Managing HPC Software Complexity with Spack

PEARC25 Tutorial
June 21, 2025

Todd Gamblin, Kathleen Shea, Caetano Melone, and Alec Scott

THE LINUX FOUNDATION

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

Todd Gamblin
LLNL

Alec Scott
LLNL

Kathleen Shea
LLNL

Caetano Melone
LLNL

Tutorial Materials

Find these slides and associated scripts here:

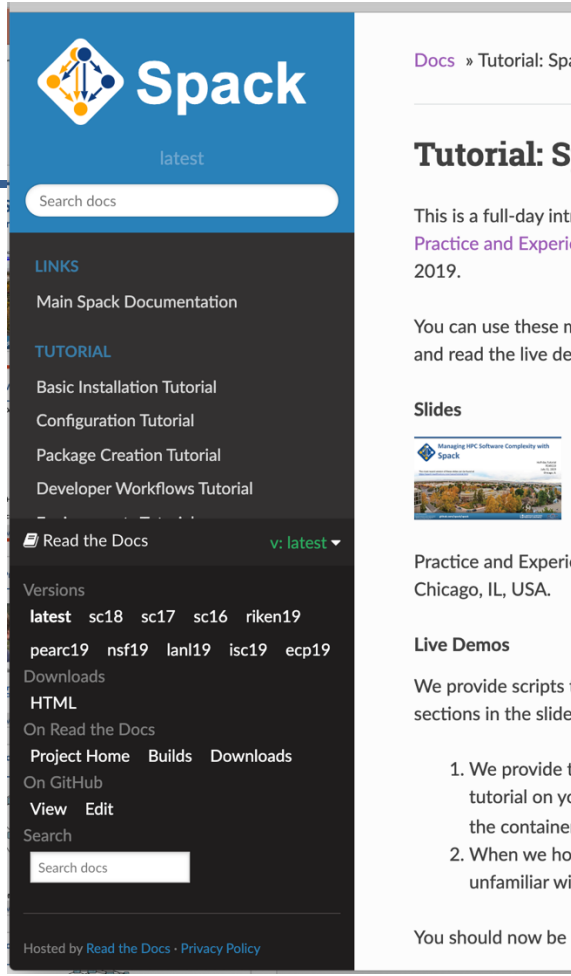
spack-tutorial.rtfd.io

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 **3,400 people** can help you on Slack!



Claim a VM instance at: bit.ly/spack-vms

	A	B	C	D	E
1	**Spack Tutorial VM Instances**				
2	Instructions:	1. Put your name in a box below to claim an account on a VM insta			
3		2. Log in to your VM:			
4		ssh <IP address>			
5		Login/password are both the username from your column below			
6					
7		**Login / Password**			
8	**IP Address**	spack1	spack2	spack3	spack4
9	35.90.43.21				
10	35.91.36.120		Your Name		
11	34.217.149.171				

If you're in the **spack2** column, your login and password are *both* **spack2**

ssh spack2@35.91.36.120

Claim a login by putting your name in the Google Sheet

Agenda (approximate)

Morning


Intro 9:00 am


Basics


Concepts

Break 10:30 am


Environments 11:00 am


Configuration

Lunch 12:30 pm

Afternoon


Software Stacks 1:30 pm


Packaging

Break 3:00 pm


Developer Workflows 3:30 pm


Mirrors & Binary Caches


Scripting

End 5:00 pm

We build codes from hundreds of small, complex pieces

> *Just when we're starting to solve the problem of how to create software using reusable parts, it founders on the nuts-and-bolts problems outside the software itself.*
>
> P. DuBois & T. Epperly. ***Why Johnny Can't Build***. Scientific Programming. Sep/Oct 2003.

- Component-based software development dates back to the 60's
  - M.D. McIlroy, *Mass Produced Software Components*. NATO SE Conf., 1968
- **Pros are well known:**
  - Teams can and must reuse each others' work
  - Teams write less code, meet deliverables faster
- **Cons:**
  - Teams must ensure that components work together
  - Integration burden increases with each additional library
  - Integration must be repeated with each update to components
  - **Components must be vetted!**
- **Managing changes over time is becoming intractable**

Build-time incompatibility; fail fast

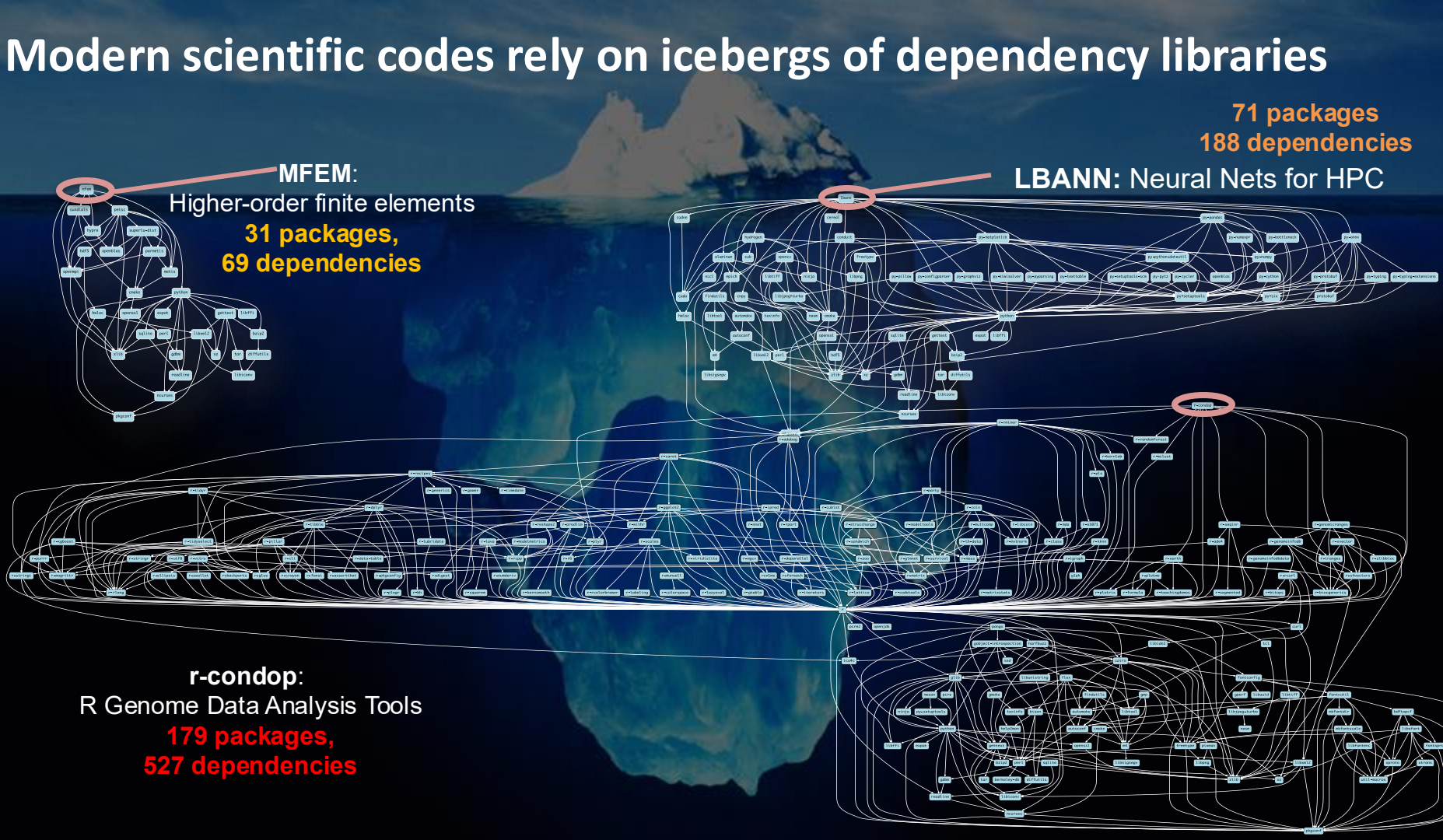
Appears to work; subtle errors later

Modern scientific codes rely on icebergs of dependency libraries

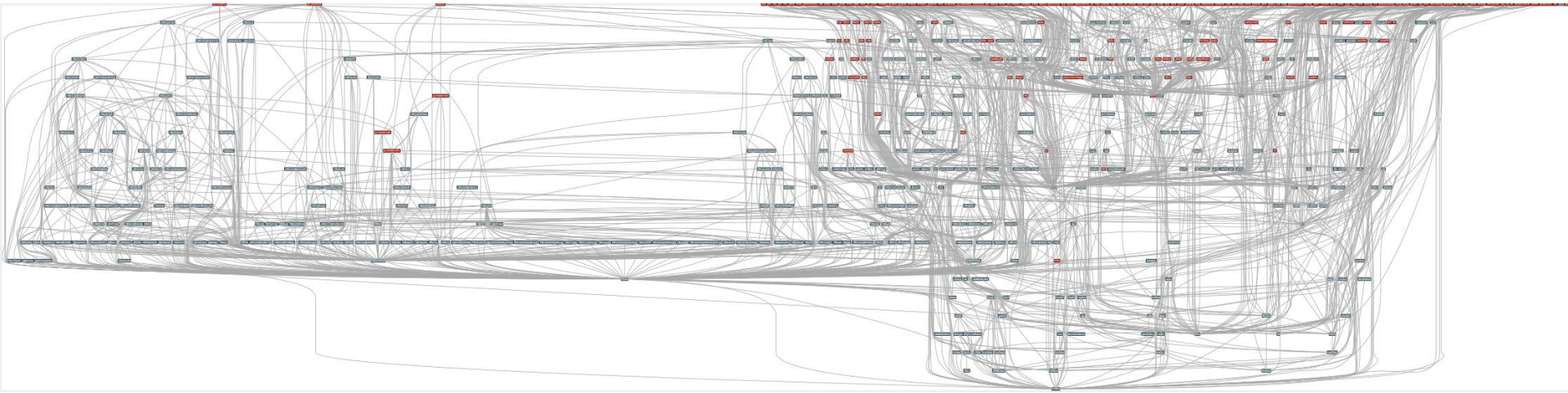
MFEM:
Higher-order finite elements
31 packages,
69 dependencies

71 packages
188 dependencies
LBANN: Neural Nets for HPC

r-condop:
R Genome Data Analysis Tools
179 packages,
527 dependencies

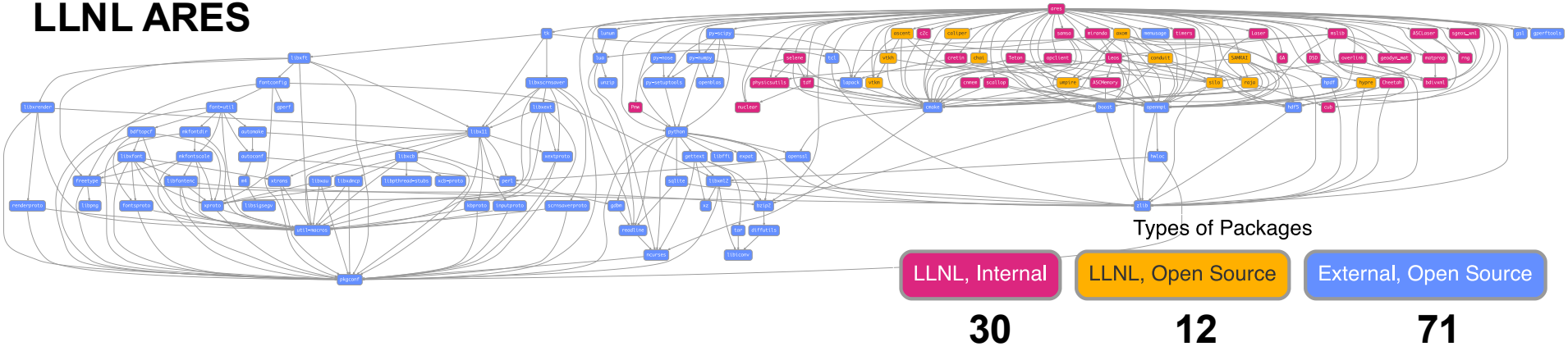


ECP's E4S stack is even larger than these codes



- 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

Modern software integrates open source and internal packages



- Most modern software uses *tons* of open source
- We *cannot* replace all these OSS components with our own
  - How do we put them all together effectively?
  - Do you *have* to integrate this stuff by hand?

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

Oak Ridge National Lab
Power9 / NVIDIA

RIKEN
Fujitsu/ARM a64fx

Lawrence Berkeley National Lab
AMD Zen / NVIDIA

Argonne National Lab
Intel Xeon / Xe

Oak Ridge National Lab
AMD Zen / Radeon

Lawrence Livermore National Lab
AMD Zen / Radeon

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

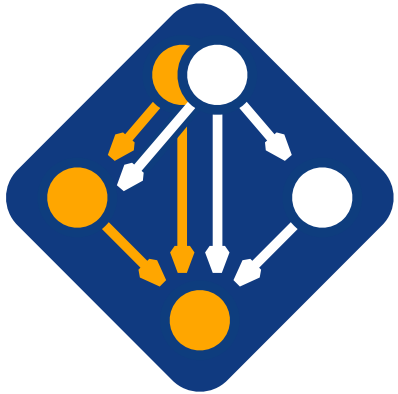
- Spack automates the build and installation of scientific software
- Packages are *parameterized,* so that users can easily tweak and tune configuration

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 cppflags="-O3 –g3"
$ spack install hdf5@1.10.5 %clang@6.0         $ spack install hdf5@1.10.5 target=haswell
$ spack install hdf5@1.10.5 +threadssafe       $ spack install hdf5@1.10.5 +mpi ^mpich@3.2
```

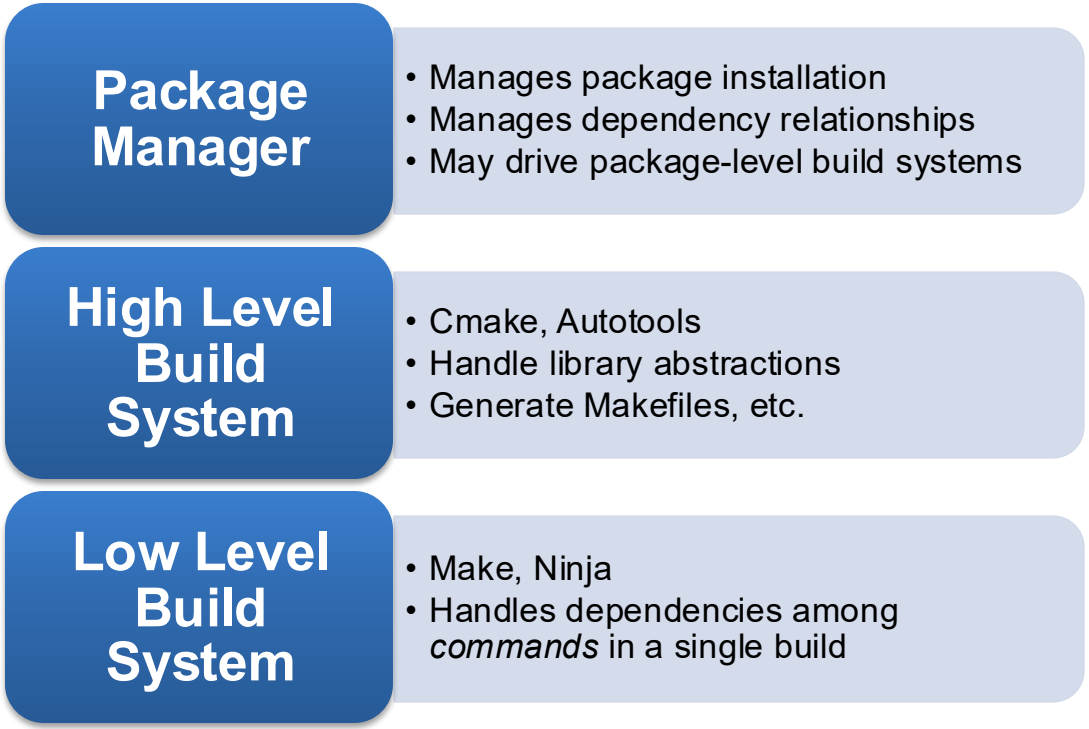


github.com/spack/spack

- 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



Who can use Spack?

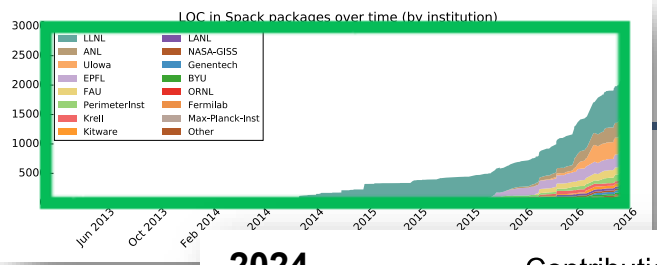
People who want to use or distribute software for HPC!

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 a Linux Foundation Project!

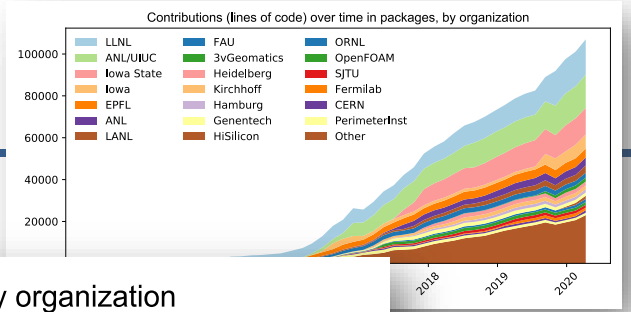
- What does that mean?
  - Project has a neutral legal entity
    - 501(c)(6) non-profit company
  - Project has a Technical Steering Committee (TSC)
    - Charter mandates TSC to make decisions
    - Governance defined at github.com/spack/governance
  - Trademark (Spack name, logo) assigned to Linux Foundation
  - Project resources owned by Linux Foundation
    - spack.io website
    - GitHub Organization
- Spack is part of the High Performance Software Foundation
  - A funded umbrella for collaborating HPC projects

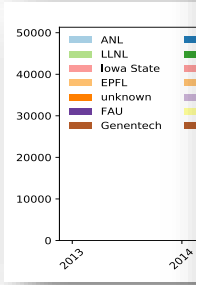
HPSF
HIGH PERFORMANCE SOFTWARE FOUNDATION

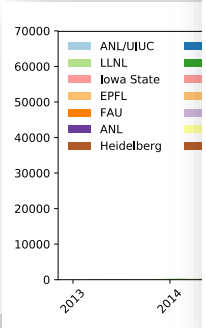


2016

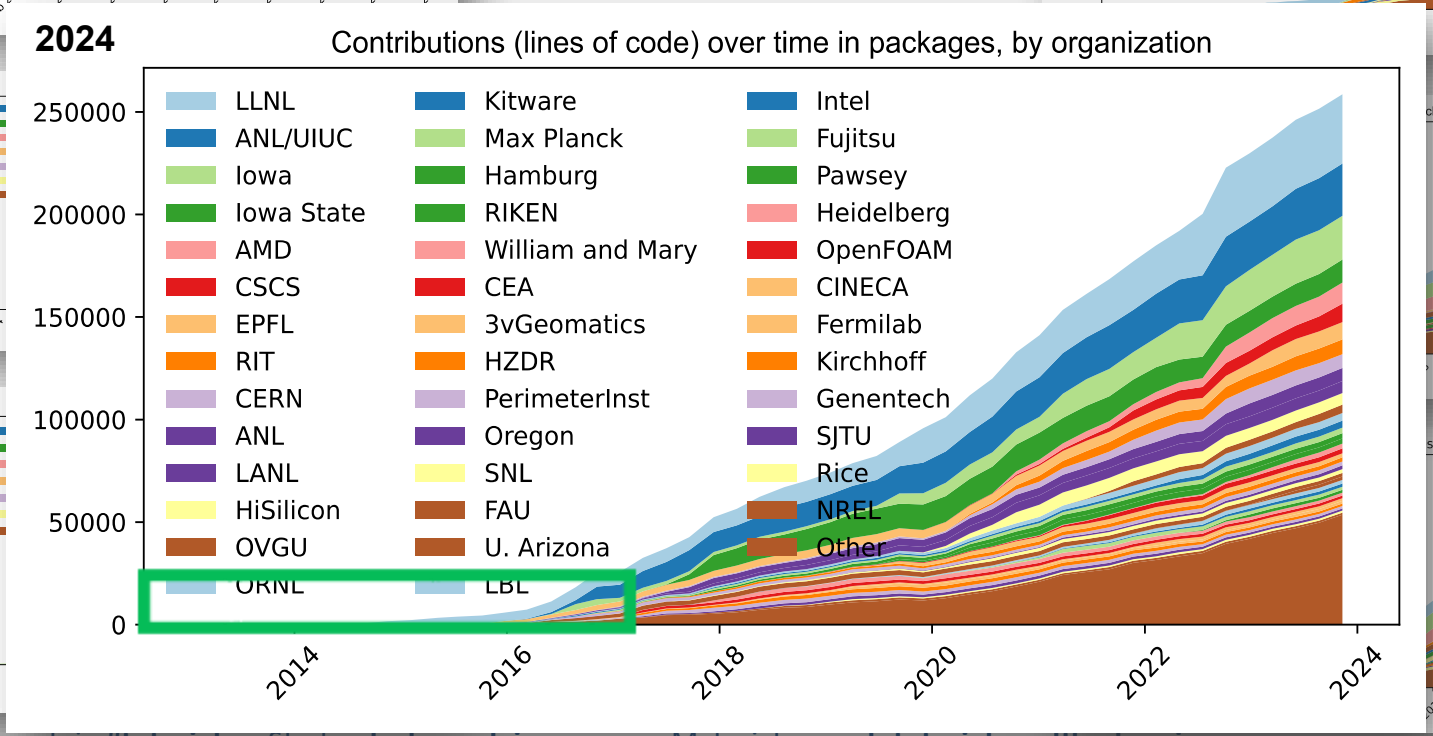
Fall 2020

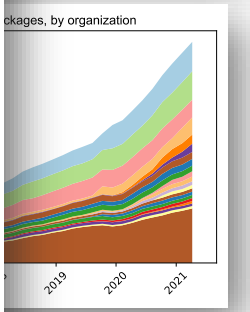


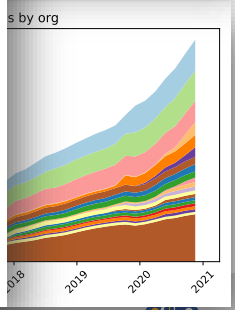




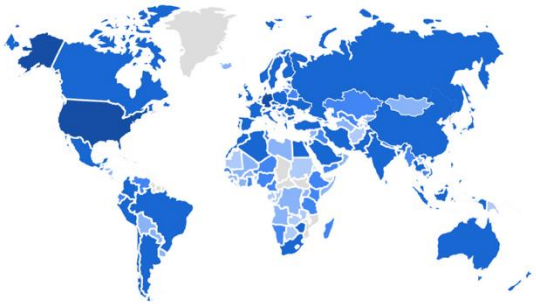
2024







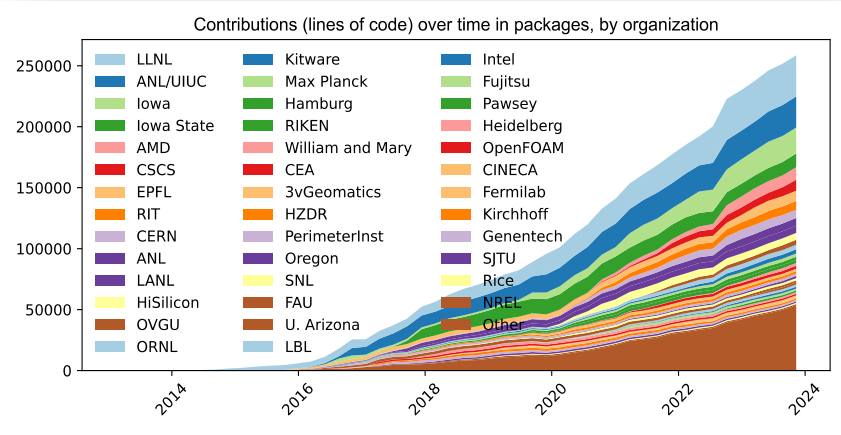
Spack sustains the HPC software ecosystem with the help of many contributors



COUNTRY	USERS
United States	23K
Germany	5.3K
China	4.6K
India	4.5K
United Kingdom	3.3K
France	3K
Japan	2.4K

2023 aggregate documentation user counts from GA4
(note: yearly user counts are almost certainly too large)

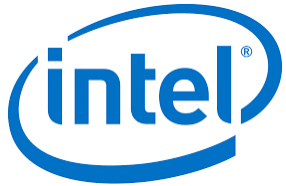
Over 8,500 software packages
Over 1,500 contributors



Contributors continue to grow worldwide!

Spack's widespread adoption has made it a de facto standard, drawing contribution and collaboration from vendors

- **AWS** is investing significantly in cloud credits for Spack
  - Supporting highly scalable cloud CI system with ~250k+/year in credits
  - Integrating Spack with ParallelCluster product
  - Joint Spack tutorial with AWS drew 125+ participants
- **Google** is using Spack in their HPC Toolkit cloud cluster product
  - List packages to deploy; automatically built and cached in cluster deployment
- **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 allowing us to do CI in the cloud for the Cray PE environment
  - Looking at tighter Spack integration with Cray PE
- **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

One month of Spack development is pretty busy!

October 17, 2024 – November 17, 2024

Period: 1 month

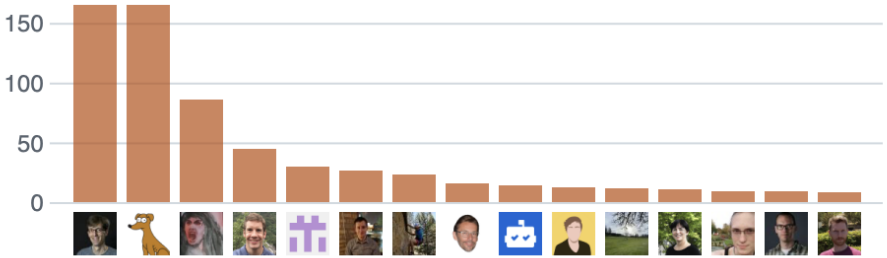
Overview

583 Active pull requests

99 Active issues

457	126	61	38
Merged pull requests	Open pull requests	Closed issues	New issues

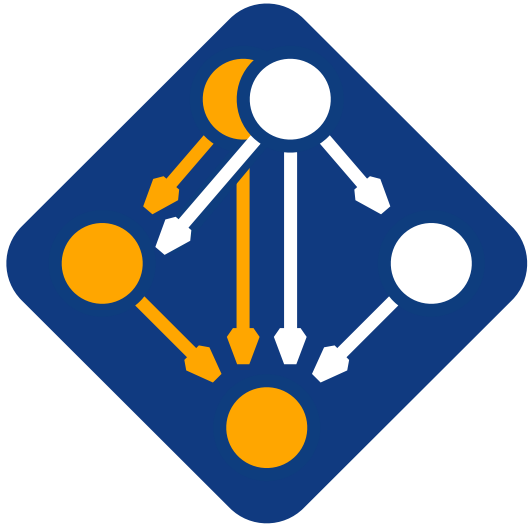
Excluding merges, **142 authors** have pushed **461 commits** to develop and **793 commits** to all branches. On develop, **1,428 files** have changed and there have been **18,717 additions** and **9,238 deletions**.



This tutorial is based on Spack v0.23.0, but Spack 1.0 was just released!

Spack v1.0 (July 2025):

1. Compiler dependencies
2. Toolchains
3. More build parallelism
4. Packages separate from Spack
5. Content-addressable buildcaches
6. New buildcache and repository format
7. More!



Spack v0.23 (November 2024):

1. Language Runtimes: `depends_on("c"|"cxx"|"fortran")`
2. ABI splicing: Build binaries with `mpich`, deploy w/`mvapich2`, etc.
3. Broader variant propagation: `++shared, build_system==Release`
4. Query by namespace with `spack find namespace=myrepo`
5. More!

github.com/spack/spack

Full release notes: https://github.com/spack/spack/releases/tag/v0.23.0

Spack is not the only HPC/AI/data science package manager out there

1. **"Functional" Package Managers**
   - Nix — https://nixos.org
   - Guix — https://hpc.guix.info

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

Other tools in the HPC Space:

- **Easybuild** — https://easybuild.io
  - 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 / Mamba / Pixi**
  - Very popular binary package ecosystem for data science — https://conda.io, https://mamba.readthedocs.io, https://prefix.dev
  - Not targeted at HPC; generally has unoptimized binaries

Claim a VM instance at: bit.ly/spack-vms

	A	B	C	D	E
1	**Spack Tutorial VM Instances**				
2	Instructions:	1. Put your name in a box below to claim an account on a VM insta			
3		2. Log in to your VM:			
4		ssh <IP address>			
5		Login/password are both the username from your column below			
6					
7		**Login / Password**			
8	**IP Address**	spack1	spack2	spack3	spack4
9	35.90.43.21				
10	35.91.36.120		Your Name		
11	34.217.149.171				

If you're in the **spack2** column, your login and password are *both* **spack2**

ssh spack2@3.73.129.196

Claim a login by putting your name in the Google Sheet

Hands-on Time: Spack Basics

Follow script at spack-tutorial.readthedocs.io

Core Spack Concepts

Spack supports 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

```
$ 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=cascadelake    set target microarchitecture
$ spack install mpileaks@3.3 ^mpich@3.2 %gcc@4.9.3  ^ dependency constraints
```

- 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 packages are *parameterized* using the spec syntax

Python DSL defines many ways to build

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

Base package
(CMake support)

Metadata at the class level

Versions

Variants (build options)

Dependencies
(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.

***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

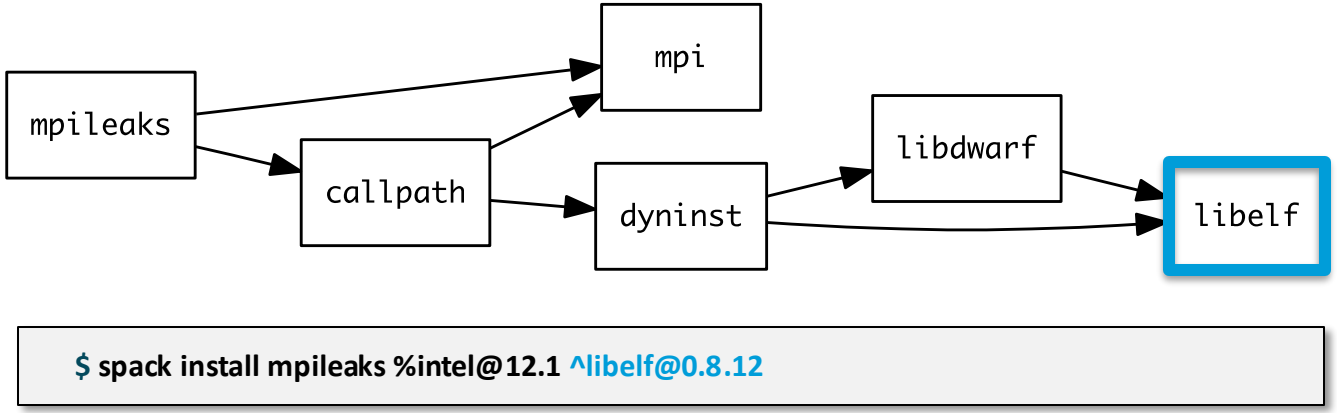
dependency on cuda, but only if cuda is enabled

constraints on cuda version

compiler support for x86_64 and ppc64le

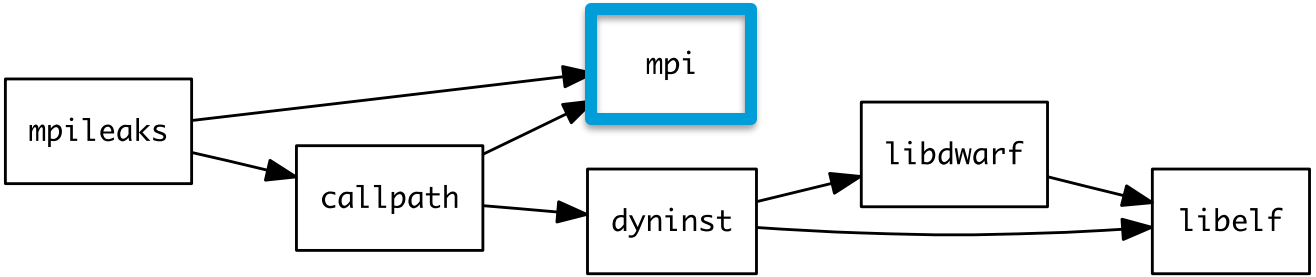
There is a lot of expressive power in the Spack package DSL.

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.

Spack handles ABI-incompatible, versioned interfaces like MPI



- mpi is a *virtual dependency*
- Install the same package built with two different MPI implementations:

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

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

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

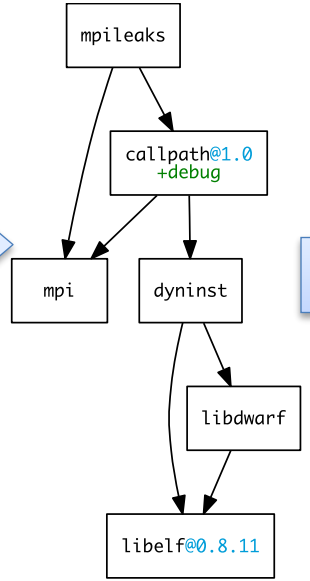
```
$ spack install mpileaks ^mpi@2
```

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

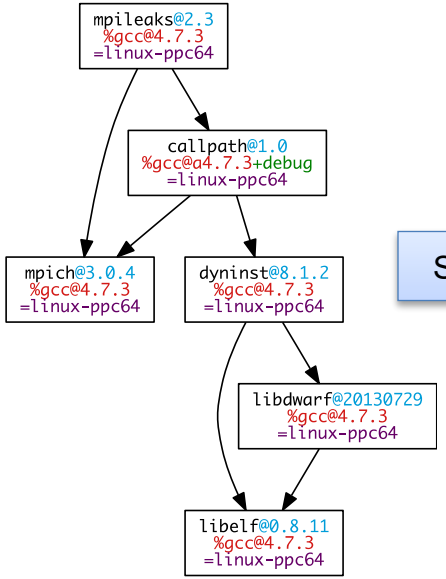
```
mpileaks ^callpath@1.0+debug ^libelf@0.8.11
```

User input: *abstract* spec with some constraints

Normalize



Concretize



Store

spec.yaml

```
spec:
- mpileaks:
    arch: linux-x86_64
    compiler:
      name: gcc
      version: 4.9.2
    dependencies:
      adept-utils: kszrtkpbzac3ss2ixcjkcorlaybnptp4
      callpath: bah5f4h4d2n47mgycej2mtmrivvxy77
      mpich: aa4ar6ifj23yijqmdabeakpejcli72t3
    hash: 33hjjhxi7p6gyzn5ptgyes7sghyprujh
    variants: {}
    version: '1.0'
- adept-utils:
    arch: linux-x86_64
    compiler:
      name: gcc
      version: 4.9.2
    dependencies:
      boost: teesjv7ehpe5kspjim5dk43a7qnowlq
      mpich: aa4ar6ifj23yijqmdabeakpejcli72t3
    hash: kszrtkpbzac3ss2ixcjkcorlaybnptp4
    variants: {}
    version: 1.0.1
- boost:
    arch: linux-x86_64
    compiler:
      name: gcc
      version: 4.9.2
    dependencies: {}
    hash: teesjv7ehpe5kspjim5dk43a7qnowlq
    variants: {}
    version: 1.59.0
...
```

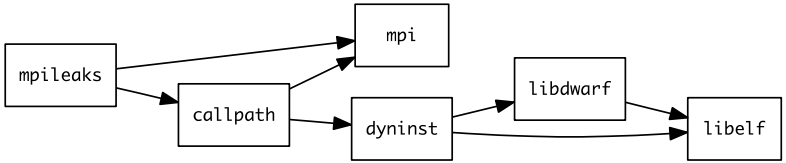
Abstract, normalized spec with some dependencies.

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

Detailed provenance is stored with the installed package

Hashing allows us to handle combinatorial complexity

Dependency DAG



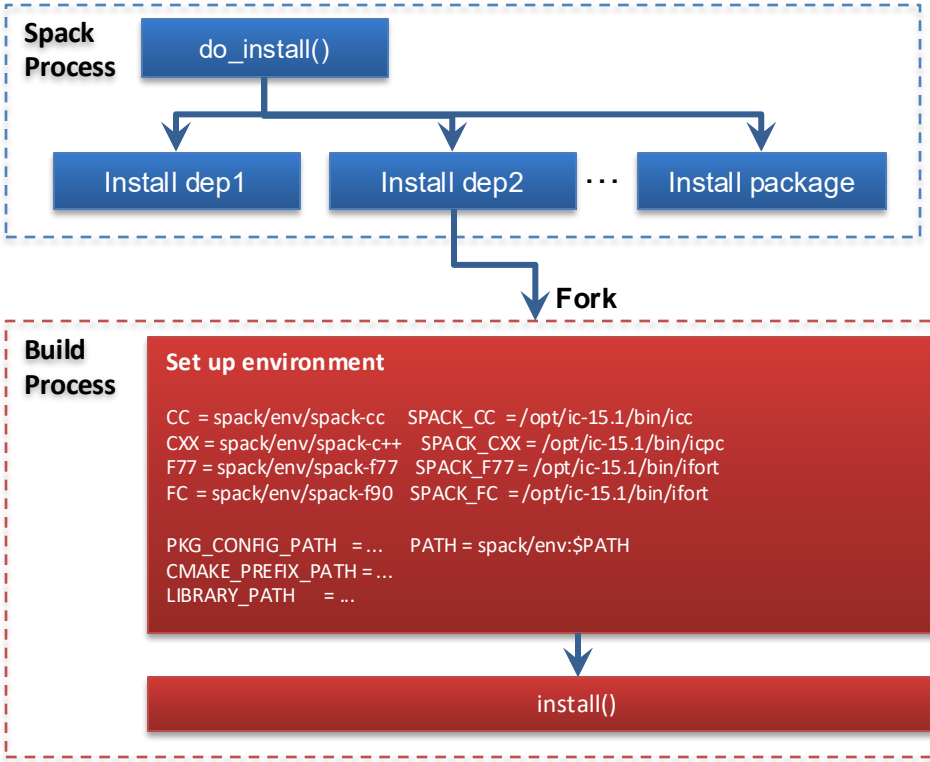
Installation Layout

Hash

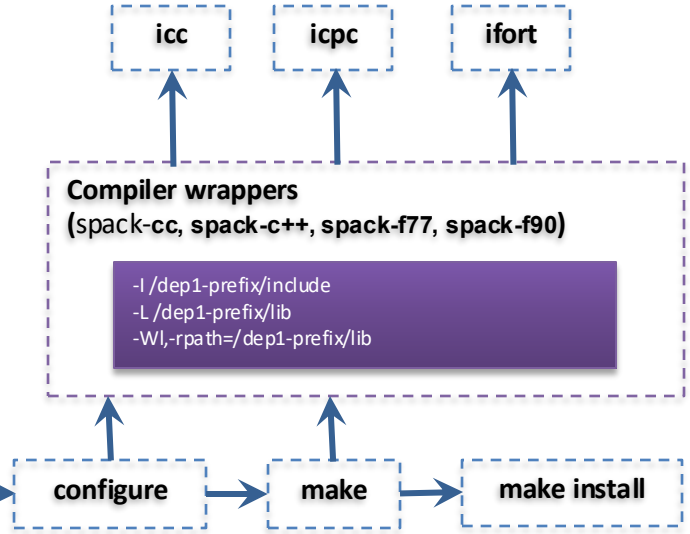
```
opt
└── spack
    ├── darwin-mojave-skylake
    │   └── clang-10.0.0-apple
    │       ├── bzip2-1.0.8-hc4sm4vuzpm4znmvrfzri4ow2mkphe2e
    │       ├── python-3.7.6-daqqpssxb6qbfrztsezkmhus3xoflbsy
    │       ├── sqlite-3.30.1-u64v26igxvxyn23hysmklfums6tgjv5r
    │       ├── xz-5.2.4-u5eawkvaoc7vonabe6nndkcfwuv233cj
    │       └── 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*

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)

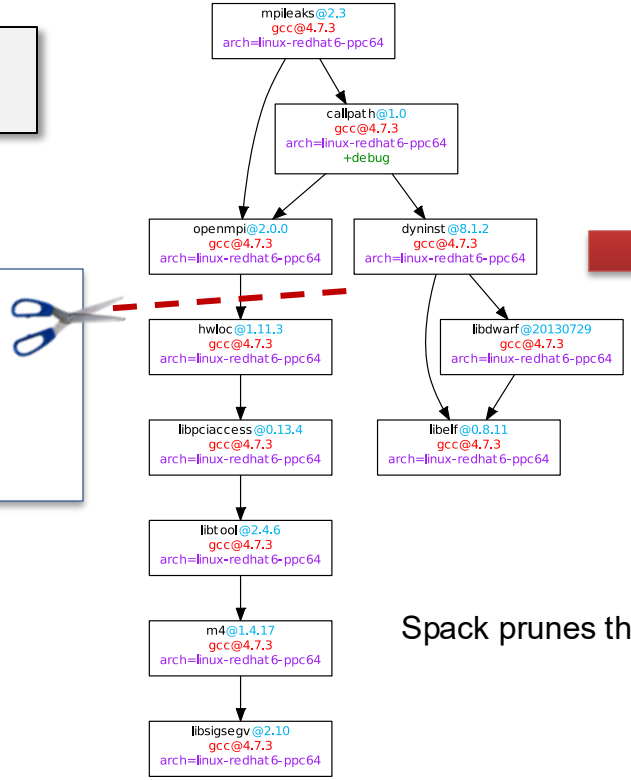


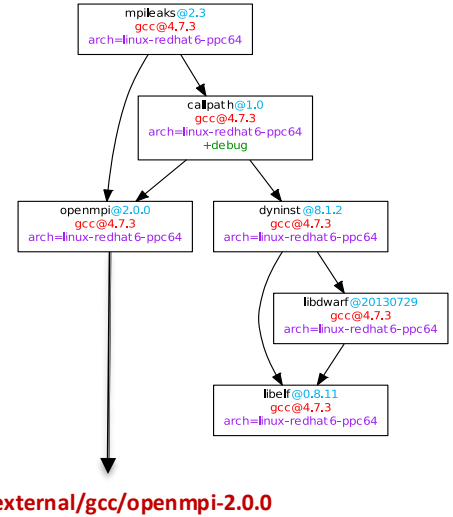
We can configure Spack to build with external software

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

packages.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
      ...
```





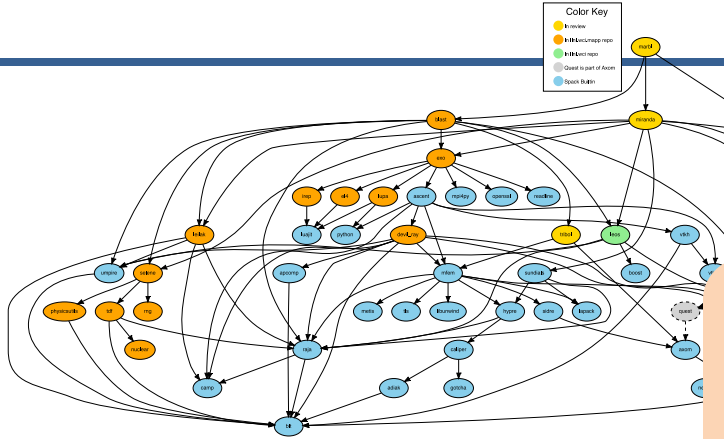
/path/to/external/gcc/openmpi-2.0.0

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

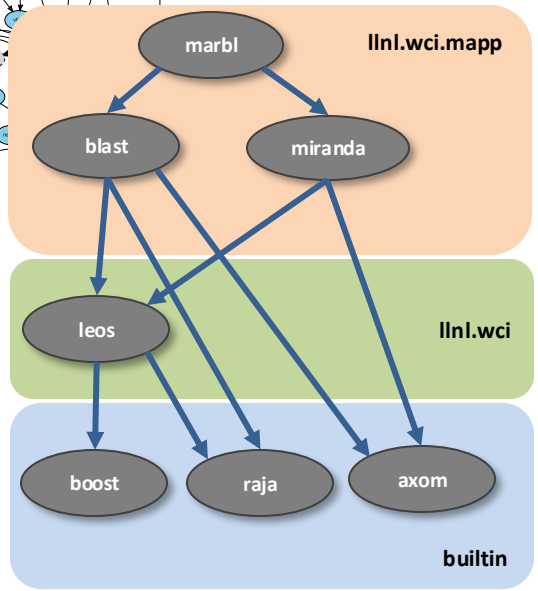
Spack prunes the DAG when adding external packages.

Spack package repositories allow stacks to be layered

LLNL MARBL multi-physics application



MARBL Application Packages



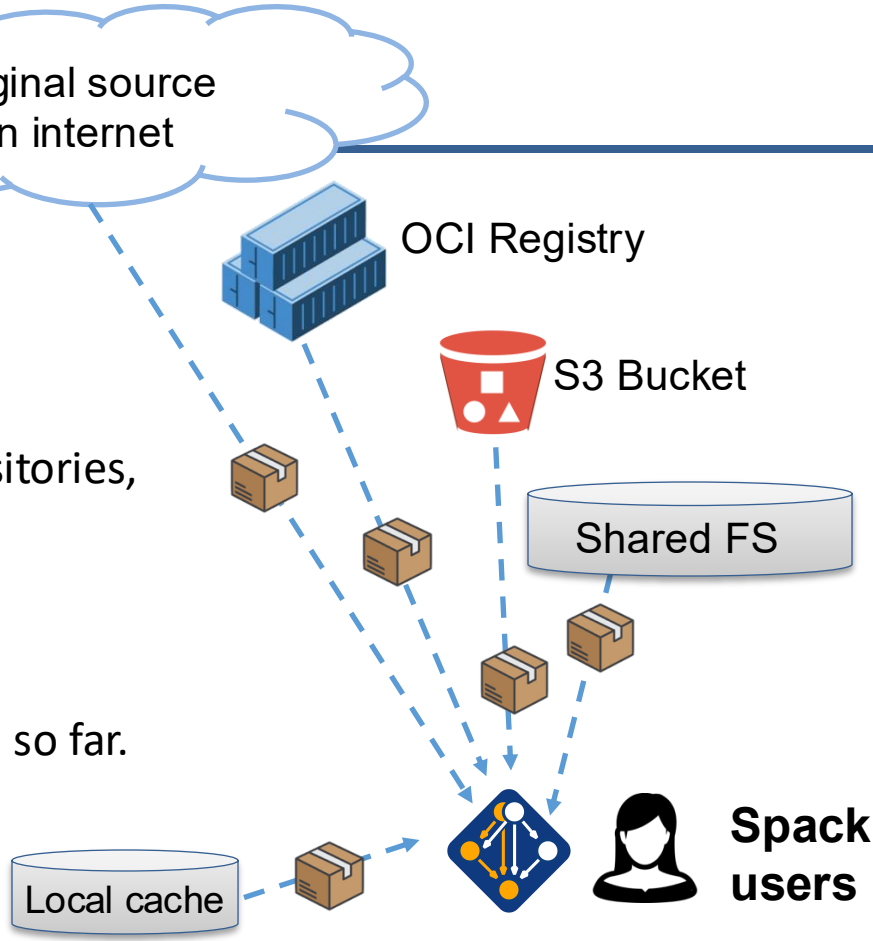
Common internal packages

Open Source Spack packages

```
$ 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

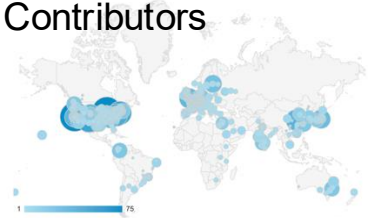
- Spack allows you to define *mirrors:*
  - Directories in the filesystem
  - On a web server
  - In an S3 bucket
- Mirrors are archives of fetched tarballs, repositories, and other resources needed to build
  - Can also contain binary packages
- By default, Spack maintains a mirror in var/spack/cache 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

Dependency solving is NP-hard

Contributors



- new versions
- new dependencies
- new constraints

package.py repository

concretizer

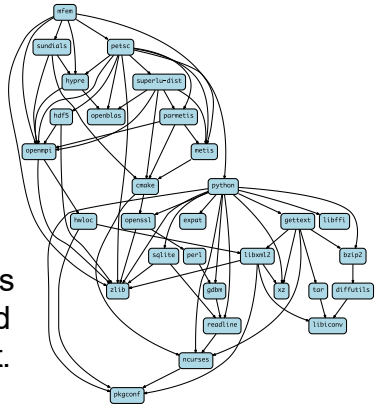
spack developers — yaml — default config: packages.yaml

admins, users — yaml — local preferences config: packages.yaml

users — yaml — local environment config: spack.yaml

users — Command line constraints: `spack install hdf5@1.12.0 +debug`

Concrete spec is fully constrained and can be built.



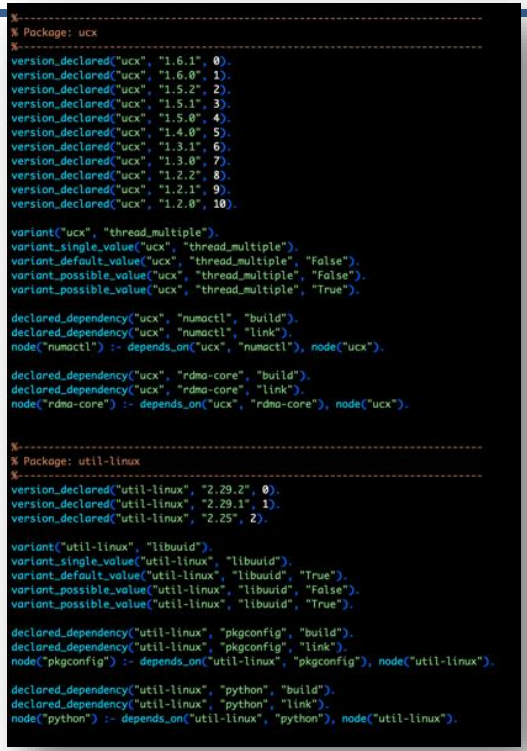
We use logic programming to simplify package solving

- 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

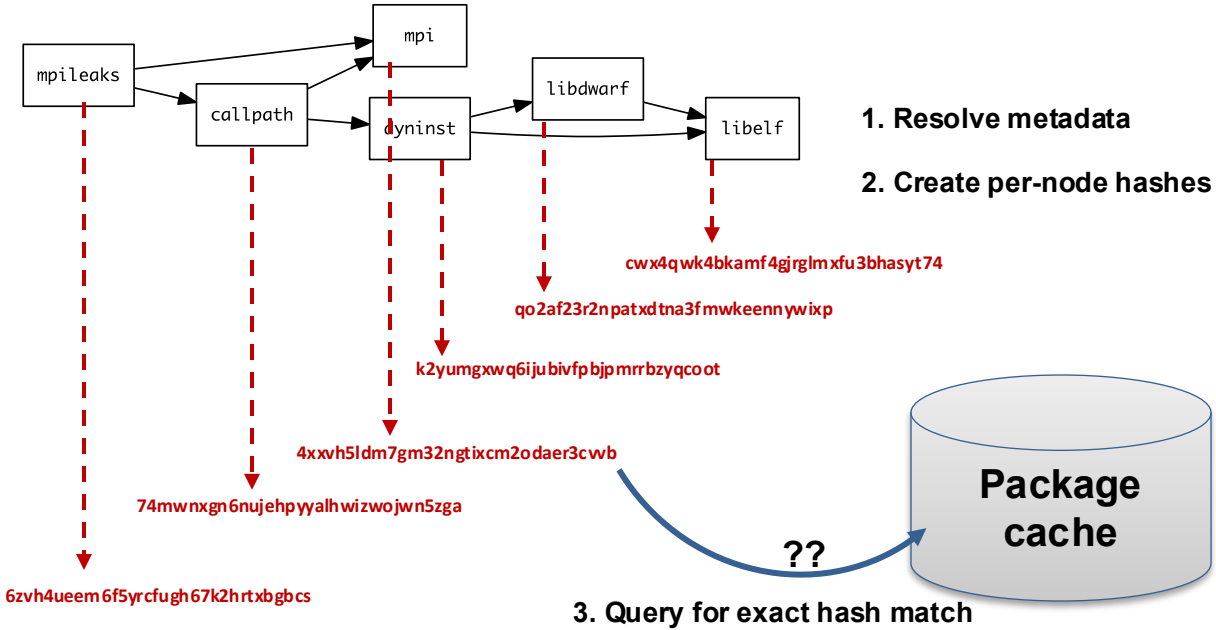
- New concretizer solves many specs that old concretizer can't
  - Backtracking is a huge win – many issues resolved
  - Conditional logic that was complicated before is now much easier



```
%-----------------------------------------------------------------------------
% 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").
```

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

--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:

```
installed_hash("openssl","lwatuuysmwkhuahrncywvn77icdhs6mn").
imposed_constraint("lwatuuysmwkhuahrncywvn77icdhs6mn","node","openssl").
imposed_constraint("lwatuuysmwkhuahrncywvn77icdhs6mn","version","openssl","1.1.1g").
imposed_constraint("lwatuuysmwkhuahrncywvn77icdhs6mn","node_platform_set","openssl","darwin").
imposed_constraint("lwatuuysmwkhuahrncywvn77icdhs6mn","node_os_set","openssl","catalina").
imposed_constraint("lwatuuysmwkhuahrncywvn77icdhs6mn","node_target_set","openssl","x86_64").
imposed_constraint("lwatuuysmwkhuahrncywvn77icdhs6mn","variant_set","openssl","systemcerts","True").
imposed_constraint("lwatuuysmwkhuahrncywvn77icdhs6mn","node_compiler_set","openssl","apple-clang").
imposed_constraint("lwatuuysmwkhuahrncywvn77icdhs6mn","node_compiler_version_set","openssl","apple-clang","12.0.0").
imposed_constraint("lwatuuysmwkhuahrncywvn77icdhs6mn","concrete","openssl").
imposed_constraint("lwatuuysmwkhuahrncywvn77icdhs6mn","depends_on","openssl","zlib","build").
imposed_constraint("lwatuuysmwkhuahrncywvn77icdhs6mn","depends_on","openssl","zlib","link").
imposed_constraint("lwatuuysmwkhuahrncywvn77icdhs6mn","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:

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

2. Choosing a hash means we impose its constraints:

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

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

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

4. Minimize builds!

```
#minimize { 1@100,Package : build(Package) }.
```

With and without --reuse optimization

```
(spackle):solver> spack solve -Il hdf5
==> Best of 9 considered solutions.
==> Optimization Criteria:
  Priority  Criterion                                                  Installed  ToBuild
 1         number of packages to build (vs. reuse)                            -       20
 2         deprecated versions used                                           0        0
 3         version weight                                                     0        0
 4         number of non-default variants (roots)                             0        0
 5         preferred providers for roots                                      0        0
 6         default values of variants not being used (roots)                  0        0
 7         number of non-default variants (non-roots)                         0        0
 8         preferred providers (non-roots)                                    0        0
 9         compiler mismatches                                                0        0
 10        OS mismatches                                                      0        0
 11        non-preferred OS's                                                 0        0
 12        version badness                                                    0        2
 13        default values of variants not being used (non-roots)              0        0
 14        non-preferred compilers                                            0        0
 15        target mismatches                                                  0        0
 16        non-preferred targets                                              0        0

  -   zzngfs3  hdf5@1.10.7%apple-clang@13.0.0~cxx~fortran~hl~ipo~java+mpi+shared~szip~threadsafe+tools api=default b
  -   nsylovq      ^cmake@3.21.4%apple-clang@13.0.0~doc+ncurses+openssl+ownlibs~qt build_type=Release arch=darwin-bi
  -   xdbaqeo          ^ncurses@6.2%apple-clang@13.0.0~symlinks+termlib abi=none arch=darwin-bigsur-skylake
  -   kfureok              ^pkgconf@1.8.0%apple-clang@13.0.0 arch=darwin-bigsur-skylake
  -   5ekd4ap          ^openssl@1.1.1l%apple-clang@13.0.0~docs certs=system arch=darwin-bigsur-skylake
  -   xz6a265              ^perl@5.34.0%apple-clang@13.0.0+cpanm+shared+threads arch=darwin-bigsur-skylake
  -   xgt3tls                  ^berkeley-db@18.1.40%apple-clang@13.0.0+cxx~docs+stl patches=b231fcc4d5cff05e5c3a4814
  -   65edjf6                  ^bzip2@1.0.8%apple-clang@13.0.0~debug~pic+shared arch=darwin-bigsur-skylake
  -   662adoo                      ^diffutils@3.8%apple-clang@13.0.0 arch=darwin-bigsur-skylake
  -   fu7tfsr                          ^libiconv@1.16%apple-clang@13.0.0 libs=shared,static arch=darwin-bigsur-skyla
  -   vjg67nd                  ^gdbm@1.19%apple-clang@13.0.0 arch=darwin-bigsur-skylake
  -   tjceldr                      ^readline@8.1%apple-clang@13.0.0 arch=darwin-bigsur-skylake
  -   xevvljj                  ^zlib@1.2.11%apple-clang@13.0.0+optimize+pic+shared arch=darwin-bigsur-skylake
  -   xelfobh      ^openmpi@4.1.1%apple-clang@13.0.0~atomics~cuda~cxx~cxx_exceptions+gpfs~internal-hwloc~java~legacy
  -   zruns75          ^hwloc@2.6.0%apple-clang@13.0.0~cairo~cuda~gl~libudev+libxml2~netloc~nvml~opencl~pci~rocm+sh
  -   ib4fnkf              ^libxml2@2.9.12%apple-clang@13.0.0~python arch=darwin-bigsur-skylake
  -   dwiv2ys                  ^xz@5.2.5%apple-clang@13.0.0~pic libs=shared,static arch=darwin-bigsur-skylake
  -   blitnbl          ^libevent@2.1.12%apple-clang@13.0.0+openssl arch=darwin-bigsur-skylake
  -   h7jalyu          ^openssh@8.7p1%apple-clang@13.0.0 arch=darwin-bigsur-skylake
  -   7v7bqx2              ^libedit@3.1-20210216%apple-clang@13.0.0 arch=darwin-bigsur-skylake
```

Pure hash-based reuse: all misses

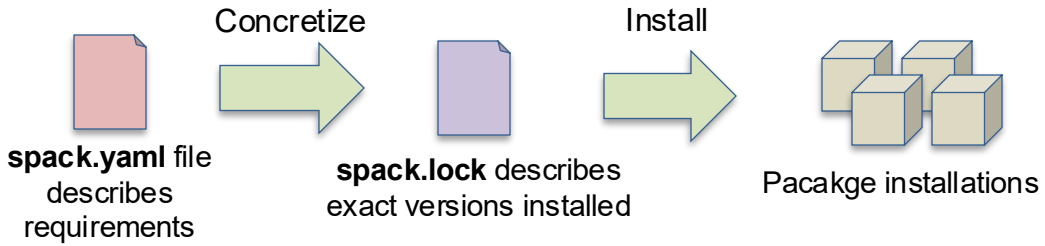
```
(spackle):spack> spack solve --reuse -Il hdf5
==> Best of 10 considered solutions.
==> Optimization Criteria:
  Priority  Criterion                                                  Installed  ToBuild
 1         number of packages to build (vs. reuse)                            -        4
 2         deprecated versions used                                           0        0
 3         version weight                                                     0        0
 4         number of non-default variants (roots)                             0        0
 5         preferred providers for roots                                      0        0
 6         default values of variants not being used (roots)                  0        0
 7         number of non-default variants (non-roots)                         2        0
 8         preferred providers (non-roots)                                    0        0
 9         compiler mismatches                                                0        0
 10        OS mismatches                                                      0        0
 11        non-preferred OS's                                                 0        0
 12        version badness                                                    6        0
 13        default values of variants not being used (non-roots)              1        0
 14        non-preferred compilers                                           15        4
 15        target mismatches                                                  0        0
 16        non-preferred targets                                              0        0

  -   yfkfnsp  hdf5@1.10.7%apple-clang@12.0.5~cxx~fortran~hl~ipo~java+mpi+shared~szip~threadsafe+tools api=defaul
 [+]  zd4m26e      ^cmake@3.21.1%apple-clang@12.0.5~doc+ncurses+openssl+ownlibs~qt build_type=Release arch=darwin-
 [+]  53i52xr          ^ncurses@6.2%apple-clang@12.0.5~symlinks+termlib abi=none arch=darwin-bigsur-skylake
 [+]  us36bwr          ^openssl@1.1.1l%apple-clang@12.0.5~docs+systemcerts arch=darwin-bigsur-skylake
 [+]  74mwnxg              ^zlib@1.2.11%apple-clang@12.0.5+optimize+pic+shared arch=darwin-bigsur-skylake
  -   3ijfnel          ^openmpi@4.1.1%apple-clang@12.0.5~atomics~cuda~cxx~cxx_exceptions+gpfs~internal-hwloc~java~lega
  -   jxexyb7              ^hwloc@2.6.0%apple-clang@12.0.5~cairo~cuda~gl~libudev+libxml2~netloc~nvml~opencl~pci~rocm+
 [+]  ckdn5zf                  ^libxml2@2.9.12%apple-clang@12.0.5~python arch=darwin-bigsur-skylake
 [+]  k7auat3                      ^libiconv@1.16%apple-clang@12.0.5 libs=shared,static arch=darwin-bigsur-skylake
 [+]  k2yumgx                      ^xz@5.2.5%apple-clang@12.0.5~pic libs=shared,static arch=darwin-bigsur-skylake
 [+]  grgtlcd                  ^pkgconf@1.8.0%apple-clang@12.0.5 arch=darwin-bigsur-skylake
  -   nnc66ug                  ^libevent@2.1.12%apple-clang@12.0.5+openssl arch=darwin-bigsur-skylake
 [+]  63xbksk                  ^openssh@8.6p1%apple-clang@12.0.5 arch=darwin-bigsur-skylake
 [+]  snhgldt                      ^libedit@3.1-20210216%apple-clang@12.0.5 arch=darwin-bigsur-skylake
 [+]  qbkmtdd                  ^perl@5.34.0%apple-clang@12.0.5+cpanm+shared+threads arch=darwin-bigsur-skylake
 [+]  tnvkifs                      ^berkeley-db@18.1.40%apple-clang@12.0.5+cxx~docs+stl patches=b231fcc4d5cff05e5c3a4814f
 [+]  7d5woqt                      ^bzip2@1.0.8%apple-clang@12.0.5~debug~pic+shared arch=darwin-bigsur-skylake
 [+]  vh6di3i                      ^gdbm@1.19%apple-clang@12.0.5 arch=darwin-bigsur-skylake
 [+]  qgy3v4l                          ^readline@8.1%apple-clang@12.0.5 arch=darwin-bigsur-skylake
```

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~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 environments enable users to build customized stacks from an abstract description



- spack.yaml 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

```
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": {
   "6s63so2kstp3zyvjezglndmavy6l3nul": {
     "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: 11:00am EDT

Get a VM here →

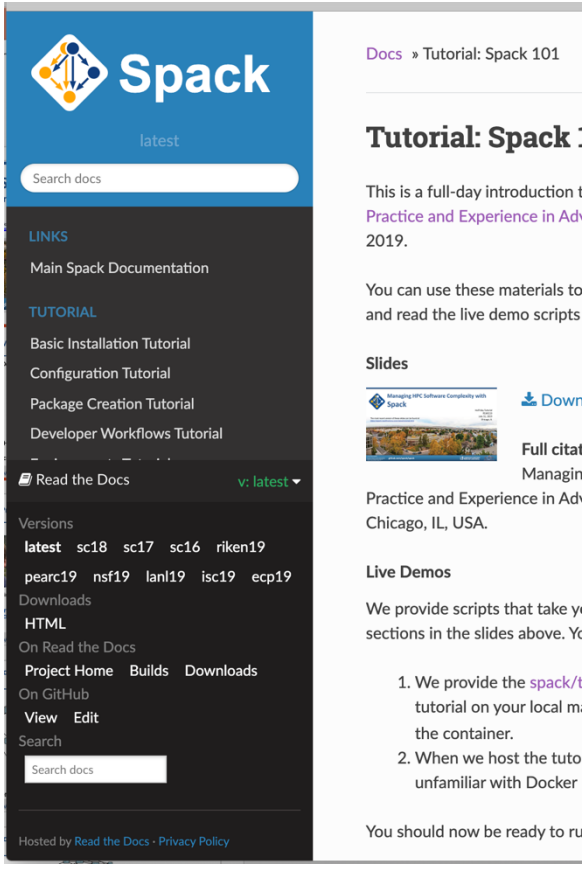
Find the slides and associated scripts here:

spack-tutorial.rtfd.io

Remember to join Spack slack so you can get help later!

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 EDT

Get a VM here →

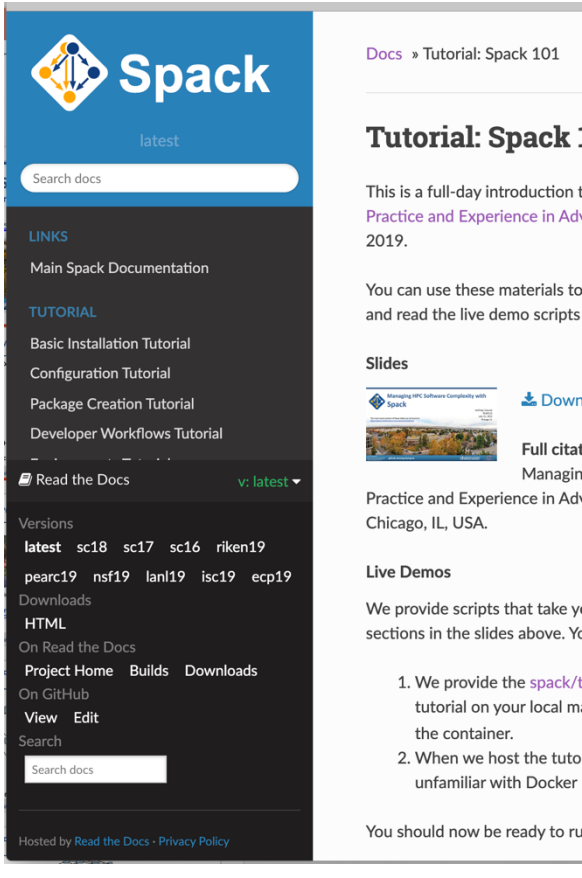
Find the slides and associated scripts here:

spack-tutorial.rtfd.io

Remember to join Spack slack so you can get help later!

slack.spack.io

Join the **#tutorial** channel!



Hands-on Time: Creating Packages

Follow script at spack-tutorial.readthedocs.io

Spack packages are *parameterized* using the spec syntax

Python DSL defines many ways to build

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

Base package
(CMake support)

Metadata at the class level

Versions

Variants (build options)

Dependencies
(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.

***One* package.py file per software project!**

We'll resume at: 3:30pm EDT

Get a VM here →

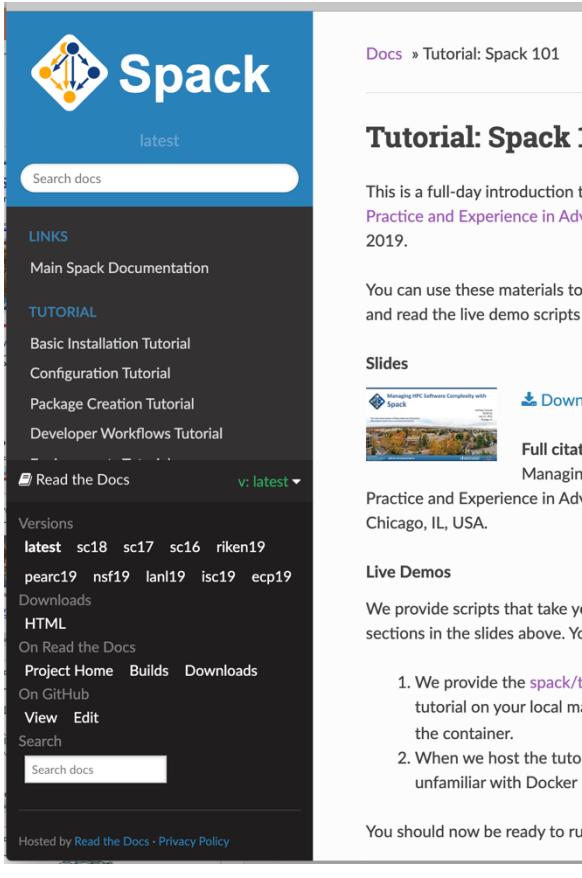
Find the slides and associated scripts here:

spack-tutorial.rtfd.io

Remember to join Spack slack so you can get help later!

slack.spack.io

Join the **#tutorial** channel!



Hands-on Time:
Binary Caches and Mirrors

Follow script at spack-tutorial.readthedocs.io

Hands-on Time:
Developer Workflows

Follow script at spack-tutorial.readthedocs.io

Hands-on Time:
Scripting

Follow script at spack-tutorial.readthedocs.io

Upcoming changes in Spack v1.0

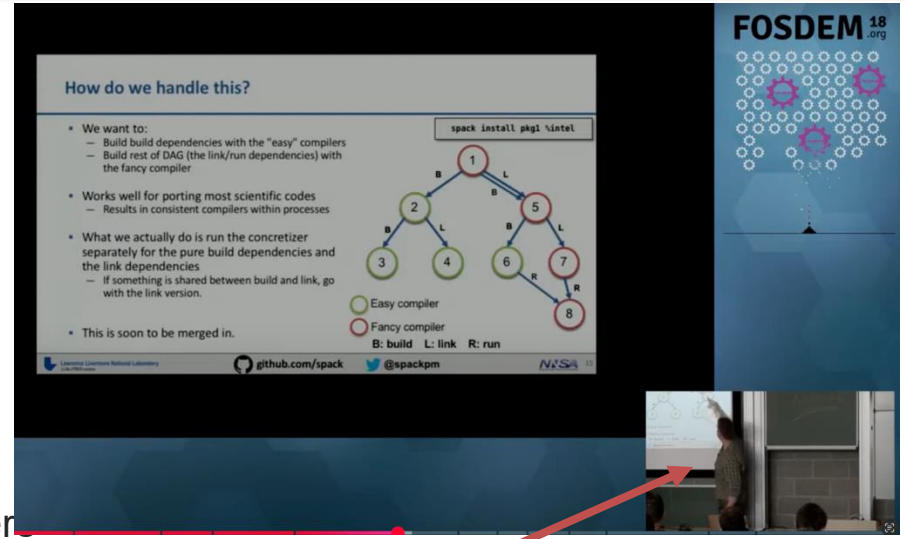
Tutorial Session
ISC 2025, Hamburg, Germany
Jun 13, 2025

THE LINUX FOUNDATION

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

The road to v1.0 has been long

- We wanted:
  - ✓ 2020 New ASP-based concretizer
  - ✓ 2021 Reuse of existing installations
  - ✓ 2022 Stable production CI
  - ✓ 2022 Stable binary cache
  - ✓ 2025 Compiler dependencies
  - ✓ 2025 Separate builtin repo
  - ✓ 2025 Stable package API
- v1.0:
  - Changes the dependency model for compilers
  - Enables users to use entirely custom packages
  - Improves reproducibility
  - Improves stability 🤞
- This is the largest change to Spack… ever.



Todd, presenting how simple all this would be at FOSDEM in 2018

Split the package repository

- Spack is two things:
  - Command line tool `spack`
  - Package repository with 8,400+ recipes
- Community wants
  - package updates without tool changes (e.g. new bugs)
  - tool updates without package changes (reproducibility)
- But coupling between tool and packages was tight
  1. Package classes are in core: `CMakePackage`, `AutotoolsBuilder`, etc.
  2. Compiler wrapper was not a package until recently
  3. Packages *live* in Spack's GitHub repository with a long (git) history

Package split process

- Sync packages to **`spack/spack-packages`**
  - Git history is preserved 😌
- Turn package repositories into Python namespace packages
  - **`spack.pkg.builtin`** is now **`spack_repo.builtin`**
- Move build systems to **`spack_repo.builtin.build_systems`**
- Update packages to use fewer Spack internals
- Enable CI on **`spack/spack-packages`**
- Make Spack support Git-based package repositories

Build systems moved to **spack/spack-packages**

```
1 from spack_repo.builtin.build_systems.autotools import AutotoolsPackage
2 from spack_repo.builtin.build_systems.cmake import CMakePackage
3
4 from spack.package import *
5
6
7 class ZlibNg(AutotoolsPackage, CMakePackage):
8     ...
```

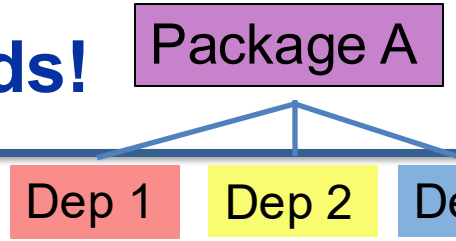
Useful commands after repo split

1. `spack repo migrate`: fixes imports in custom repos for you
2. `spack repo set --destination ~/spack-pkgs builtin`: put packages in your favorite location
3. (`spack repo update`: update & pin package repos 

™)

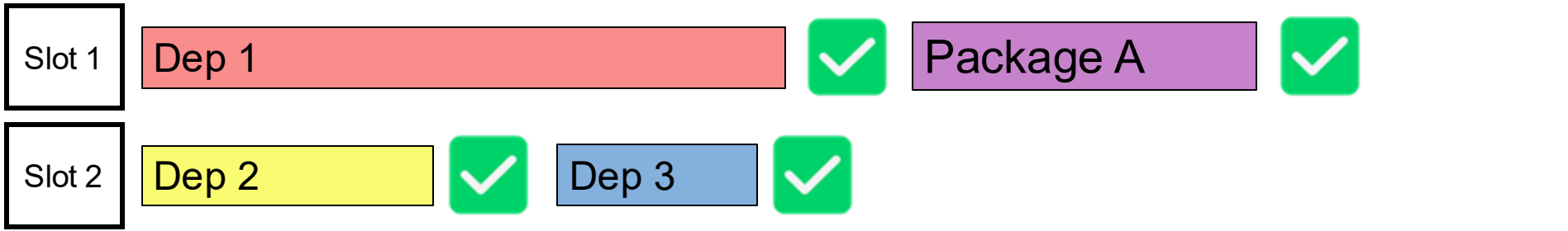
New docs: **https://spack.readthedocs.io/en/latest/repositories.html**

Spack now supports concurrent builds!



- We *sort of* supported this already
  - But the user had to launch multiple spack processes
  - e.g., `srun -N 4 -n 16 spack install hdf5`

- Now spack handles on-node parallelism itself!
  - Spack now has a scheduler loop
  - Monitors dependencies, starts multiple processes, polls for completion
  - User can control max concurrent processes with '-p'



You can now specify the package repo version in an environment or config

Pin a commit

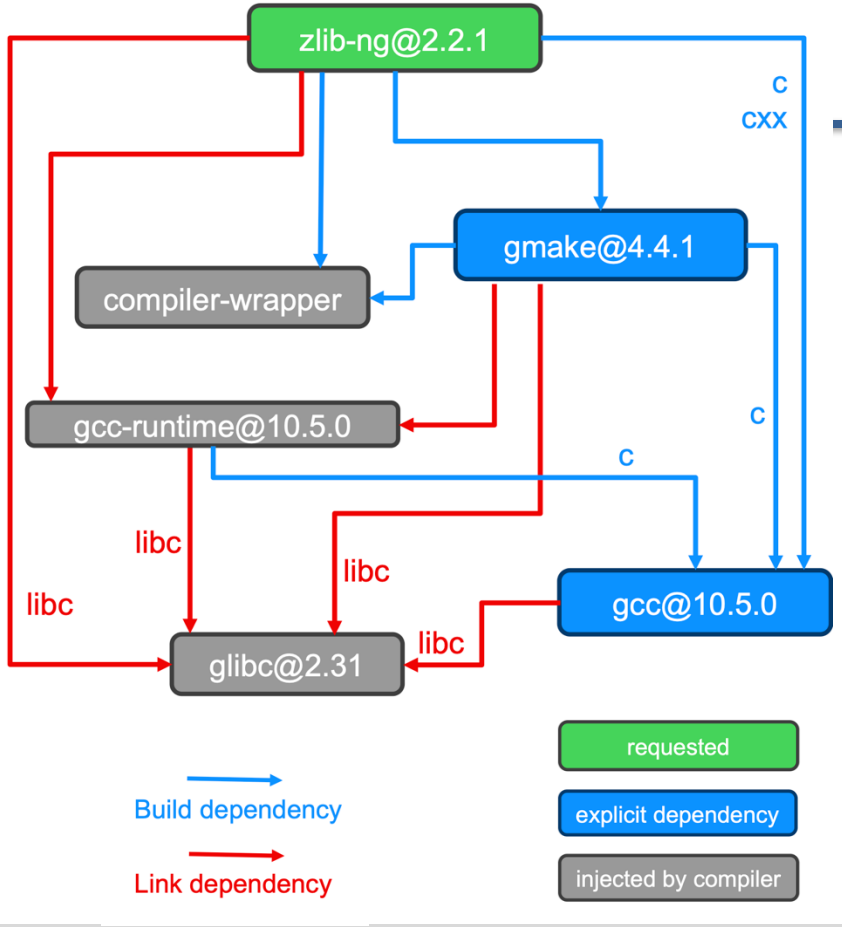
```
spack:
  repos:
    builtin:
      git: https://github.com/spack/spack-packages.git
      commit: aec1e3051c0e9fc7ef8feadf766435d6f8921490
```

Work on a branch

```
spack:
  repos:
    builtin:
      git: https://github.com/spack/spack-packages.git
      destination: /path/to/clone/of/spack-packages
      branch: develop
```

Compiler Dependencies

- Compilers are now *build dependencies*
- Runtime libraries modeled as packages
  - gcc-runtime is injected as link dependency by gcc
  - packages depend on c, cxx,
    fortran virtuals, which are satisfied by gcc node
- glibc is an automatically detected external
  - Injected as a `libc` virtual dependency
  - Does not require user configuration
- Will eventually be able to choose implementations (e.g., musl)



Language dependencies

```
depends_on("c", type="build")
depends_on("cxx", type="build")
depends_on("fortran", type="build")
```

- You now need to specify these to use c, cxx, or fortran
  - No-op in the release as we prepare for compilers as dependencies
  - Backported to v0.22 release to assist teams working across Spack releases
- Spack has historically made these compilers available to every package
  - A compiler was actually “something that supports c + cxx + fortran + f77”
  - Made for a lot of special cases
  - Also makes for duplication of purely interpreted packages (e.g. python)

Configuring compilers in Spack v1.*

Spack v0.x

compilers.yaml

```yaml
compilers:
  - compiler:
      spec: gcc@12.3.1
      paths:
         c: /usr/bin/gcc
         cxx: /usr/bin/g++
         fc: /usr/bin/gfortran
      modules: [...]
```

Spack v1.x

packages.yaml

```yaml
packages:
  gcc:
    externals:
    - spec: gcc@12.3.1+binutils
      prefix: /usr
      extra_attributes:
        compilers:
          c: /usr/bin/gcc
          cxx: /usr/bin/g++
          fc: /usr/bin/gfortran
        modules: [...]
```

- We automatically convert compilers.yaml, when no compiler is configured
- We will still support *reading* the old configuration until at *least* v1.1
- All fields from `compilers.yaml` are supported in `extra_attributes`

More on direct dependencies with %

- You could previously write:

```
pkg %gcc +foo      # +foo would associate with pkg, not gcc - will error in 1.0
```

- Now you'll need to write:

```
pkg +foo %gcc      # +foo associates with pkg
```

- We want these to be symmetric:

```
pkg +foo %dep +bar  # `pkg +foo` depends on `dep +bar` directly
pkg +foo ^dep +bar  # `pkg +foo` depends on `dep +bar` directly or transitively
```

- spack style --spec-strings --fix can remedy this automatically
  - Fixes YAML files, scripts, package.py files
  - Alternative was to have a very hard-to-explain syntax – we surveyed users and they decided it was better to break a bit than to explaining the subtleties of the first 10 years of Spack forever

Breaking changes 

1. It is no longer safe to assume every node has a compiler.
   a. The tokens `{compiler}`, `{compiler.version}`, and `{compiler.name}` in `Spec.format` expand to `none` if a Spec does not depend on C, C++, or Fortran.
   b. `spec.compiler` will default to the c compiler if present, else cxx, else fortran for backwards compatibility.
   c. The new default install tree projection is `{architecture.platform}/{architecture.target}/{name}-{version}-{hash}`

2. The syntax `spec["name"]` will only search link/run dependencies and *direct* build dependencies.
   - Previously, this would find deep, transitive deps, which was almost always the wrong behavior.
   - You can still hop around in the graph, e.g. spec["cmake"][”bzip2"] will find cmake’s link dependency

3. The % sigil in specs means “direct dependency”.
   - Can now say: `foo %cmake@3.26 ^bar %cmake@3.31`
   - `^` dependencies are unified, `%` dependencies are not

Spack 1.x introduces *toolchains*

toolchains.yaml

```
toolchains:
  clang_gfortran:
    - spec: %c=clang
      when: %c
    - spec: %cxx=clang
      when: %cxx
    - spec: %fortran=gcc
      when: %fortran
    - spec: cflags="-O3 -g"
    - spec: cxxflags="-O3 -g"
    - spec: fflags="-O3 -g"
```

```
toolchains:
  intel_mvapich2:
    - spec: %c=intel-oneapi-compilers @2025.1.1
      when: %c
    - spec: %cxx=intel-oneapi-compilers @2025.1.1
      when: %cxx
    - spec: %fortran=intel-oneapi-compilers @2025.1.1
      when: %fortran
    - spec: %mpi=mvapich2 @2.3.7-1 +cuda
      when: %mpi
```

```
spack install foo %clang_gfortran
```

```
spack install foo %intel_mvapich2
```

- Can lump many dependencies, flags together and use them with a single name
- Any spec in a toolchain can be *conditional*
  - Only apply when needed

Other new 1.0 features of interest

- Concurrent builds (`spack install -p 4 -j 16`)
- Toolchains: aliases for dependency combinations
- Content-addressed build caches
- New package repository layout
  - Packages are standard python packages (better for editors/tooling)

 1.0 was released on July 20, 2025! 

But wait! There's more!

HPSF

Spack is part of the
High Performance Software Foundation

Join us at the Spack User Meeting at
HPSFCon 2026 next year!

@hpsf.bsky.social

hpsf.io

Join us after ISC!

- Join us and 3,700+ others on Spack slack
- Contribute packages, docs, and features on GitHub
- Continue the tutorial at spack-tutorial.rtfd.io

slack.spack.io

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

@spackpm.bsky.social

@spack@hpc.social

@spackpm

spack.io

We hope to make distributing & using HPC software easy!