Managing HPC Software Complexity with Spack

SC22 Full Day Tutorial
November 13, 2022

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

Find these slides and associated scripts here:

spack-tutorial.rtfd.io

We also have a chat room on Spack slack. You can join here:

slack.spack.io

Join the #tutorial channel!

You can ask questions here after the conference is over. Over 2,000 people can help you on Slack!
Tutorial Presenters

Greg Becker
LLNL

Adam Stewart
UIUC

Todd Gamblin
LLNL
# Agenda (approximate)

## Morning
- **Intro**: 8:30 am
- **Basics**: 8:45 am
- **Concepts**: 9:30 am
- **Break**: 10:00 am
- **Environments**: 10:30 am
- **Configuration**: 11:15 am
- **Lunch**: 12:00 pm

## Afternoon
- **Stacks**: 1:30 pm
- **Packaging**: 2:00 pm
- **Break**: 3:00 pm
- **Developer Workflows**: 3:30 pm
- **Mirrors & Binary Caches**: 4:00 pm
- **Scripting**: 4:20 pm
- **Roadmap / Questions**: 4:35 pm
- **End**: 5:00 pm
Modern scientific codes rely on icebergs of dependency libraries

MFEM: Higher-order finite elements
31 packages, 69 dependencies

LBANN: Neural Nets for HPC
71 packages 188 dependencies

r-condop: R Genome Data Analysis Tools
179 packages, 527 dependencies
ECP’s E4S stack is even larger than these codes

- Red boxes are the packages in it (about 100)
- Blue boxes are what *else* you need to build it (about 600)
- It’s infeasible to build and integrate all of this manually
Some fairly common (but questionable) assumptions made by package managers (conda, pip, apt, etc.)

- **1:1 relationship between source code and binary (per platform)**
  - Good for reproducibility (e.g., Debian)
  - Bad for performance optimization

- **Binaries should be as portable as possible**
  - What most distributions do
  - Again, bad for performance

- **Toolchain is the same across the ecosystem**
  - One compiler, one set of runtime libraries
  - Or, no compiler (for interpreted languages)

Outside these boundaries, users are typically on their own
High Performance Computing (HPC) violates many of these assumptions

- **Code is typically distributed as source**
  - With exception of vendor libraries, compilers

- **Often build many variants of the same package**
  - Developers’ builds may be very different
  - Many first-time builds when machines are new

- **Code is optimized for the processor and GPU**
  - Must make effective use of the hardware
  - Can make 10-100x perf difference

- **Rely heavily on system packages**
  - Need to use optimized libraries that come with machines
  - Need to use host GPU libraries and network

- **Multi-language**
  - C, C++, Fortran, Python, others
  all in the same ecosystem

Some Supercomputers

- **Current**
  - Summit
    - Oak Ridge National Lab
    - Power9 / NVIDIA
  - Fugaku
    - RIKEN
    - Fujitsu/ARM a64fx

- **Upcoming**
  - Perlmutter
    - Lawrence Berkeley National Lab
    - AMD Zen / NVIDIA
  - Aurora
    - Argonne National Lab
    - Intel Xeon / Xe
  - Frontier
    - Lawrence Livermore National Lab
    - AMD Zen / Radeon
  - El Capitan
    - Lawrence Livermore National Lab
    - AMD Zen / Radeon

Join #tutorial on Slack: slack.spack.io
Materials: spack-tutorial.readthedocs.io
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 enables software distribution for HPC

- Spack automates the build and installation of scientific software
- Packages are parameterized, so that users can easily tweak and tune configuration

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
$ spack install hdf5@1.10.5 cppflags="-O3 -g3"
$ spack install hdf5@1.10.5 target=haswell
$ spack install hdf5@1.10.5 +mpi ^mpich@3.2
```

- Ease of use of mainstream tools, with flexibility needed for HPC
- In addition to CLI, Spack also:
  - Generates (but does not require) modules
  - Allows conda/virtualenv-like environments
  - Provides many devops features (CI, container generation, more)
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
- May drive 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

---

Join *tutorial* on Slack: slack.spack.io  
Materials: spack-tutorial.readthedocs.io
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 sustains the HPC software ecosystem with the help of its many contributors.

Over 6,700 software packages
Over 1,100 contributors

Most package contributions are not from DOE
But they help sustain the DOE ecosystem!

Nearly 6,000 monthly active users
(per documentation site)

Materials: spack-tutorial.readthedocs.io
Spack is critical for ECP’s mission to create a robust, capable exascale software ecosystem.

- Spack will be used to build software for the three upcoming U.S. exascale systems
- ECP has built the Extreme Scale Scientific Software Stack (E4S) with Spack – more at https://e4s.io
- Spack will be integral to upcoming ECP testing efforts.

https://e4s.io

Spack is the most depended-upon project in ECP
One month of Spack development is pretty busy!

October 12, 2021 – November 12, 2021

Overview

- 671 Active Pull Requests
- 145 Active Issues
- 536 Merged Pull Requests
- 135 Open Pull Requests
- 75 Closed Issues
- 70 New Issues

Excluding merges, 173 authors have pushed 571 commits to develop and 634 commits to all branches. On develop, 703 files have changed and there have been 20,730 additions and 3,807 deletions.

- 1 Release published by 1 person

v0.17.0
published 7 days ago

- 536 Pull requests merged by 151 people

Join #tutorial on Slack: slack.spack.io
Materials: spack-tutorial.readthedocs.io
Spack’s widespread adoption has drawn contributions and collaborations with many vendors

- **AWS** invests significantly in cloud credits for Spack build farm
  - Joint Spack tutorial with AWS had 125+ participants
  - Joint AWS/AHUG Spack Hackathon drew 60+ participants

- **AMD** has contributed ROCm packages and compiler support
  - 55+ PRs mostly from AMD, also others
  - ROCm, HIP, aocc packages are all in Spack now

- **HPE/Cray** is doing internal CI for Spack packages, in the Cray environment

- **Intel** contributing OneApi support and licenses for our build farm

- **NVIDIA** contributing NVHPC compiler support and other features

- **Fujitsu and RIKEN** have contributed a huge number of packages for ARM/a64fx support on Fugaku

- **ARM and Linaro** members contributing ARM support
  - 400+ pull requests for ARM support from various companies

Join #tutorial on Slack: slack.spack.io  
Materials: spack-tutorial.readthedocs.io
Spack v0.19.0 was released last week!

- Major new features:
  1. Package requirements
  2. Environment UI improvements
  3. Packages with multiple build systems
  4. Compiler/variant propagation
  5. Enhanced git versions
  6. Better Cray EX Support
  7. Testing and CI improvements
  8. Experimental binding link model

*Bold items covered in today’s tutorial*

Full release notes:
https://github.com/spack/spack/releases/tag/v0.19.0
Spack is not the only tool that automates builds

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

2. Build-from-source Package Managers
   - Homebrew, LinuxBrew
   - MacPorts
   - Gentoo

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 has unoptimized binaries

Join #tutorial on Slack: slack.spack.io  Materials: pack-tutorial.readthedocs.io
Hands-on Time: Spack Basics

Follow script at [spack-tutorial.readthedocs.io](http://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 provides a spec syntax to describe customized package 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

```bash
$ spack install mpirleaks            unconstrained
$ spack install mpirleaks@3.3       @ custom version
$ spack install mpirleaks@3.3 %gcc@4.7.3 % custom compiler
$ spack install mpirleaks@3.3 %gcc@4.7.3 +threads +/- build option
$ spack install mpirleaks@3.3 cppflags="-O3 -g3" set compiler flags
$ spack install mpirleaks@3.3 target=cascadelake set target microarchitecture
$ spack install mpirleaks@3.3 ^mpich@3.2 %gcc@4.9.3 ^ dependency constraints
```
Spack packages are parameterized using the spec syntax.

Python DSL defines many ways to build:

```python
from spack import *

class Kripke(CMakePackage):
    
    """Kripke is a simple, scalable, 3D Sn deterministic particle transport 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='3ff72eef0d1ba582578d626741eb0b3f026a096048d7ec4794d2a7dfbe2b8a6')
    version('1.2.2', sha256='eaf9ddf562416974157b34d00c3a1c880fc5296fcede2aa2efa039a8be0976f3a3')
    version('1.1', sha256='232d74072fc7b848fa2adc0a1bc839ae0fb5f90d50224186601f5555a25f64a')

    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):
        mkdirp(prefix.bin)
        install('..../spack-build/kripke', prefix.bin)
```

One package .py file per software project!
Conditional variants simplify packages

CudaPackage: a mix-in for packages that use CUDA

```
class CudaPackage(PackageBase):
    variant('cuda', default=False,
        description='Build with CUDA')

    variant('cuda_arch',
        description='CUDA architecture',
        values=any_combination_of(cuda_arch_values),
        when='+cuda')

depends_on('cuda', when='+cuda')

depends_on('cuda@9.0:', when='cuda_arch=70')
depends_on('cuda@9.0:', when='cuda_arch=72')
depends_on('cuda@10.0:', when='cuda_arch=75')

conflicts('gcc@9:', when='+cuda ^cuda@:10.2.89 target=x86_64:')
conflicts('gcc@9:', when='+cuda ^cuda@:10.1.243 target=ppc64le:')
```

cuda is a variant (build option)
cuda_arch is only present if cuda is enabled
dependency on cuda, but only if cuda is enabled
constraints on cuda version
compiler support for x86_64 and ppc64le

There is a lot of expressive power in the Spack package DSL.

Join #tutorial on Slack: spackpm.herokuapp.com
Materials: spack-tutorial.readthedocs.io
Spack Specs can constrain versions of dependencies

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

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

```sh
$ 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.

**mpileaks**

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

User input: *abstract* spec with some constraints

---

### Abstract, normalized spec with some dependencies.

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

---

### Concretize

- **mpileaks**
  - `^callpath@1.0`
  - `^libelf@0.8.11`

- **mpich@3.0.4**
  - `%gcc@4.7.3`
  - `=linux-ppc64`

- **dyninst@8.1.2**
  - `%gcc@4.7.3`
  - `=linux-ppc64`

---

### Store

```
spec.yaml
```

```yaml
spec:
  - mpileaks:
    arch: linux-x86_64
    compiler:
      name: gcc
      version: 4.9.2
    dependencies:
      - adept-utils: kszrtkpbzac332ilxjkkc0laidmnpv4
      - callpath: bah5f44d0474nq5ysjej2tmmirvuy7
      - mpich: os4ar6lifj23jyjndabdpfejclj17tc3
    hash: 33htjx7p766gznspgyes3ghguyrujgh
    variants: {} version: "1.0"
  - mpich:
    arch: linux-x86_64
    compiler:
      name: gcc
      version: 4.9.2
    dependencies:
      - adept-utils: kszrtkpbzac332ilxjkkc0laidmnpv4
    hash: kszrtkpbzac332ilxjkkc0laidmnpv4
    variants: {} version: "1.0.2"
  - boost:
    arch: linux-x86_64
    compiler:
      name: gcc
      version: 4.9.2
    dependencies: {}
    hash: teesjv74hee5ksspmj5dk43o7qownlq
    variants: {} version: "1.59.0"
...
```

Detailed provenance is stored with the installed package.
Hashing allows us to handle combinatorial complexity

**Dependency DAG**

- mpileaks
- mpi
- callpath
- dyninst
- libdwarf
- libelf

**Installation Layout**

```
opt
  spack
    linux-rhel7-skylake
      gcc-8.3.0
        mpileaks-1.0-hc4sm4vuzpm4znmvrfzri4ow2mkpche2e
        callpath-1.0.4-daqqppssxb6qbfrztsezkmhus3xoflbsy
        openmpi-4.1.4-u64v261gxxyn23hysmklfums6tgjv5r
        dyninst-12.1.0-u64v261gxxyn23hysmklfums6tgjv5r
        libdwarf-20180129-u5eawkvaoc7vonabe6ndkcfwuv233cj
        libelf-0.8.13-x46q4wm46ay4pltriijbgizxjrhbaka6
```

- 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
An isolated compilation environment allows Spack to easily swap compilers

- 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)

### Compiler wrappers

(spack-cc, spack-c++, spack-f77, spack-f90)

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

---

**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 = ...
  - CMAKE_PREFIX_PATH = ...
  - LIBRARY_PATH = ...
  - PATH = spack/env:$PATH

- do_install()

- Install dep1
- Install dep2
- ... Install package

- Fork

- Build

  - cc
  - icpc
  - ifort

  - configure
  - make
  - make install
We can configure Spack to build with external software

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

packages.yml

```
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).

Spack prunes the DAG when adding external packages.
Spack package repositories allow stacks to be layered

LLNL MARBL multi-physics application

$ 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
The concretizer includes information from packages, configuration, and CLI

- new versions
- new dependencies
- new constraints

Contributors

- spack developers
- admins, users
- users

Dependecy solving is NP-hard

Concrete spec is fully constrained and can be built.

```bash
spack install hdf5@1.12.0 +debug
```
We use logic programming to simplify package solving

- New concretizer leverages Clingo (see potassco.org)

- Clingo is an Answer Set Programming (ASP) solver
  - ASP looks like Prolog; leverages SAT solvers for speed/correctness
  - ASP program has 2 parts:
    1. Large list of facts generated from our package repositories and config
    2. Small logic program (~800 lines)
       - includes constraints and optimization criteria

- New algorithm on the Spack side is conceptually simpler:
  - Generate facts for all possible dependencies, send to logic program
  - Optimization criteria express preferences more clearly
  - Build a DAG from the results

- New concretizer solves many specs that old concretizer can’t
  - Backtracking is a huge win – many issues resolved
  - Conditional logic that was complicated before is now much easier

Some facts for the HDF5 package
--fresh only reuses builds if hashes match

- Hash matches are very sensitive to small changes
- In many cases, a satisfying cached or already installed spec can be missed
- Nix, Spack, Guix, Conan, and others reuse this way

1. Resolve metadata
2. Create per-node hashes
3. Query for exact hash match
--reuse (now the default) is more aggressive

- --reuse tells the solver about all the installed packages!
- Add constraints for all installed packages, with their hash as the associated ID:

```python
installed_hash("openssl", "lwatuysmwhuhrnycvyn77icdhs6mn").
imposed_constraint("lwatuysmwhuhrnycvyn77icdhs6mn", "node", "openssl").
imposed_constraint("lwatuysmwhuhrnycvyn77icdhs6mn", "version", "openssl", "1.1.1g").
imposed_constraint("lwatuysmwhuhrnycvyn77icdhs6mn", "node_platform_set", "openssl", "darwin").
imposed_constraint("lwatuysmwhuhrnycvyn77icdhs6mn", "node_os_set", "openssl", "catalina").
imposed_constraint("lwatuysmwhuhrnycvyn77icdhs6mn", "node_target_set", "openssl", "x86_64").
imposed_constraint("lwatuysmwhuhrnycvyn77icdhs6mn", "variant_set", "openssl", "systemcerts", "True").
imposed_constraint("lwatuysmwhuhrnycvyn77icdhs6mn", "node_compiler_set", "openssl", "apple-clang").
imposed_constraint("lwatuysmwhuhrnycvyn77icdhs6mn", "node_compiler_version_set", "openssl", "apple-clang", "12.0.0").
imposed_constraint("lwatuysmwhuhrnycvyn77icdhs6mn", "node_os_set", "openssl", "apple-clang").
imposed_constraint("lwatuysmwhuhrnycvyn77icdhs6mn", "depends_on", "openssl", "zlib", "build").
imposed_constraint("lwatuysmwhuhrnycvyn77icdhs6mn", "depends_on", "openssl", "zlib", "link").
imposed_constraint("lwatuysmwhuhrnycvyn77icdhs6mn", "hash", "zlib", "x2anksgssxsxa7pcnhzg5k3dhgacglze").
```
Telling the solver to minimize builds is surprisingly simple in ASP

1. Allow the solver to *choose* a hash for any package:

   ```asp
   { hash(Package, Hash) : installed_hash(Package, Hash) } 1 :- node(Package).
   ```

2. Choosing a hash means we impose its constraints:

   ```asp
   impose(Hash) :- hash(Package, Hash).
   ```

3. Define a build as something *without* a hash:

   ```asp
   build(Package) :- not hash(Package, _), node(Package).
   ```

4. Minimize builds!

   ```asp
   #minimize { 1@100,Package : build(Package) }.
   ```
With and without --reuse optimization

Pure hash-based reuse: all misses

With reuse: 16 packages were reusable
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+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 environments enable users to build customized stacks from an abstract description

- `spack.yaml` file describes project requirements
- `spack.lock` describes exactly what versions/configurations were installed, allows them to be reproduced.
- Can be used to maintain configuration of a software stack.
  - Can easily version an environment in a repository

---

**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": {
    "6s63ss02kstpszyvjezglndmavy613nul": {
      "hdf5": {
        "version": "1.10.5",
        "arch": {
          "platform": "darwin",
          "platform_os": "mojave",
          "target": "x86_64"
        }
      },
      "compiler": {
        "name": "clang",
        "version": "10.0.0-apple"
      },
      "namespace": "built",
      "parameters":
    }
  }
}
We’ll resume at: 10:30am

Find the slides and associated scripts here:

spack-tutorial.readthedocs.io

Remember to join Spack slack so you can get help after ISC!

slack.spack.io

Join the #tutorial channel!
Environments, spack.yaml and spack.lock

Follow script at spack-tutorial.readthedocs.io
Hands-on Time: Configuration

Follow script at spack-tutorial.readthedocs.io
We’ll resume at: 1:30pm

Find the slides and associated scripts here: spack-tutorial.readthedocs.io

Remember to join Spack slack so you can get help after ISC! slack.spack.io

Join the #tutorial channel!
Hands-on Time: Stacks

Follow script at spack-tutorial.readthedocs.io
Hands-on Time: Creating Packages

Follow script at spack-tutorial.readthedocs.io
Find the slides and associated scripts here:

spack-tutorial.readthedocs.io

Remember to join Spack slack so you can get help after ISC!

slack.spack.io

Join the #tutorial channel!
Hands-on Time: Developer Workflows

Follow script at spack-tutorial.readthedocs.io
Hands-on Time: Scripting

Follow script at spack-tutorial.readthedocs.io
Hands-on Time: Mirrors and Build Caches

Follow script at spack-tutorial.readthedocs.io
More Features and the Road Ahead
Environments have enabled us to add build many features to support developer workflows

**spack external find**
Automatically find and configure external packages on the system

**spack test**
Packages know how to run their own test suites

**spack ci**
Automatically generate parallel build pipelines (more on this later)

**spack containerize**
Turn environments into container build recipes

Join #tutorial on Slack: slack.spack.io
Materials: pack-tutorial.readthedocs.io
Spack environments are the foundation of Spack CI

- `spack ci` enables any environment to be turned into a build pipeline
- Pipeline generates a `.gitlab-ci.yml` file from `spack.lock`
- Pipelines can be used just to build, or to generate relocatable binary packages
  - Binary packages can be used to keep the same build from running twice
- Same repository used for `spack.yaml` can generate pipelines for project
We are building a supply chain for HPC

Spack Contributions on GitHub

gitlab.spack.io

spack ci

spack.yaml configurations (E4S, SDKs, others)

GitLab CI builds (changed) packages
- On every pull request
- On every release branch

- New security model supports untrusted contributions from forks
  - Sandboxed build caches for test builds; Authoritative builds on mainline only after approved merge

This CI has greatly increased reliability of builds for users
Spack’s model lowers the maintenance burden of optimized software stacks

Traditional OS package manager

Recipe per package configuration
(need rewrites for new systems)

Build farm

Portable (unoptimized) x86_64 binaries

One software stack upgraded over time

Spack

Parameterized recipe per package
(Same recipe evolves for all targets)

Build farm / CI

Optimized Graviton2 binaries

Optimized Skylake binaries

Optimized GPU binaries

Many software stacks

Built for specific:
- Systems
- Compilers
- OS’s
- MPIs
- etc.

Users/developers can also build directly from source

Join #tutorial on Slack: slack.spack.io
Materials: spack-tutorial.readthedocs.io
We started providing public binaries in June 2022

# latest v0.18.x release binaries
spack mirror add https://binaries.spack.io/releases/v0.18

# rolling release: bleeding edge binaries
spack mirror add https://binaries.spack.io/develop

- Over 3,000 builds in the cache so far:
  - Amazon Linux 2 x86_64_v4
  - Amazon Linux 2 aarch64
  - Amazon Linux 2 graviton2
  - Ubuntu 18.04 x86_64

- Expect this list to expand!
Our infrastructure enables us to sustainably manage a binary distro

Separate, untrusted S3 buckets

Per-PR build caches
- github/pr-28468
- github/pr-28469
- ...

Public, signed binaries in CloudFront distribution
- https://binaries.spack.io
- develop
- releases/v0.18
- ...

Contributors submit package changes
- Iterate on builds in PR
- Caches prevent unnecessary rebuilds

Maintainers review PRs
- Verify PR build succeeded
- Review package code
- Merge to develop

Rebuild and Sign
- Published binaries built ONLY from approved code
- Protected signing runners
- Ephemeral keys

- Moves bulk of binary maintenance upstream, onto PRs
  - Production binaries never reuse binaries from untrusted environment

Join #tutorial on Slack: slack.spack.io
Materials: spack-tutorial.readthedocs.io
Spack v0.20 roadmap: Separate concretization of build dependencies

- We want to:
  - Build build dependencies with the "easy" compilers
  - Build rest of DAG (the link/run dependencies) with the fancy compiler

- 2 approaches to modify concretization:
  1. Separate solves
     - Solve run and link dependencies first
     - Solve for build dependencies separately
     - May restrict possible solutions (build <-> run env constraints)
  2. Separate models
     - Allow a bigger space of packages in the solve
     - Solve all runtime environments together
     - May explode (even more) combinatorially
Spack 0.20 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
Spack’s long-term strategy is based around broad adoption and collaboration

- **Not sustainable without a community**
  - Broad adoption incentivizes contributors
  - Cloud resources and automation absolutely necessary

- **Spack preserves build knowledge in a cross-platform, reusable way**
  - Minimize rewriting recipes when porting

- **CI ensures builds continue to work as packages evolve**
  - Keep packages flexible but verify key configurations

- **Growing contributor base and continuing to automate are the most important priorities**
  - 377 contributors to 0.18 release!
When would we go 1.0?

- Big things we’ve wanted for 1.0 are:
  - New concretizer
  - production CI
  - production public build cache
  - Compilers as dependencies
  - Stable package API
    - Enables separate package repository

- After 0.19 we will hopefully have all of these
  - Maybe there won’t be a 0.20!
An active-learning-based approach for identifying high-fidelity package build configurations

- Iterative sampling method using only a limited set of samples. Suitable when the true objective function evaluations are expensive

- Surrogate model is used to compute the value of the objective for a configuration

Reliabuild iterates between fitting model and using it to select samples

*Reliabuild: Searching for High-Fidelity Builds Using Active Learning; H. Menon, K. Parasyris, T. Scogland, T. Gamblin; MSR'2022*
Reliabuild has significantly higher precision than *Random* selection.
### Package Importance Analysis

<table>
<thead>
<tr>
<th>Root package</th>
<th>Dependency ranking</th>
</tr>
</thead>
<tbody>
<tr>
<td>abyss</td>
<td>autoconf: 0.37</td>
</tr>
<tr>
<td></td>
<td>autoconf+m4: 0.37</td>
</tr>
<tr>
<td>adios</td>
<td>autoconf+perl: 0.27</td>
</tr>
<tr>
<td></td>
<td>autoconf+m4: 0.27</td>
</tr>
<tr>
<td>ascent</td>
<td>vtk-h+openmpi: 0.14</td>
</tr>
<tr>
<td></td>
<td>vtk-h: 0.14</td>
</tr>
<tr>
<td>axom</td>
<td>lua: 0.08</td>
</tr>
<tr>
<td></td>
<td>lua+n curses: 0.08</td>
</tr>
<tr>
<td>bolt</td>
<td>autoconf+perl: 0.37</td>
</tr>
<tr>
<td></td>
<td>autoconf+m4: 0.37</td>
</tr>
<tr>
<td>hypre</td>
<td>openblas+perl: 0.07</td>
</tr>
<tr>
<td></td>
<td>openblas: 0.07</td>
</tr>
<tr>
<td>hpx</td>
<td>hpx+boost: 0.24</td>
</tr>
<tr>
<td></td>
<td>hpx+hwloc: 0.24</td>
</tr>
<tr>
<td>heffe</td>
<td>heffe: 0.35</td>
</tr>
<tr>
<td></td>
<td>heffe+openmpi: 0.30</td>
</tr>
<tr>
<td>hdf5</td>
<td>mpich+findutils: 0.03</td>
</tr>
<tr>
<td></td>
<td>mpich+pkgconf: 0.03</td>
</tr>
<tr>
<td>ninja</td>
<td>ninja+python: 0.03</td>
</tr>
<tr>
<td></td>
<td>python+n curses: 0.01</td>
</tr>
<tr>
<td>omega-h</td>
<td>omega-h+zlib: 0.24</td>
</tr>
<tr>
<td>openmpi</td>
<td>json-c: 0.30</td>
</tr>
<tr>
<td>openpmd-api</td>
<td>hdfs: 0.19</td>
</tr>
<tr>
<td>papyrus</td>
<td>papyrus+mpich: 0.11</td>
</tr>
<tr>
<td></td>
<td>cmake+n curses: 0.08</td>
</tr>
<tr>
<td>plasma</td>
<td>plasma: 0.52</td>
</tr>
<tr>
<td></td>
<td>plasma+openblas: 0.26</td>
</tr>
</tbody>
</table>

- A particular choice of version for packages can significantly affect the build outcome.
- Importance metric: We use Jensen-Shannon (JS) divergence to compute the difference between the good and bad distribution.
- Some packages impact the build outcome more than others.

Relative ranking of dependencies based on importance can guide the exploration process.
Pairwise Version Constraints Analysis
Join us for more SC22 events!

• Wed Nov 16:
  – 2 Spack-related papers:
    • Mapping out the HPC Dependency Chaos 1:30pm
    • Using Answer Set Programming for HPC Dependency Solving 2:00pm
  – Spack Community BOF 5:15pm

We’ll be giving away more T-shirts at the BOF! Tell your friends!
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!