

Software Packaging and Environment Management

In the context of RSE/HPC

Introduction

- Today you'll see a lot about building and using containers
- How you building the software that goes into containers is also important
- This session covers:
 - Software packaging
 - Overview
 - Pitfalls to avoid
 - Solutions
 - 'Traditional'/Centralised/HPC environment deployment
 - Overview
 - Spack
 - Pixi (Conda/Mamba)


Software Packaging

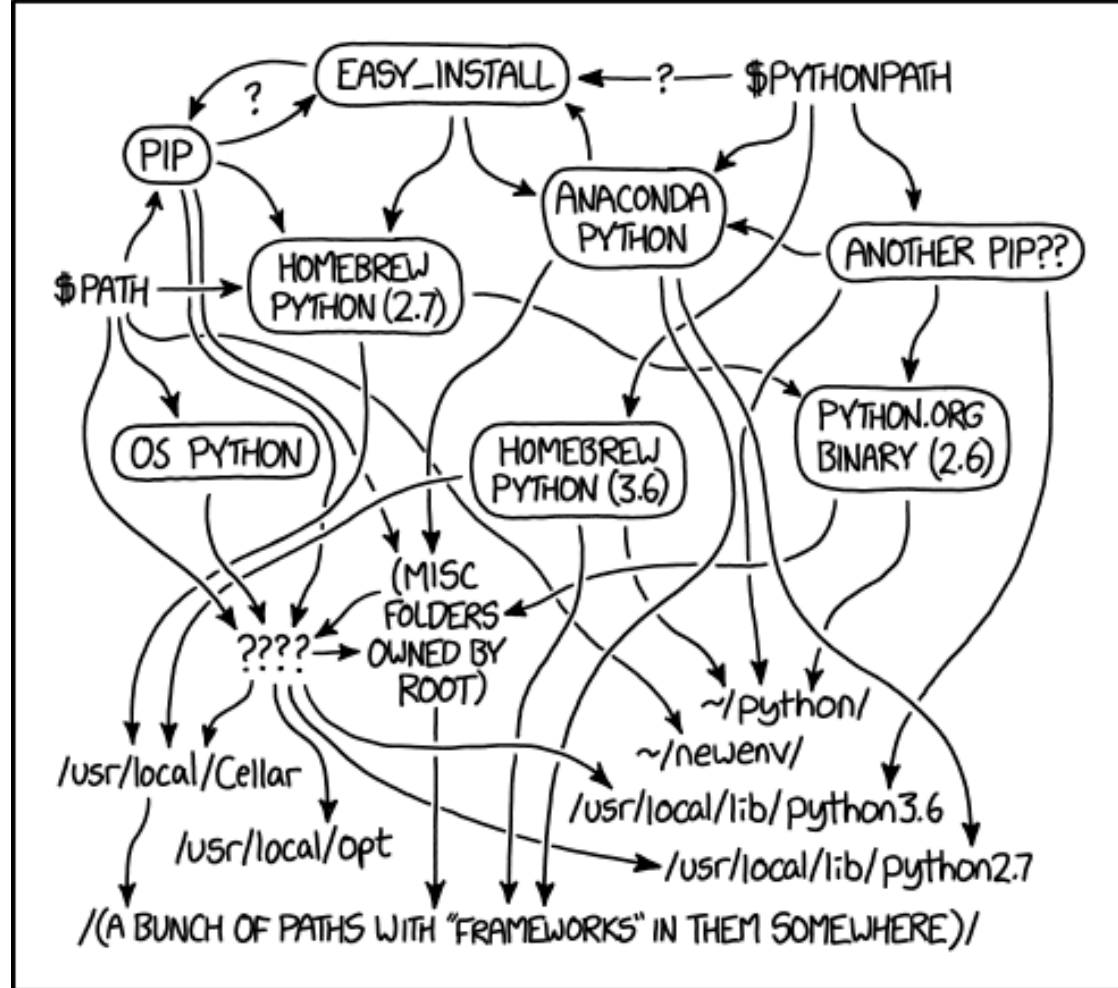
Everything you don't want to know, but should

Dependency Hell

Historically a pain in the ass

Dependency hell is a colloquial term for the frustration of some software users who have installed software packages which have dependencies on specific versions of other software packages.

- Many dependencies
- Long chains of dependencies
- Version conflicts (explicit or not) 
- Circular dependencies
- 'Meta' dependencies (multiple package managers)



MY PYTHON ENVIRONMENT HAS BECOME SO DEGRADED
THAT MY LAPTOP HAS BEEN DECLARED A SUPERFUND SITE.

Escaping Hell

Luckily we're in ✨*the future*✨ and there are some solutions

Know your use case:

- Application - can be completely isolated?
- Library - installed alongside other packages?
- Support requirements - concrete (``python==3.13``), range (``python>=3.10``), wildcard (``python==*``)?
- Targeting hosts?
 - Single - ``linux x86-64``
 - Many - Linux, Windows, MacOS, etc...
- Specific requirements - ``mpi`` , ``avx512`` , etc...?

Keep control over your environment:

- Self contained - avoid external/system dependencies
- Use isolated environments
- Keep concretized dependency lock files

Leverage CI/CD:

- Build and test across supported environments/versions
 - Need system packages? Use matrix of os versions, e.g. ``ubuntu:22`` , ``ubuntu:lts``
 - Supporting range of dependencies? Matrix of major versions
- Check for issues before users notice
 - Test on nightly/release candidates to get heads up
 - Automated PRs to trigger tests on upstream changes

Python

Python has (too) many packaging tools

- Minimalistic:
 - setuptools - most basic packaging tool
 - Flit - minimalistic packaging, quick and simple
 - Hatch - newer project from PyPI, plugins, matrix support
- Locking:
 - Poetry - very (most?) popular
 - Pipenv - builds on top of pip/virtualenv
 - PDM - standards based
- Misc:
 - UV - 'drop-in' replacement for pip/venv, much faster
 - Rye/UV - multiple python version matrix
 - Conda/Pixi - external binary dependencies

Which to pick...? There's no right answer

My personal opinion:

- **Strongly** recommend one with support for lock files
- Past that it doesn't matter much
- Poetry is good in general, others offer 'niche' features
- Try poetry, if it doesn't do what you need test others

Python - Poetry

1. Create a new project
 2. Add a dependency
 3. Dependency contains broad semver bounds by default
 4. This will create an isolated `venv`` for your project
 5. Creates `poetry.lock`` file - exact versions of all deps
 6. Commit both files to repo
- Developers `poetry install``
 - Installs isolated venv
 - With exactly what is in lock file
 - Users `pip install ...``
 - Uses versions from `pyproject.toml``

```
$ poetry new cool-project
```

```
$ cd cool-project
```

```
$ ll
```

```
$ cat pyproject.toml
```

```
$ poetry add pydantic
```

```
$ cat pyproject.toml
```

```
$ cat poetry.lock
```


General - Conda

What if you have non-python dependencies?

Conda is a good option:

- Large ecosystem, thousands of packages
- Recommend using Mamba or Pixi instead (faster)
- Not just python
 - Works for (pretty much) any package

Write a separate Conda 'recipe' file:

- Contains all 'normal' (PyPI) dependencies
- Can contain external dependencies
- Can contain fancier OS/architecture build rules

Note that Conda packaging is typically 'independent':

- Stored in separate 'feedstock' repository

```
context:
  name: extra-data
  version: 1.19.0

package:
  name: '{{ name|lower }}'
  version: '{{ version }}'

source:
  url: https://pypi.io/packages/source/{{ name[0] }}/{{ name }}
  sha256: ee6aa3bae6b406d50765edb502bc37faf834b18a8145a6d4a106

build:
  entry_points:
    - lsxfel = extra_data.lsxfel:main
    - karabo-bridge-serve-files = extra_data.cli.serve_files:run
    - karabo-bridge-serve-run = extra_data.cli.serve_run:main
    - extra-data-validate = extra_data.validation:main
    - extra-data-make-virtual-cxi = extra_data.cli.make_virtual_cxi:main
    - extra-data-locality = extra_data.locality:main
  noarch: python
  script: '{{ PYTHON }} -m pip install --no-deps --ignore-installed
  number: 1

requirements:
  host:
    - python >=3.9
    - pip
```

General - Spack

Spack is a package manager for supercomputers, [...] you can build a software stack in Python or R, link to libraries written in C, C++, or Fortran, and easily swap compilers or target specific microarchitectures

- Conceptually similar to Conda, uses custom recipe files
- Much more powerful
- (Can be) much more complex
- Excellent support for optimised builds
- Focus on reproducible packaging/environments
- Both compile-from-source and build cache
- Packages (recipes) written as templates in Python
- Specs (targets) support complex constraints
 - For the package itself
 - Or for its dependencies
- Can be used to build environments
- More on Spack in following section
- Over 8,000 packages in the Spack ecosystem
- Used by many HPC centres

HPC Software and Environment Management

Challenges

Most package managers make many assumptions which do not hold in HPC environments

Standard package managers assume:

- Tight coupling between source code and target
- Portable binaries/build artifacts
 - Massively simplifies packaging
 - Requires (relatively) generic builds
 - Bad for performance optimisation
- Same toolchain across ecosystem
 - Same compiler
 - Same runtime libraries
 - Same language version

But in a HPC environment:

- Often start from source code you need to compile
- Optimized builds targeted to system architecture
- Specific optimised toolchain/library present on OS
- May need many variants of same package
 - e.g. ``mpich``, ``openmpi``, ``infiniband``
- Restrictions from system
 - e.g. ancient OS
- Restrictions from packages
 - e.g. ancient Fortran package w/ extremely specific dependencies

What About Containers?

Important differences between HPC clusters and cloud/containerised usecases

	HPC	Cloud/Containerised Computing
Environment	Pre-configured/centrally managed, use of environment modules (e.g. Lmod)	User-defined containerized environments (Docker, Singularity)
Isolation	Shared dependencies, managed via modules	Strong isolation, dependencies bundled
Reproducibility & Portability	Challenging due to system libraries, optimised builds, limited portability	High reproducibility with consistent container images, definition files, repos
Management & Updates	(Mostly) admin managed updates; limited direct user control	Direct control via CI/CD pipelines for rapid iteration
Performance Optimization	(Ideally) optimized for specific hardware, max system integration (e.g., tuned MPI libraries)	Abstracted hardware details, typically generic builds, may require tuning

What About Containers?

Important differences between HPC clusters and cloud/containerised usecases

Containers are excellent for creating and distributing an **already built** software stack:

- Typically done by using OS/3rd party package managers
 - Which provide generic unoptimised binaries
 - Difficult to integrate with system dependencies
- Creating optimised container builds is as challenging as building all your software from scratch
- This is completely unfeasible to do manually with basically any non-trivial environment

*Following slides stolen borrowed from:

ATPESC 2024 Software Track
Spack: Package Management for HPC
Todd Gamblin, Spack Project Lead

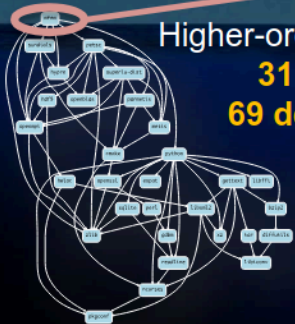
<https://extremecomputingtraining.anl.gov/wp-content/uploads/sites/96/2024/08/ATPESC-2024-Track-3-Talk-4-Gamblin-Spack.pdf>

Modern scientific codes rely on icebergs of dependency libraries

MFEM:

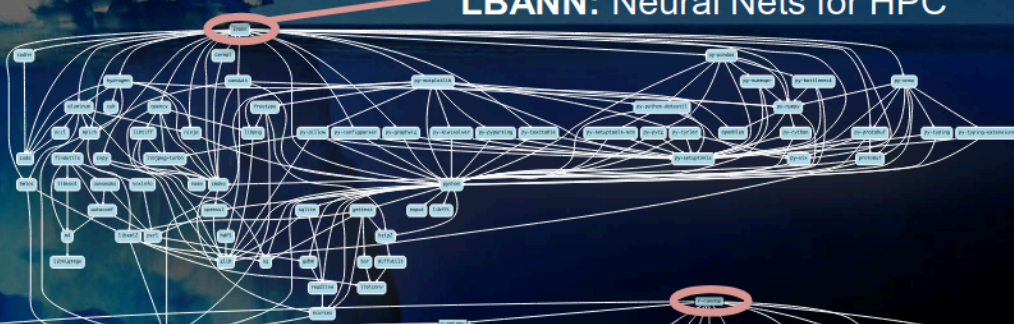
Higher-order finite elements

**31 packages,
69 dependencies**



**71 packages
188 dependencies**

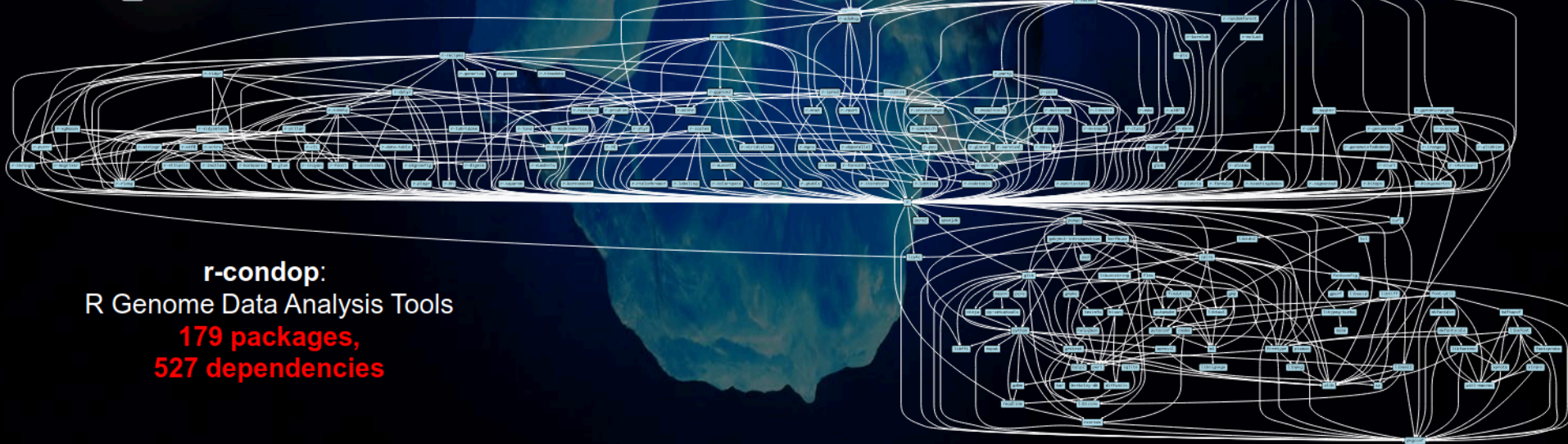
LBANN: Neural Nets for HPC



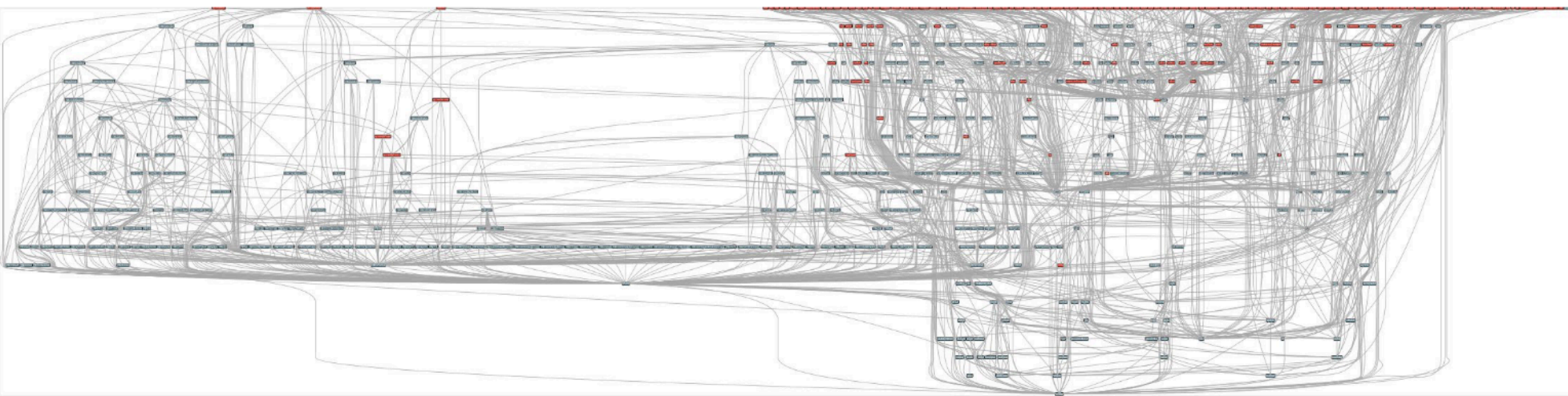
r-condop:

R Genome Data Analysis Tools

**179 packages,
527 dependencies**



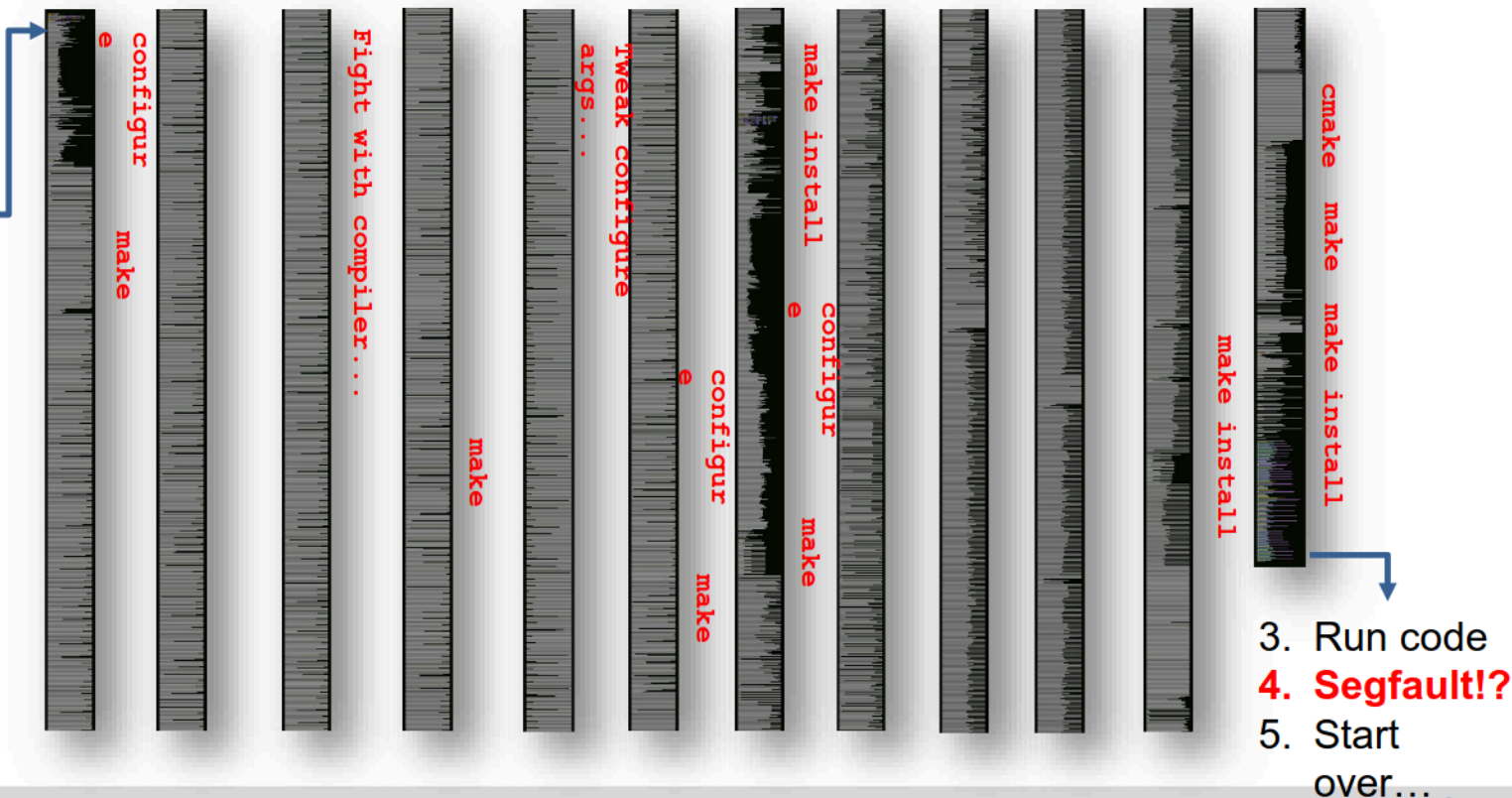
ECP's E4S stack is even larger than these codes



- Red boxes are the packages in it (about 100)
- Blue boxes are what *else* you need to build it (about 600)
- It's infeasible to build and integrate all of this manually

How to install software on a supercomputer, circa 2013

- ## 2. Start building!



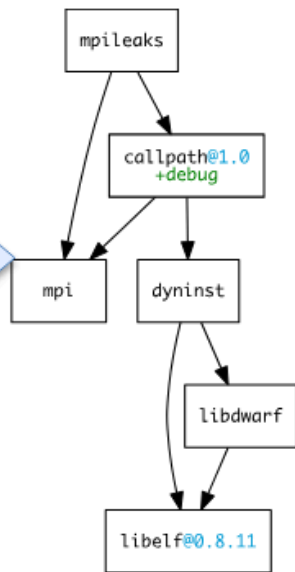
Concretization fills in missing configuration details when the user is not explicit.

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

User input: *abstract* spec with some constraints

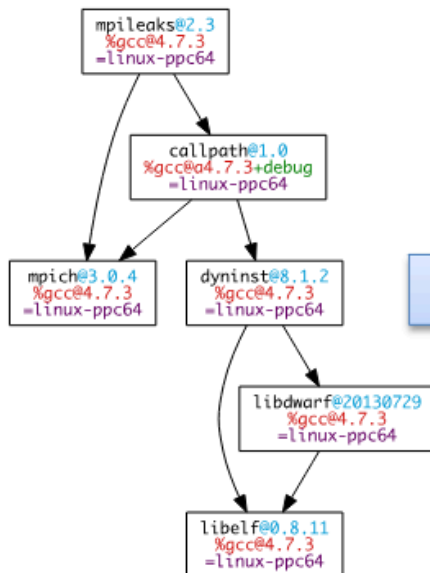
spec.yaml

Normalize



Abstract, normalized spec with some dependencies

Concretize



Concrete spec is fully constrained and can be passed to install

Store

```
spec:
- mpileaks:
  arch: linux-x86_64
  compiler:
    name: gcc
    version: 4.9.2
  dependencies:
    adept-utils: kszrtkpbzac3ss2ixcjkorlaybnpt4
    callpath: bah3f4h4d2n47mgycej2mtrnrivvxy77
    mpich: aa4ar0ifj23yjqmdabeakepejcl17213
    hash: 33hjhx7p0gyzn3ptgyes7sghypruh
    variants: {}
    version: '1.0'
- adept-utils:
  arch: linux-x86_64
  compiler:
    name: gcc
    version: 4.9.2
  dependencies:
    boost: teesjv7ehpe5kssjim5dk43a7qnowlq
    mpich: aa4ar0ifj23yjqmdabeakepejcl17213
    hash: kszrtkpbzac3ss2ixcjkorlaybnpt4
    variants: {}
    version: 1.0.1
- boost:
  arch: linux-x86_64
  compiler:
    name: gcc
    version: 4.9.2
  dependencies: {}
  hash: teesjv7ehpe5kssjim5dk43a7qnowlq
  variants: {}
  version: 1.59.0
...
```

Detailed provenance stored with installed package

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

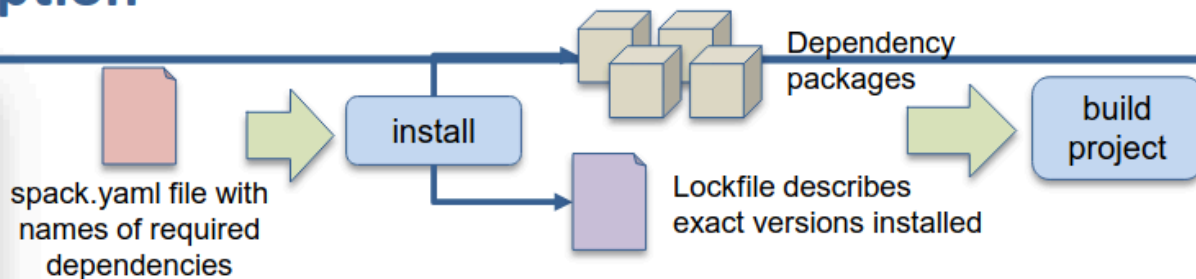
Simple spack.yaml file

```
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": {
    "6s63so2kstp3zyvjzeglndmavy6l3nul": {
      "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": {
```



- 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 Cmdr(Package):
    executables = ['cmdr']

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

        cmdr = spawn.util.Executable.Executable(exe_to_path['cmdr'])
        output = cmdr('-version', stdout=True)
        if output:
            match = re.search(r'cmdr,version=(\S+)', output)
            if match:
                version_str = match.group(1)
                return SpecInfo(execs_in_prefix, version_str)
```

package.py

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

spack.yaml configuration

spack external find

Automatically find and configure external packages on the system

spack test

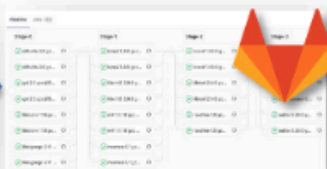
Packages know how to run their own test suites

[illegible]

package.py

[illegible]

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

[illegible]

spack.io



Pixi

Pixi is a package management tool for developers. It allows the developer to install libraries and applications in a reproducible way. Use pixi cross-platform, on Windows, Mac and Linux.

- Descendent of projects like Mamba/Boa which aimed to improve Conda performance

Main features:

- Extremely fast
- Locking
- Packable/relocatable
- Think of it as Conda + locking + matrix envs

```
$ pixi init      # create new workspace
$ pixi add ...   # add dep to project
$ pixi install   # install all deps
$ pixi shell     # spawn shell in env
```

References

- Poetry
- Spack
- Pixi
- ATPESC 2024 - Spack
- Flit
- Hatch
- Pipenv
- PDM
- Rye

Questions?