Spack: A Package Manager for HPC

Todd Gamblin
Advanced Technology Office, Livermore Computing
Lawrence Livermore National Laboratory

NCSA Blue Waters Webinar
October 30, 2019
Software complexity in HPC is growing

Ascent: Lightweight, in-situ, many-core visualization and analysis
Software complexity in HPC is growing

MFEM: Arbitrary high-order finite elements
Software complexity in HPC is growing

Ascent: Lightweight, in-situ, many-core visualization and analysis

MFEM: Arbitrary high-order finite elements

LBANN: Artificial Neural Nets for HPC
The complexity of the exascale ecosystem threatens productivity.

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

= up to 1,260,000 combinations!

- 15+ applications
- 80+ software packages
- 5+ target architectures/platforms
  - Xeon
  - Power
  - KNL
  - NVIDIA
  - ARM
  - Laptops?
- Up to 7 compilers
  - Intel
  - GCC
  - Clang
  - XL
  - PGI
  - Cray
  - NAG
- 10+ Programming Models
  - OpenMPI
  - MPICH
  - MVAPICH
  - OpenMP
  - CUDA
  - OpenACC
  - Dharma
  - Legion
  - RAJA
  - Kokkos
- 2-3 versions of each package + external dependencies

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

github.com/spack/spack
How to install software on a Mac laptop, circa 2013

```bash
(gluon):~$ port install libelf
---> Computing dependencies for libelf
---> Fetching distfiles for libelf
---> Verifying checksum(s) for libelf
---> Extracting libelf
---> Applying patches to libelf
---> Configuring libelf
---> Building libelf
---> Staging libelf into destroot
---> Installing libelf @0.8.13_2
---> Activating libelf @0.8.13_2
---> Cleaning libelf
---> Updating database of binaries: 100.0%
---> Scanning binaries for linking errors: 100.0%
---> No broken files found.
(gluon):~$ 
```
How to install software on a supercomputer

1. Download all 16 tarballs you need
2. Start building!
How to install software on a supercomputer

1. Download all 16 tarballs you need
2. Start building!
How to install software on a supercomputer

1. Download all 16 tarballs you need
2. Start building!
3. Run code
4. Segfault!?
5. Start over…
What about modules?

- Most supercomputers deploy some form of *environment modules*
  - TCL modules (dates back to 1995) and Lmod (from TACC) are the most popular

  ```
  $ gcc
  -bash: gcc: command not found
  
  $ module load gcc/7.0.1
  $ gcc --dumpversion
  7.0.1
  ```

- Modules don’t handle installation!
  - They only modify your environment (things like PATH, LD_LIBRARY_PATH, etc.)

- Someone (likely a team of people) has already installed gcc for you!
  - Also, you can *only* `module load` the things they’ve installed
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

github.com/spack/spack
Spack is a flexible package manager for HPC

- Spack automates the build and installation of scientific software
- Packages are **templated**, so that users can easily tune for the host environment

**No installation required: clone and go**

```
$ git clone https://github.com/spack/spack
$ spack install hdf5
```

**Simple syntax enables complex installs**

```
$ spack install hdf5@1.10.5
$ spack install hdf5@1.10.5 %clang@6.0
$ spack install hdf5@1.10.5 +threadssafe
```

- Ease of use of mainstream tools, with flexibility needed for HPC tuning

- Major victories:
  - ARES porting time on a new platform was reduced from **2 weeks to 3 hours**
  - Deployment time for 1,300-package stack on Summit supercomputer reduced from **2 weeks to a 12-hour overnight build**
  - Used by teams across ECP to **accelerate development**

```bash
$ git clone https://github.com/spack/spack
git clone http://github.com/spack/spack
```
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

[github.com/spack/spack](https://github.com/spack/spack)
Spack is used worldwide!

Over 3,500 software packages
Over 2,000 monthly active users (on docs site)

Over 450 contributors from labs, academia, industry

Plot shows users on spack.readthedocs.io for one month
Active Users on the spack.readthedocs.io

Oct 22, 2017 - Oct 28, 2019

Active Users

- 1 Day Active Users
- 7 Day Active Users
- 14 Day Active Users
- 28 Day Active Users

1 Day Active Users: 152 (100.00% of total)
7 Day Active Users: 684 (100.00% of total)
14 Day Active Users: 1,264 (100.00% of total)
28 Day Active Users: 2,270 (100.00% of total)
Spack is being used on many of the top HPC systems

- Official deployment tool for the U.S. Exascale Computing Project
- 7 of the top 10 supercomputers
- High Energy Physics community — Fermilab, CERN, collaborators
- Astra (Sandia)
- Fugaku (Japanese National Supercomputer Project)

Fugaku coming to RIKEN in 2021
DOE/MEXT collaboration

Summit (ORNL), Sierra (LLNL)
SuperMUC-NG (LRZ, Germany)
Edison, Cori, Perlmutter (NERSC)

github.com/spack/spack
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 3,500 packages
- Most packages are from external contributors!
- Many contributions in core, as well.
- We are committed to sustaining Spack’s open source ecosystem!
Related Work

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

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

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

Other tools in the HPC Space:

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

- Conda
  — Very popular binary package manager for data science
  — Not targeted at HPC; generally unoptimized binaries
Spack Basics
Spack provides a spec syntax to describe customized DAG configurations

- Each expression is a **spec** for a particular configuration
  - Each clause adds a constraint to the spec
  - Constraints are optional – specify only what you need.
  - Customize install on the command line!

- Spec syntax is recursive
  - Full control over the combinatorial build space

```
$ spack install mpileaks
$ spack install mpileaks@3.3   unconstrained
$ spack install mpileaks@3.3 %gcc@4.7.3 @ custom version
$ spack install mpileaks@3.3 %gcc@4.7.3 +threads % custom compiler
$ spack install mpileaks@3.3 cppflags="-O3 -g3" +/- build option
$ spack install mpileaks@3.3 target=skylake set target microarchitecture
$ spack install mpileaks@3.3 ^mpich@3.2 %gcc@4.9.3 ^ dependency information
```
`spack list` shows what packages are available

$ spack list

```bash
===> 303 packages.
```

- Spack has over 3,500 packages now.
All the versions coexist! — Multiple versions of the 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.

```bash
$ 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@0.2.0.0  libtool@2.4.2  parmetis@4.0.3  qt@5.4.0
    adept-utils@1.0.1  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  hdfs@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
    fontconfig@2.11.1  libffi@8.1.2  ocr@2015-02-16  py-setuptools@11.3.1  zlib@1.2.8
    freetype@2.5.3  libmg@2.0.2  otf@1.12.5salmon  python@2.7.8
    gdk-pixbuf@2.31.2  libpng@1.6.16  otf2@1.4  qhull@1.0.0

  -- 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
```
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
```

```
$ spack find -dl callpath
==> 2 installed packages.
-- linux-rhel6-x86_64 / clang@3.4 -----  -- linux-rhel6-x86_64 / gcc@4.9.2 -----.
udltshs callpath@1.0.2
```

Expand dependencies with `spack find -d`

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

github.com/spack/spack
Spack manages installed compilers

- Compilers are automatically detected
  - Automatic detection determined by OS
  - Linux: PATH
  - Cray: `module avail`

- Compilers can be manually added
  - Including Spack-built compilers

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

-- clang -----------------------------
clang@6.0
compilers.yaml
```

```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
```
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

- 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

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

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

```bash
$ spack install mpileaks %intel@12.1 ^libelf@0.8.12
```
Spack handles ABI-incompatible, versioned interfaces like MPI

- **mpi** is a *virtual dependency*

- Install the same package built with two different MPI implementations:
  
  ```
  $ spack install mpileaks ^mvapich@1.9  
  $ spack install mpileaks ^openmpi@1.4:
  ```

- Let Spack choose MPI implementation, as long as it provides MPI 2 interface:
  
  ```
  $ spack install mpileaks ^mpi@2
  ```
Concretization fills in missing configuration details when the user is not explicit.

User input: abstract spec with some constraints

```yaml
spec:
  - mpileaks:
    arch: linux-x86_64
    compiler:
      name: gcc
      version: 4.9.2
    dependencies:
      - adept-utils
        hash: kszrtkpbzac3ss2ixcjkcorlaybnptp4
      - mpich
        hash: aa4ar6ifj23yijqmdabeakpejcli72t3
    callpath: bah5f4h4d2n47mgycej2mtrnrivvxy77
    version: 1.0+
  - adept-utils:
    arch: linux-x86_64
    compiler:
      name: gcc
      version: 4.9.2
    dependencies:
      - boost
        hash: teesjv7ehpe5ksspjim5dk43a7qnowlq
    callpath: ooa49f12j3yiymdabeakpejcli72t3
    hash: kszrtkpbzac3ss2ixcjkcorlaybnptp4
    version: 1.0.1
  - boost:
    arch: linux-x86_64
    compiler:
      name: gcc
      version: 4.9.2
    dependencies:
      - adept-utils
        hash: teesjv7ehpe5ksspjim5dk43a7qnowlq
    callpath: ooa49f12j3yiymdabeakpejcli72t3
    hash: kszrtkpbzac3ss2ixcjkcorlaybnptp4
    version: 1.0.1
    variants: {}
    version: 1.59.0
...
Use `spack spec` to see the results of concretization

```bash
$ 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 packages are templates
They use a simple Python DSL to define how to build

```python
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='3f7f2eef0d1ba5825780d626741eb0b3f026a0966b48d7ec4794d2a7dfbe2b8a6')
    version('1.2.2', sha256='eaf9ddf562416974157b34d00c3a1c880fc5296fcee2aa2efa039a6e0976f3a3')
    version('1.1', sha256='232d74072fc7b048fa2adc0a1bc839ae07b5f96d50224186681f55554a25f64a')

    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)
```

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 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)

### Build Process

**Set up environment**

- **CC** = spack/env/spack-cc
- **CXX** = spack/env/spack-c++
- **F77** = spack/env/spack-f77
- **FC** = spack/env/spack-f90
- **PKG_CONFIG_PATH** = ...
- **PATH** = spack/env:$PATH
- **CMAKE_PREFIX_PATH** = ...
- **LIBRARY_PATH** = ...

**install()**

### Fork

**do_install()**

Install dep1 → Install dep2 → ... → Install package

**Compiler wrappers**

- **icc**
- **icpc**
- **ifort**

```
- I /dep1-prefix/include
- L /dep1-prefix/lib
- Wl,-rpath=/dep1-prefix/lib
```

**configure** → **make** → **make install**
Some advanced features
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.

```
+ external-blas

spec['blas'].libs
spec['lapack'].libs
```

```
~ external-blas

spec['blas'].libs
spec['lapack'].libs
```
What’s New?
What’s on the Road Map?
We are working to enable optimized software distribution for HPC

• Distribution effort required is similar to efforts like Red Hat, Debian, Ubuntu, etc.
  – Curation and vetting of software
  – Packaging, building
  – Wide distribution

• HPC community is not as mainstream, not as widespread as these distributions

• HPC platform complexity poses challenges
  – Many (often unique) platforms
  – Many software ecosystems
  – From-source distribution
  – Must support Optimization, GPUs, fast networks

• Much more automation is required to practically support our ecosystem!
Our strategy is to enable exascale software distribution on both bare metal and containers

- New capabilities to make HPC packaging easy and automated
  - Optimized builds and package binaries that exploit the hardware
  - Workflow automation for facilities, developers, and users
  - Strong integration with containers as well as bare metal deployments

- Work with ECP and other partners to harden packages
  - Build pipelines at facilities
  - Coordination on multi-site testing
  - Security integration

- Outreach to users
  - Tutorials, workshops, BOFs

github.com/spack/spack
We have developed a cross-platform library to detect and compare microarchitecture metadata
   - Detects based on /proc/cpuinfo (Linux), sysctl (Mac)
   - Allows comparisons for compatibility, e.g.:

```
  skylake > broadwell
  zen2 > x86_64
```

Key features:
   - Know which compilers support which chips/which flags
   - Determine compatibility
   - Enable creation and reuse of optimized binary packages
   - Easily query available architecture features for portable build recipes

We will be extracting this as a standalone library for other tools & languages
   - Hope to make this standard!

```
$ spack arch --known-targets
Generic architectures (families)
  aarch64  ppc64  ppc64le  x86  x86.64
IBM - ppc64
  power7  power8  power9
IBM - ppc64le
  power8le  power9le
AuthenticAMD - x86_64
  barcelonaintel  bulldozer  piledriver  steamroller  excavator  zen  zen2
GenuineIntel - x86_64
  nocona  westmere  haswell  mic_knl  castellated  core2  sandybridge  broadwell  skylake_avx512  icelake  nehalem ivybridge  skylake  cannonlake
GenuineIntel - x86
  i686  pentium2  pentium3  pentium4  prescott
```

```
$ spack install lbann target=cascadelake
$ spack install petsc target=zen2
```

```
class OpenBlas(Package):
    def configure_args(self, spec, env):
        args = []
        if 'avx512' in spec.target:
            args.append('--with-avx512')
        ...
        return args
```

Extensive microarchitecture knowledge

Simple feature query

Specialized installations

Spack now understands specific target microarchitectures
Spack environments enable users to build customized stacks from an abstract description

- 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
```
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)
```
{
  "concrete_specs": {
    "6s63so2kstp3zyvjezglr": {
      "hdf5": {
        "version": "1.10",
        "arch": {
          "platform":
        }
      }
    }
  }
```

github.com/spack/spack
We have simplified container deployments using Spack Environments

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

```bash
FROM spack/centos:7
WORKDIR /build
COPY spack.yaml .
RUN spack install

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

Build with `docker build`.

Run with Singularity (or some other tool)

Base image with Spack in PATH

Copy in `spack.yaml`

Then run `spack install`

List of packages to install, with constraints

[GitHub link](https://github.com/spack/spack)
We have developed Spack stacks: combinatorial environments for entire facility deployments

- Allow users to easily express a huge cross-product of specs
  - All the packages needed for a facility
  - Generate modules tailored to the site
  - Generate a directory layout to browse the packages

- Build on the environments workflow
  - Manifest + lockfile
  - Lockfile enables reproducibility

- Relocatable binaries allow the same binary to be used in a stack, regular install, or container build.
  - Difference is how the user interacts with the stack
  - Single-PATH stack vs. modules.

```spack
spack:
definitions:
  compilers:
    [gcc@5.4.0, clang@3.8, intel@18.0.0]
  mpi:
    [mvapich2@2.2, ^mvapich2@2.3, ^openmpi@3.1.3]
packages:
  - nalu
  - hdf5
  - hypre
  - trilinos
  - petsc
  - ...
specs:
  # cartesian product of the lists above
  matrix:
    - [$packages]
    - [$compilers]
    - [$mpis]
modules:
  lmod:
    core_compilers: [gcc@5.4.0]
    hierarchy: [mpi, lapack]
    hash_length: 0
```

[github.com/spack/spack](https://github.com/spack/spack)
We have been heavily involved in the ECP CI project.

- We have added security features to the open source GitLab product.
  - Integration with center identity management
  - Integration with schedulers like SLURM, LSF

- We are democratizing testing at Livermore Computing
  - Users can run tests across 30+ machines by editing a file
  - Previously, each team had to administer own servers

- ECP sites are deploying GitLab CI for users
  - All HPC centers can leverage these improvements
  - NNSA labs plan to deploy common high-side CI infrastructure
  - We are developing new security policies to allow external open source code to be tested safely on key machines

User commits to GitLab

GitLab test runners are now integrated with HPC machines
Spack has added GitLab CI integration to automate package build pipelines

- Builds on Spack environments
  - Support auto-generating GitLab CI jobs
  - Can run in a Kube cluster or on bare metal runners at an HPC site
  - Sends progress to CDash
ECP is working towards a periodic, hierarchical release process

- ECP teams work to ensure that libraries and components work together
  - Historically, HPC codes used very few dependencies

- Now, groups of teams work together on small releases of “Software Development Kits”

- SDKs are rolled into a larger, periodic release.
Automated builds using ECP CI will enable a robust, widely available HPC software ecosystem.

With pipeline efforts at E6 labs, users will no longer need to build their own software for high performance.

github.com/spack/spack
Spack focus areas in FY20

- Multi-stage container generation with Spack
  - Add support to Spack to generate *multi-stage* container builds that exclude build dependencies from artifacts automatically

- Build Hardening with Spack Pipelines
  - Continue working with E4S team to harden container builds

- Parallel builds
  - “srun spack install” will use the entire allocation to build

- New concretizer based on fast ASP/SAT solvers

- Improved dependency models for compilers
  - icpc depends on g++ for its libstdc++, and other ABI nightmares

github.com/spack/spack
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 Slack channel (see GitHub repo for info)
  - Join our Google Group (see GitHub repo for info)
  - Submit GitHub issues and pull requests!

- Docs and a full day tutorial are available at: spack.readthedocs.io