Spack: Package Management for HPC

ATPESC 2025 Software Sustainability Track

Todd Gamblin

Spack Project Lead
Distinguished Member of Technical Staff
Lawrence Livermore National Laboratory

THE LINUX FOUNDATION

We build codes from hundreds of small, complex pieces

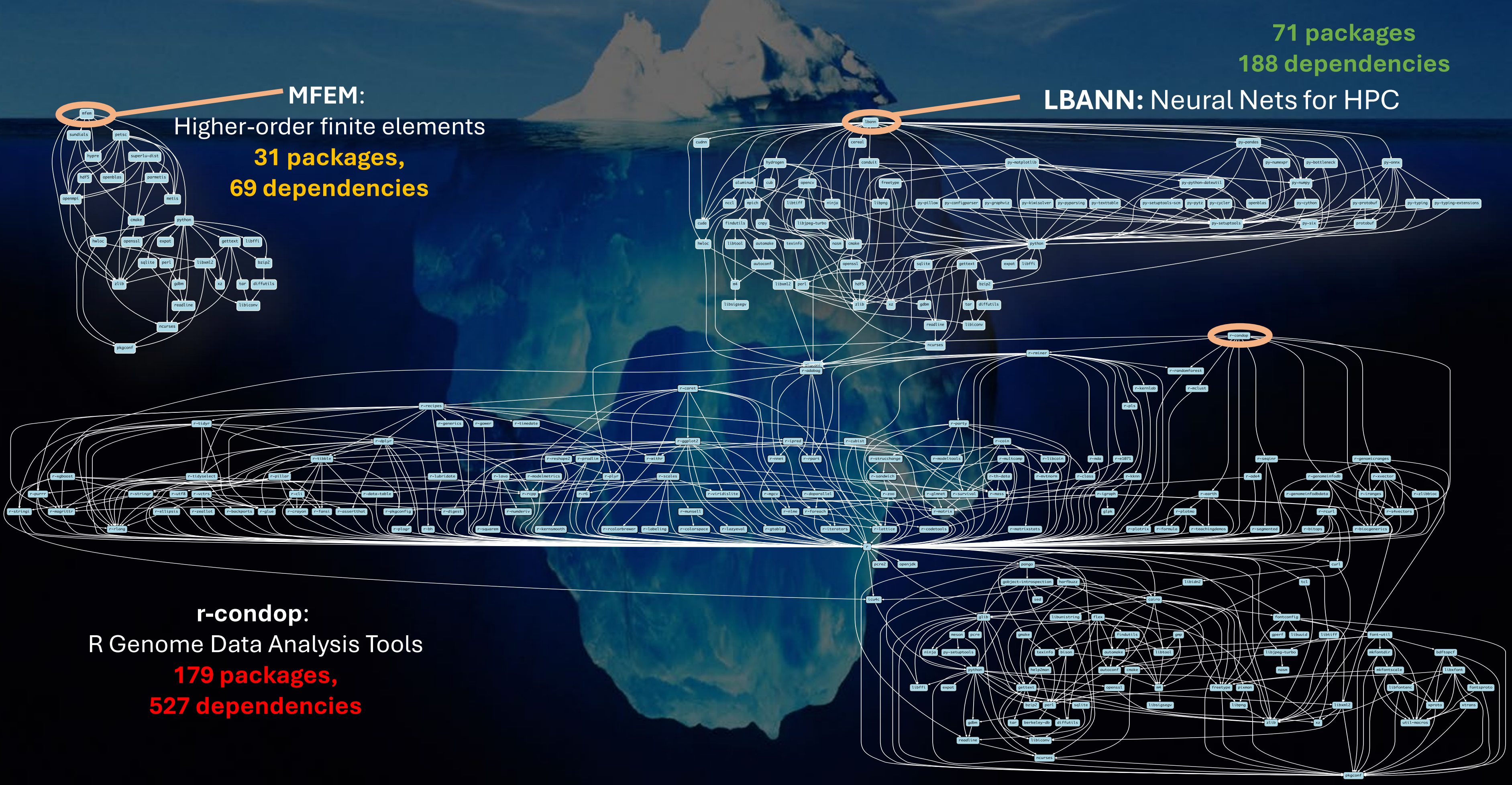
> *Just when we're starting to solve the problem of how to create software using reusable parts, it founders on the nuts-and-bolts problems outside the software itself.*
>
> P. DuBois & T. Epperly. ***Why Johnny Can't Build***. Scientific Programming. Sep/Oct 2003.

- **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

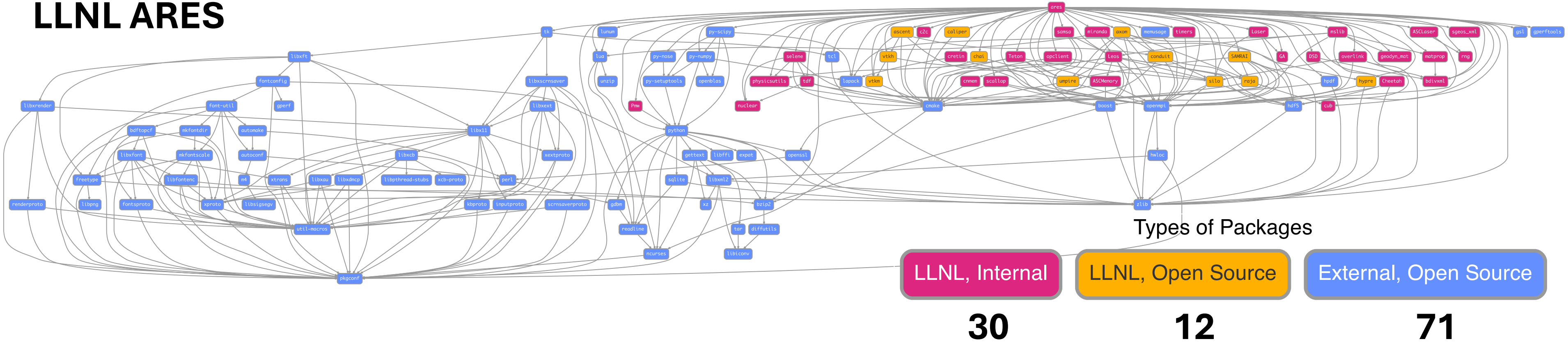
71 packages
188 dependencies

LBANN: Neural Nets for HPC

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

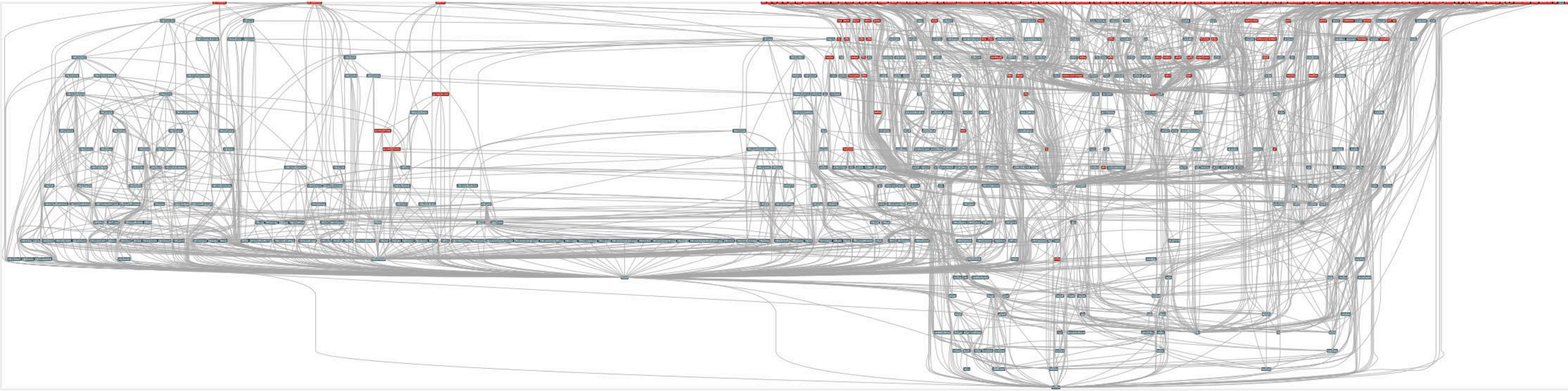
Even proprietary software builds on top of open source

LLNL ARES



- Open source is critical for nearly every application
- 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?

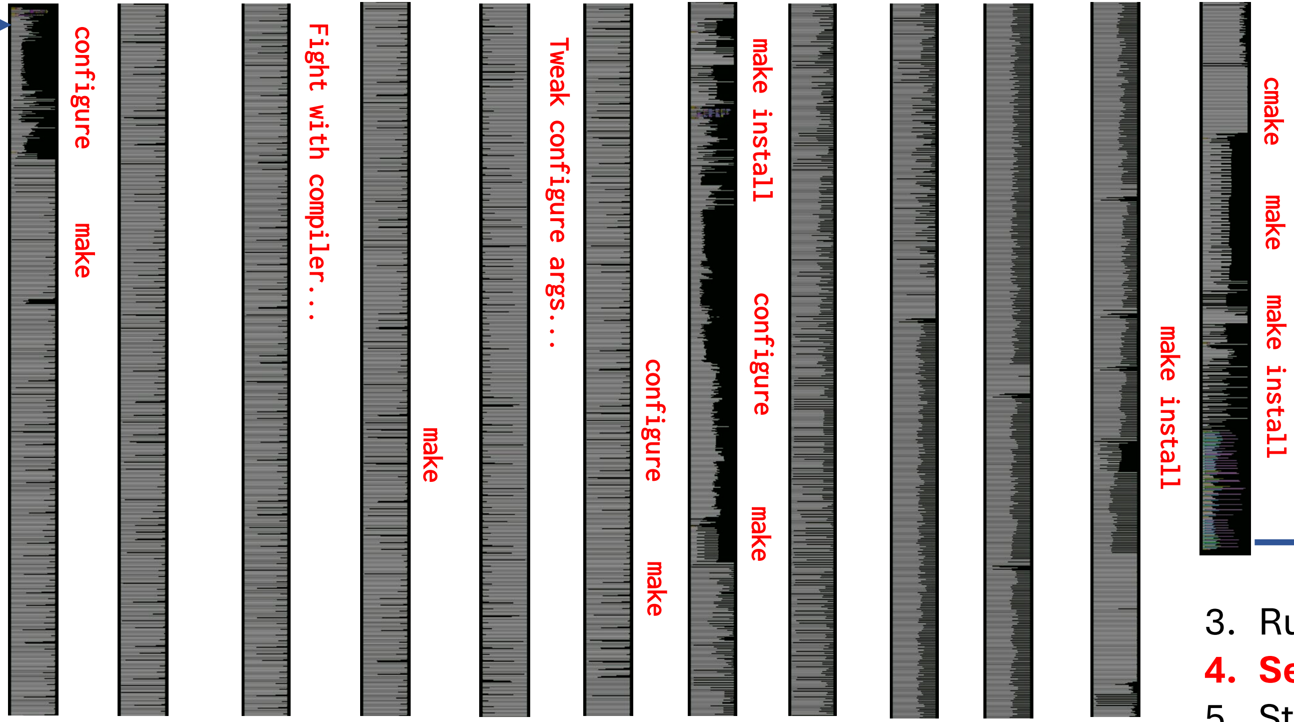
ECP’s E4S stack is even larger than these codes



- 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 history:
How to install software on a supercomputer, circa 2013

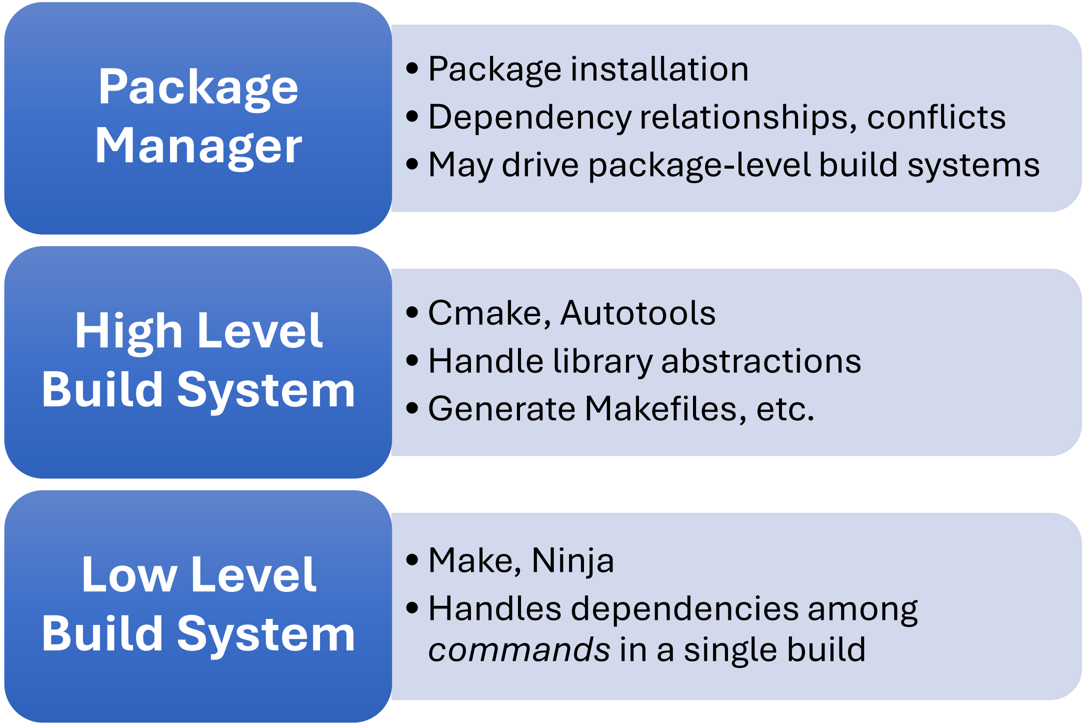
1. Download all 16 tarballs you need
2. Start building!



3. Run code
4. **Segfault!?**
5. Start over...

What does a package managers do to help?

- **Does not** replace Cmake/Autotools
- Manages ***dependencies***
  - Drives package-level build systems
  - Ensures **consistency** and **compatibility** among builds of packages in the ecosystem
- Stores **community knowledge**
  - Cache of package build recipes
  - Determining magic configure lines takes time



Package Manager

- Package installation
- Dependency relationships, conflicts
- 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

Many package managers (conda, pip, apt, etc.) make simplifying assumptions about the software ecosystem

- **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) and *just one language*

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**

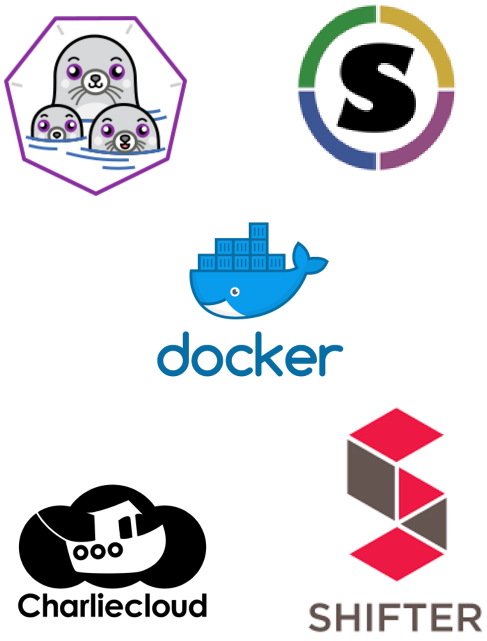
Argonne National Lab
Intel **Xeon** / **Xe**

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 are often optimized per-system**
  - Not clear that we can ever build one container for all facilities



We need something more flexible to **build** versions of containers

Overview & Community

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)
  - Supports *binary "buildcaches"* so that you don't have to build everything from source

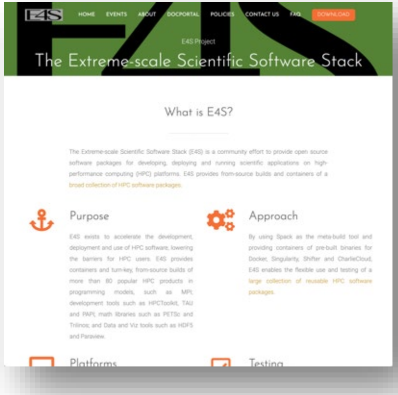
Anyone can use Spack!

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

Spack was critical for ECP's mission to create a robust, capable exascale software ecosystem.

ECP

EXASCALE COMPUTING PROJECT



E4S Project

The Extreme-scale Scientific Software Stack

What is E4S?

The Extreme-scale Scientific Software Stack (E4S) is a community effort to provide open source software packages for developing, deploying and running scientific applications on high-performance computing (HPC) platforms. E4S provides from-source builds and containers of a broad collection of HPC software packages.

Purpose

E4S exists to accelerate the development, deployment and use of HPC software, lowering the barriers for HPC users. E4S provides containers and turn-key, from-source builds of more than 80 popular HPC products in programming models, such as MPI; development tools such as HPCToolkit, TAU and PAPI; math libraries such as PETSc and Trilinos; and Data and Viz tools such as HDF5 and Paraview.

Approach

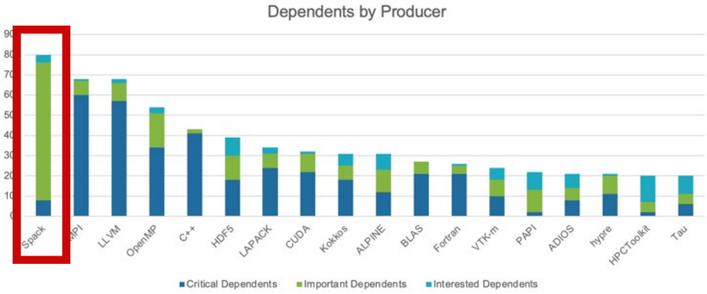
By using Spack as the meta-build tool and providing containers of pre-built binaries for Docker, Singularity, Shifter and CharlieCloud, E4S enables the flexible use and testing of a large collection of reusable HPC software packages.

Platforms

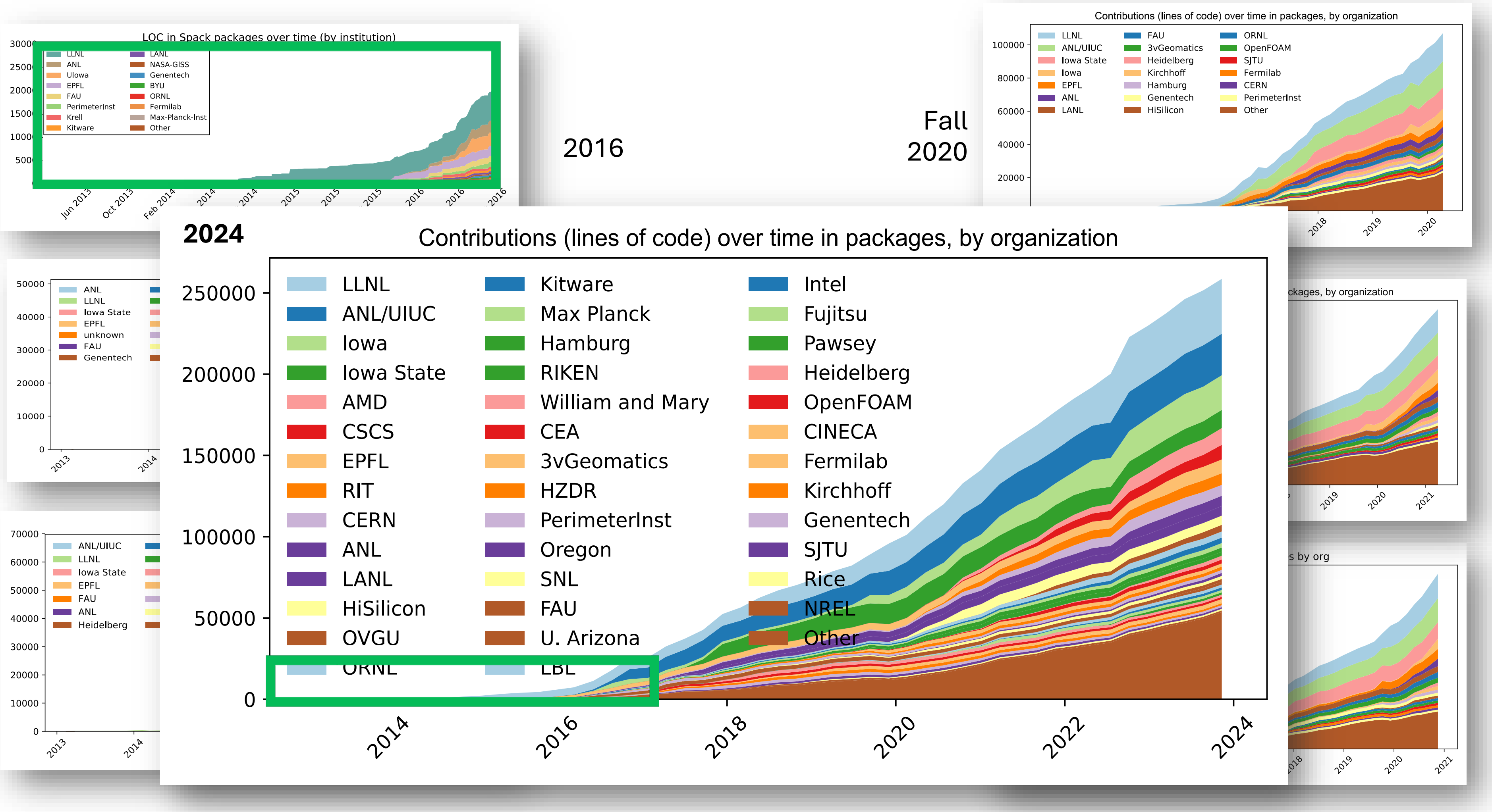
Testing

https://e4s.io

- Used for building software on the three U.S. exascale systems
- ECP built the Extreme Scale Scientific Software Stack (E4S) with Spack – more at https://e4s.io
- Project continues on ASC and ASCR funding



Spack was the most depended-upon project in ECP



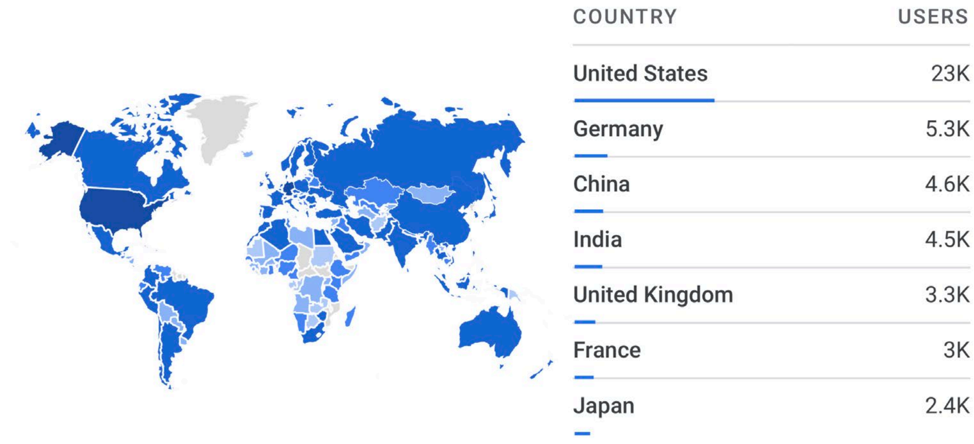
2016

Fall
2020

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

Over 8,000 software packages

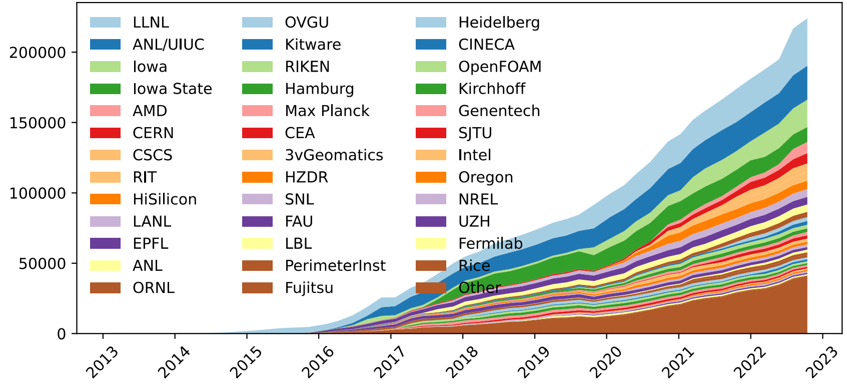
Over 1,300 contributors



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)

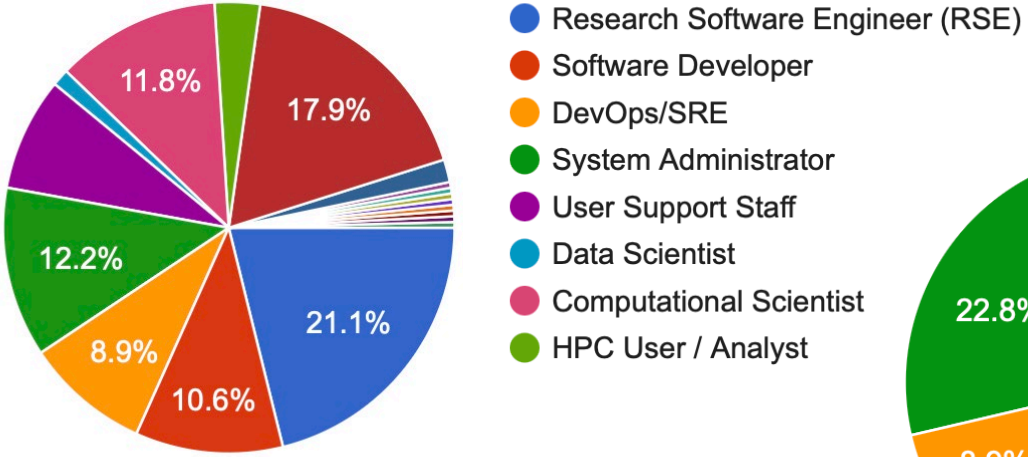
Contributions (lines of code) over time in packages, by organization



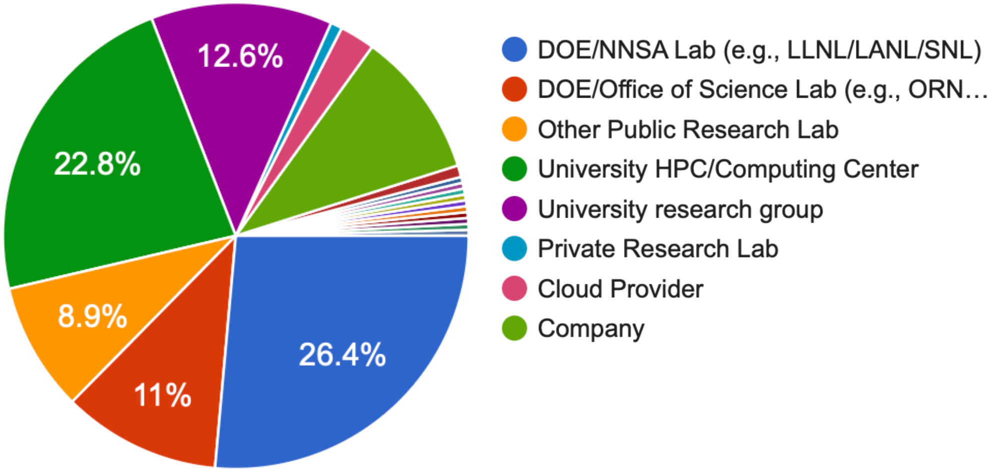
Contributors continue to grow worldwide!

Spack users have diverse roles across many types of institutions

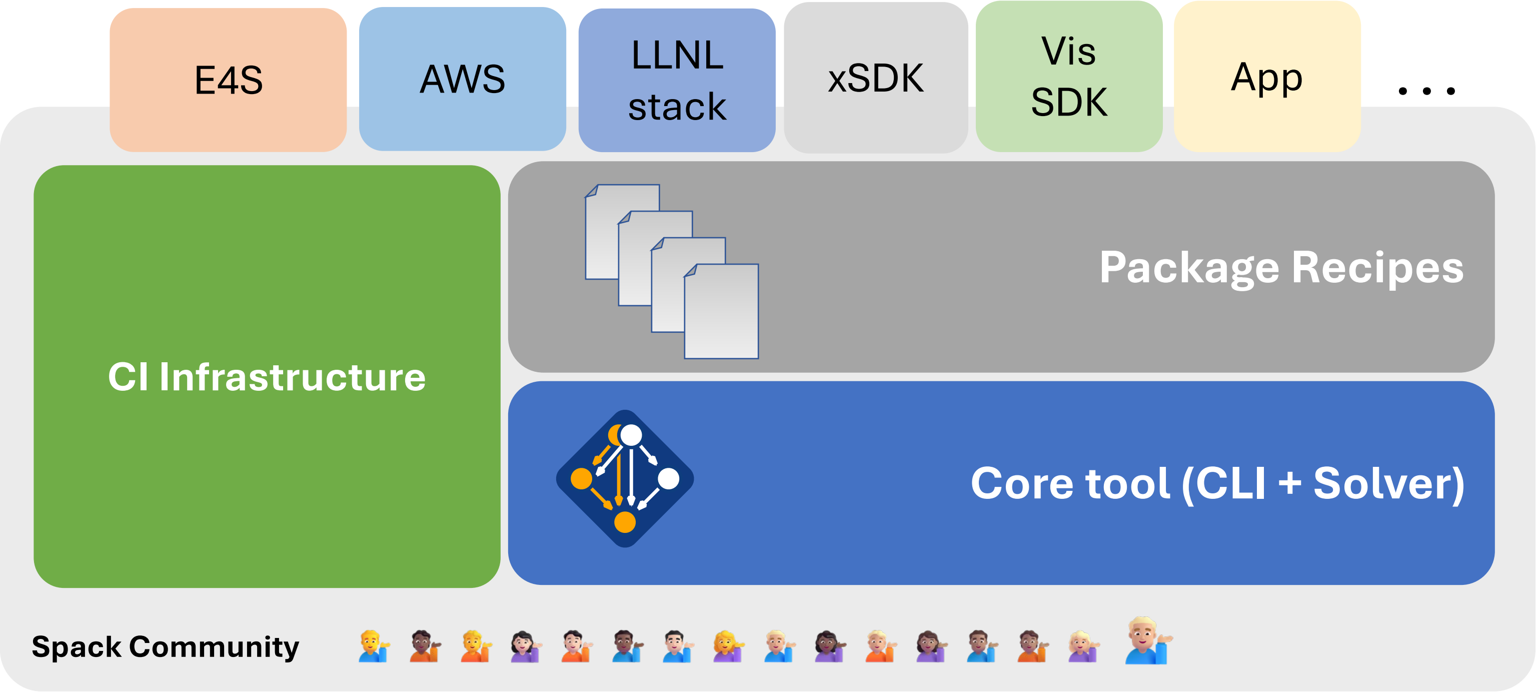
What type of user are you?



Where do you work?



What does the Spack project look like now?



E4S

AWS

LLNL stack

xSDK

Vis SDK

App

. . .

CI Infrastructure

Package Recipes

Core tool (CLI + Solver)

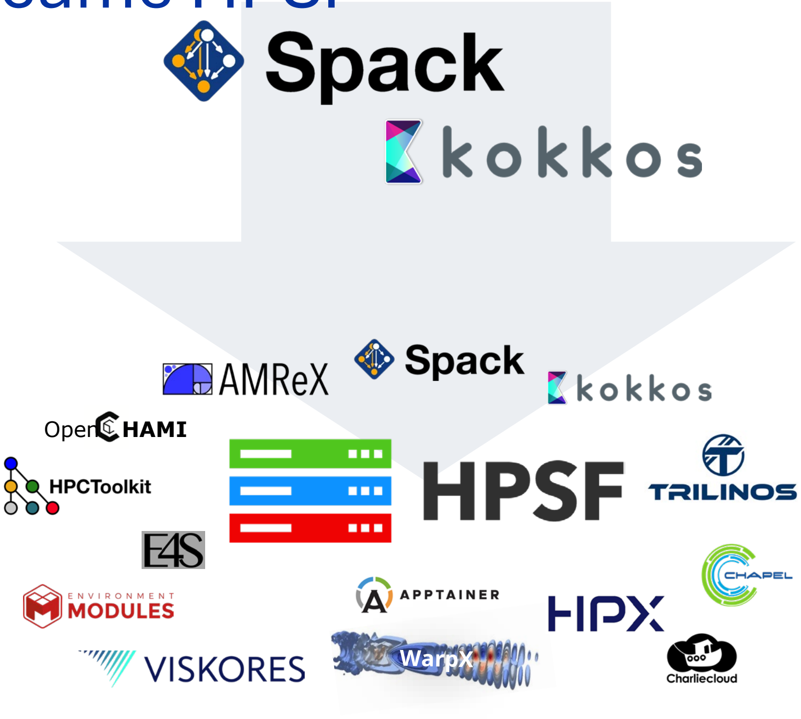
Spack Community

We started conversations with Linux Foundation in December 2021, and talked through mid-2022

- We wanted:
  - A neutral project home
    - To encourage more participation in the project
  - A way to fund project activities:
    - More continuous integration resources
    - User meetings, Slack, etc.
- Talked to LF onboarding team
- Learned about LF's basic requirements:
  - Technical charter
  - Open Governance
  - Trademark assignment

We joined forces with Kokkos to start a larger umbrella, which eventually became HPSF

- Spack and Kokkos were two of the most adopted projects during ECP
- Enable performance portability at different levels
  - Spack: build level
  - Kokkos: application/runtime level
- Goals:
  - Leverage proven track record of community building
  - Leverage industry and labs' familiarity
  - Get more projects on board to build an umbrella organization



With HPSF, we've formalized our governance with the Technical Steering Committee

Todd Gamblin, LLNL
TSC Chair

Greg Becker
LLNL

Massimiliano Culpo
n.p. complete s.r.l

Tammy Dahlgren
LLNL

Wouter Deconinck
U. Manitoba

Ryan Krattiger
Kitware

Mark Krentel
Rice University

John Parent
Kitware

Marc Paterno
Fermilab

Luke Peyralans
U. Oregon

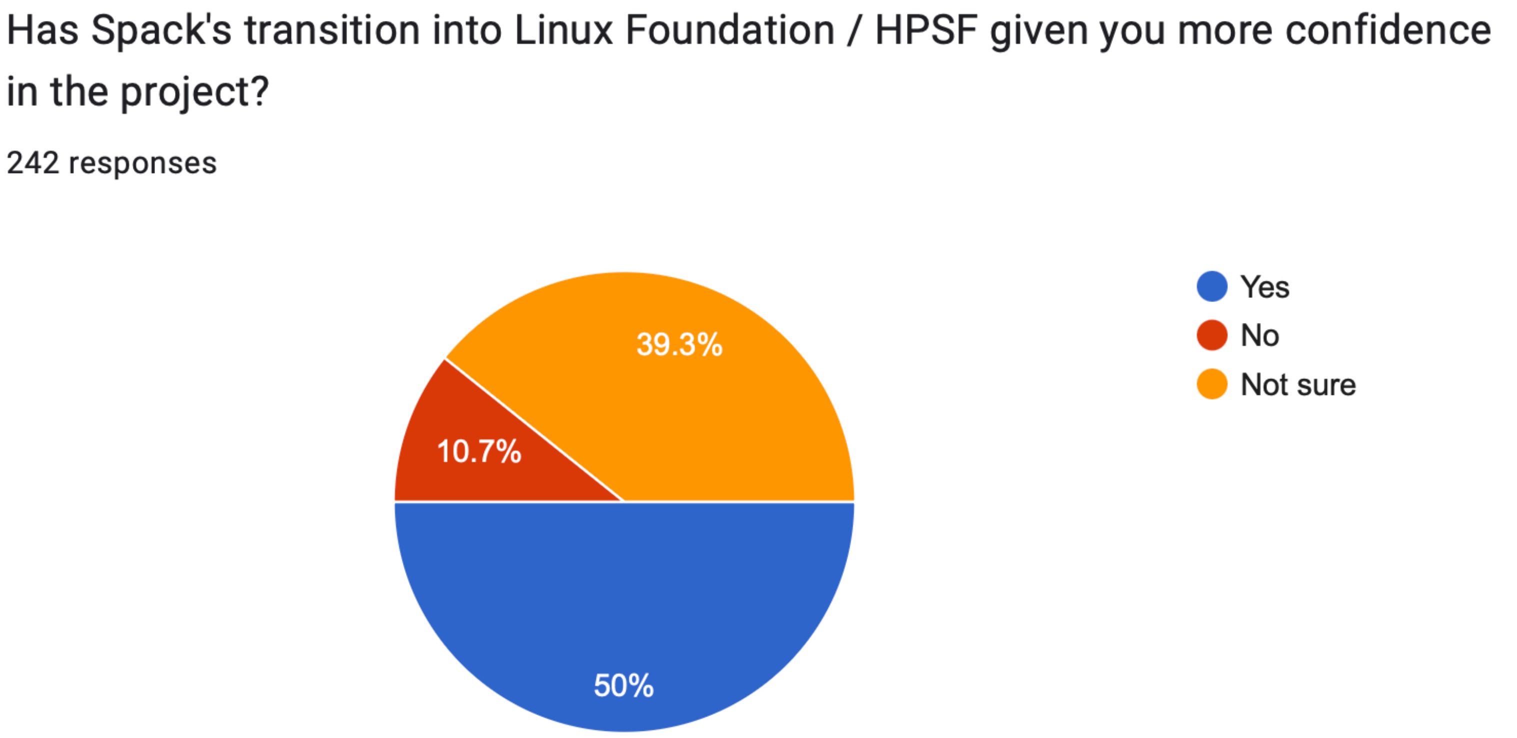
Phil Sakievich
Sandia

Peter Scheibel
LLNL

Adam Stewart
TU Munich

Harmen Stoppels
Stoppels Consulting

Response to LF/HPSF seems positive



Has Spack's transition into Linux Foundation / HPSF given you more confidence in the project?

242 responses

Spack Usage

Spack provides a *spec* syntax to describe customized installations

```
$ 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=zen2                  set target microarchitecture
$ spack install mpileaks@3.3 ^mpich@3.2 %gcc@4.9.3        ^ dependency information
```

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

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

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

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

```
from spack import *

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

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

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

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

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

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

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

Base package (CMake support)

Metadata at the class level

Versions

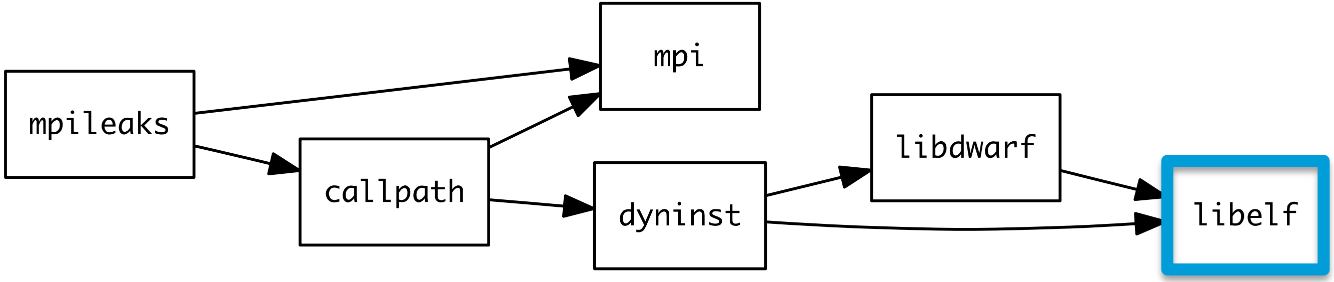
Variants (build options)

Dependencies (same spec syntax)

Install logic in instance methods

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

Spack Specs can constrain versions of dependencies

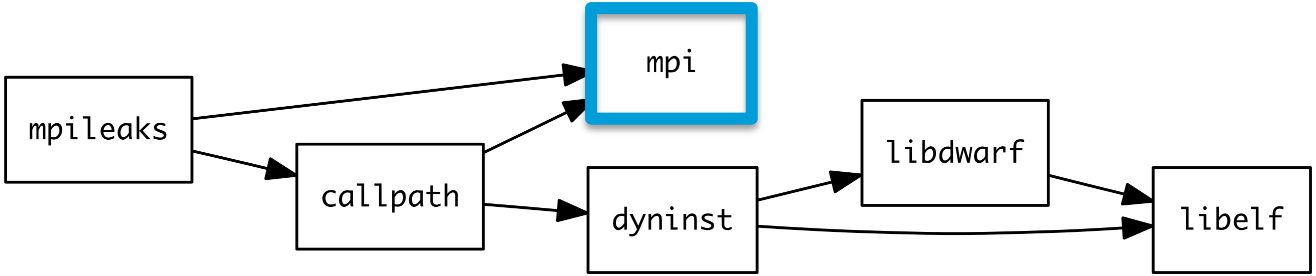


```
$ spack install mpileaks %intel@12.1 ^libelf@0.8.12
```

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

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

Spack handles ABI-incompatible, versioned interfaces like MPI



- mpi is a *virtual dependency*

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

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

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

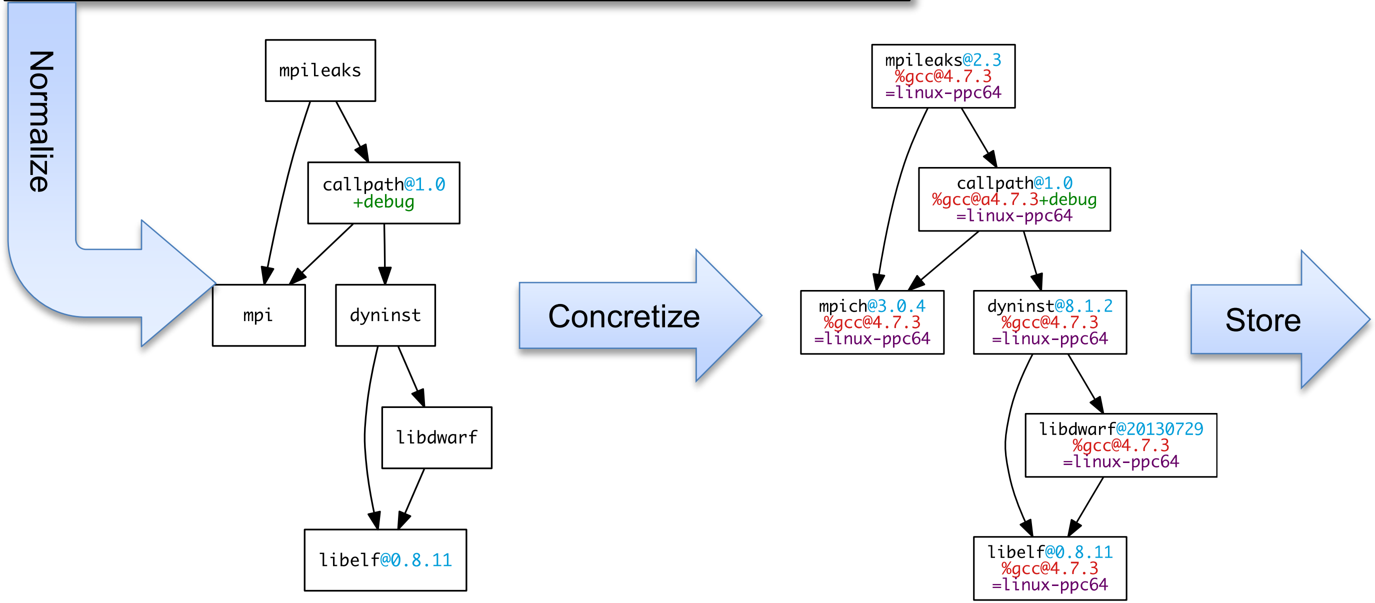
- Let Spack choose MPI implementation, as long as it provides MPI 2 interface:

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

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

mpileaks ^callpath@1.0+debug ^libelf@0.8.11

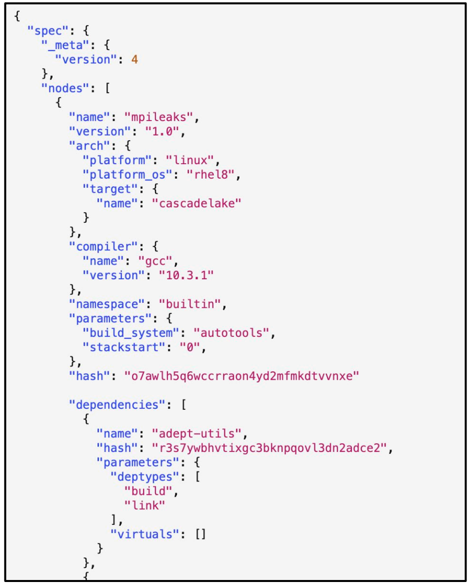
User input: *abstract* spec with some constraints



Abstract, normalized spec with some dependencies

Concrete spec is fully constrained and can be passed to install

spec.json

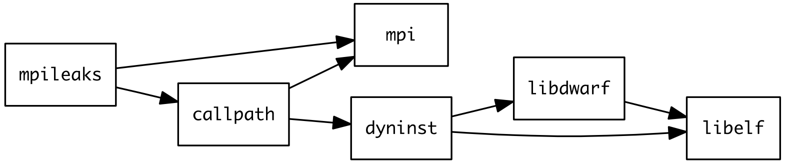


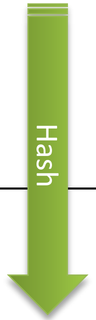
```
{
  "spec": {
    "_meta": {
      "version": 4
    },
    "nodes": [
      {
        "name": "mpileaks",
        "version": "1.0",
        "arch": {
          "platform": "linux",
          "platform_os": "rhel8",
          "target": {
            "name": "cascadelake"
          }
        },
        "compiler": {
          "name": "gcc",
          "version": "10.3.1"
        },
        "namespace": "builtin",
        "parameters": {
          "build_system": "autotools",
          "stackstart": "0",
        },
        "hash": "o7awlh5q6wccrraon4yd2mfmkdtvvnxe"

        "dependencies": [
          {
            "name": "adept-utils",
            "hash": "r3s7ywbhvtixgc3bknpqovl3dn2adce2",
            "parameters": {
              "deptypes": [
                "build",
                "link"
              ],
              "virtuals": []
            }
          },
          {
```

Detailed provenance stored with installed package

Spack handles combinatorial software complexity

Dependency DAG





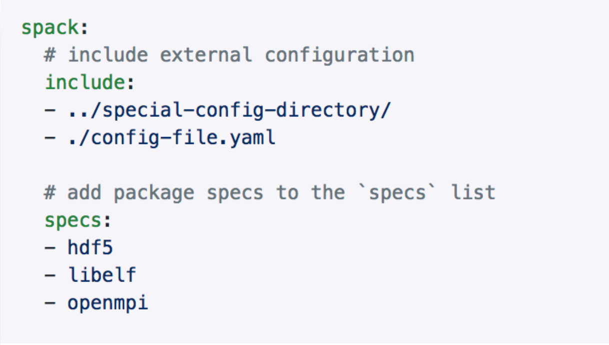
Installation Layout

```
opt
└── spack
    ├── linux-rhel7-skylake
    │   └── gcc-8.3.0
    │       ├── mpileaks-1.0-hc4sm4vuzpm4znmvrfzri4ow2mkphe2e
    │       ├── callpath-1.0.4-daqqpssxb6qbfrztsezkmhus3xoflbsy
    │       ├── openmpi-4.1.4-u64v26igxvxyn23hysmklfums6tgjv5r
    │       ├── dyninst-12.1.0-u64v26igxvxyn23hysmklfums6tgjv5r
    │       ├── libdwarf-20180129-u5eawkvaoc7vonabe6nndkcfwuv233cj
    │       └── 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*

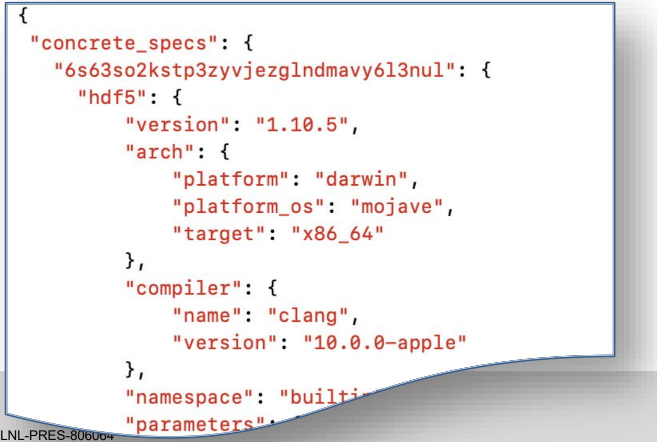
Spack environments enable users to build customized stacks from an abstract description

Simple spack.yaml file

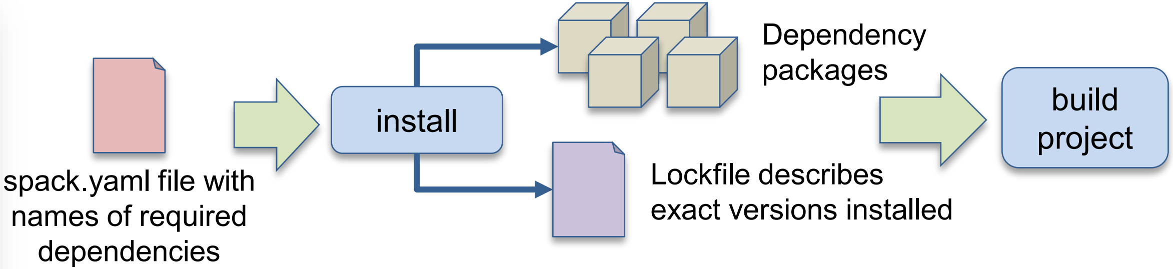


```
spack:
  # include external configuration
  include:
  - ../special-config-directory/
  - ./config-file.yaml

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

Concrete spack.lock file (generated)



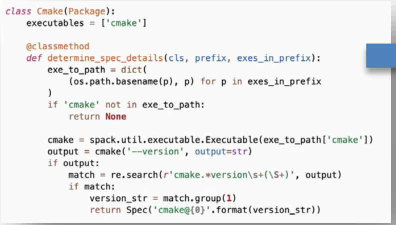
```
{
 "concrete_specs": {
   "6s63so2kstp3zyvjezglndmavy6l3nul": {
     "hdf5": {
         "version": "1.10.5",
         "arch": {
             "platform": "darwin",
             "platform_os": "mojave",
             "target": "x86_64"
         },
         "compiler": {
             "name": "clang",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



spack.yaml file with names of required dependencies
install
Dependency packages
Lockfile describes exact versions installed
build project

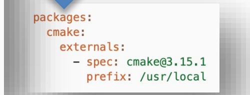
- spack.yaml describes project requirements
- spack.lock describes exactly what versions/configurations were installed, allows them to be reproduced.
- Can also be used to maintain configuration together with Spack packages.
  - E.g., versioning your own local software stack with consistent compilers/MPI implementations
  - Allows developers and site support engineers to easily version Spack configurations in a repository

Environments have enabled us to add build many features to support developer workflows



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

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

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

package.py



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

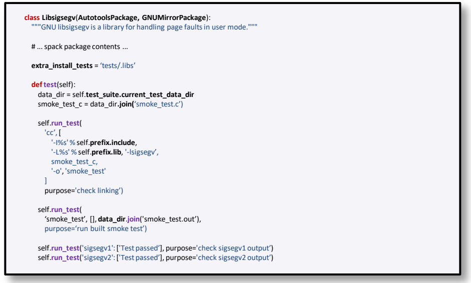
spack.yaml configuration

spack external find

Automatically find and configure external packages on the system

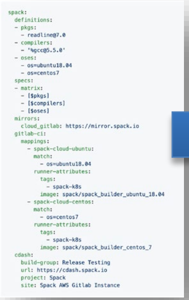
spack test

Packages know how to run their own test suites



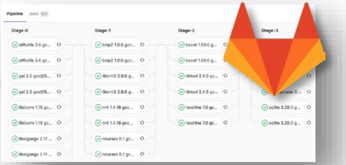
```
class Libsigsegv(AutotoolsPackage, GNUMirrorPackage):
    """GNU libsigsegv is a library for handling page faults in user mode."""

    # ... spack package contents ...

    extra_install_tests = 'tests/.libs'

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

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

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

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

package.py



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

spack.yaml



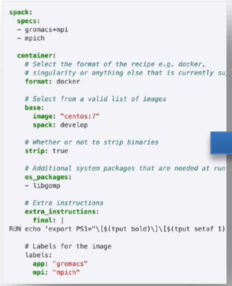
.gitlab-ci.yml CI pipeline

spack ci

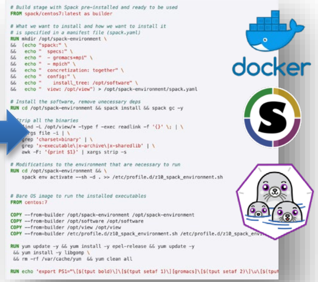
Automatically generate parallel build pipelines
(more on this later)

spack containerize

Turn environments into container build recipes



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

  container:
    # Select the format of the recipe e.g. docker,
    # singularity or anything else that is currently su
    format: docker

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

    # Whether or not to strip binaries
    strip: true

    # Additional system packages that are needed at run
    os_packages:
    - libgomp

    # Extra instructions
    extra_instructions:
      final: |
RUN echo 'export PS1="\[$(tput bold)\]\[$(tput setaf 1)

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



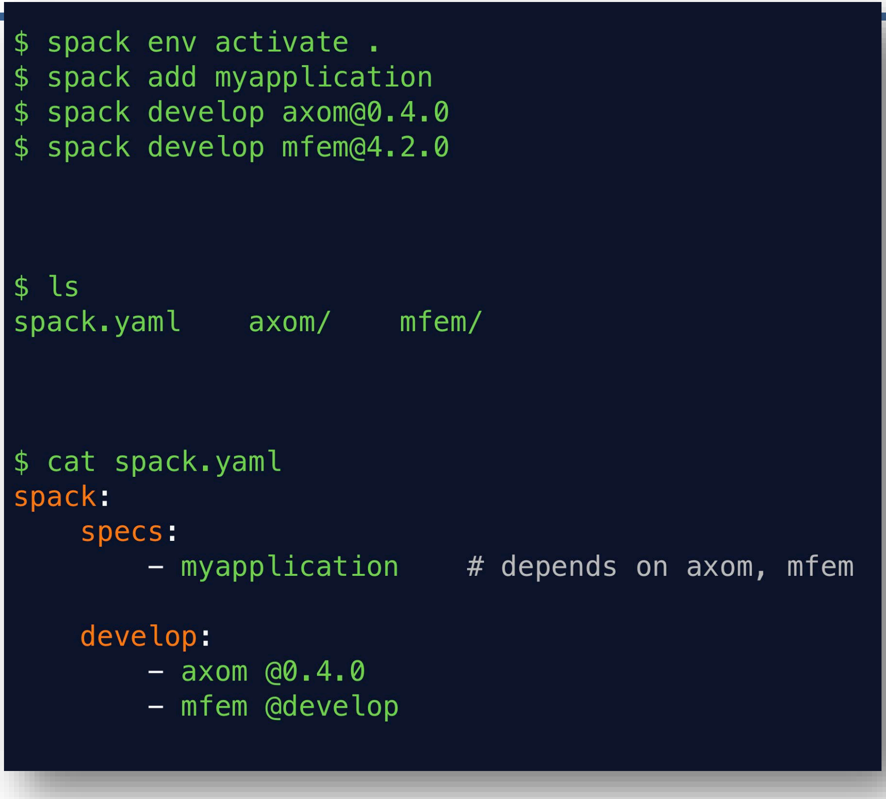
```
# Build stage with Spack pre-installed and ready to be used
FROM spack/centos7:latest as builder

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

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

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

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


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

COPY --from=builder /opt/spack-environment /opt/spack-environment
COPY --from=builder /opt/software /opt/software
COPY --from=builder /opt/view /opt/view
COPY --from=builder /etc/profile.d/z10_spack_environment.sh /etc/profile.d/z10_spack_envi

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

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

spack develop lets developers work on many packages at once

- Developer features so far have focused on single packages (spack dev-build, etc.)

- New spack develop feature enables development environments
  - Work on a code
  - Develop multiple packages from its dependencies
  - Easily rebuild with changes

- Builds on spack environments
  - Required changes to the installation model for dev packages
  - dev packages don't change paths with configuration changes
  - Allows devs to iterate on builds quickly



```
$ spack env activate .
$ spack add myapplication
$ spack develop axom@0.4.0
$ spack develop mfem@4.2.0
```

```
$ ls
spack.yaml    axom/    mfem/
```

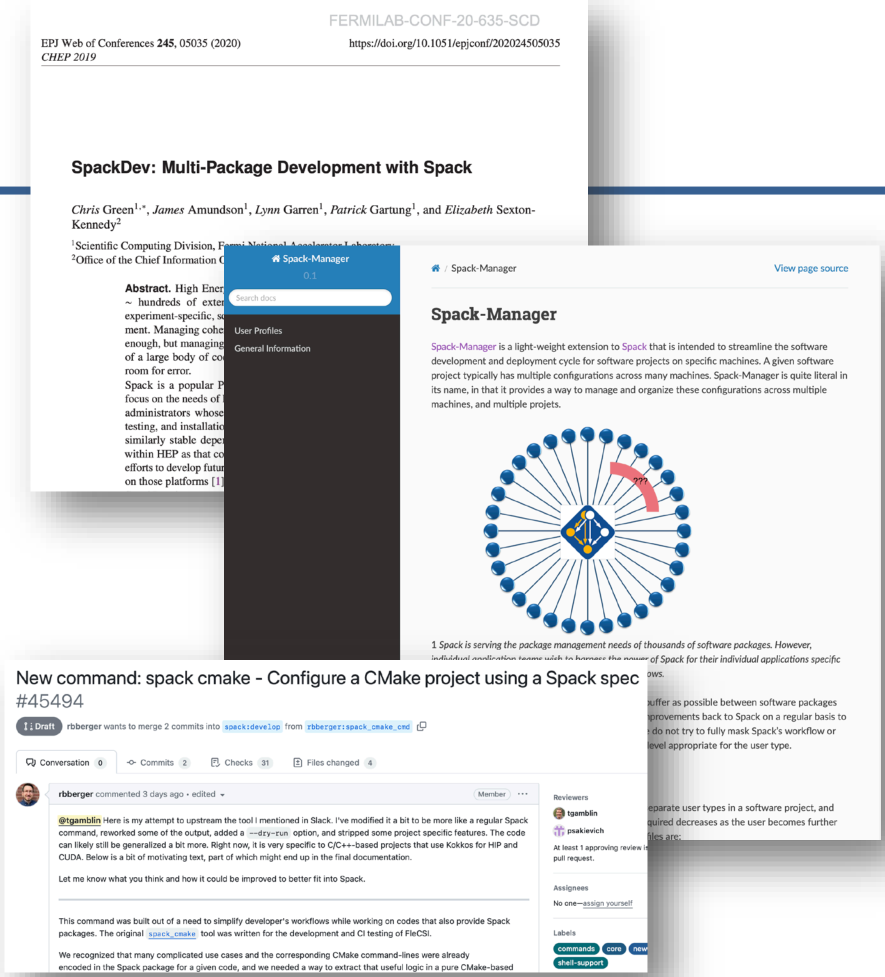
```
$ cat spack.yaml
spack:
    specs:
        - myapplication    # depends on axom, mfem

    develop:
        - axom @0.4.0
        - mfem @develop
```

Teams are building development front-ends top of on Spack

- Many different approaches:
  - Thin **mack** wrapper for MARBL team at LLNL
  - **spack-manager** at Sandia
  - **SpackDev** at Fermilab
  - **spack-organizer** at CEA
  - **spack_cmake** at LANL

- Workflow seems to be converging around environments + spack develop

- We are trying to bring many of the features from these scripts into core
  - Most of the front-ends are opinionated in one way or another
  - We want spack to support but not force these workflows



FERMILAB-CONF-20-635-SCD

EPJ Web of Conferences **245**, 05035 (2020)
CHEP 2019

https://doi.org/10.1051/epjconf/202024505035

SpackDev: Multi-Package Development with Spack

Chris Green[1,*], *James* Amundson[1], *Lynn* Garren[1], *Patrick* Gartung[1], and *Elizabeth* Sexton-Kennedy[2]

[1]Scientific Computing Division, F
[2]Office of the Chief Information O

Abstract. High Ener
~ hundreds of exter
experiment-specific, so
ment. Managing cohe
enough, but managing
of a large body of co
room for error.
Spack is a popular P
focus on the needs of
administrators whose
testing, and installatio
similarly stable deper
within HEP as that co
efforts to develop futu
on those platforms [1]

Spack-Manager

Spack-Manager is a light-weight extension to Spack that is intended to streamline the software development and deployment cycle for software projects on specific machines. A given software project typically has multiple configurations across many machines. Spack-Manager is quite literal in its name, in that it provides a way to manage and organize these configurations across multiple machines, and multiple projets.

1 *Spack is serving the package management needs of thousands of software packages. However, individual application teams wish to harness the power of Spack for their individual applications specific* …ows.

…uffer as possible between software packages
…provements back to Spack on a regular basis to
… do not try to fully mask Spack's workflow or
…level appropriate for the user type.

…eparate user types in a software project, and
…quired decreases as the user becomes further
…files are:

New command: spack cmake - Configure a CMake project using a Spack spec #45494

rbberger wants to merge 2 commits into `spack:develop` from `rbberger:spack_cmake_cmd`

rbberger commented 3 days ago • edited

@tgamblin Here is my attempt to upstream the tool I mentioned in Slack. I've modified it a bit to be more like a regular Spack command, reworked some of the output, added a `--dry-run` option, and stripped some project specific features. The code can likely still be generalized a bit more. Right now, it is very specific to C/C++-based projects that use Kokkos for HIP and CUDA. Below is a bit of motivating text, part of which might end up in the final documentation.

Let me know what you think and how it could be improved to better fit into Spack.

This command was built out of a need to simplify developer's workflows while working on codes that also provide Spack packages. The original `spack_cmake` tool was written for the development and CI testing of FleCSI.

We recognized that many complicated use cases and the corresponding CMake command-lines were already encoded in the Spack package for a given code, and we needed a way to extract that useful logic in a pure CMake-based

Reviewers
tgamblin
psakievich
At least 1 approving review i… pull request.

Assignees
No one—assign yourself

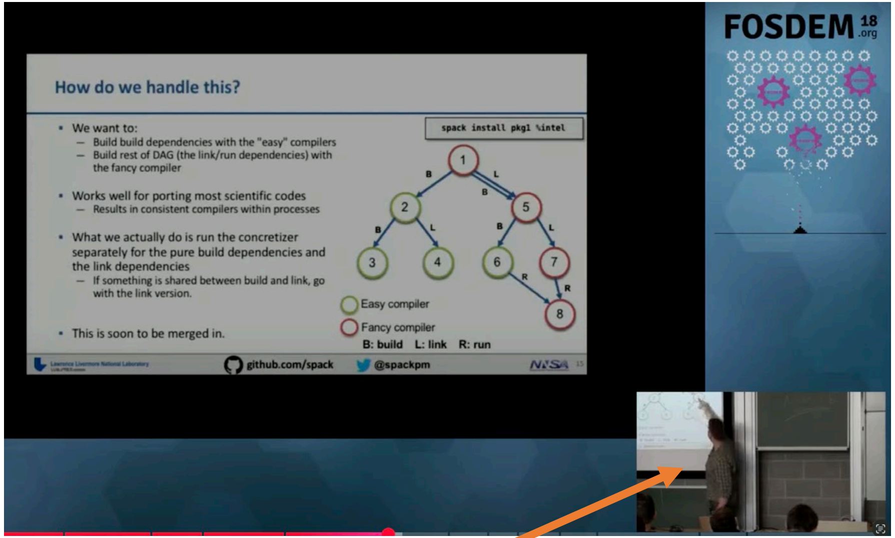
The Road to Spack v1.0

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.



How do we handle this?

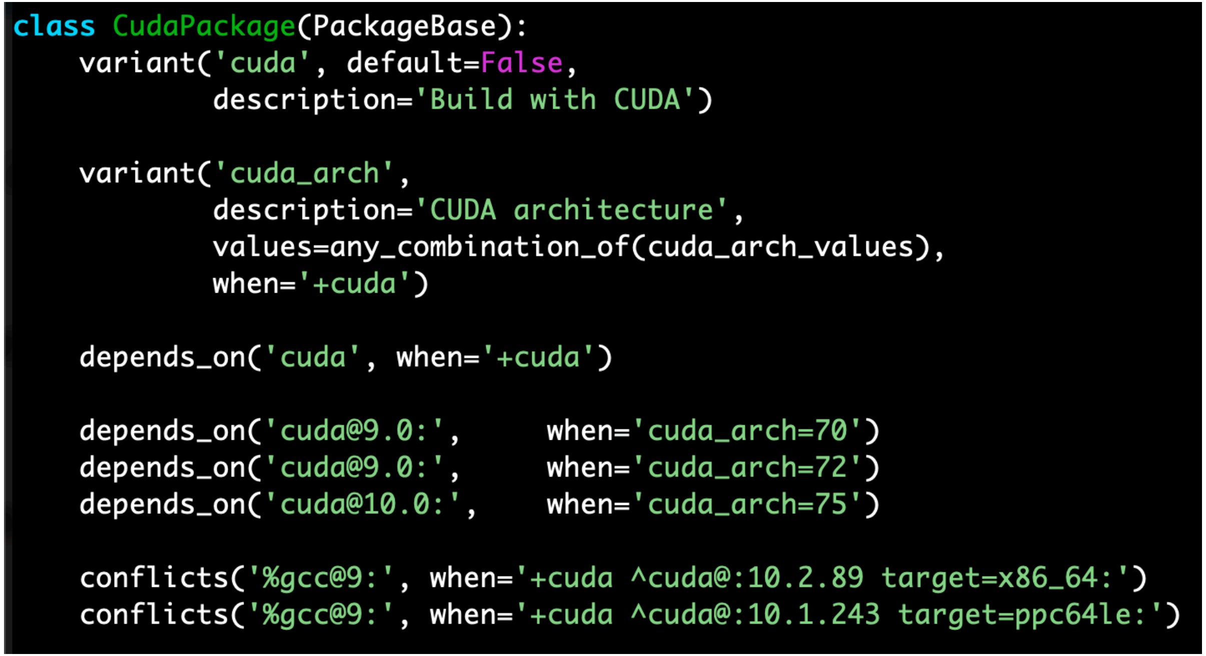
- We want to:
  - Build build dependencies with the "easy" compilers
  - Build rest of DAG (the link/run dependencies) with the fancy compiler
- Works well for porting most scientific codes
  - Results in consistent compilers within processes
- What we actually do is run the concretizer separately for the pure build dependencies and the link dependencies
  - If something is shared between build and link, go with the link version.
- This is soon to be merged in.

```
spack install pkg1 %intel
```

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

Spack packages use a *lot* of (declarative) conditional logic

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

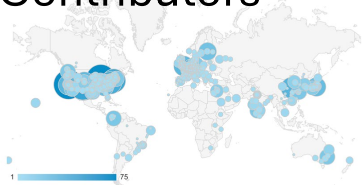
constraints on cuda version

compiler support for x86_64
and ppc64le

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

First challenge: we needed a new concretizer to model the expressiveness of the DSL

Contributors



- new versions
- new dependencies
- new constraints

package.py repository

This part is NP-hard!

concretizer

spack developers

yaml

default config packages.yaml

admins, users

yaml

local preferences config packages.yaml

users

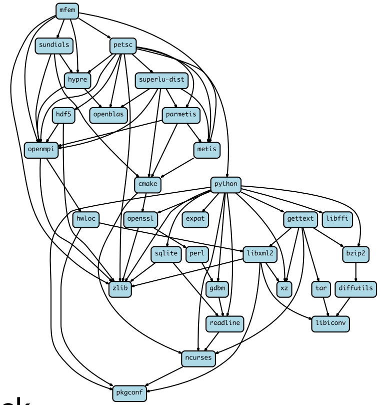
yaml

l environment config spack.yaml

users

Command line constraints

```
spack install hdf5@1.12.0 +debug
```

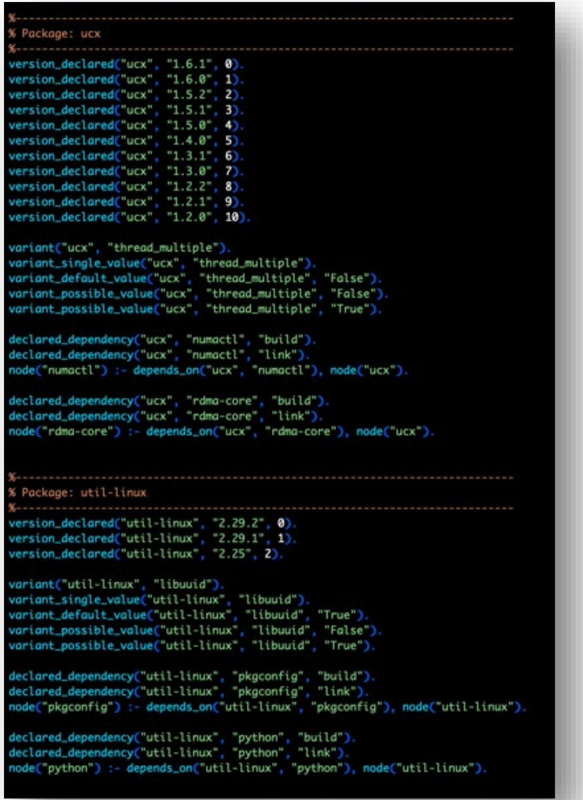


Concrete spec is fully constrained and can be built.

Is stored in spack.lock file after solve.

We reimplemented Spack’s concretizer using *Answer Set Programming (ASP)*

- Originally a greedy, custom Python algorithm
- ASP is a *declarative* programming paradigm
  - Looks like Prolog
  - Built around modern CDCL SAT solver techniques
- ASP program has 2 parts:
  1. Large list of facts generated from recipes (problem instance)
  2. Small logic program (~700 lines of ASP code)
- Algorithm is conceptually simpler:
  - Generate facts for all possible dependencies
  - Send facts and our logic program to the solver
  - Read results and rebuild the resolved DAG
- Using **Clingo**, the Potassco grounder/solver package



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

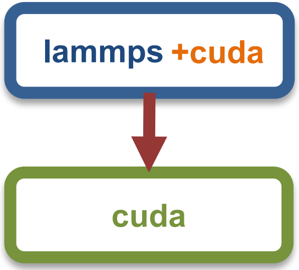
Some facts for HDF5 package

Spack’s concretizer is implemented using Answer Set Programming (ASP)

ASP looks like Prolog but is converted to SAT with optimization

Facts describe the graph

```
node("lammps").
node("cuda").
variant_value("lammps", "cuda", "True").
depends_on("lammps", "cuda").
```



First-order rules (with variables) describe how to resolve nodes and metadata

```
node(Dependency) :- node(Package), depends_on(Package, Dependency).
```

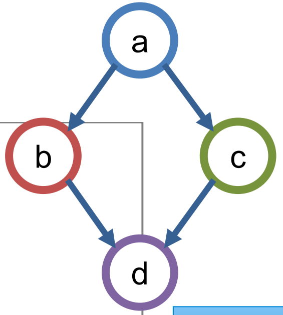
```
node("mpi")
```

←

```
node("hdf5").
depends_on("hdf5", "mpi").
```

Ground Rule

Grounding converts a first-order logic program into a propositional logic program, which can be solved.



```
depends_on(a, b).
depends_on(a, c).
depends_on(b, d).
depends_on(c, d).

node(Dep)
  :- node(Pkg),
     depends_on(Pkg, Dep).

% at least one is true
1 { node(a); node(b) }.
```

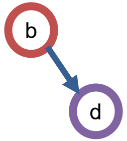
First-order Logic Program

ound

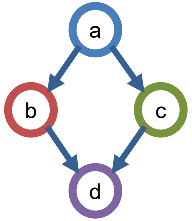
```
depends_on(a, b).
depends_on(a, c).
depends_on(b, d).
depends_on(c, d).

node(b) :- node(a).
node(c) :- node(a).
node(d) :- node(c).
node(d) :- node(b).

% at least one is true
1 { node(a); node(b) }.
```

Propositional Program

```
Answer 1:
node(b)
node(d)
```



```
Answer 2:
node(a)
node(b)
node(c)
node(d)
```



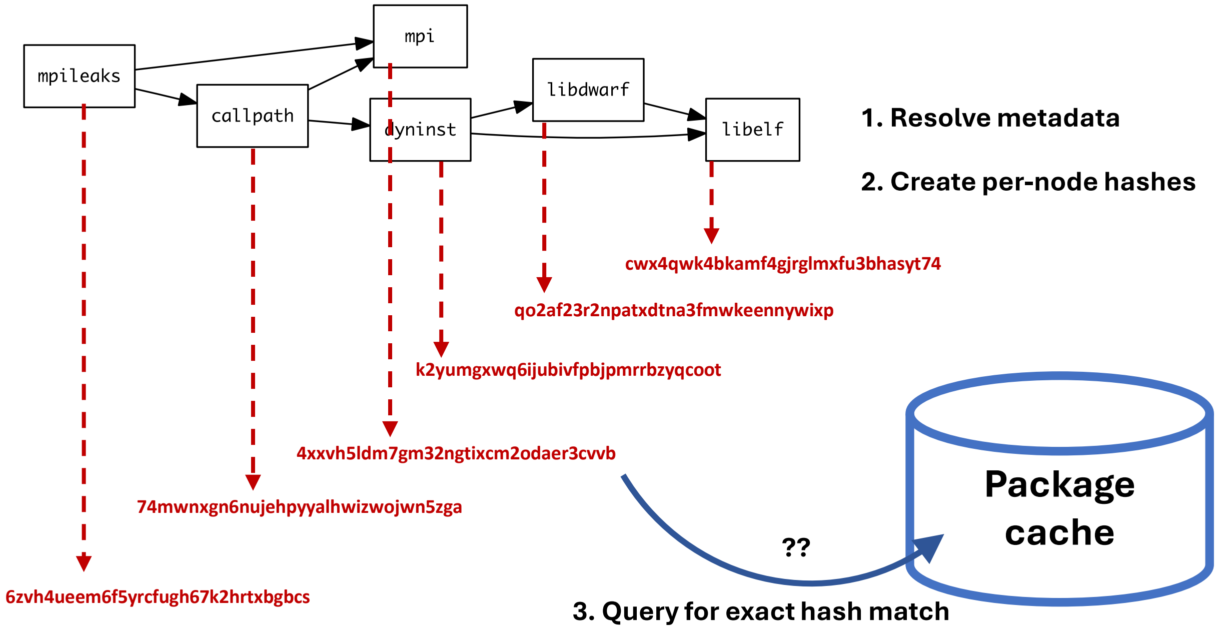
Stable Models (Answer Sets)

Answer 1: Only node(b) is true
Answer 2: Both node(a) and node(b) are true

ASP searches for *stable models* of the input program

- Stable models are also called ***answer sets***
- A ***stable model*** (loosely) is a set of true atoms that can be deduced from the inputs, where every rule is idempotent.
  - Similar to fixpoints
  - Put more simply: a *set of atoms where all your rules are true!*
- Unlike Prolog:
  - Stable models contain everything that can be derived (vs. just querying values)
  - Good ways to do optimization to select the “best” stable model
  - ASP is guaranteed to complete!

Second challenge: Spack's original concretizer did not reuse existing installations



mpileaks
mpi
callpath
dyninst
libdwarf
libelf
cwx4qwk4bkamf4gjrglmxfu3bhasyt74
qo2af23r2npatxdtna3fmwkeennywixp
k2yumgxwq6ijubivfpbjpmrrbzyqcoot
4xxvh5ldm7gm32ngtixcm2odaer3cvvb
74mwnxgn6nujehpyyalhwizwojwn5zga
6zvh4ueem6f5yrcfugh67k2hrtxbgbcs
??
Package cache

1. Resolve metadata

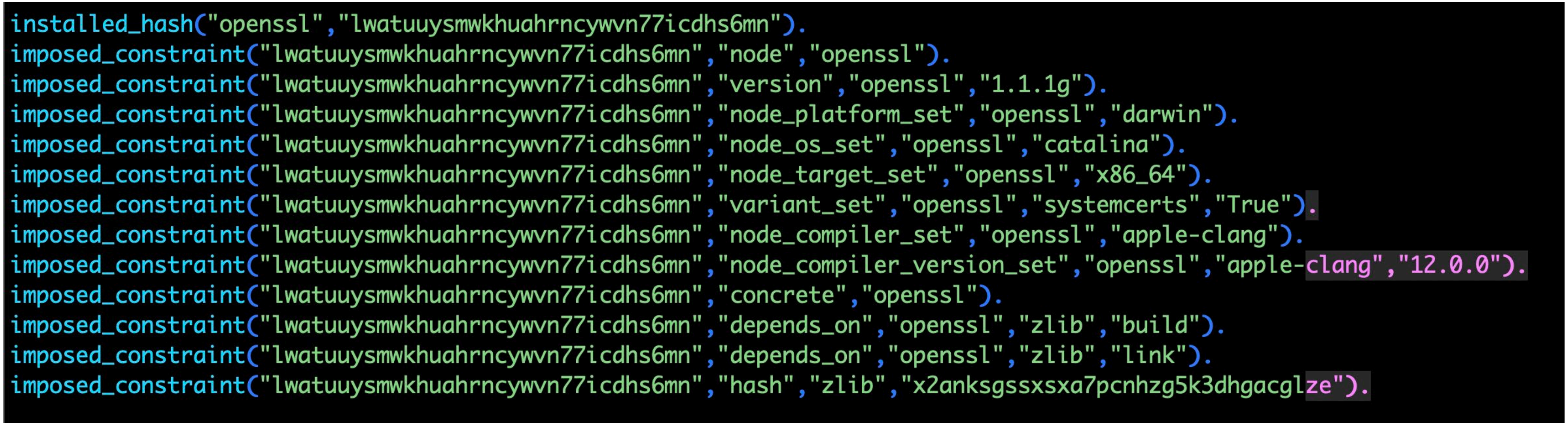
2. Create per-node hashes

3. Query for exact hash 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) was enabled by ASP

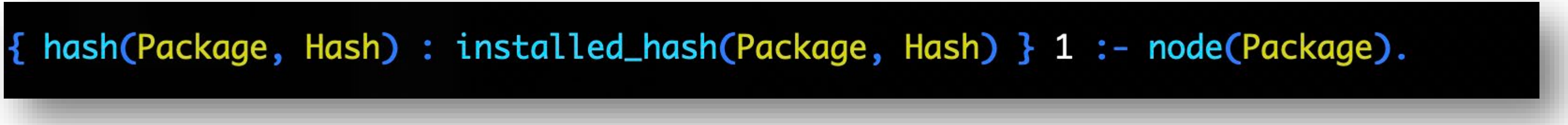
- --reuse tells the solver about all the installed packages!
- Add constraints for all installed packages, with their hash as the associated ID:



```
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").
```

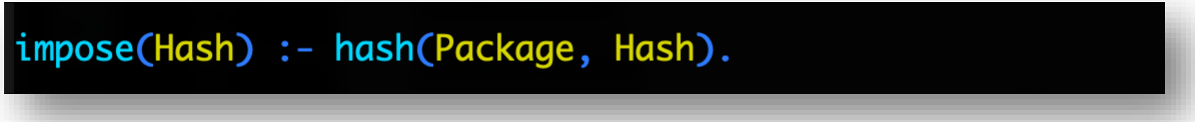
Minimizing builds is surprisingly simple in ASP

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



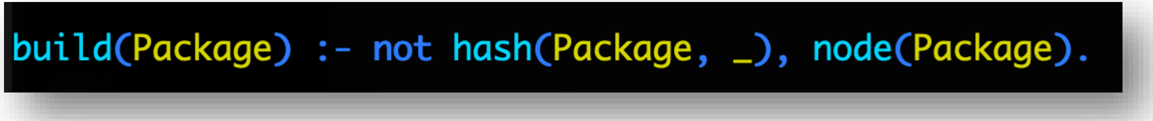
```
{ hash(Package, Hash) : installed_hash(Package, Hash) } 1 :- node(Package).
```

2. Choosing a hash means we impose its constraints:



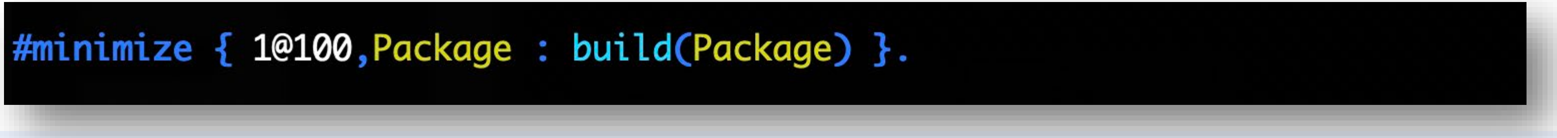
```
impose(Hash) :- hash(Package, Hash).
```

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



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

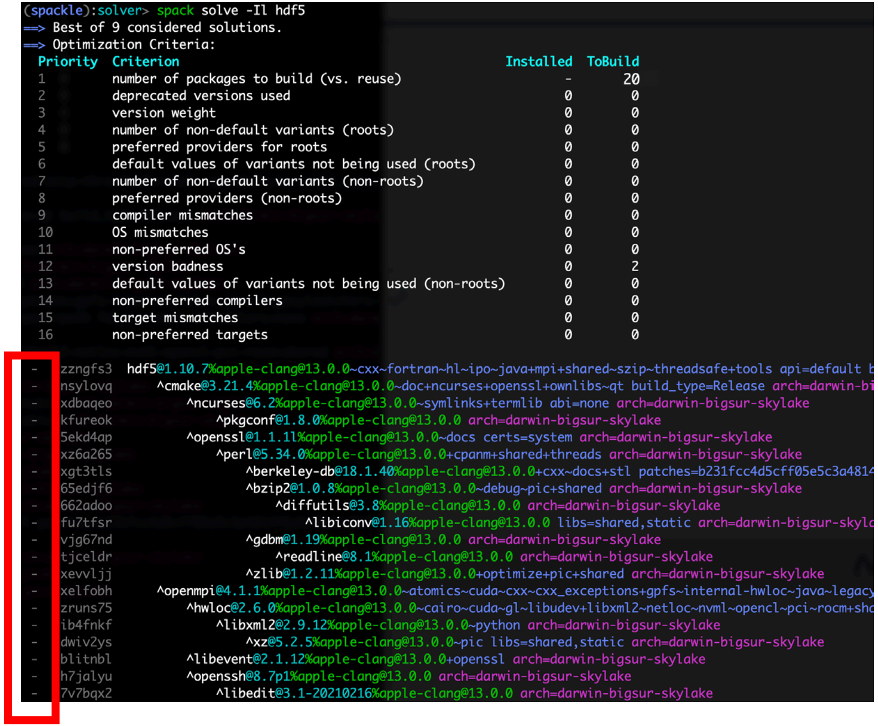
4. Minimize builds!



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

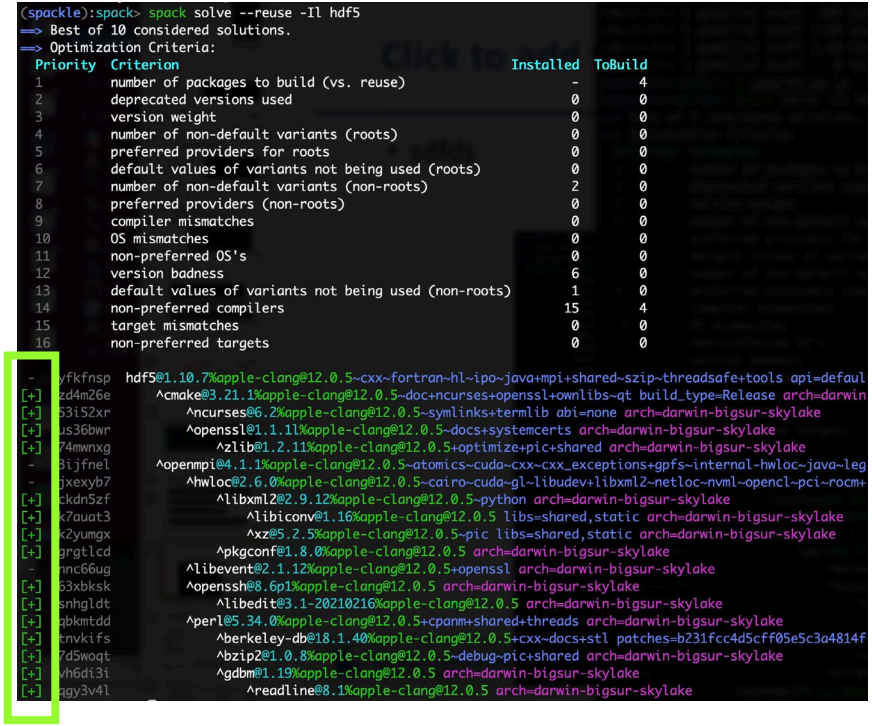
There's more to it than this, but you get the idea...

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+sha
 -   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
```

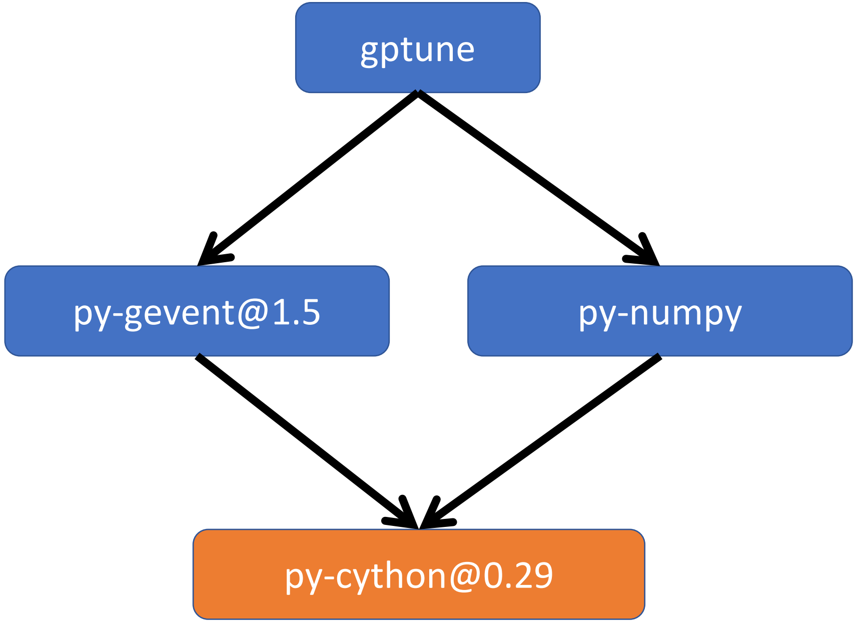
Pure hash-based reuse: all misses



```
(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~leg
 -   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
```

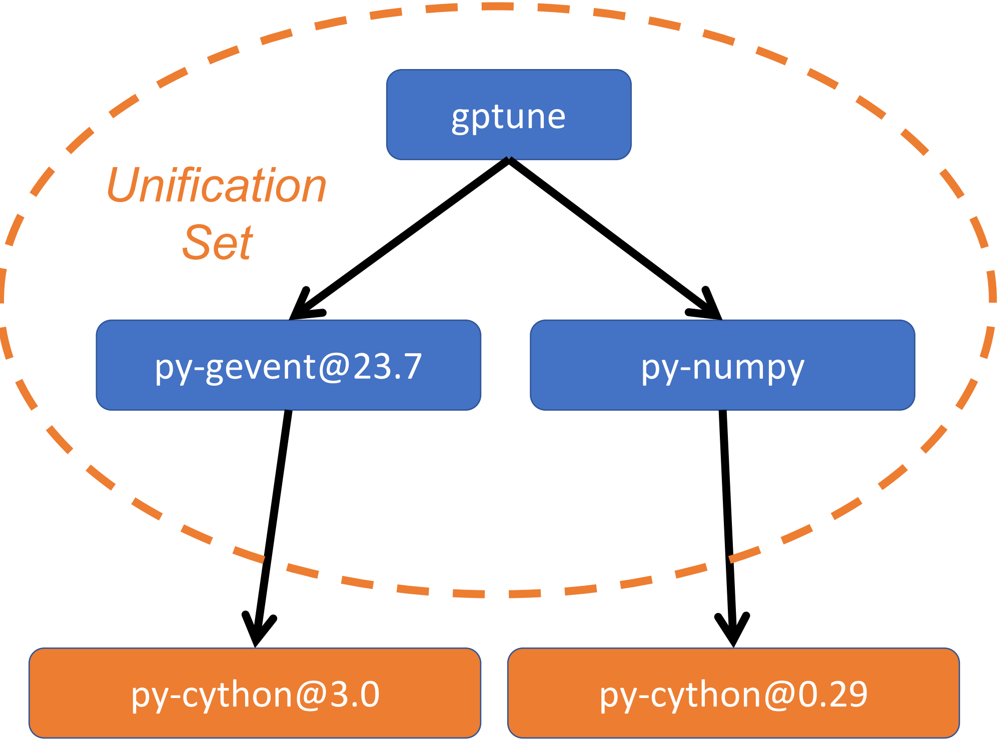
With reuse: 16 packages were reusable

Third challenge: we needed to allow multiple versions of build dependencies in the DAG



- Only one configuration per package allowed in the DAG
  - Ensures ABI compatibility but is too restrictive
- Needed to enable compiler mixing with compiler dependencies
- Also needed for Python ecosystem
  - In the example py-numpy needs to use py-cython@0.29 <u>as a build tool</u>
  - That enforces using an old py-gevent, because newer versions depend on py-cython@3.0 or greater

Objective: dependency splitting



- The constraint on build dependencies can be relaxed, without compromising the ABI compatibility
- Having a single configuration of a package is now enforced on unification sets
- These are the set of nodes used together at runtime (the one shown is for gptune)
- This allows us to use the latest version of py-gevent, because now we can have two versions of py-cython

We want to dynamically “split” nodes when needed

1. Start with deducing single dependency nodes:

```
node(DependencyName)
   :- dependency_holds(PkgName, DependencyName)
```

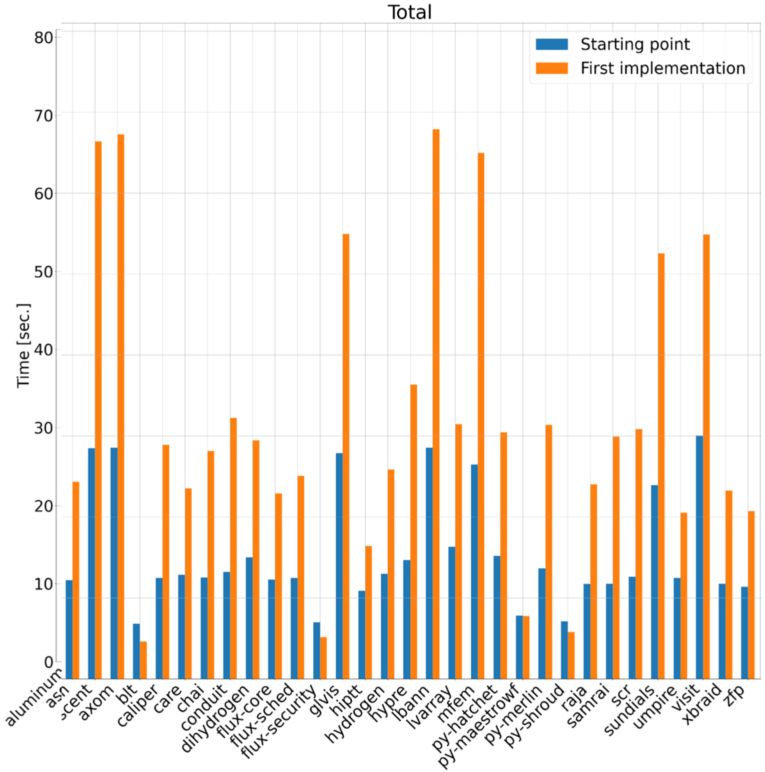
2. Allow solver to **choose** to duplicate a node:

Converted node identifier from **name** to **(name, id)**

```
   1 {
      depends_on(PkgNode, node(0..Y-1, DepNode), Type)
      : max_dupes(DepNode, Y)
   } 1
      :- dependency_holds(PkgNode, DepNode).
```

3. Re-encode package metadata so that it can be associated with duplicates

First try at allowing duplicates in a single solve



Increased runtimes by >> 2x in some cases

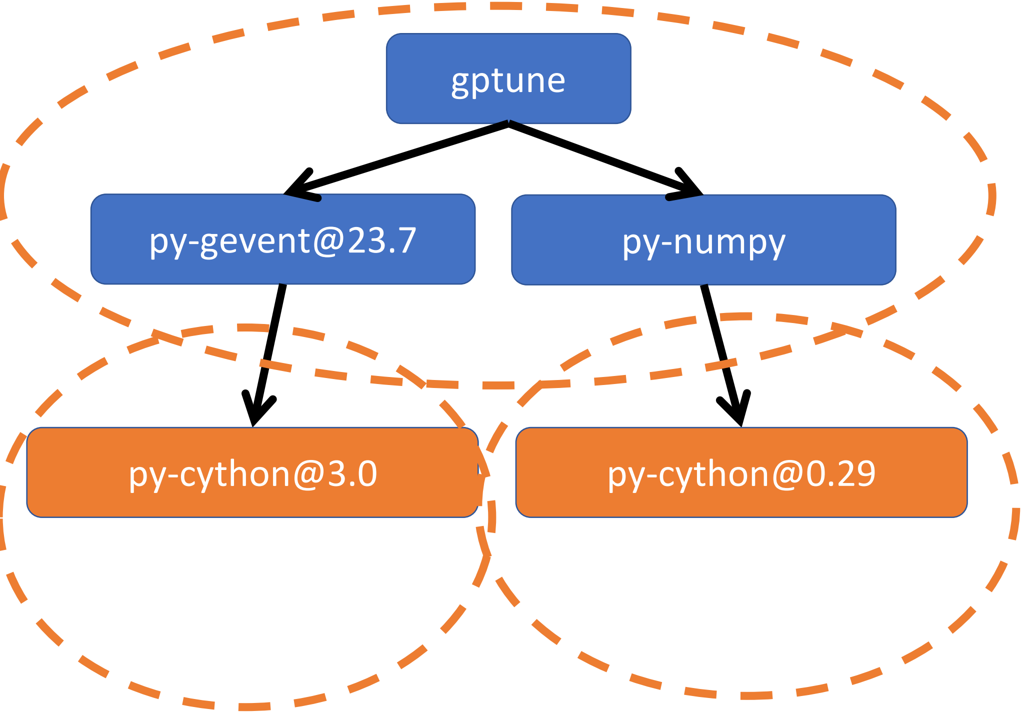
Cycle detection in the solver is *expensive*

```
path(A, B) :- depends_on(A, B).
path(A, C) :- path(A, B), depends_on(B, C).

% this constraint says "no cycles"
:- path(A, B), path(B, A).
```

- Has to maintain path() predicate representing paths between nodes
- Cycles are actually rare in solutions
  - Switched to post-processing for cycle detection
  - Only do expensive solve if a cycle is detected in a solution
- Eventually moved this calculation *into* the solver using some custom directives from the developers

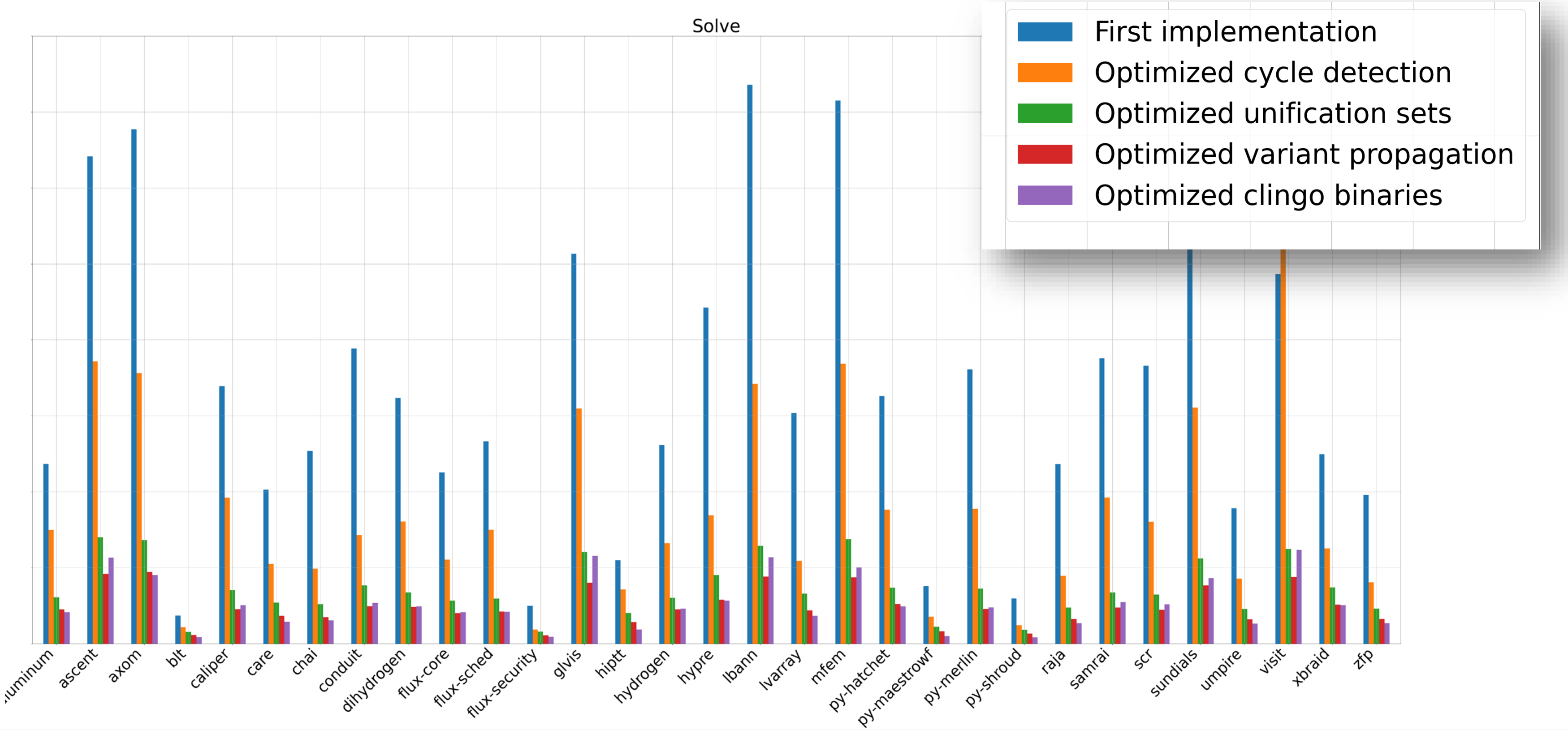
50%+ improvement in solve time

Unification sets can be expensive too

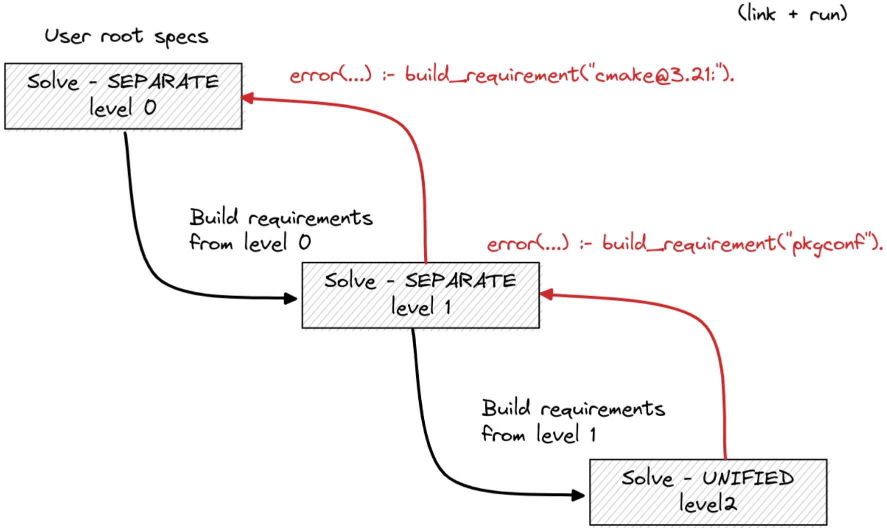


- Unification set creation was originally recursive for *any* build dependencies
  - Ends up blowing up grounding
- Mitigation:
  - Only create new sets for explicitly *marked* build tools
  - Transitive build dependencies that are not from marked build tools go into a *common* unification set
- Need better heuristics to split when necessary for full generality

Through many different optimizations, we were able to reclaim enough performance to make duplicate build dependencies tractable



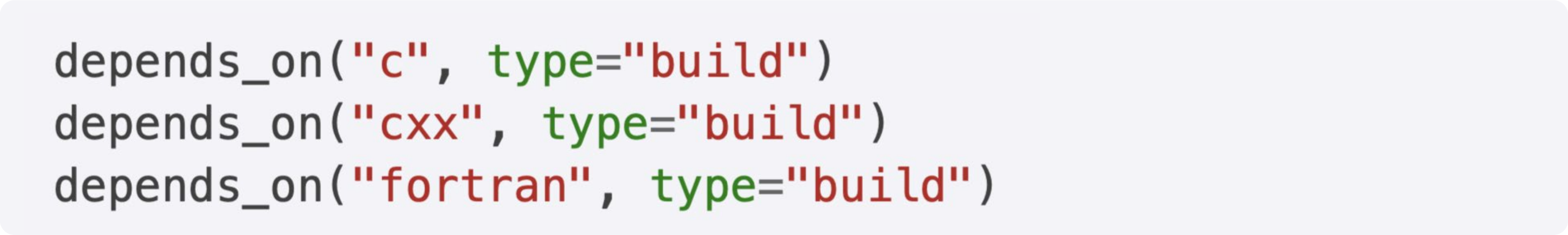
It was not trivial to find a model that was both performant *and* tightly coupled



(link + run)
User root specs
Solve - SEPARATE level 0
error(...) :- build_requirement("cmake@3.21:").
Build requirements from level 0
error(...) :- build_requirement("pkgconf").
Solve - SEPARATE level 1
Build requirements from level 1
Solve - UNIFIED level2

- We tried an iterative version with multiple solves
- Multiple solves had some disadvantages:
  - Slower due to overhead of multiple solves
  - Not coupled, so feedback from build to run environment (and back) was awkward
  - Packagers needed to "help" the solver
- Requiring packagers to provide solve hints in packages isn't practical

Fourth Challenge: v1.0 adds language dependencies

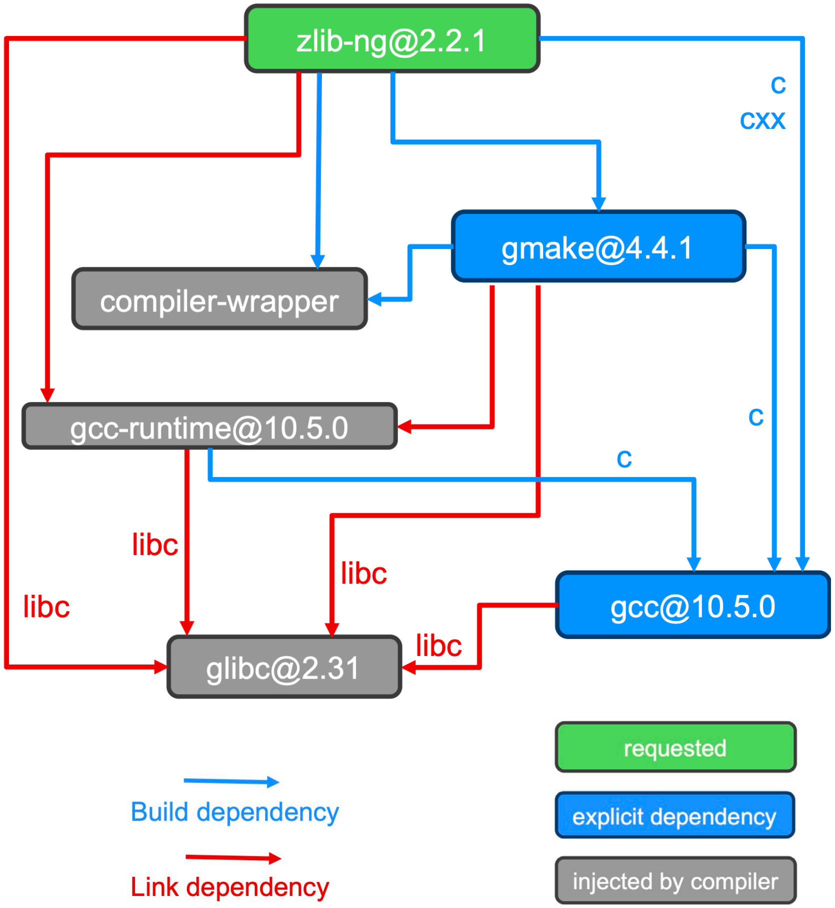


```
depends_on("c", type="build")
depends_on("cxx", type="build")
depends_on("fortran", type="build")
```

- 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)
- Required in 1.0 if you want to use c, cxx, or fortran
  - No-op in v0.23 and prior as we prepared for this feature

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)



Spack 1.x introduces *toolchains*

toolchains.yaml

```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"
```

```
spack install foo %clang_gfortran
```

```yaml
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 %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

Configuring compilers in Spack v1.*

Spack v0.x

compilers.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

```
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: [...]
```

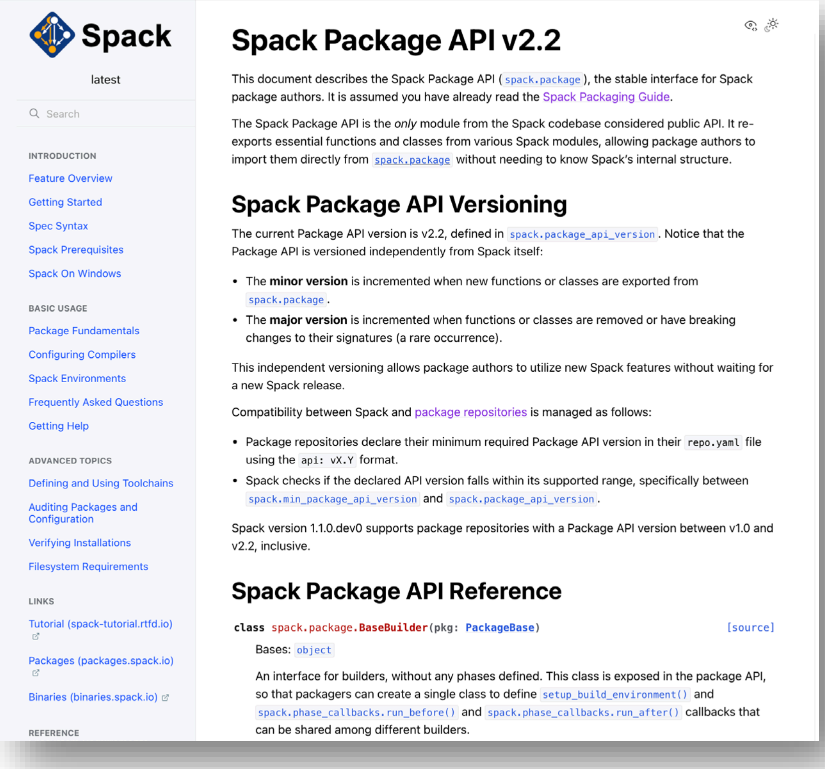
- We automatically convert compilers.yaml, when no compiler is configured
- We will still support *reading* the old configuration until at *least* v1.1
- All fields from `compilers.yaml` are supported in `extra_attributes`

Final challenge: Splitting the package repository

- Spack is two things:
  - Command line tool `spack`
  - Package repository with 8,500+ recipes
- Community wanted
  - 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

Spack now has a Stable Package API

- Repositories define API version used
  - Versioned *per commit*
- Spack defines API version(s) supported
  - Will complain if a repo is too new
- Packages can only import from:
  - spack.package
  - Core Python
- Any 1.x Spack will support the same package API as all prior 1.x versions
  - Won't break package API unless we bump the major version



Spack Package API v2.2

This document describes the Spack Package API (`spack.package`), the stable interface for Spack package authors. It is assumed you have already read the Spack Packaging Guide.

The Spack Package API is the *only* module from the Spack codebase considered public API. It re-exports essential functions and classes from various Spack modules, allowing package authors to import them directly from `spack.package` without needing to know Spack's internal structure.

Spack Package API Versioning

The current Package API version is v2.2, defined in `spack.package_api_version`. Notice that the Package API is versioned independently from Spack itself:

- The **minor version** is incremented when new functions or classes are exported from `spack.package`.
- The **major version** is incremented when functions or classes are removed or have breaking changes to their signatures (a rare occurrence).

This independent versioning allows package authors to utilize new Spack features without waiting for a new Spack release.

Compatibility between Spack and package repositories is managed as follows:

- Package repositories declare their minimum required Package API version in their `repo.yaml` file using the `api: vX.Y` format.
- Spack checks if the declared API version falls within its supported range, specifically between `spack.min_package_api_version` and `spack.package_api_version`.

Spack version 1.1.0.dev0 supports package repositories with a Package API version between v1.0 and v2.2, inclusive.

Spack Package API Reference

```
class spack.package.BaseBuilder(pkg: PackageBase)                    [source]
```

Bases: `object`

An interface for builders, without any phases defined. This class is exposed in the package API, so that packagers can create a single class to define `setup_build_environment()` and `spack.phase_callbacks.run_before()` and `spack.phase_callbacks.run_after()` callbacks that can be shared among different builders.

`https://spack.rtfd.io/en/latest/package_api.html`

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

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

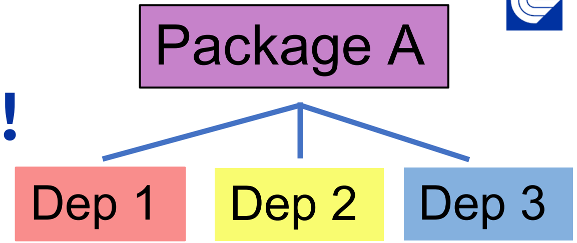
```
spack:
  repos:
    builtin:
      git: https://github.com/spack/spack-packages.git
      destination: /path/to/clone/of/spack-packages
      branch: develop
```

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

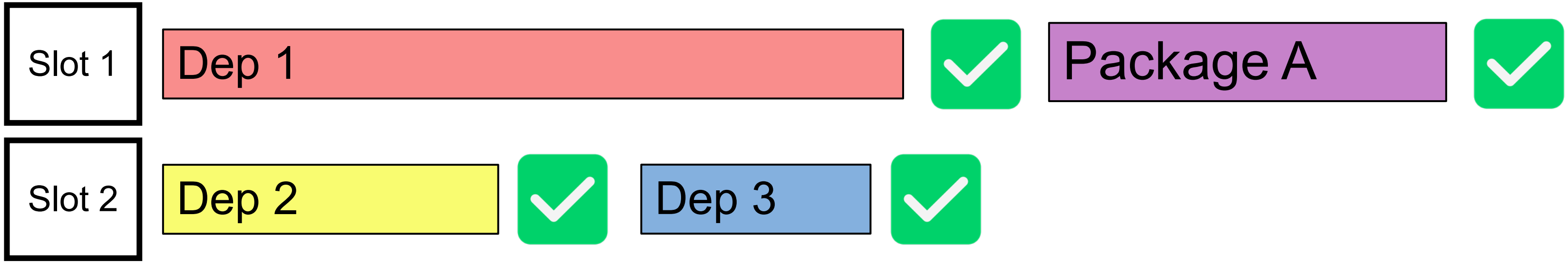
Bonus feature:
Spack now supports concurrent builds!



- 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’

Queue:



But wait! There's more!

Spack is a core project in 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 3,800+ 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

𝕏 @spackpm

spack.io

We hope to make distributing & using HPC software easy!

Hands-on time!

THE LINUX FOUNDATION

Tutorial Materials

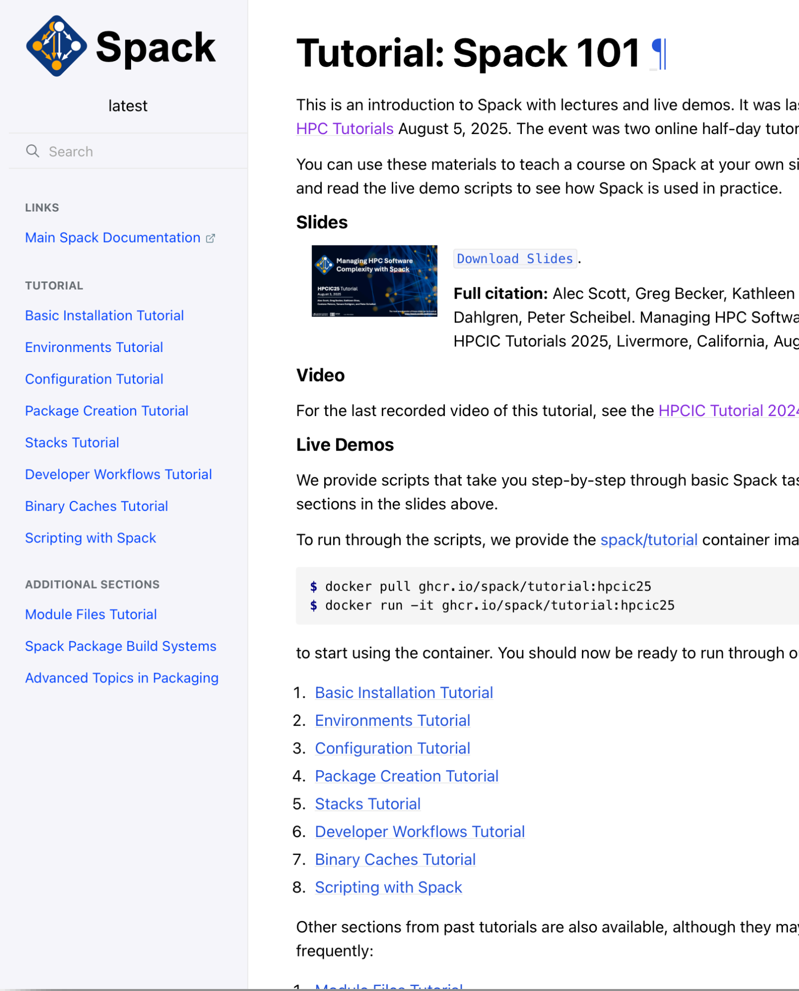
Find these slides and associated scripts here:

spack-tutorial.rtfd.io

We also have a `#tutorial` chat room on Spack slack. Join at:

slack.spack.io

You can ask questions here any time!



LINKS

Main Spack Documentation

TUTORIAL

Basic Installation Tutorial

Environments Tutorial

Configuration Tutorial

Package Creation Tutorial

Stacks Tutorial

Developer Workflows Tutorial

Binary Caches Tutorial

Scripting with Spack

ADDITIONAL SECTIONS

Module Files Tutorial

Spack Package Build Systems

Advanced Topics in Packaging

Tutorial: Spack 101 ¶

This is an introduction to Spack with lectures and live demos. It was la
HPC Tutorials August 5, 2025. The event was two online half-day tutor

You can use these materials to teach a course on Spack at your own si
and read the live demo scripts to see how Spack is used in practice.

Slides

`Download Slides`.

Full citation: Alec Scott, Greg Becker, Kathleen
Dahlgren, Peter Scheibel. Managing HPC Softwa
HPCIC Tutorials 2025, Livermore, California, Aug

Video

For the last recorded video of this tutorial, see the HPCIC Tutorial 202

Live Demos

We provide scripts that take you step-by-step through basic Spack ta
sections in the slides above.

To run through the scripts, we provide the spack/tutorial container ima

```
$ docker pull ghcr.io/spack/tutorial:hpcic25
$ docker run -it ghcr.io/spack/tutorial:hpcic25
```

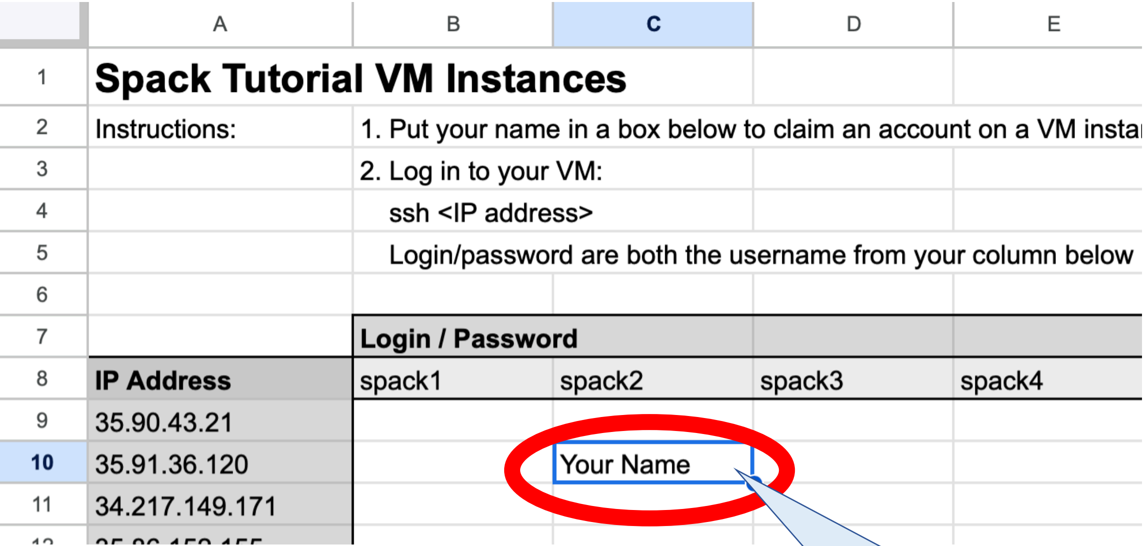
to start using the container. You should now be ready to run through o

1. Basic Installation Tutorial
2. Environments Tutorial
3. Configuration Tutorial
4. Package Creation Tutorial
5. Stacks Tutorial
6. Developer Workflows Tutorial
7. Binary Caches Tutorial
8. Scripting with Spack

Other sections from past tutorials are also available, although they ma
frequently:

1. Module Files Tutorial

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				
12	35.86.152.155				

bitly

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

```
ssh spack2@3.73.129.196
```

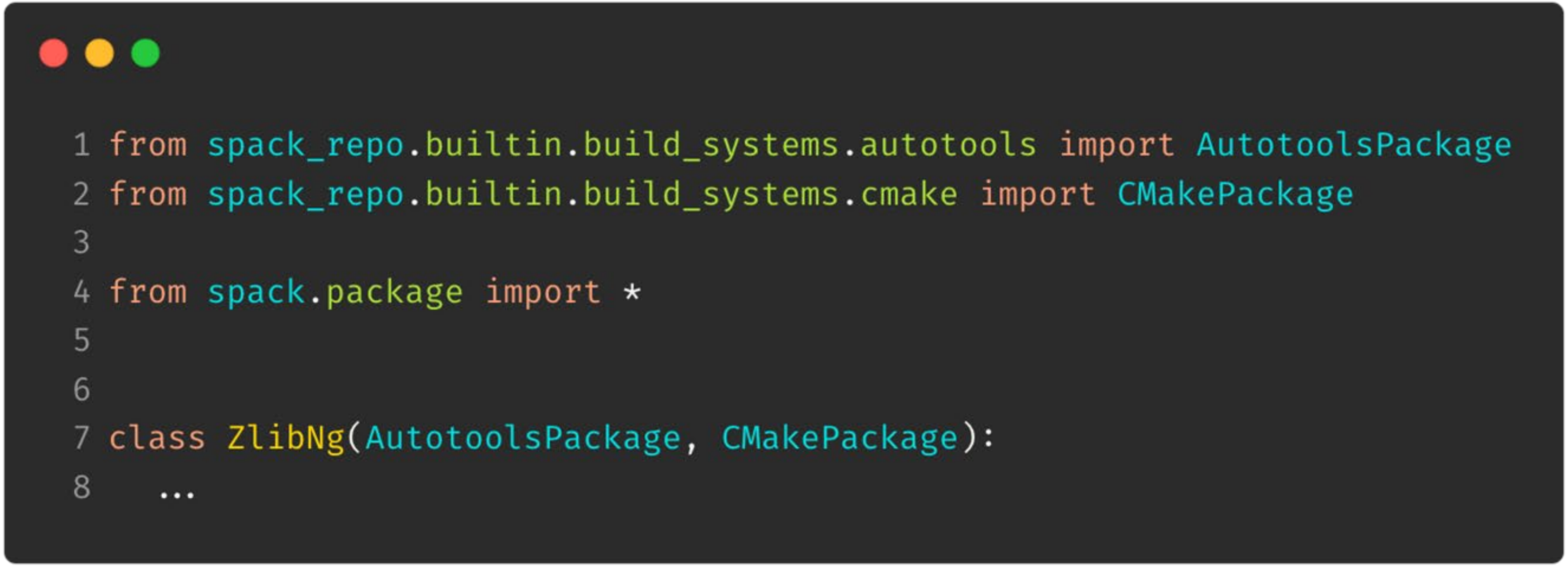
Claim a login by putting your name in the Google Sheet

Disclaimer

This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

THE LINUX FOUNDATION

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



```python
from spack_repo.builtin.build_systems.autotools import AutotoolsPackage
from spack_repo.builtin.build_systems.cmake import CMakePackage

from spack.package import *


class ZlibNg(AutotoolsPackage, CMakePackage):
    ...
```

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

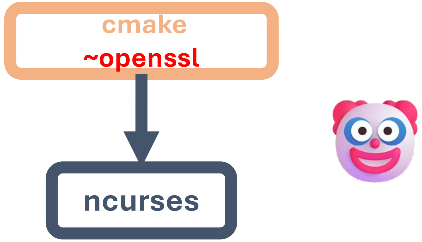
3. The % sigil in specs means “direct dependency”.

- Can now say: `foo %cmake@3.26 ^bar %cmake@3.31`
- ^ dependencies are unified, % dependencies are not

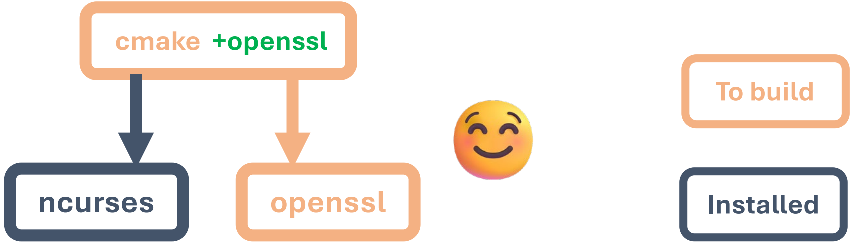
Art vs. science: reusing builds is not quite enough

- We get strange behavior when we have to build new packages
  - E.g.: Cmake depends on openssl for https
  - Minimizing builds will toggle this feature *off* to avoid a dependency
- **We want to prioritize reusing a package *if* the user already installed it**
  - Has to be more important than defaults, or we would never reuse
- **We want to prioritize package defaults *if* the package had to be built anyway**
  - Make *new* builds follow defaults

How can we do both?



minimize builds > package defaults

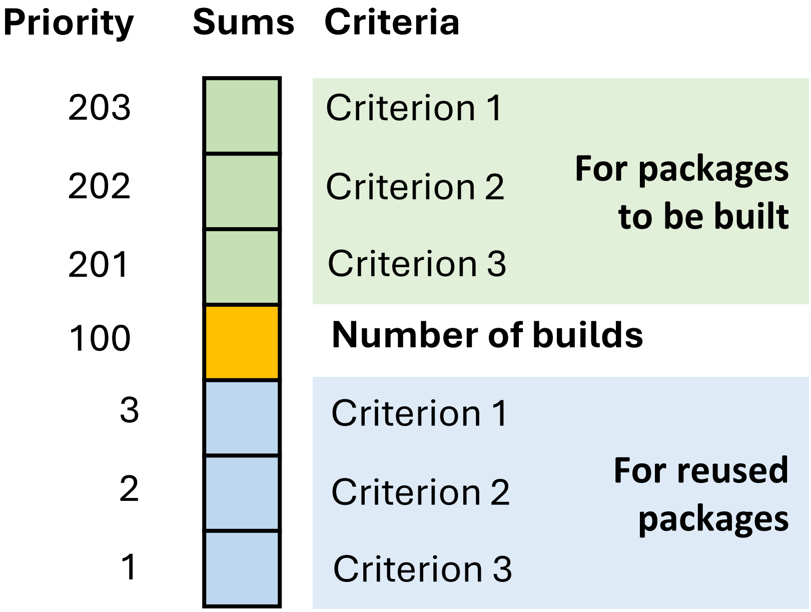


package defaults > minimize builds

We devised a two-level optimization scheme

```
build_priority(P, 200) :- build(P),     node(P).
build_priority(P, 0)   :- not build(P), node(P).

% priority + 200 IF we are building
#minimize{
   W@2+Priority,P
   : version_weight(P, W), build_priority(P, Priority)
}.
```

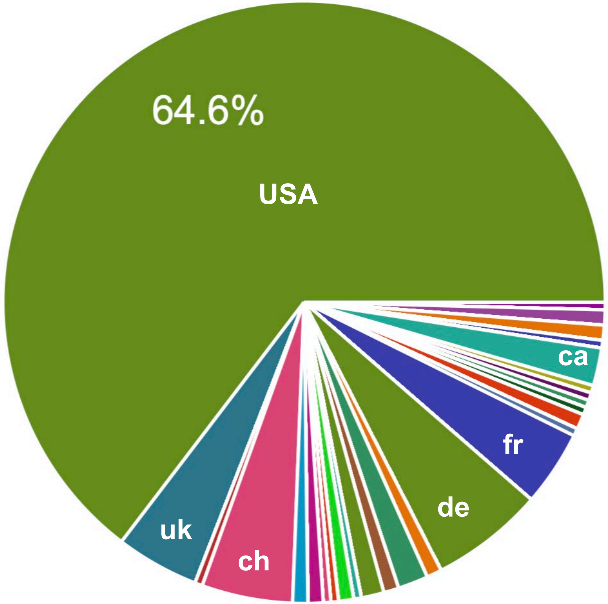
- Minimize builds, *unless we have to build*
  - Prioritize *defaults* for specs we *have* to build
- Last trick to get this to work:
  - All criteria must be formulated as minimizations
  - No built configuration can be “better” than a reused configuration



Priority	Sums	Criteria	
203	🟩	Criterion 1	For packages to be built
202	🟩	Criterion 2	
201	🟩	Criterion 3	
100	🟨	**Number of builds**	
3	🟦	Criterion 1	For reused packages
2	🟦	Criterion 2	
1	🟦	Criterion 3	

Objective vectors of sums are compared lexicographically from highest to lowest priority

About 2/3 of respondents are from the US



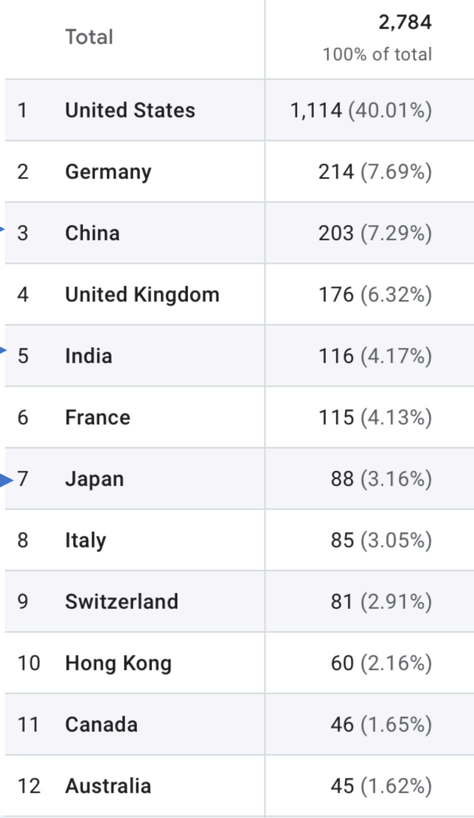
2025 Survey

Comparing to our documentation data, some countries are under-represented.

1 response → China

4 responses → India

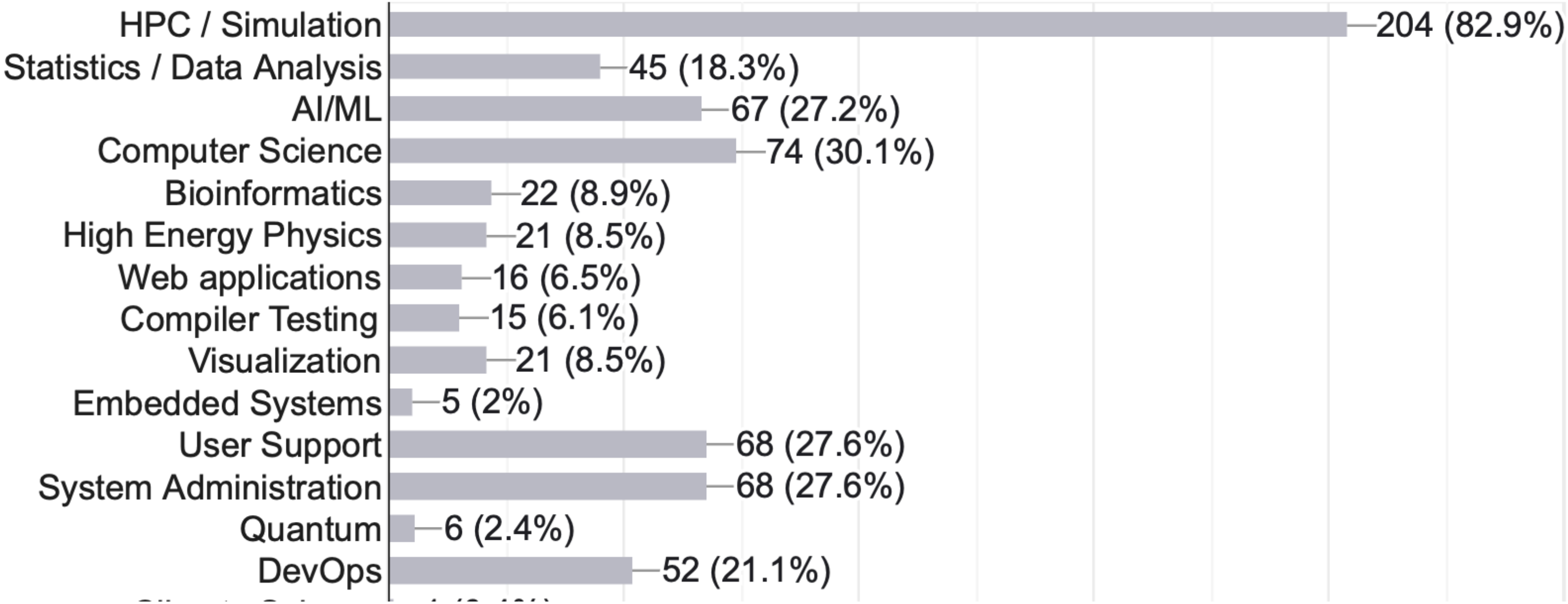
3 responses → Japan



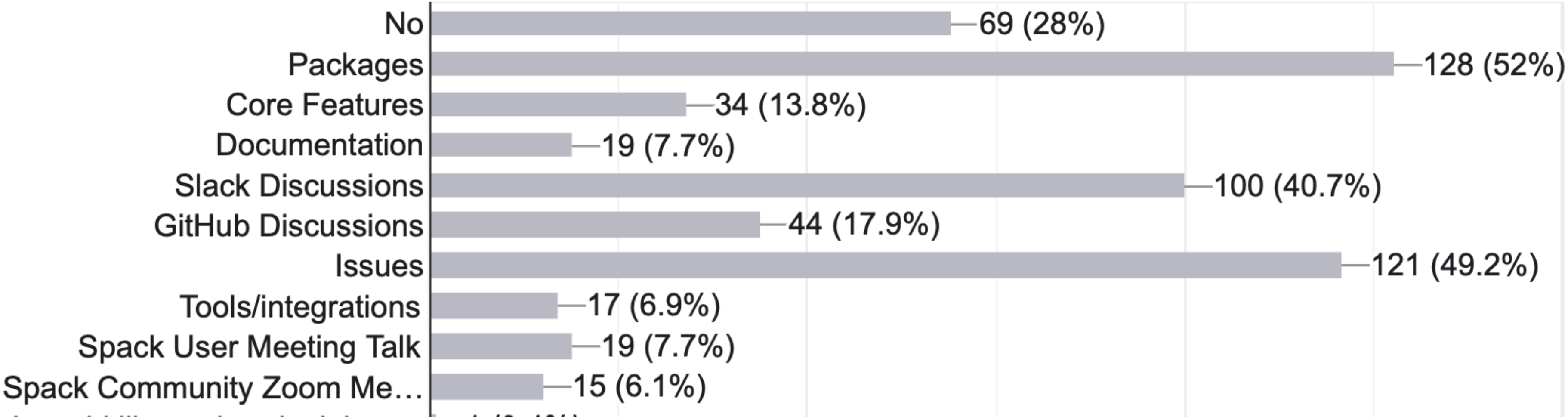
	Total	2,784 100% of total
1	United States	1,114 (40.01%)
2	Germany	214 (7.69%)
3	China	203 (7.29%)
4	United Kingdom	176 (6.32%)
5	India	116 (4.17%)
6	France	115 (4.13%)
7	Japan	88 (3.16%)
8	Italy	85 (3.05%)
9	Switzerland	81 (2.91%)
10	Hong Kong	60 (2.16%)
11	Canada	46 (1.65%)
12	Australia	45 (1.62%)

Past month: active users at spack.readthedocs.io

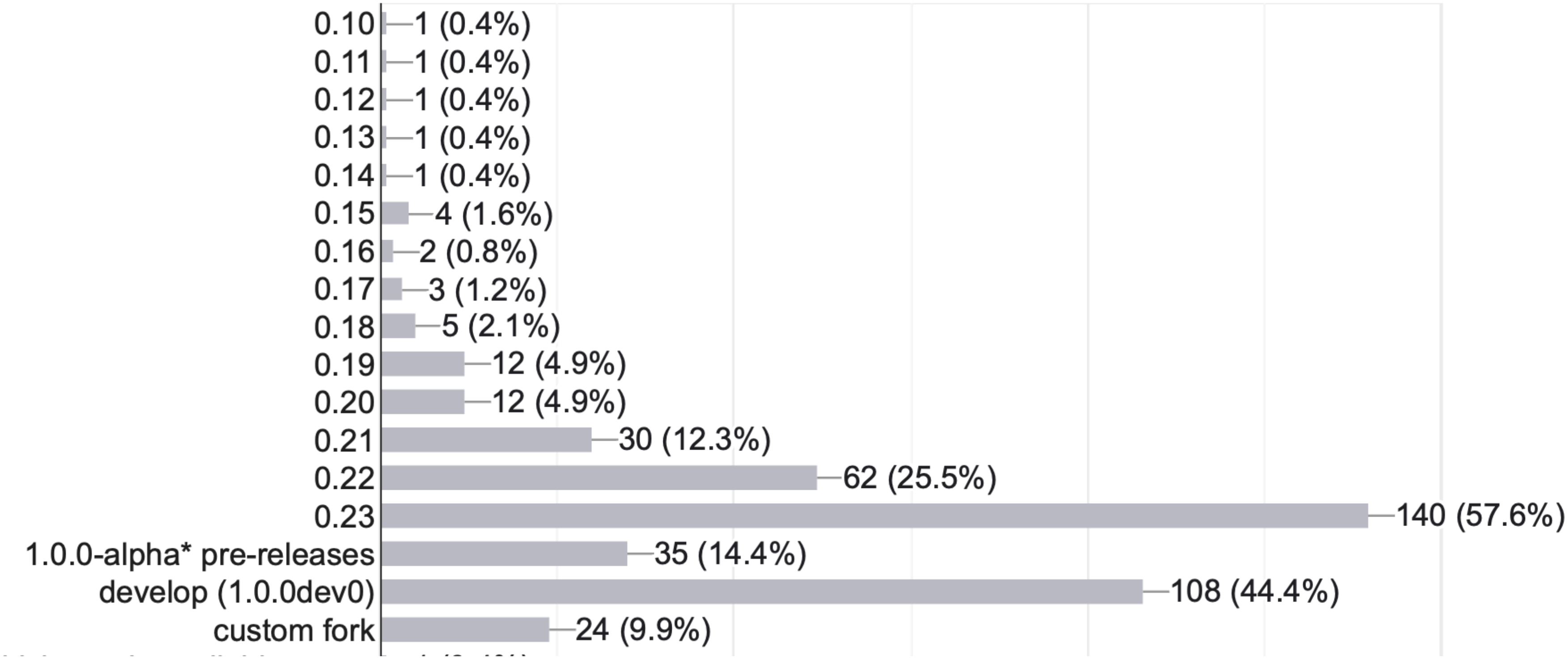
82% of users are doing HPC; 27% AI



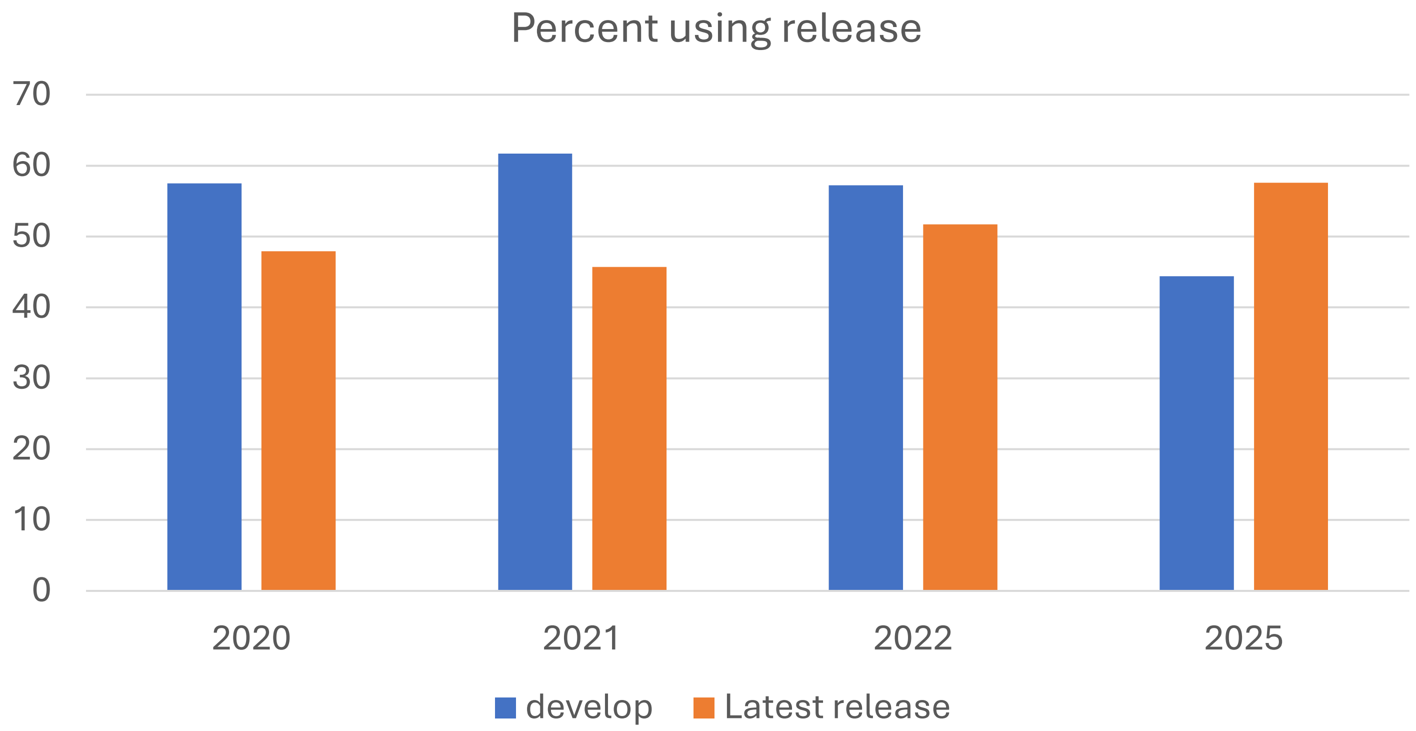
Most users are also contributors!



Most are within 2 releases of latest

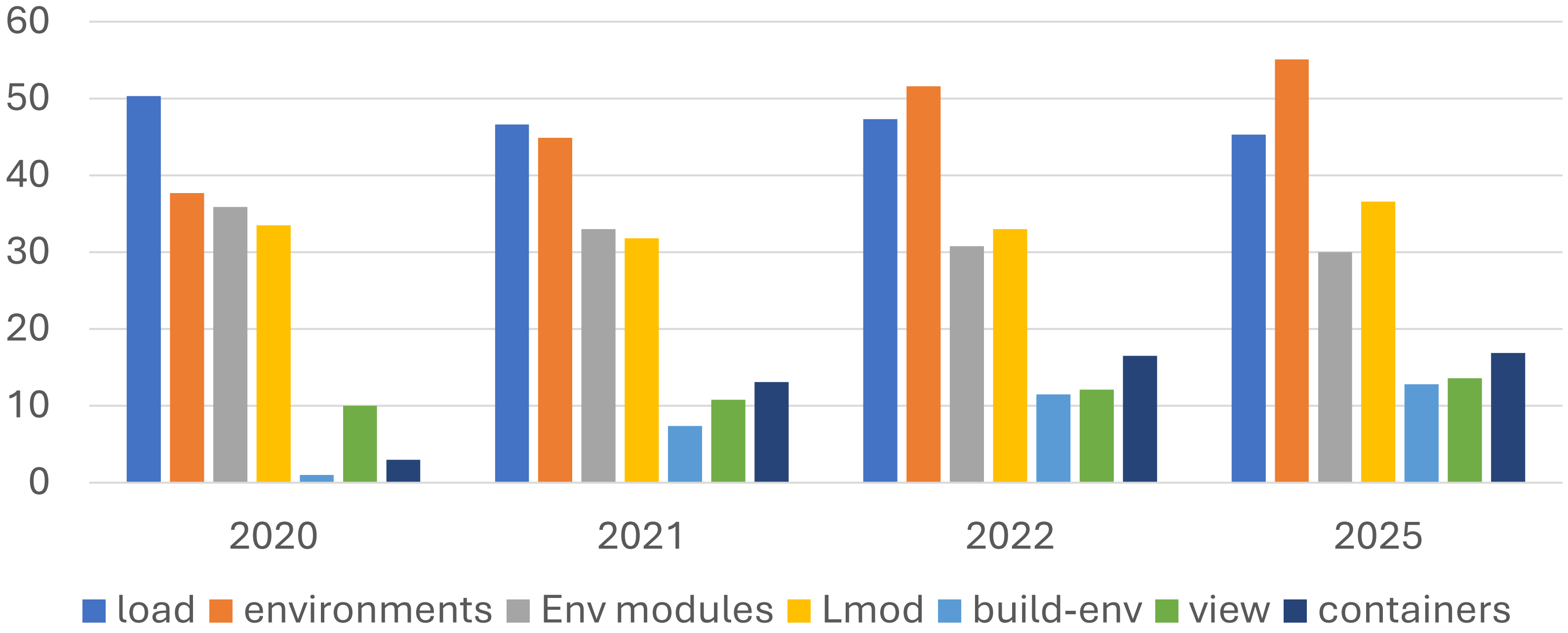


More users are on stable releases now than on develop

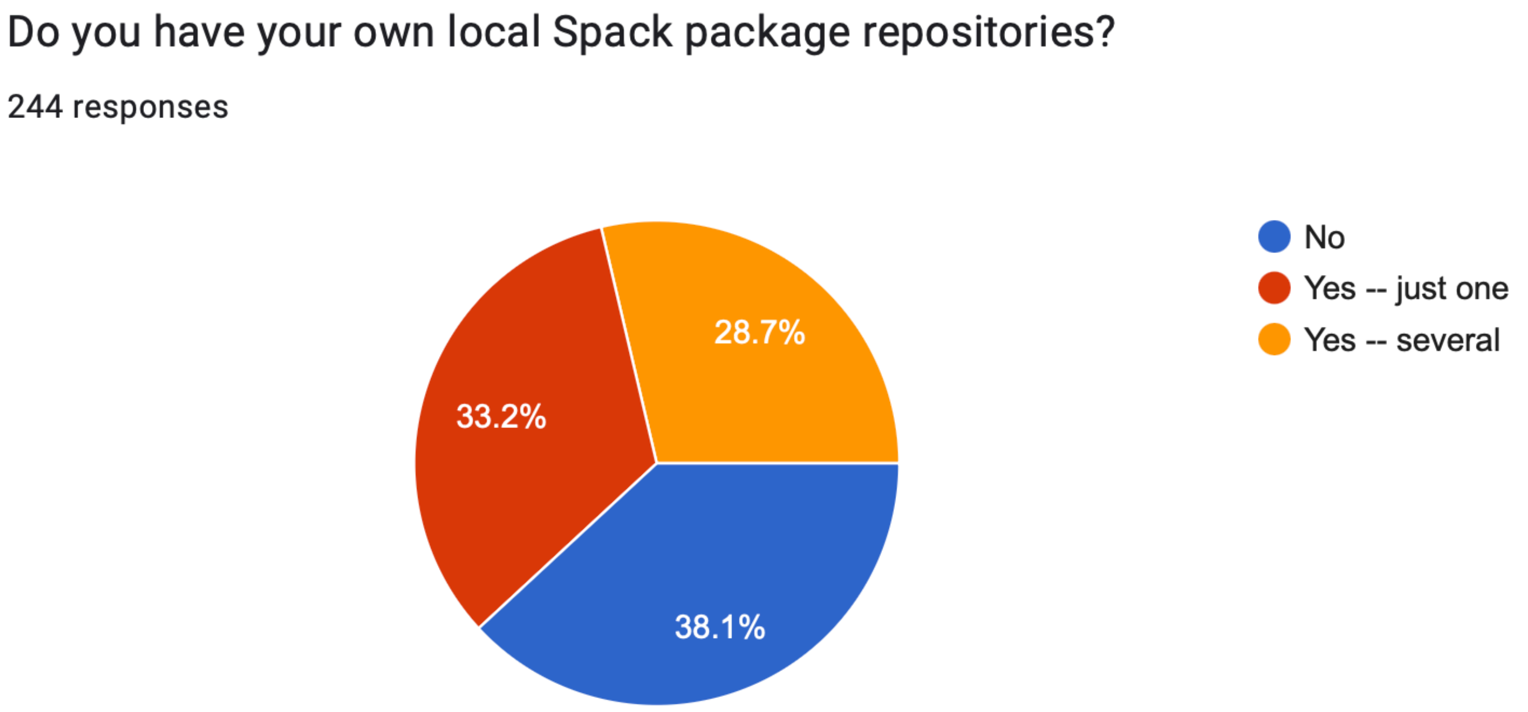


Environments have become the preferred way to load packages

How do you get installed Spack packages into your environment?



There are many packages outside of Spack’s builtin repo

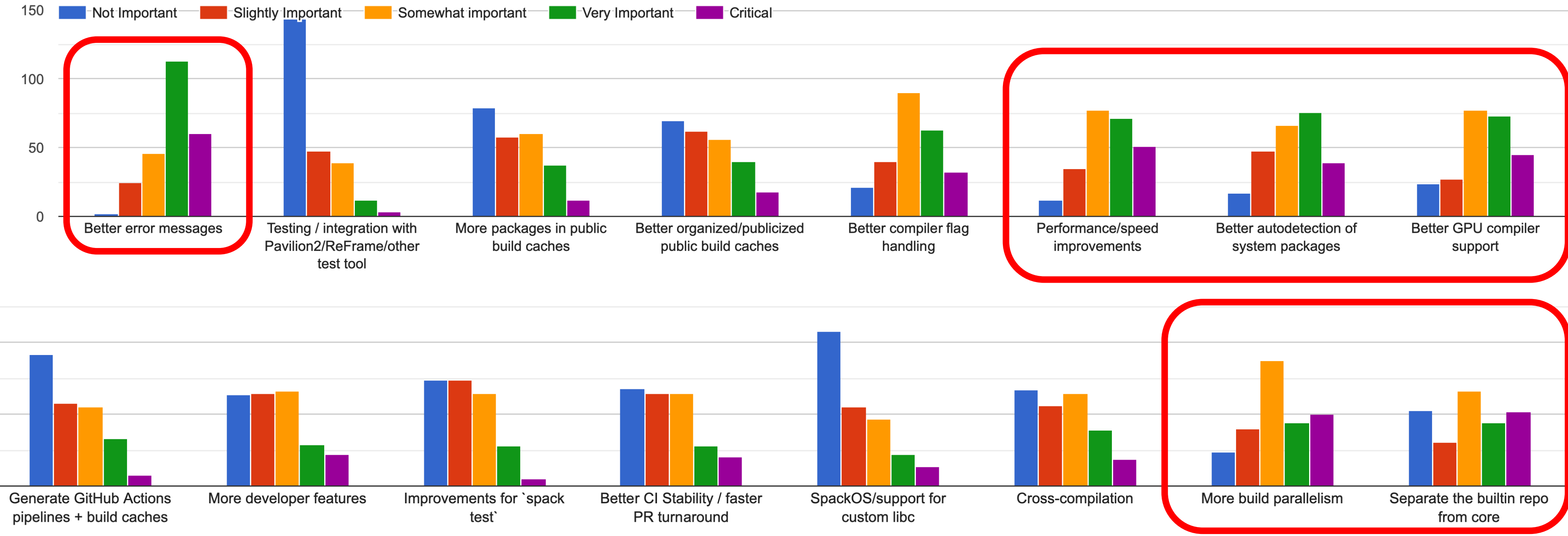


Do you have your own local Spack package repositories?

244 responses

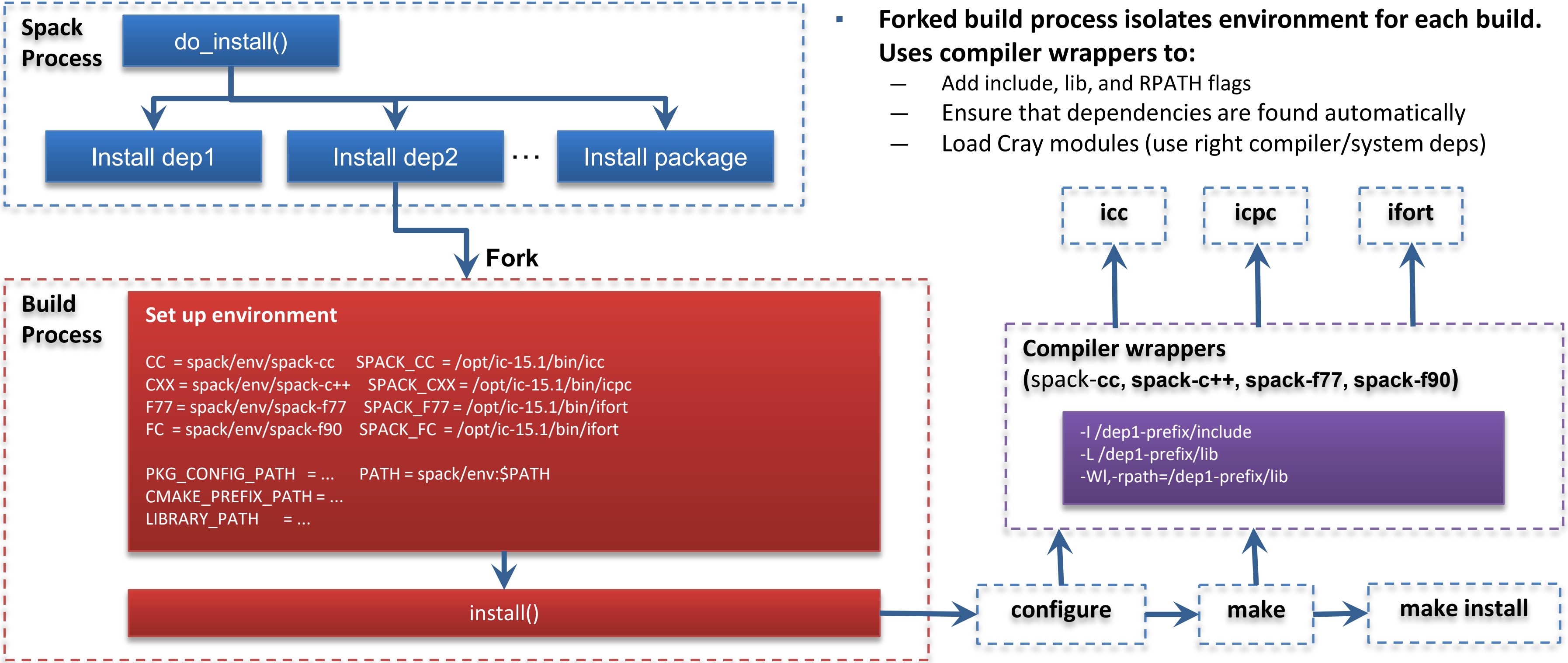
Users want better error messages, more performance, and a separate package repo

Rank these TBD Spack features by importance



An isolated compilation environment allows Spack to easily swap compilers

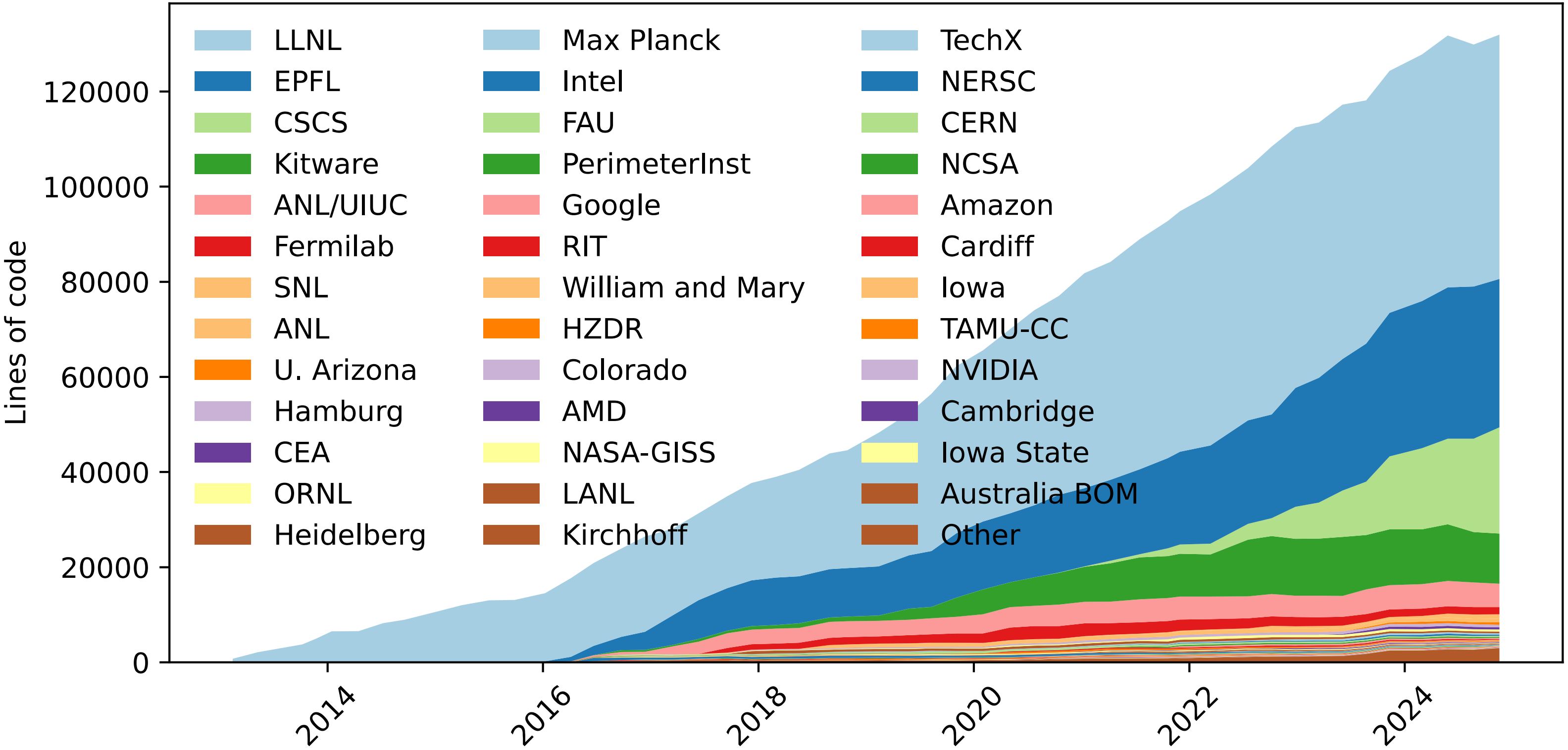
- **Forked build process isolates environment for each build. Uses compiler wrappers to:**
  - Add include, lib, and RPATH flags
  - Ensure that dependencies are found automatically
  - Load Cray modules (use right compiler/system deps)



```
CC  = spack/env/spack-cc      SPACK_CC  = /opt/ic-15.1/bin/icc
CXX = spack/env/spack-c++     SPACK_CXX = /opt/ic-15.1/bin/icpc
F77 = spack/env/spack-f77     SPACK_F77 = /opt/ic-15.1/bin/ifort
FC  = spack/env/spack-f90     SPACK_FC  = /opt/ic-15.1/bin/ifort

PKG_CONFIG_PATH   = ...       PATH = spack/env:$PATH
CMAKE_PREFIX_PATH = ...
LIBRARY_PATH      = ...
```

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

Core contributions are less diverse, but still comprise many organizations


Lines of code
LLNL
EPFL
CSCS
Kitware
ANL/UIUC
Fermilab
SNL
ANL
U. Arizona
Hamburg
CEA
ORNL
Heidelberg
Max Planck
Intel
FAU
PerimeterInst
Google
RIT
William and Mary
HZDR
Colorado
AMD
NASA-GISS
LANL
Kirchhoff
TechX
NERSC
CERN
NCSA
Amazon
Cardiff
Iowa
TAMU-CC
NVIDIA
Cambridge
Iowa State
Australia BOM
Other