What's new in Spack?

New features and the Spack Roadmap

IDEAS Best Practices for HPC Software Developers Webinar
July 15, 2020

Todd Gamblin
Advanced Technology Office
Livermore Computing

LLNL-PRES-811119

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC

Lawrence Livermore National Laboratory

Spack provides a *spec* syntax to describe customized DAG configurations

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

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

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

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

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

```
from spack import *

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

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

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

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

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

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

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

Base package
(CMake support)

Metadata at the class level

Versions

Variants (build options)

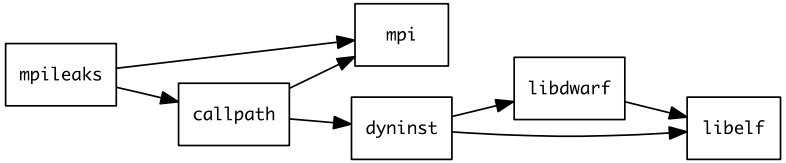
Dependencies
(note: same spec syntax)

Install logic
in instance methods

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

Spack handles combinatorial software complexity.

Dependency DAG



Installation Layout

Hash

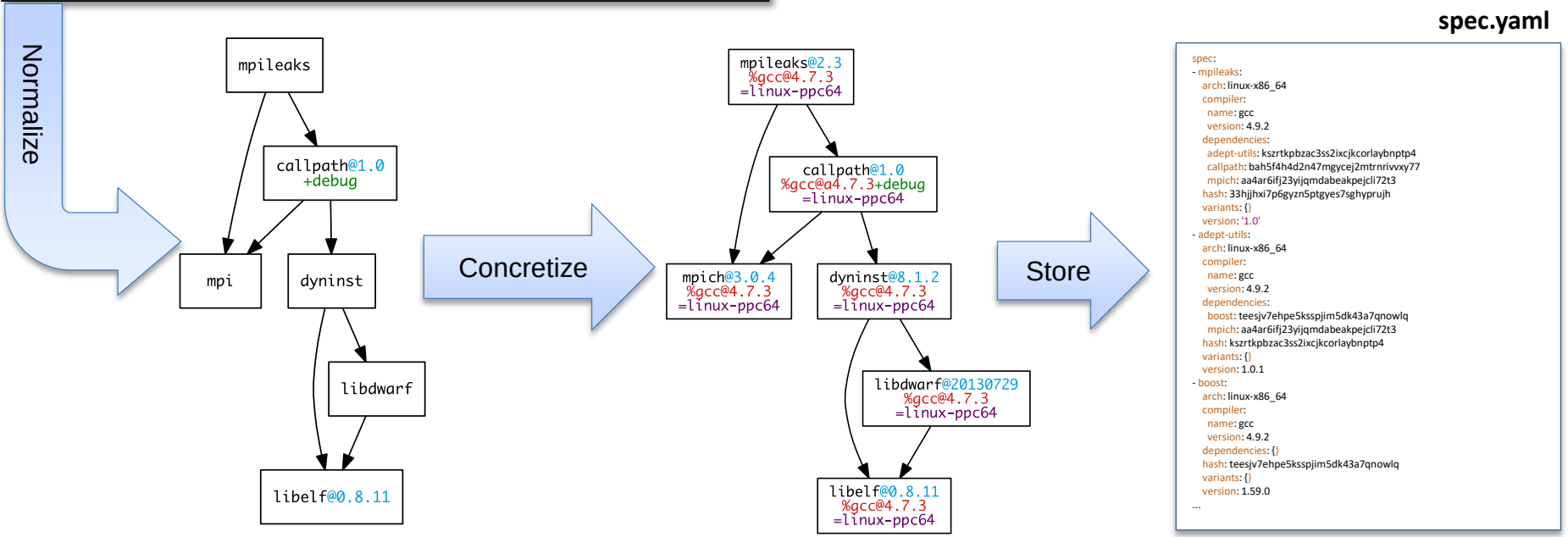
```
spack/opt/
  linux-x86_64/
    gcc-4.7.2/
      mpileaks-1.1-0f54bf34cadk/
    intel-14.1/
      hdf5-1.8.15-lkf14aq3nqiz/
  bgq/
    xl-12.1/
      hdf5-1-8.16-fqb3a15abrwx/
  ...
```

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

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.

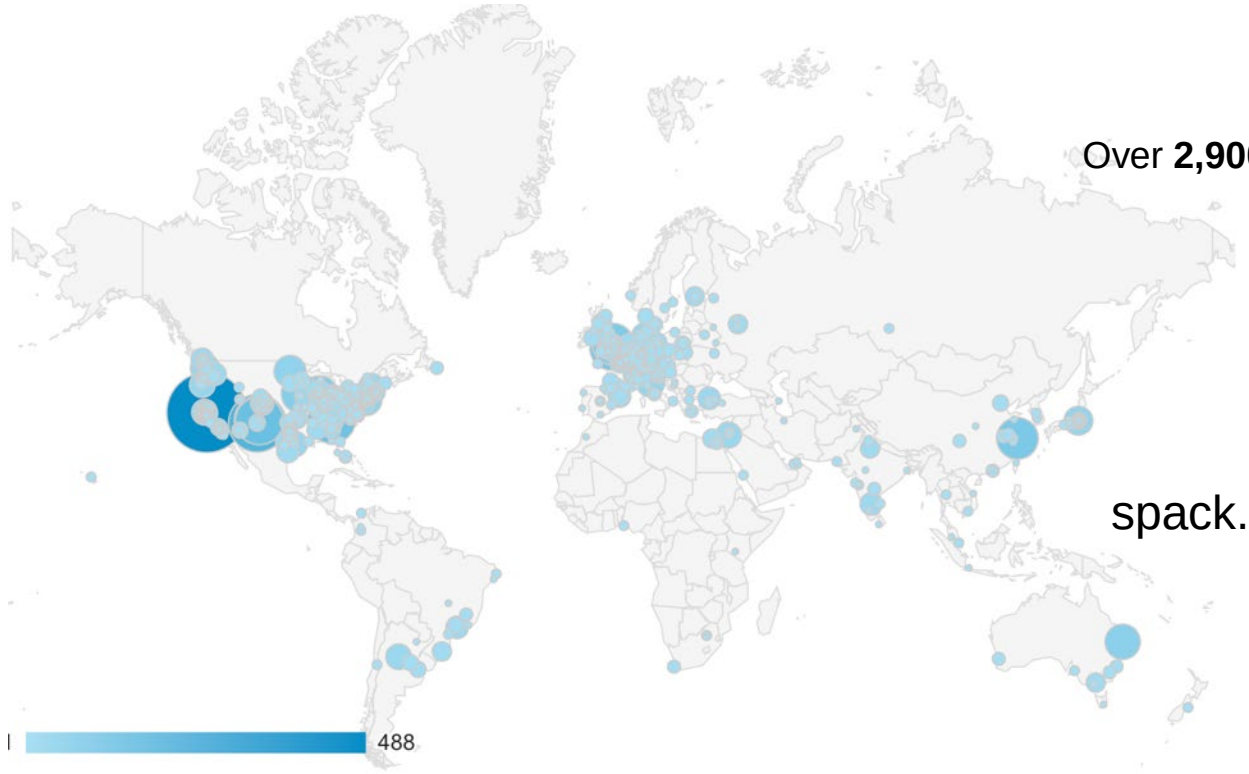
Concrete spec is fully constrained and can be passed to install.

Detailed provenance is stored with the installed package

Use `spack spec` to see the results of concretization

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

Concretized
--------------------------------
 mpileaks@1.0%gcc@5.3.0 arch=darwin-elcapitan-x86_64
    ^adept-utils@1.0.1%gcc@5.3.0 arch=darwin-elcapitan-x86_64
       ^boost@1.61.0%gcc@5.3.0+atomic+chrono+date_time~debug+filesystem~graph
        ~icu_support+iostreams+locale+log+math~mpi+multithreaded+program_options
        ~python+random +regex+serialization+shared+signals+singlethreaded+system
        +test+thread+timer+wave arch=darwin-elcapitan-x86_64
          ^bzip2@1.0.6%gcc@5.3.0 arch=darwin-elcapitan-x86_64
          ^zlib@1.2.8%gcc@5.3.0 arch=darwin-elcapitan-x86_64
       ^openmpi@2.0.0%gcc@5.3.0~mxm~pmi~psm~psm2~slurm~sqlite3~thread_multiple~tm~verbs+vt arch=darwin-elcapitan-x86_64
          ^hwloc@1.11.3%gcc@5.3.0 arch=darwin-elcapitan-x86_64
             ^libpciaccess@0.13.4%gcc@5.3.0 arch=darwin-elcapitan-x86_64
                ^libtool@2.4.6%gcc@5.3.0 arch=darwin-elcapitan-x86_64
                   ^m4@1.4.17%gcc@5.3.0+sigsegv arch=darwin-elcapitan-x86_64
                      ^libsigsegv@2.10%gcc@5.3.0 arch=darwin-elcapitan-x86_64
    ^callpath@1.0.2%gcc@5.3.0 arch=darwin-elcapitan-x86_64
       ^dyninst@9.2.0%gcc@5.3.0~stat_dysect arch=darwin-elcapitan-x86_64
          ^libdwarf@20160507%gcc@5.3.0 arch=darwin-elcapitan-x86_64
             ^libelf@0.8.13%gcc@5.3.0 arch=darwin-elcapitan-x86_64
```

Spack is used worldwide!

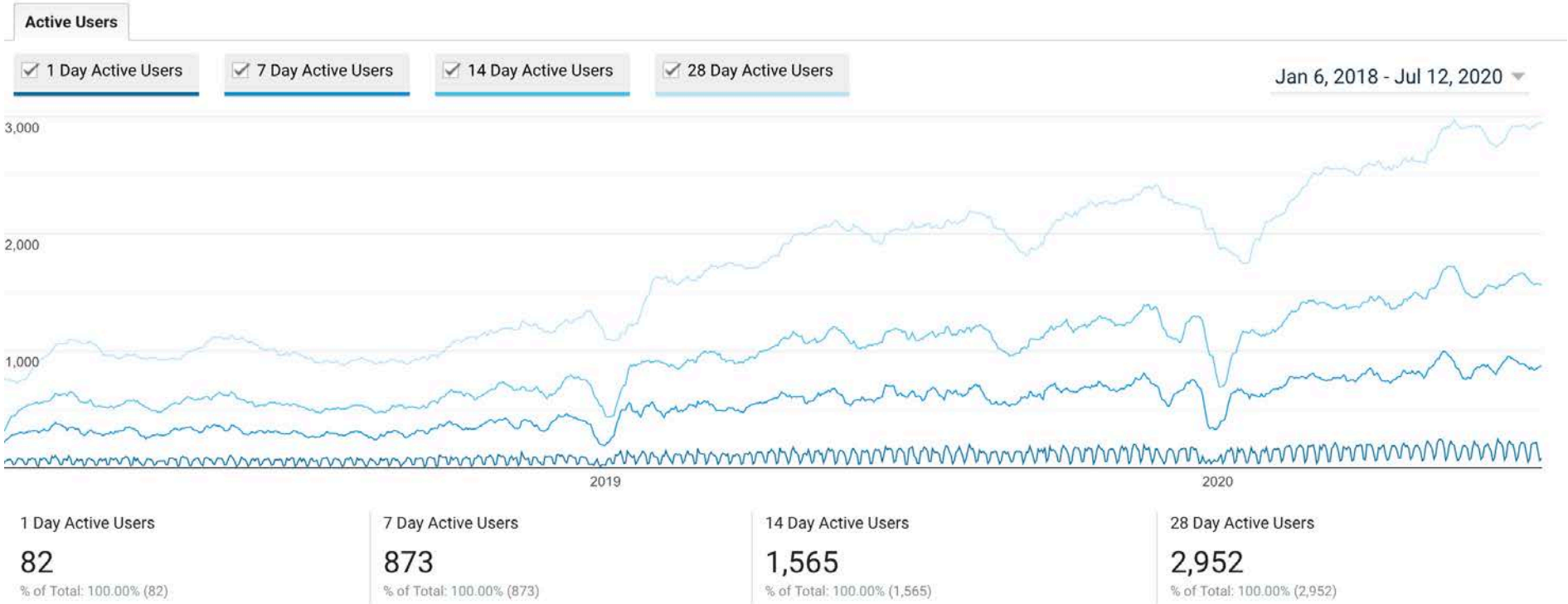


Over **4,300** software packages
Over **2,900** monthly active users (on docs site)

Over **600** contributors
from labs, academia, industry

Plot shows sessions on
spack.readthedocs.io for one month

Users on our documentation site have also been increasing



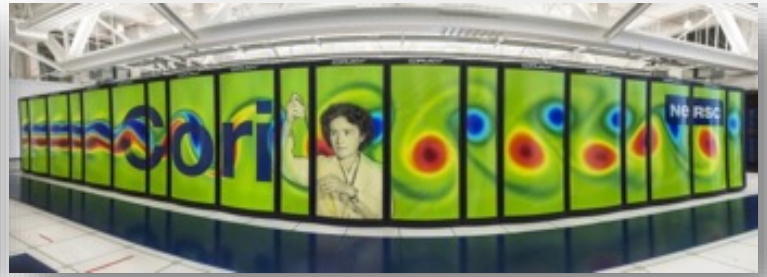
Spack is being used on many of the top HPC systems

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

Fugaku coming to RIKEN in 2021
DOE/MEXT collaboration

Summit (ORNL), Sierra (LLNL)

SuperMUC-NG (LRZ, Germany)



Edison, Cori, Perlmutter (NERSC)

One month of Spack development is pretty busy!

June 13, 2020 – July 13, 2020

Period: 1 month

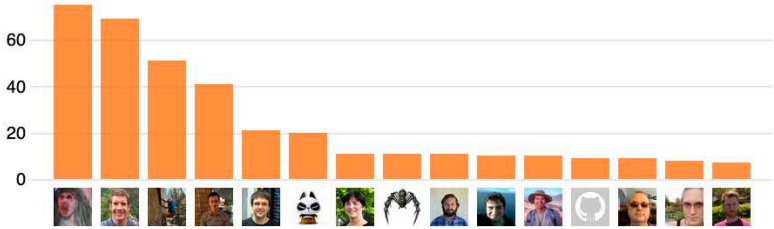
Overview

398 Active Pull Requests

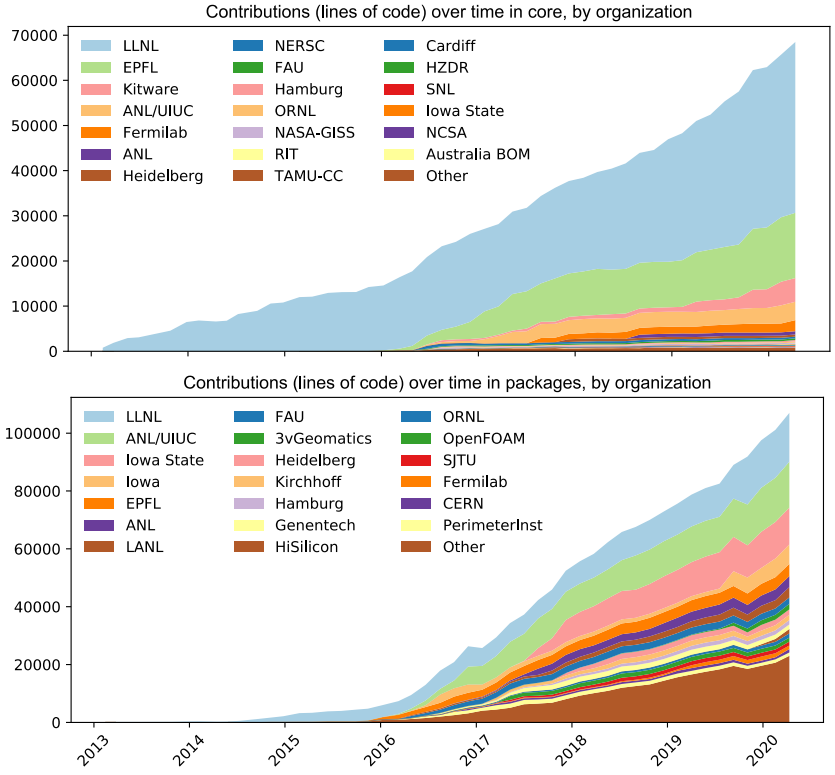
111 Active Issues

333	65	61	50
Merged Pull Requests	Proposed Pull Requests	Closed Issues	New Issues

Excluding merges, **113 authors** have pushed **345 commits** to develop and **532 commits** to all branches. On develop, **567 files** have changed and there have been **16,144 additions** and **2,496 deletions**.



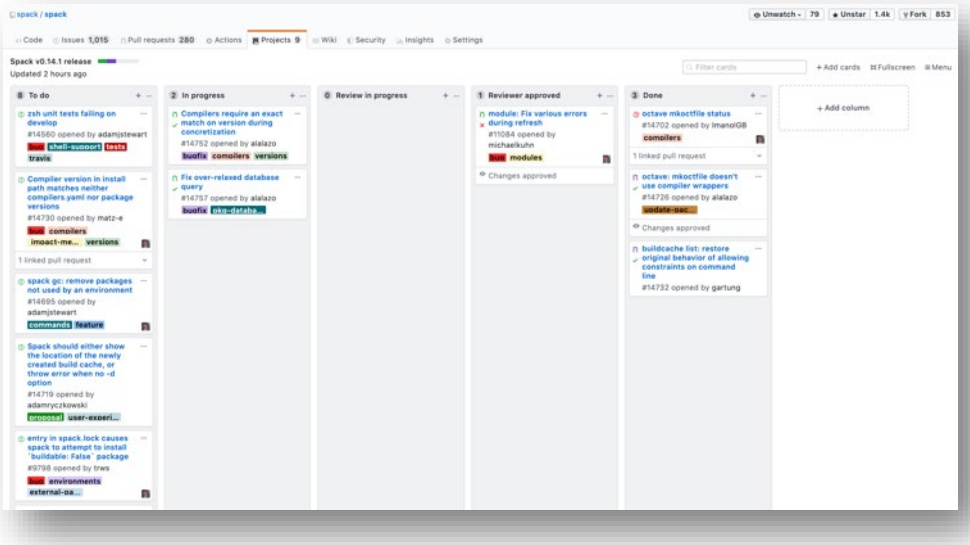
Contributions to Spack continue to grow!



- In November 2015, LLNL provided most of the contributions to Spack
- Since then, we've gone from 300 to over 4,000 packages
- Most packages are from external contributors!
- Many contributions in core, as well.
- We are committed to sustaining Spack's open source ecosystem!

Spack has a release workflow

- We are creating GitHub projects (Kanban boards) per release
  - Includes major (0.13.0, 0.14.0) and minor (0.13.1, 0.13.2, etc.) releases
  - Each release shows the timeframe
  - You can easily see what's on the roadmap!
- Makes it easy to rely on release branches
  - You can expect us to backport fixes for critical bugs onto these branches
- Shooting for quarterly releases
  - Expect some movement of features from release to release
  - If we don't finish some things, we'll move them forward



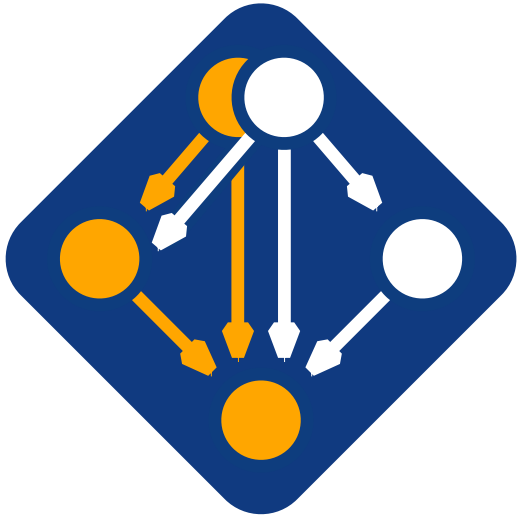
Per-release Kanban boards allow the community to track releases better!

Spack has stable release branches

- Develop is where most of the action happens
  - Latest commits from pull requests
  - Package updates
- Release branches have release tags, minimize churn
  - Only bugfixes are backported from develop to stable releases
  - Major new features and package recipe changes happen in develop
- releases/v0.14 is the release branch for:
  - v0.14.0
  - v0.14.1
  - v0.14.2
  - Etc.

Spack 0.13 was released in November, at SC19

- **Spack stacks:** combinatorial environments for facility deployment
- Spack detects and builds for **specific microarchitectures**
- **Chaining**: use dependencies from external "upstream" Spack instances



Ever tried to figure out what your processor is?

- You can get a lot of information from:
  - **/proc/cpuinfo** on linux
  - sysctl tool on macs
- But it's not exactly intuitive

Humans call this architecture "broadwell"

oh.

```
$ cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 79
model name      : Intel(R) Xeon(R) CPU E5-2695 v4 @ 2.10GHz
stepping        : 1
microcode       : 0xb000038
cpu MHz         : 2101.000
cache size      : 46080 KB
physical id     : 0
siblings        : 18
core id         : 0
cpu cores       : 18
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 20
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge
 mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe sy
scall nx pdpe1gb rdtscp lm constant_tsc arch_perfmon pebs bts rep_good
 nopl xtopology nonstop_tsc aperfmperf eagerfpu pni pclmulqdq dtes64 m
onitor ds_cpl vmx smx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid dca s
se4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c
 rdrand lahf_lm abm 3dnowprefetch epb cat_l3 cdp_l3 invpcid_single int
el_ppin intel_pt ssbd ibrs ibpb stibp tpr_shadow vnmi flexpriority ept
 vpid fsgsbase tsc_adjust bmi1 hle avx2 smep bmi2 erms invpcid rtm cqm
 rdt_a rdseed adx smap xsaveopt cqm_llc cqm_occup_llc cqm_mbm_total cq
m_mbm_local dtherm ida arat pln pts md_clear spec_ctrl intel_stibp flu
sh_l1d
bogomips        : 4190.37
clflush size    : 64
cache_alignment : 64
address sizes   : 46 bits physical, 48 bits virtual
power management:
```

what!?

what!?

Spack now understands specific target microarchitectures

- Spack knows what type of machine you're on
  - Detects based on /proc/cpuinfo (Linux), sysctl (Mac)
  - Allows comparisons for compatibility, e.g.:

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

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

```
$ spack arch --known-targets
Generic architectures (families)
    aarch64  ppc64  ppc64le  x86  x86_64

IBM - ppc64
    power7  power8  power9

IBM - ppc64le
    power8le  power9le

AuthenticAMD - x86_64
    barcelona  bulldozer  piledriver  steamroller  excavator  zen  zen2

GenuineIntel - x86_64
    nocona    westmere     haswell    mic_knl         cascadelake
    core2     sandybridge  broadwell  skylake_avx512  icelake
    nehalem   ivybridge    skylake    cannonlake

GenuineIntel - x86
    i686  pentium2  pentium3  pentium4  prescott
```

```
class OpenBlas(Package):

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

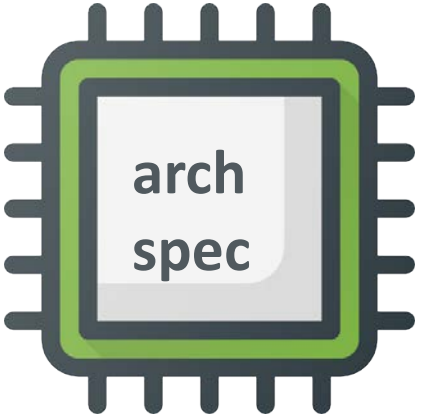
Simple feature query

```
$ spack install lbann target=cascadelake

$ spack install petsc target=zen2
```

Specialized installations

Archspec: a library for reasoning about microarchitectures

- Standalone library, extracted from Spack
- Use fine-grained, human-readable labels, e.g.:
  - broadwell, haswell, skylake
  - instead of x86_64, aarch64, ppc64 etc.
- Query capabilities
  - “Does haswell support AVX-512?” “no.”
- Query compiler flags
  - “How do I compile for broadwell with icc?”
- Python package for now, but we want more bindings!
  - Actual data is in a common JSON file w/schema



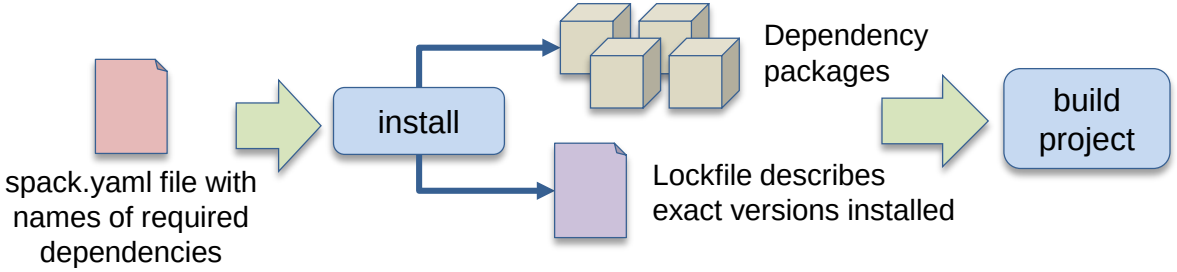
github.com/archspec

ReadTheDocs: archspec.rtfd.io

License: Apache 2.0 OR MIT

pip3 install archspec

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



- Allows developers to bundle Spack configuration with their repository
- Can also be used to maintain configuration together with Spack packages.
  - E.g., versioning your own local software stack with consistent compilers/MPI implementations
- Manifest / Lockfile model pioneered by Bundler is becoming standard
  - spack.yaml describes project requirements
  - spack.lock describes exactly what versions/configurations were installed, allows them to be reproduced.

Simple spack.yaml file

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

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

Concrete spack.lock file (generated)


```
{
 "concrete_specs": {
  "6s63so2kstp3zyvjezglndmavy6l3nul": {
   "hdf5": {
       "version": "1.10.5",
       "arch": {
           "platform": "darwin",
           "platform_os": "mojave",
           "target": "x86_64"
       },
       "compiler": {
           "name": "clang",
           "version": "10.0.0-apple"
       },
       "namespace": "builtin",
       "parameters": {
           "cxx": false,
           "debug": false,
           "fortran": false,
           "hl": false,
           "mpi": true,
           "pic": true,
           "shared": true,
           "szip": false,
           "threadsafe": fal
```

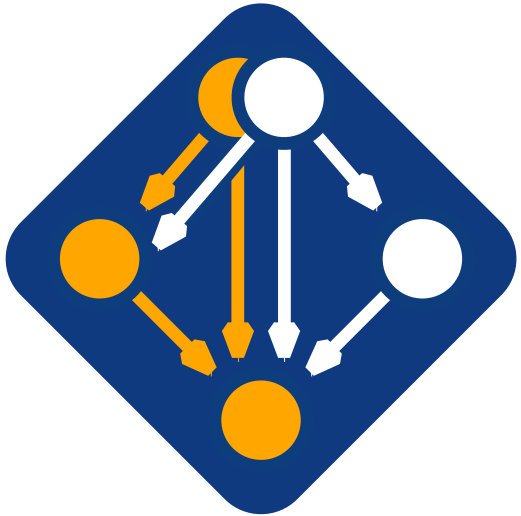
We have developed Spack stacks: combinatorial environments for entire facility deployments

```yaml
spack:
    definitions:
        compilers:
            [%gcc@5.4.0, %clang@3.8, %intel@18.0.0]
        mpis:
            [^mvapich2@2.2, ^mvapich2@2.3, ^openmpi@3.1.3]
        packages:
            - nalu
            - hdf5
            - hypre
            - trilinos
            - petsc
            - ...

    specs:
        # cartesian product of the lists above
        matrix:
            - [$packages]
            - [$compilers]
            - [$mpis]

    modules:
        lmod:
            core_compilers: [gcc@5.4.0]
            hierarchy:      [mpi, lapack]
            hash_length:    0
```

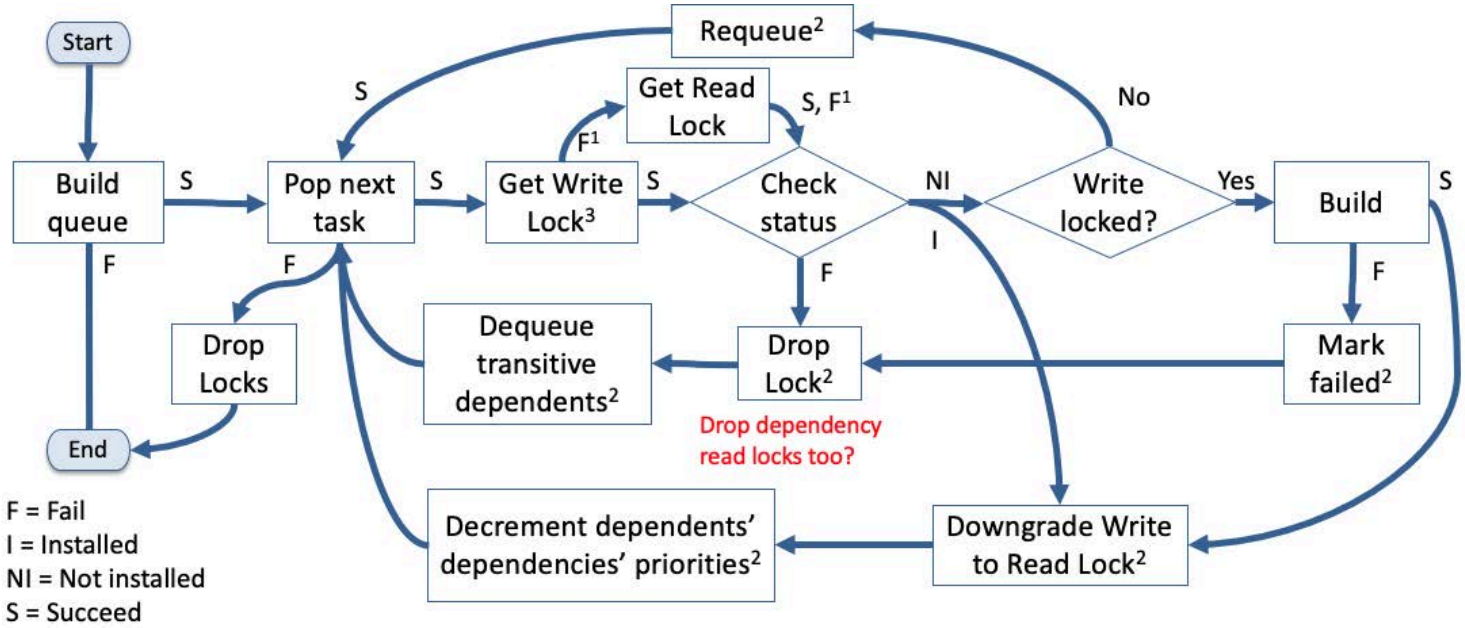
- Allow users to easily express a huge cross-product of specs
  - All the packages needed for a facility
  - Generate modules tailored to the site
  - Generate a directory layout to browse the packages
- Build on the environments workflow
  - Manifest + lockfile
  - Lockfile enables reproducibility
- Relocatable binaries allow the same binary to be used in a stack, regular install, or container build.
  - Difference is how the user interacts with the stack
  - Single-PATH stack vs. modules.

Spack 0.14.0 was released at the end of February

- Completely reworked **GitLab pipeline generation**
  - spack ci command
- **Generate container recipes** from environments
  - spack containerize command
- **Distributed/parallel builds**
  - srun –N 8 spack install
  - Spack instances coordinate effectively via locks



New distributed locking algorithm enables parallel builds (0.14)



- **Spack instances can coordinate with each other using only filesystem locks (no MPI required)**
  - Independently run instances on login nodes, or
  - srun –N 8 –n 32 spack install -j 16 <package>

Generate container images from environments (0.14)

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

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

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

    # Whether or not to strip binari
    strip: true

    # Additional system packages tha
    os_packages:
    - libgomp

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

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

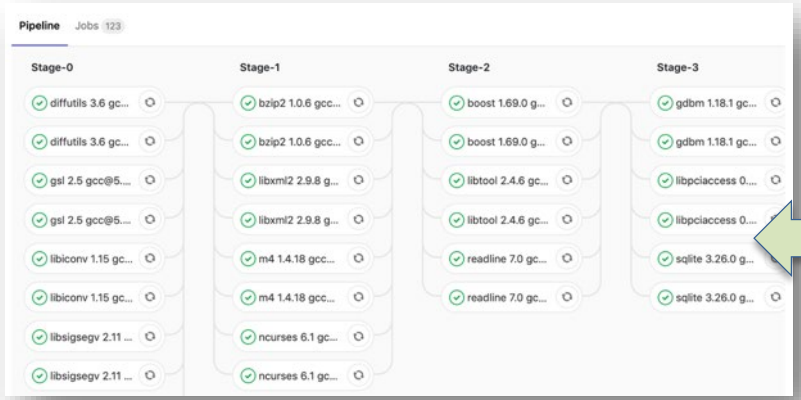
```
# Build stage with Spack pre-installed and ready to be used
FROM spack/centos7:latest as builder

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

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

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

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


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

COPY --from=builder /opt/spack-environment /opt/spack-environment
COPY --from=builder /opt/software /opt/software
COPY --from=builder /opt/view /opt/view
C  --from=builder /etc/profile.d/z10_spack_environment.sh /etc/profile.d/z10_spack_en

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

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

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

spack containerize

Spack can now generate CI Pipelines from environments

- User adds a gitlab-ci section to environment
  - Spack maps builds to GitLab runners
  - Generate gitlab-ci.yml with spack ci command
- Can run in a Kube cluster or on bare metal at an HPC site
  - Sends progress to CDash



spack ci

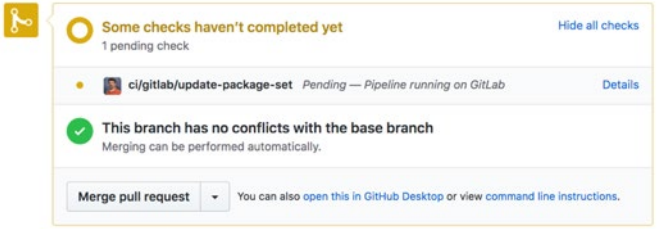
```yaml
spack:
  definitions:
  - pkgs:
    - readline@7.0
  - compilers:
    - '%gcc@5.5.0'
  - oses:
    - os=ubuntu18.04
    - os=centos7
  specs:
  - matrix:
    - [$pkgs]
    - [$compilers]
    - [$oses]
  mirrors:
    cloud_gitlab: https://mirror.spack.io
  gitlab-ci:
    mappings:
      - spack-cloud-ubuntu: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

Making use of the new workflow

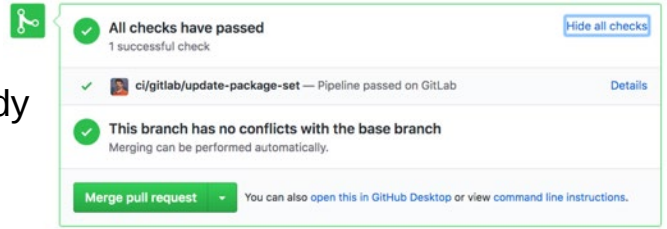
GitHub

Environment repo

Create PR, or push PR branch

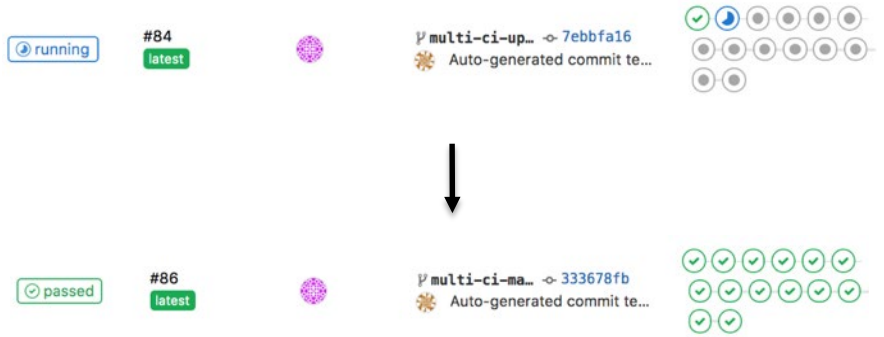


Merge ready



GitLab

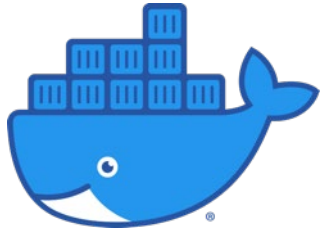
"CI/CD" only repo



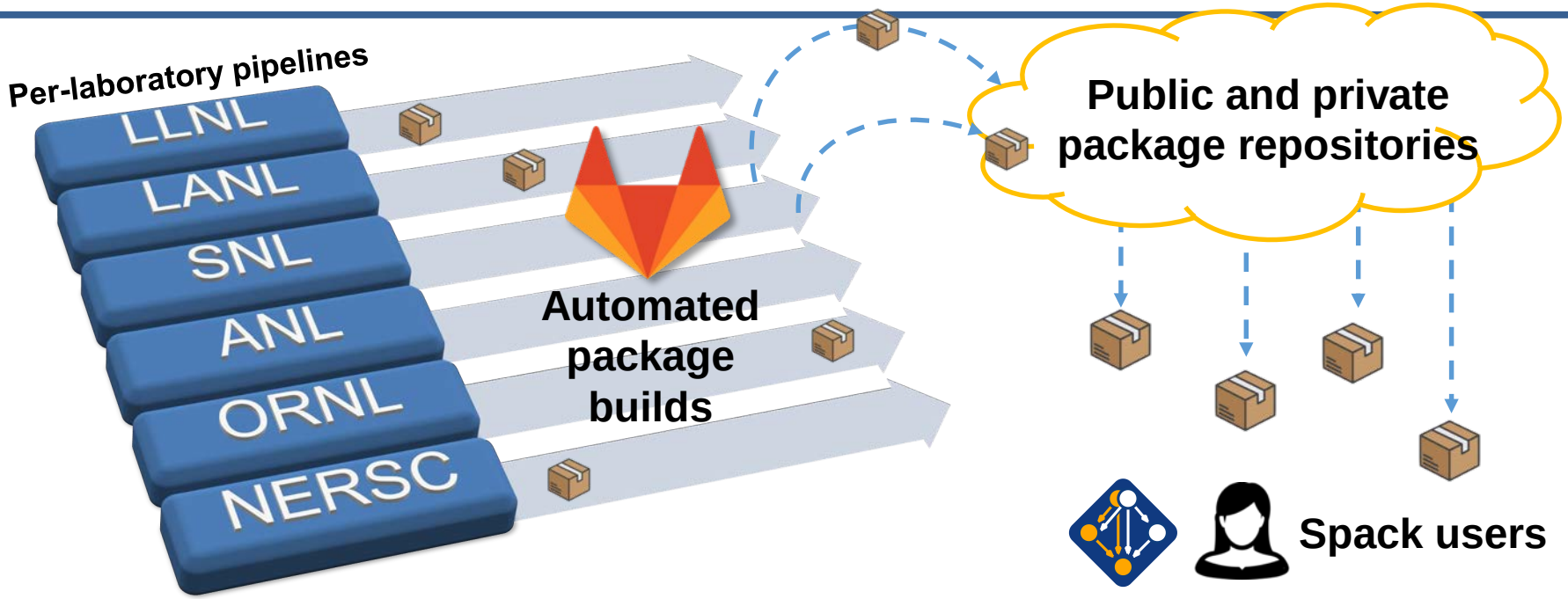
Merge PR

docker hub

Automated build creates container with contents of mirror



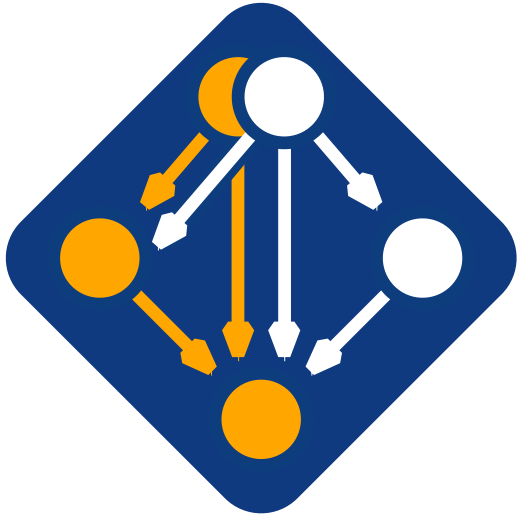
Automated builds using GitLab CI will enable a robust, widely available HPC software ecosystem.



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

Spack 0.15 was released 2 weeks ago

- Better Cray support
- **Packages can specify how they should be found on the system**
  - **spack external find** command
- Better compiler optimization support on macOS
  - apple-clang now its own compiler
- Enhancements and simplification to configuration
  - **spack config add / spack config remove**



spack external find

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

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

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

Logic for finding external installations in package.py

```
packages:
    cmake:
        paths:
            cmake@3.15.1: /usr/local
```

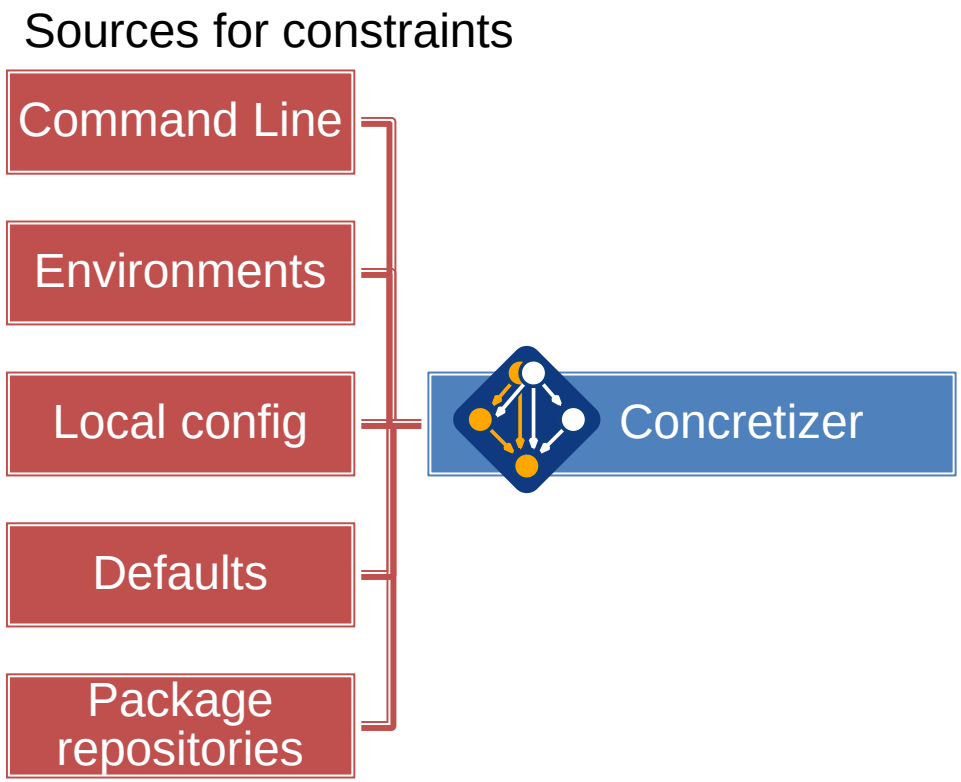
packages.yamlconfiguration

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

Getting external libraries right is tricky

- Current support for external finding is really for build dependencies
- Can work for dependencies like MPI that have well-defined commands
  - mpicc –showme can be used to query information about libraries
  - Provides well defined versions, link path
- Without this, we'd need to inspect libraries, which hard:
  - Are they built for the right architecture?
  - Are they ABI compatible?
  - What variants are enabled?
  - What version is the library?
- Future work: figure out how to detect more libraries **safely**
  - Could look at tools like pkg-config for this

The concretizer has gotten pretty complicated



- Current implementation is ad-hoc:
  - Traverse the DAG
  - Evaluate conditions, add dependencies
  - Fill in defaults from many sources
  - Repeat until DAG doesn't change
- Issues:
  - Limited support for backtracking causes some graphs to resolve incorrectly
  - Some constraints are strictly ordered
  - Lots of conditional complexity
- Design doesn't scale to all the criteria
  - Hard to add new features/logic
  - Can be slow

What is managed by dependency managers?

1. **What packages does this project depend on?**
   - This is a property of the project.
   - Developers determine this

2. **What version of each package should I install?**
   - Specified by developers of project and dependencies
   - Version pinning may be too specific
   - Leaving version ranges open leaves room for error

Concerns:

- Correct/compatible versions? Developers manage
- Latest vs. most tested version? Developers manage
- Most secure version? Developers manage

```
{
    "name": "foo",
    "version": "1.0",
    "depends_on": {
        "bar": ">= 2.0",
        "baz": ">= 3.0"
    }
}
```

Simple package model

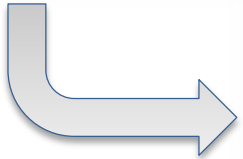
It's hard for developers just to manage packages and versions

With a more diverse ecosystem, there's more that needs to be managed

- **Build configuration options**
  - Optional features/interfaces
  - Choice of parallelism model
    - OpenMP, CUDA, HIP, etc.
- **Interfaces: which library implementation**
  - MPI: MPICH, OpenMPI, MVAPICH
  - BLAS: OpenBLAS, Intel MKL, ARM math libs, etc.
  - CUDA versions
- **Which compiler**
  - Intel, gcc, PGI, clang, XL, AMD, Cray, NAG, others…
  - Compiler version?
  - Which optimization flags?
  - Which runtime libraries (libstdc++, fortran ABIs)
  - Potentially mixed compilers
- **Microarchitecture**
  - Mostly SIMD instruction features
    - AVX-512, AVX-256, SSE, ARM SVE, etc.

```
{
    "name": "foo",
    "version": "1.0",
    "depends_on": {
        "bar": ">= 2.0",
        "baz": ">= 3.0"
    }
}
```

Simple package model



Complex package model


```
{
 "hdf5": {
  "version": "1.10.6",
  "arch": {
   "platform": "darwin",
   "platform_os": "mojave",
   "target": {
    "name": "skylake",
   }
  },
  "compiler": {
   "name": "clang",
   "version": "10.0.0-apple"
  },
  "namespace": "builtin",
  "parameters": {
   "cxx": false,
   "debug": false,
   "fortran": false,
   "hl": false,
   "mpi": true,
   "pic": true,
   "shared": true,
   "szip": false,
   "threadsafe": false,
   "cflags": ["-O3", "-g"],
   "cxxflags": ["-O3", "-g"],
   "fflags": ["-O3", "-g", "-fdefault-double-8"],
  },
  "dependencies": {
   "openmpi": {
    "hash": "tvjyinwp7x3eb5eit3xwgwjrt73o4s6r",
    "type": ["build", "link"]
   },
   "zlib": {
    "hash": "licwn64il2mmstixvgfdmjjgby7ay3ey",
    "type": ["build", "link"]
   }
  },
  "hash": "qwzmhcempklkpf5ih7dw6vasxpoklt5c",
  "build_hash": "d44a2dh37ecyex3ifnztq7iosalnrqnr"
},
```

SAT solvers look appealing, but they're very low-level

Some options:

- **picosat** (used by Conda): basic Boolean SAT solver
  - A basic SAT solver finds *any* valid solution
  - We need to optimize for a lot of different criteria
  - We'd like to be able to use numbers, some math in the solve

- **libsolv**: very targeted towards traditional package model
  - Packages, versions, standard formats, picking latest version

Doing optimization in a SAT solver is hard!

- Conda implements its own math routines in pure SAT
- This is kind of like implementing your own binary adders and multipliers ☺
- Apparently a lot slower than libsolv (cf. Mamba project using libsolv in Conda)

Some higher-level solver options

- **SMT: Satisfiability modulo theories**
  - Z3 seems to be the industry standard: very powerful, very active community
  - Support for integer math, implications, higher level logic operations
  - Support for multi-criteria optimization
  - Traction in the formal verification community
  - Nice high level Python interface
  - Can generate unsatisfiable cores and proofs for error cases (but proofs are complex)

- **ASP: Answer Set Programming (not the other ASP)**
  - Potassco project seems to be the most actively developed/active (and very fast)
  - Nice prolog-like first-order logic syntax; boils down to SAT
  - Support for multi-criteria optimization
  - Python interface
  - No support for generating unsat cores or proofs

We ended up implementing a prototype concretizer in ASP

- Used Clingo, the Potassco grounder/solver package
- ASP program has 2 parts:
  1. Large list of facts generated from our package repositories
     - 6,000 – 9,000 facts is typical – includes dependencies, options, etc.
  2. Small logic program (~130 lines)
- New algorithm (at least our part) is conceptually simpler:
  - Generate facts for all possible dependencies
  - Send facts and our logic program to the solver
  - Build a DAG from the results
- Solve time is much faster than existing concretizer
  - Typically a fraction of a second (so far), plus parsing
  - *Can* fall off a cliff – it's NP-complete after all

```
%-----------------------------------------------------------------------------
% 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

ASP makes it easy to put the logic in one place

Define the space:
each package must be assigned exactly one version.

Disallow conflicted versions

Minimize the total of all version Weights (more on this later)

```
% If something is a package, it has only one version and that must be a
% possible version.
1 { version(P, V) : version_possible(P, V) } 1 :- node(P).

% If a version is declared but conflicted, it's not possible.
version_possible(P, V) :- version_declared(P, V), not version_conflict(P, V).

% version weight and optimization
version_weight(P, V, N) :- version(P, V), version_declared(P, V, N).
#minimize{ N@8,P,V : version_weight(P, V, N) }.
```

Previously complicated logic became very simple

- Every node in the DAG has a compiler and a target architecture
  - Some compilers don't support generating code for some targets
  - But we want to pick the best target possible for each compiler

- Previously this required some complicated logic mixed in with the rest of the solve

Each node has 1 target assigned

Disallow cases where the compiler doesn't support the target.

Minimize the total weight of all targets

```
% one target per node -- optimization will pick the "best" one
1 { node_target(P, T) : target(T) } 1 :- node(P).

% can't use targets on node if the compiler for the node doesn't support them
:- node_target(P, T), not compiler_supports_target(C, V, T),
   node_compiler(P, C), node_compiler_version(P, C, V).

% if a target is set explicitly, respect it
node_target(P, T) :- node(P), node_target_set(P, T).

% each node has the weight of its assigned target
node_target_weight(P, N) :- node(P), node_target(P, T), target_weight(T, N).
#minimize{ N@5,P : node_target_weight(P, N) }.
```

Dependency logic is pretty concise, too (even with virtuals)

- It was easy to express what were previously pretty complicated constraints:
  - There can be at most one provider of any virtual dependency in the DAG
  - Depending on a virtual means you depend on one of its providers
  - Preferences for virtuals can come from multiple sources
  - Pick the most preferred virtual packages
- Each of these sections stands alone and is easy to compose with others

```
% declared dependencies are real if they're not virtual
depends_on(P, D, T) :- declared_dependency(P, D, T), not virtual(D), node(P).

% if you declare a dependency on a virtual, you depend on one of its providers
1 { depends_on(P, Q, T) : provides_virtual(Q, V) } 1
    :- declared_dependency(P, V, T), virtual(V), node(P).

% if a virtual was required by some root spec, one provider is in the DAG
1 { node(P) : provides_virtual(P, V) } 1 :- virtual_node(V).

% for any virtual, there can be at most one provider in the DAG
provider(P, V) :- node(P), provides_virtual(P, V).
0 { provider(P, V) : node(P) } 1 :- virtual(V).

% give dependents the virtuals they want
provider_weight(D, N)
    :- virtual(V), depends_on(P, D), provider(D, V),
       pkg_provider_preference(P, V, D, N).
provider_weight(D, N)
    :- virtual(V), depends_on(P, D), provider(D, V),
       not pkg_provider_preference(P, V, D, _),
       default_provider_preference(V, D, N).

% pick most preferred virtual providers
#minimize{ N*R@9,D : provider_weight(D, N), root(P, R) }.
```

Not everything was simple

- The learning curve for ASP is fairly high.
  - If you haven't been exposed to this before, it can take a while to get in the right mindset
- The shorter the program, the more thought per line
  - The examples before are simple to talk about and they're easy to maintain
  - Writing all of this from scratch took a lot of thought (at least for me)
- Structuring optimization criteria can be a challenge
  - Took a little while to really think through the implications
  - Maximizing criteria tend to expand the DAG unnecessarily, so had to learn to prefer minimization to maximization for most things.
  - Deciding the order in which to optimize different criteria involves some tradeoffs
- The solver is *very* aggressive, which can lead to some surprising cases
  - hdf5~mpi ^mpich

Sometimes the solver can be overly aggressive

- Previous solver couldn't figure out how to toggle build options, e.g.:

  spack install hdf5 ^mpich

- This would fail because mpich is optional; it's only in the DAG if the mpi variant is enabled:

  spack install hdf5 +mpi ^mpich

- But the new solver can be too smart for its own good . Consider:

  spack install hdf5 -mpi ^mpich

- This quickly finds a really obscure way to depend on MPI:

  hdf5 → libaec → cmake → libarchive → lz4 → valgrind → mpi

- Need to disable searches through build dependencies (cmake) to avoid this kind of weirdness

Getting information about errors is still tough

- **Good error messages are important for unsatisfiable cases**
  - Need to be able to tell the user something useful about the problem
  - PubGrub is very good at this
- **PubGrub essentially generates a proof of *why* the DAG isn't satisfiable**
  - Tells you the salient constraints, points you to what to change
- **Potassco currently doesn't have great ways to get this information**
  - No unsatisfiable cores or proofs
- **Z3 has support for proof generation, so we're looking at trying it**
  - Z3 proofs are complicated; challenge to translate them to good messages
  - This is a work in progress

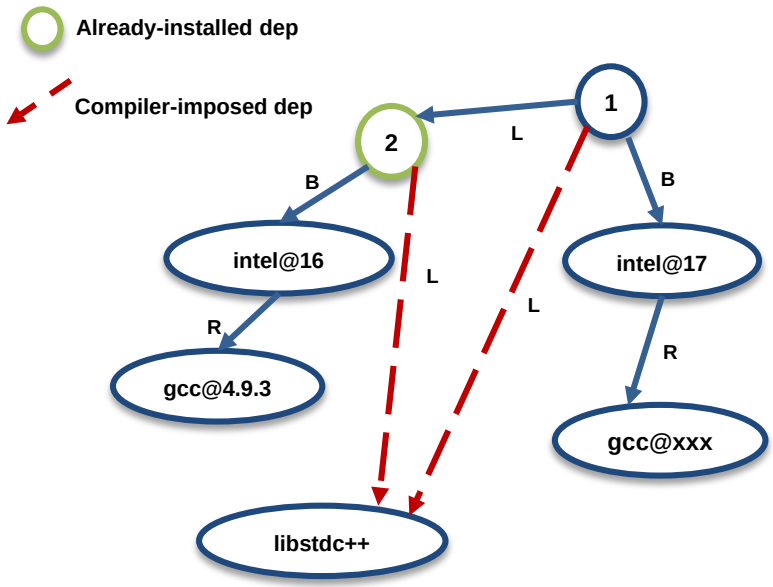
Spack 0.16 Roadmap: permissions and directory structure

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

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

Spack 0.16 roadmap: compilers as dependencies

- **We need deeper modeling of compilers to handle complex ABI issues**
  - libstdc++, libc++ compatibility
  - Compilers that depend on compilers
- **Future GPU, OpenMP target, etc. libraries have similar issues**
  - Entire stack for a large code needs to be consistent
  - We currently do not have visibility into what's under the compiler
- **Packages that depend on languages**
  - Depend on **cxx@2011**, **cxx@2017**, **fortran@1995**, etc.
  - Model langauges, openmp, cuda, etc. as virtuals



Compilers and runtime libs fully modeled as dependencies

PubGrub

- Natalie Weizenbaum implemented awesome error reporting in Pub, the package manager for Dart
- Builds on a basic CDCL SAT solver with a data structure to keep track of conflicts and to generate great error messages
  - Model of PubGrub so far seems to be package/version
  - Has some custom callbacks to evaluate version constraints
- Optimization is done by exploring versions in order
  - We need multi-criteria optimization – more complex tactics
  - lots of peoples' life work has gone into faster solvers than we think we could implement ourselves.
- Worried about implementing a custom solver in Python
  - We're solving more complex problems than most tools
  - Poetry, other Python-native solvers can already be quite slow, and they only deal with packages and versions