):
        # Kripke does not provide install target, so we have to copy
        # things into place.
        mkdirp(prefix.bin)
        install('../spack-build/kripke', prefix.bin)
```

Base package
(CMake support)

Metadata at the class level

Versions

Variants (build options)

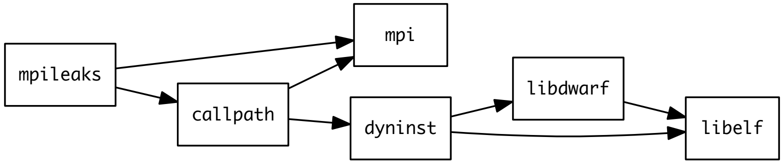
Dependencies
(note: same spec syntax)

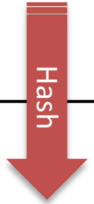
Install logic
in instance methods

Don't typically need `install()` for `CMakePackage`, but we can work around codes that don't have it.

Spack handles combinatorial software complexity

Dependency DAG





Installation Layout

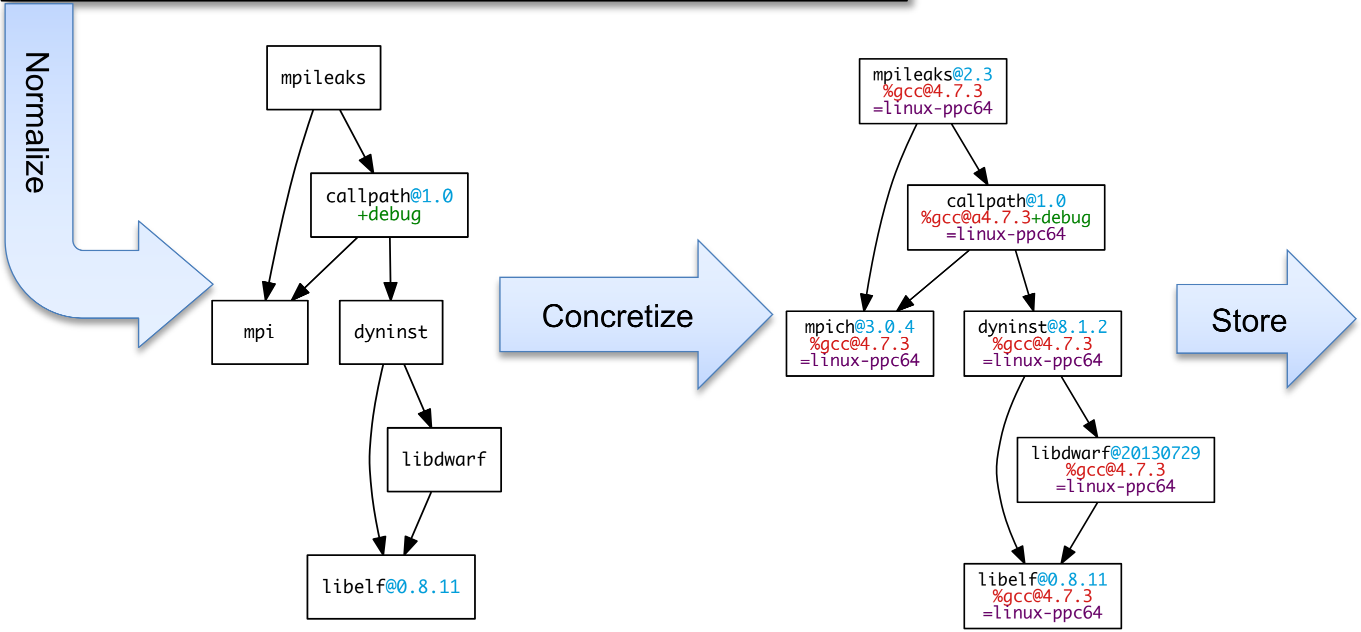
```
opt
└── spack
    ├── darwin-mojave-skylake
    │   └── clang-10.0.0-apple
    │       ├── bzip2-1.0.8-hc4sm4vuzpm4znmvrfzri4ow2mkphe2e
    │       ├── python-3.7.6-daqqpssxb6qbfrztsezkmhus3xoflbsy
    │       ├── sqlite-3.30.1-u64v26igxvxyn23hysmklfums6tgjv5r
    │       ├── xz-5.2.4-u5eawkvaoc7vonabe6nndkcfwuv233cj
    │       └── zlib-1.2.11-x46q4wm46ay4pltriijbgizxjrhbaka6
    ├── darwin-mojave-x86_64
    │   └── clang-10.0.0-apple
    │       └── coreutils-8.29-pl2kcytejqcys5dzecfrtjqxfdssvnob
```

- Each unique dependency graph is a unique ***configuration***.
- Each configuration in a unique directory.
  - Multiple 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*

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

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

User input: *abstract* spec with some constraints



spec.yaml


```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
...
```


Abstract, normalized spec with some dependencies.

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

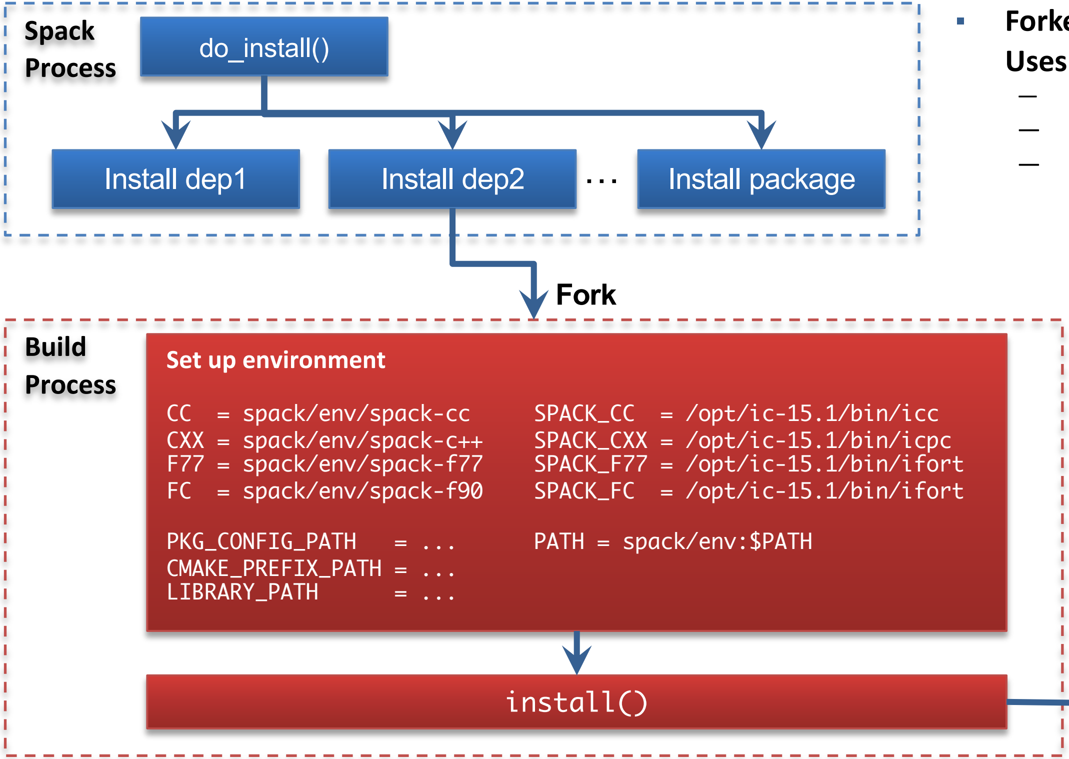
Detailed provenance is stored with the installed package

Use `spack spec` to see the results of concretization

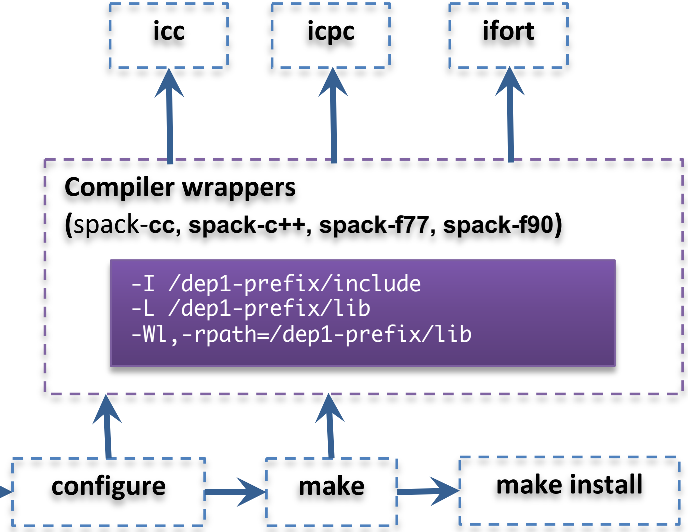
```
$ spack spec mpileaks
Input spec
------------------------------
  mpileaks

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
```

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)



Hands-on Time: Creating Packages

Follow script at spack-tutorial.readthedocs.io

Hands-on Time:
Developer Workflows

Follow script at spack-tutorial.readthedocs.io

Hands-on Time:
Binary Caches and Mirrors

Follow script at spack-tutorial.readthedocs.io

Spack Stacks

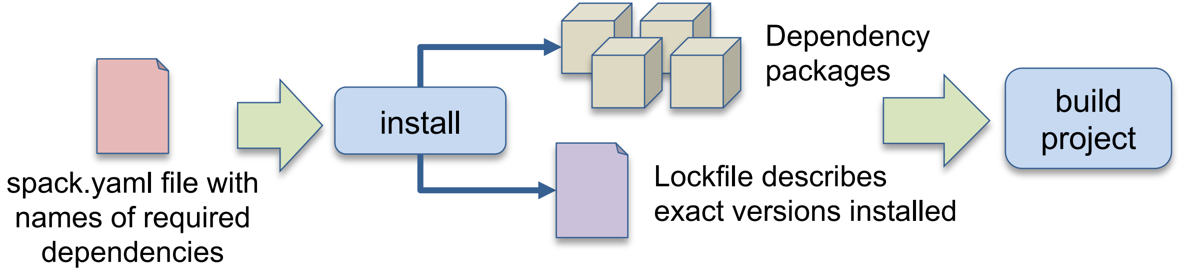
Follow script at spack-tutorial.readthedocs.io

Scripting and spack-python

Follow script at spack-tutorial.readthedocs.io

More New Features and the Road Ahead

Spack environments are the basis for complex workflows



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": {
   "6s63so2kstp3zyvjezglndmavy6l3nul": {
     "hdf5": {
         "version": "1.10.5",
         "arch": {
             "platform": "darwin",
             "platform_os": "mojave",
             "target": "x86_64"
         },
         "compiler": {
             "name": "clang",
             "version": "10.0.0-apple"
         },
         "namespace": "builti
         "parameters":
```

- Two files:
  - `spack.yaml` describes project requirements
  - `spack.lock` records installed versions and configurations exactly
  - Enables reproducibility for many configurations
- Can use environments for:
  - Creating containers (`spack containerize`)
  - Auto-generate continuous integration builds (`spack ci`)
  - Deployment (`matrix`, spack stacks)
  - **Developer workflows (new!)**

Generate container images from environments (0.14)

```yaml
spack:
  specs:
  - gromacs+mpi
  - mpich

  container:
    # Select the format of the recip
    # singularity or anything else t
    format: docker

    # Select from a valid list of im
    base:
      image: "centos:7"
      spack: develop

    # Whether or not to strip binari
    strip: true

    # Additional system packages tha
    os_packages:
    - libgomp

    # Extra instructions
    extra_instructions:
      final: |
RUN echo 'export PS1="\[$(tput bold)

    # Labels for the image
    labels:
      app: "gromacs"
      mpi: "mpich"
```

```dockerfile
# Build stage with Spack pre-installed and ready to be used
FROM spack/centos7:latest as builder

# What we want to install and how we want to install it
# is specified in a manifest file (spack.yaml)
RUN mkdir /opt/spack-environment \
&&  (echo "spack:" \
&&   echo "  specs:" \
&&   echo "  - gromacs+mpi" \
&&   echo "  - mpich" \
&&   echo "  concretization: together" \
&&   echo "  config:" \
&&   echo "    install_tree: /opt/software" \
&&   echo "  view: /opt/view") > /opt/spack-environment/spack.yaml

# Install the software, remove unecessary deps
RUN cd /opt/spack-environment && spack install && spack gc -y

# Strip all the binaries
RUN find -L /opt/view/* -type f -exec readlink -f '{}' \; | \
    xargs file -i | \
    grep 'charset=binary' | \
    grep 'x-executable\|x-archive\|x-sharedlib' | \
    awk -F: '{print $1}' | xargs strip -s

# Modifications to the environment that are necessary to run
RUN cd /opt/spack-environment && \
    spack env activate --sh -d . >> /etc/profile.d/z10_spack_environment.sh


# Bare OS image to run the installed executables
FROM centos:7

COPY --from=builder /opt/spack-environment /opt/spack-environment
COPY --from=builder /opt/software /opt/software
COPY --from=builder /opt/view /opt/view
C   --from=builder /etc/profile.d/z10_spack_environment.sh /etc/profile.d/z10_spack_env

   m update -y && yum install -y epel-release && yum update -y
     install -y libgomp \
   rm -rf /var/cache/yum  && yum clean all

RUN echo 'export PS1="\[$(tput bold)\]\[$(tput setaf 1)\][gromacs]\[$(tput setaf 2)\]\u\[$(tpu
```

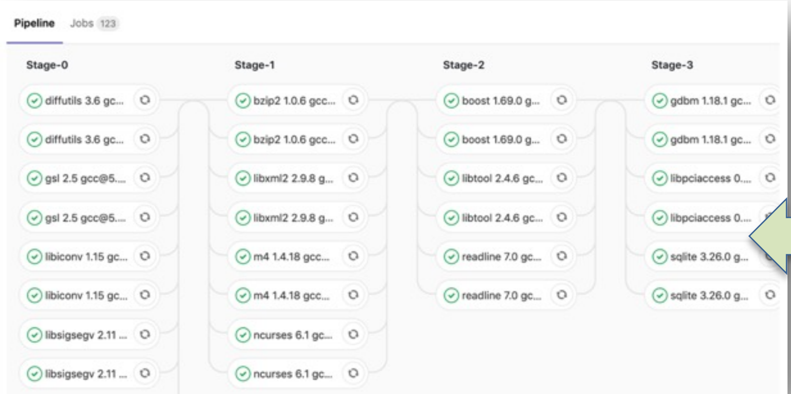
- Any Spack environment can be bundled into a container image
  - Optional container section allows finer-grained customization
- Generated Dockerfile uses multi-stage builds to minimize size of final image
  - Strips binaries
  - Removes unneeded build deps with `spack gc`
- Can also generate Singularity recipes

`spack containerize`

Spack can generate CI Pipelines from environments

- User adds a `gitlab-ci` section to environment
  - Spack maps builds to GitLab runners
  - Generate gitlab-ci.yml with `spack ci` command

- Can run in a Kube cluster or on bare metal at an HPC site
  - Sends progress to CDash



`spack ci`

```yaml
spack:
  definitions:
  - pkgs:
    - readline@7.0
  - compilers:
    - '%gcc@5.5.0'
  - oses:
    - os=ubuntu18.04
    - os=centos7
  specs:
  - matrix:
    - [$pkgs]
    - [$compilers]
    - [$oses]
  mirrors:
    cloud_gitlab: https://mirror.spack.io
  gitlab-ci:
    mappings:
      - spack-cloud-ubuntu:
        match:
          - os=ubuntu18.04
        runner-attributes:
          tags:
            - spack-k8s
          image: spack/spack_builder_ubuntu_18.04
      - spack-cloud-centos:
        match:
          - os=centos7
        runner-attributes:
          tags:
            - spack-k8s
          image: spack/spack_builder_centos_7
  cdash:
    build-group: Release Testing
    url: https://cdash.spack.io
    project: Spack
    site: Spack AWS Gitlab Instance
```

spack external find

```python
class Cmake(Package):
    executables = ['cmake']

    @classmethod
    def determine_spec_details(cls, prefix, exes_in_prefix):
        exe_to_path = dict(
            (os.path.basename(p), p) for p in exes_in_prefix
        )
        if 'cmake' not in exe_to_path:
            return None

        cmake = spack.util.executable.Executable(exe_to_path['cmake'])
        output = cmake('--version', output=str)
        if output:
            match = re.search(r'cmake.*version\s+(\S+)', output)
            if match:
                version_str = match.group(1)
                return Spec('cmake@{0}'.format(version_str))
```

Logic for finding external installations in `package.py`

```yaml
packages:
  cmake:
    externals:
    - spec: cmake@3.15.1
      prefix: /usr/local
```

`packages.yaml` configuration

- Spack has had compiler detection for a while
  - Finds compilers in your PATH
  - Registers them for use
- We can find any package now
  - Package defines:
    - possible command names
    - how to query the command
  - Spack searches for known commands and adds them to configuration
- Community can easily enable tools to be set up rapidly

spack test: write tests directly in Spack packages, so that they can evolve with the software

```
class Libsigsegv(AutotoolsPackage, GNUMirrorPackage):
    """GNU libsigsegv is a library for handling page faults in user mode."""

    # ... spack package contents ...

    extra_install_tests = ‘tests/.libs’

    def test(self):
        data_dir = self.test_suite.current_test_data_dir
        smoke_test_c = data_dir.join(‘smoke_test.c’)

        self.run_test(
            'cc’, [
                '-I%s' % self.prefix.include,
                '-L%s' % self.prefix.lib, '-lsigsegv’,
                smoke_test_c,
                '-o', 'smoke_test'
            ]
            purpose='check linking’)

        self.run_test(
            ‘smoke_test’, [], data_dir.join('smoke_test.out’),
            purpose=‘run built smoke test’)

        self.run_test('sigsegv1': ['Test passed’], purpose='check sigsegv1 output’)
        self.run_test('sigsegv2': ['Test passed’], purpose='check sigsegv2 output’)
```

Tests are part of a regular Spack recipe class

Easily save source code from the package

User just defines a `test()` method

Retrieve saved source.
Link a simple executable.

Spack ensures that `cc` is a compatible compiler

Run the built smoke test and verify output

Run programs installed with package

Build configuration is its own many-dimensional constraint optimization problem

- **The new concretizer in v0.16.0** allows us to solve this problem
  - Uses *Answer Set Programming* – framework for solving NP-hard optimization problems
  - Unlike other systems, package manager has insight into build details and configuration

- ASP program has 2 parts:
  1. Large list of facts
     - generated from our package repositories
     - 20,000 – 30,000 facts is typical
     - includes dependencies, versions, options, etc.
  2. Small logic program
     - ~800 lines of ASP code
     - 300 rules + 11 optimization criteria

```
%----------------------------------------------------------------------------
% Package: ucx
%----------------------------------------------------------------------------
version_declared("ucx", "1.6.1", 0).
version_declared("ucx", "1.6.0", 1).
version_declared("ucx", "1.5.2", 2).
version_declared("ucx", "1.5.1", 3).
version_declared("ucx", "1.5.0", 4).
version_declared("ucx", "1.4.0", 5).
version_declared("ucx", "1.3.1", 6).
version_declared("ucx", "1.3.0", 7).
version_declared("ucx", "1.2.2", 8).
version_declared("ucx", "1.2.1", 9).
version_declared("ucx", "1.2.0", 10).

variant("ucx", "thread_multiple").
variant_single_value("ucx", "thread_multiple").
variant_default_value("ucx", "thread_multiple", "False").
variant_possible_value("ucx", "thread_multiple", "False").
variant_possible_value("ucx", "thread_multiple", "True").

declared_dependency("ucx", "numactl", "build").
declared_dependency("ucx", "numactl", "link").
node("numactl") :- depends_on("ucx", "numactl"), node("ucx").

declared_dependency("ucx", "rdma-core", "build").
declared_dependency("ucx", "rdma-core", "link").
node("rdma-core") :- depends_on("ucx", "rdma-core"), node("ucx").


%----------------------------------------------------------------------------
% Package: util-linux
%----------------------------------------------------------------------------
version_declared("util-linux", "2.29.2", 0).
version_declared("util-linux", "2.29.1", 1).
version_declared("util-linux", "2.25", 2).

variant("util-linux", "libuuid").
variant_single_value("util-linux", "libuuid").
variant_default_value("util-linux", "libuuid", "True").
variant_possible_value("util-linux", "libuuid", "False").
variant_possible_value("util-linux", "libuuid", "True").

declared_dependency("util-linux", "pkgconfig", "build").
declared_dependency("util-linux", "pkgconfig", "link").
node("pkgconfig") :- depends_on("util-linux", "pkgconfig"), node("util-linux").

declared_dependency("util-linux", "python", "build").
declared_dependency("util-linux", "python", "link").
node("python") :- depends_on("util-linux", "python"), node("util-linux").
```

Sample ASP input for Spack solver

The new concretizer enables significant simplifications to packages, particularly complex constraints in SDKs

- Dependencies and other constraints within SDKs could get very messy
- The new concretizer removes the need for some of the more painful constructs
- Also allows for new constructs, like specializing dependencies
  - When conditions are now much more general
  - Can be solved together with other constraints.

In some cases we needed cross-products of dependency options:

Before

```
depends_on('foo+A+B', when='+a+b')
depends_on('foo+A~B', when='+a~b')
depends_on('foo~A+B', when='~a+b')
depends_on('foo~A~B', when='~a~b')
```

After

```
depends_on('foo')
depends_on('foo+A', when='+a')
depends_on('foo+B', when='+b')
```

Specializing a virtual did not previously work:

```
depends_on('blas')
depends_on(
    'openblas threads=openmp', when='^openblas'
)
```

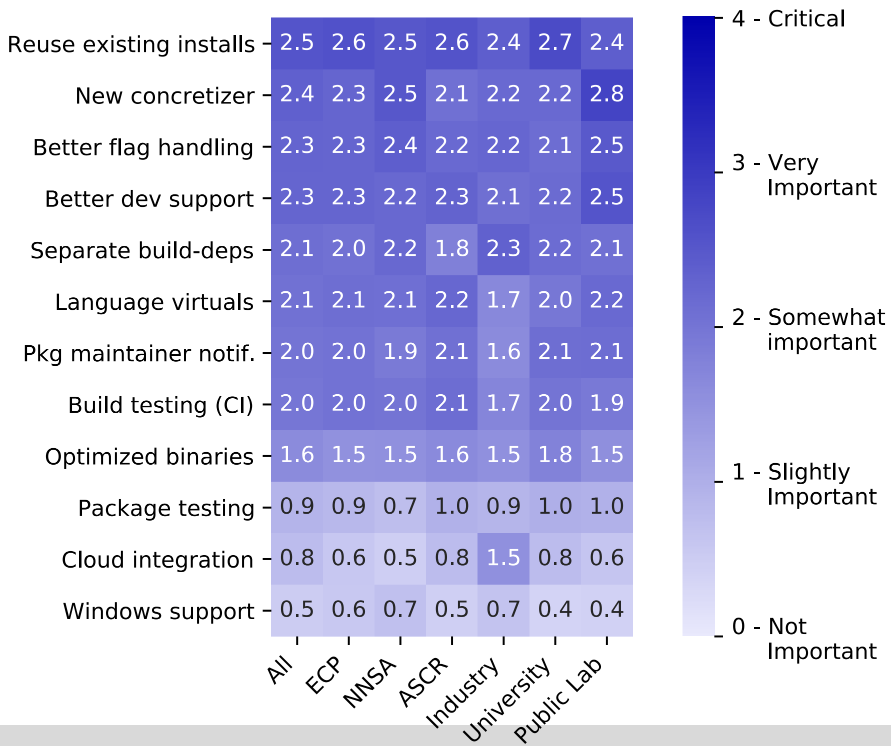
Spack 0.17 Roadmap: permissions and directory structure

- **Sharing a Spack instance**
  - Many users want to be able to install Spack on a cluster and `module load spack`
  - Installations in the Spack prefix are shared among users
  - Users would `spack install` to their home directory by default.
  - This requires us to move most state ***out*** of the Spack prefix
    - Installations would go into ~/.spack/…

- **Getting rid of configuration in ~/.spack**
  - While *installations* may move to the home directory, *configuration* there is causing issues
  - User configuration is like an unwanted global (e.g., LD_LIBRARY_PATH 😬)
    - Interferes with CI builds (many users will `rm -rf ~/.spack` to avoid it)
    - Goes against a lot of our efforts for reproducibility
    - Hard to manage this configuration between multiple machines
  - Environments are a much better fit
    - Make users keep configuration like this in an environment instead of a single config

Four of the top six most wanted features in Spack are tied to the new concretizer

Average feature importance by workplace



- Complexity of packages in Spack is increasing
  - many more package solves require backtracking than a year ago
  - Many variants, conditional dependencies, special compiler requirements
- More aggressive reuse of existing installs requires better dependency resolution
  - Need to be able to analyze how to configure the build to work with installed packages
- Separate resolution of build dependencies also requires a more sophisticated solver
  - Makes the solve even more combinatorial
  - Needed to support mixed compilers, version conflicts between different package’s build requirements

We will be releasing v0.17 in the next 1-2 months

Main goals:

1. Get rid of the old concretizer, make the new concretizer default
2. Improve and harden binary cache workflows
3. Make Spack able to optimize for reuse of installed packages and packages from binary mirrors
4. Make “shared” spack instances for facilities more manageable
5. Get rid of pain points like ~/.spack configuration

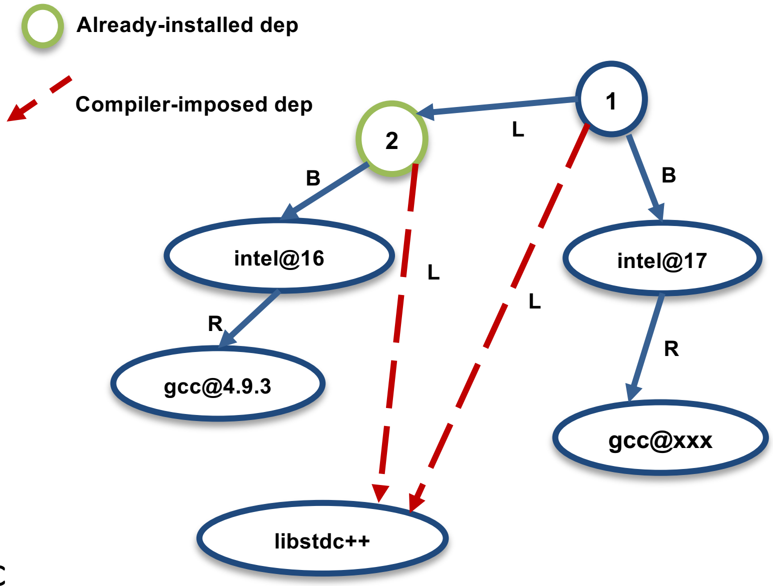
Spack 0.17 Roadmap: permissions and directory structure

- **Sharing a Spack instance**
  - Many users want to be able to install Spack on a cluster and `module load spack`
  - Installations in the Spack prefix are shared among users
  - Users would `spack install` to their home directory by default.
  - This requires us to move most state ***out*** of the Spack prefix
    - Installations would go into ~/.spack/…

- **Getting rid of configuration in ~/.spack**
  - While *installations* may move to the home directory, *configuration* there is causing issues
  - User configuration is like an unwanted global (e.g., LD_LIBRARY_PATH 😬)
    - Interferes with CI builds (many users will `rm -rf ~/.spack` to avoid it)
    - Goes against a lot of our efforts for reproducibility
    - Hard to manage this configuration between multiple machines
  - Environments are a much better fit
    - Make users keep configuration like this in an environment instead of a single config

Spack 0.18 Roadmap: compilers as dependencies

- **We need deeper modeling of compilers to handle compiler interoperability**
  - libstdc++, libc++ compatibility
  - Compilers that depend on compilers
  - Linking executables with multiple compilers
- **First prototype is complete!**
  - We've done successful builds of some packages using compilers as dependencies
  - We need the new concretizer to move forward!
- **Packages that depend on languages**
  - Depend on **cxx@2011**, **cxx@2017**, **fortran@1995**, etc
  - Depend on **openmp@4.5**, other compiler features
  - Model languages, openmp, cuda, etc. as virtuals



Compilers and runtime libs fully modeled as dependencies

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!
  - You're already on our **Slack channel** (spackpm.herokuapp.com)
  - Join our **Google Group** (see GitHub repo for info)
  - Submit **GitHub issues** and **pull requests**!

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

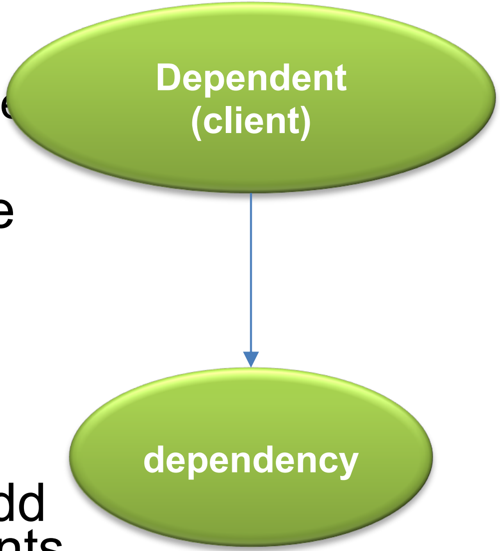
Follow us on Twitter!
@spackpm

We hope to make distributing & using HPC software easy!

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

```python
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)

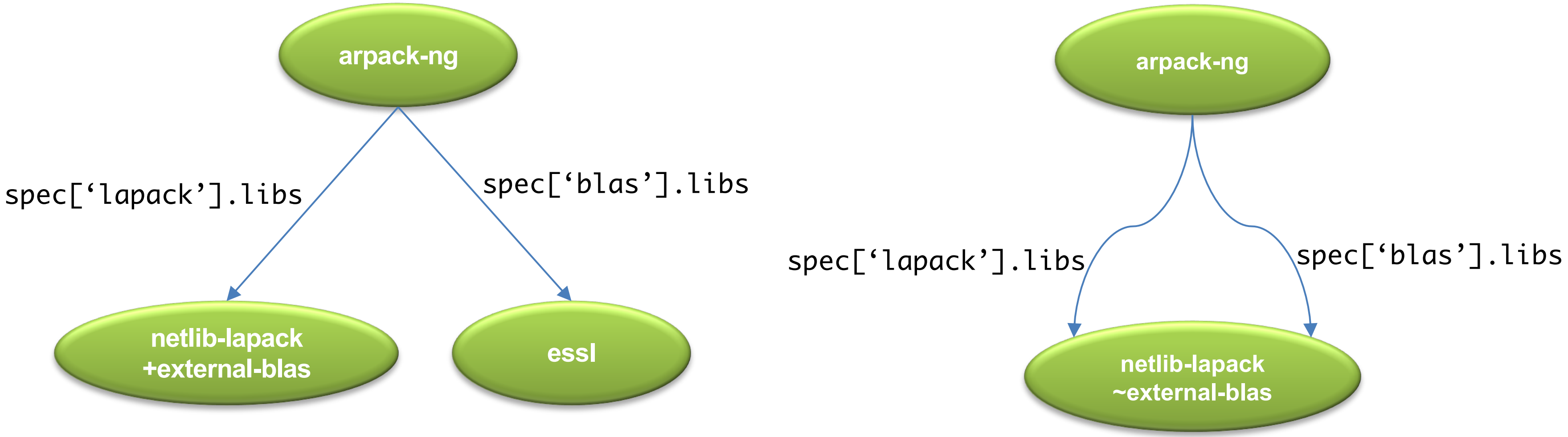
`.libs.search_flags` expresses the libraries as linker arguments like:

`"-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 g run the python exe:

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

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

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

- F mentations set the .r sociated spec

Hands-on Time: Environment Modules

Follow script at spack-tutorial.readthedocs.io