Binary packaging for HPC with Spack

HPC, Big Data, and Data Science Devroom at FOSDEM 2018

Brussels, Belgium

Feburary 4, 2018

Todd Gamblin
Center for Applied Scientific Computing
LLNL

LLNL-PRES-745747
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

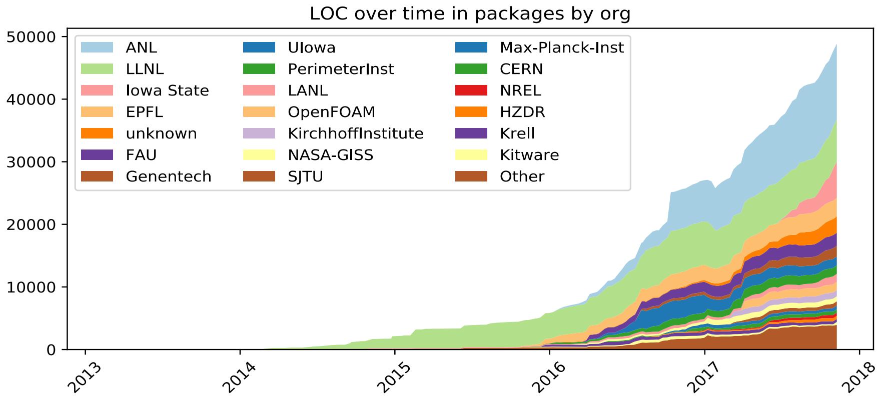
Lawrence Livermore National Laboratory

Spack is a general purpose, from-source package manager

- Inspired somewhat by homebrew and nix
- Targets HPC and scientific computing
  - Community is growing!
- Goals:
  - Facilitate experimenting with performance options
  - Flexibility. Make these things easy:
    - Build packages with many different:
      - compilers/versions/build options
    - Change compilers and flags in builds (keep provenance)
    - Swap implementations of ABI-incompatible libraries
      - MPI, BLAS, LAPACK, others like jpeg/jpeg-turbo, etc.
  - Build software stacks for scientific simulation and analysis
  - Run on laptops, Linux clusters, and some of the largest supercomputers in the world

Spack

https://spack.io



Spec CLI syntax makes it easy to install different ways

```
$ 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 cflags="-O3 –g3"                 setting compiler flags
$ spack install mpileaks@3.3 ^mpich@3.2 %gcc@4.9.3            ^ dependency constraints
```

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

- Spec syntax is recursive
  - **^** (caret) adds constraints on dependencies

Spack packages are *templates*: they define how to build a spec

```
from spack import *

class Dyninst(Package):
    """API for dynamic binary instrumentation."""

    homepage = "https://paradyn.org"
    url = "http://www.paradyn.org/release8.1.2/DyninstAPI-8.1.2.tgz"

    version('8.2.1', 'abf60b7faabe7a2e')
    version('8.1.2', 'bf03b33375afa66f')
    version('8.1.1', 'd1a04e995b7aa709')

    depends_on("cmake", type="build")

    depends_on("libelf", type="link")
    depends_on("libdwarf", type="link")
    depends_on("boost @1.42: +multithreaded")

    def install(self, spec, prefix):
        with working_dir('spack-build', create=True):
            cmake('-DBoost_INCLUDE_DIR=' + spec['boost'].prefix.include,
                  '-DBoost_LIBRARY_DIR=' + spec['boost'].prefix.lib,
                  '-DBoost_NO_SYSTEM_PATHS=TRUE'
                  '..')
            make()
            make("install")
```

Simple Python DSL

- Packages are classes (ala homebrew)
- Directives use the same spec syntax

Metadata at the class level

Versions

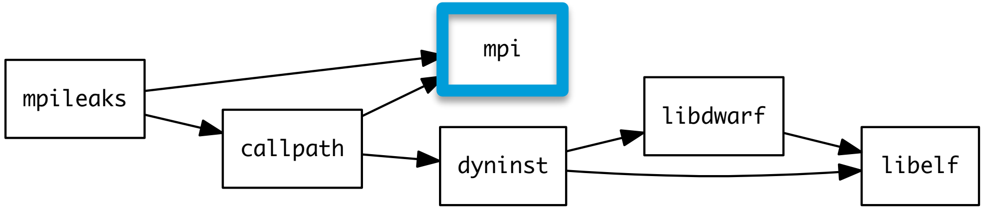
Dependencies

Patches, variants, resources, conflicts, etc. (not shown)

Install logic in instance methods

Depend on interfaces (not implementations) with virtual dependencies

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




```
$ spack install mpileaks ^mvapich
```

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

- Virtual deps are replaced with a valid implementation at resolution time.
  - If the user didn't pick something and there are multiple options, Spack picks.

Virtual dependencies can be versioned:

dependent

```
class Mpileaks(Package):
    depends_on("mpi@2:")
```

provider

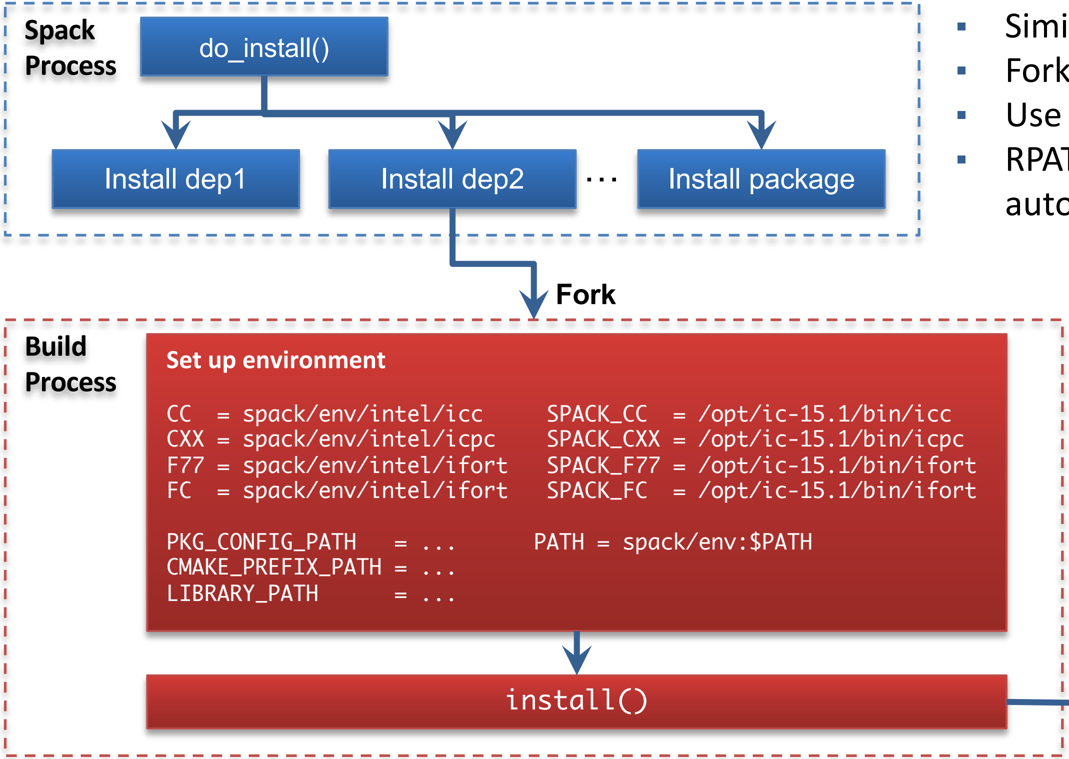
```
class Mvapich(Package):
    provides("mpi@1” when="@:1.8")
    provides("mpi@2” when="@1.9:")
```

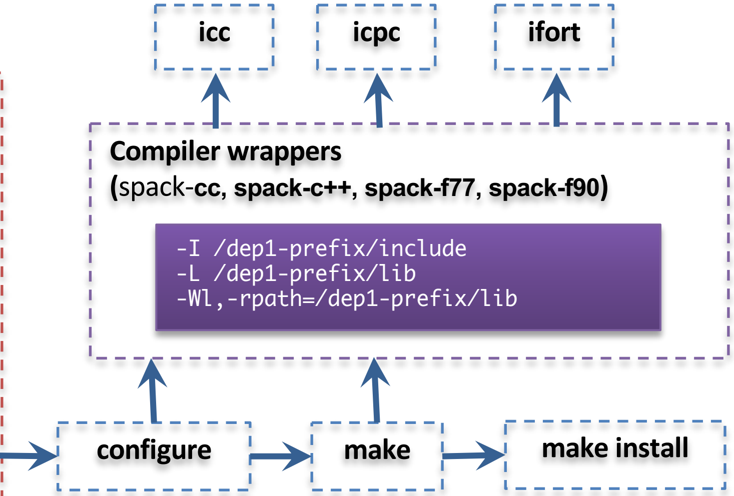
provider

```
class Openmpi(Package):
    provides("mpi@:2.2" when="@1.6.5:")
```

Spack builds packages with compiler wrappers

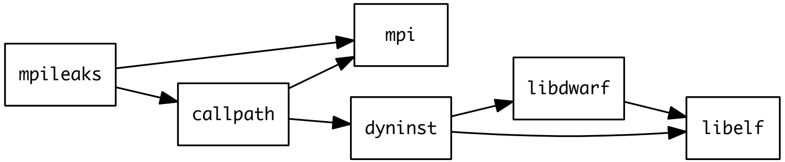
- Similar to homebrew "shims"
- Forked build process isolates environment for each build
- Use compiler wrappers to add include, lib, and RPATH flags
- RPATHs ensure that the correct dependencies are found automatically at runtime.

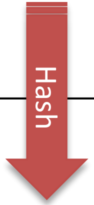




Hashes handle combinatorial software complexity.

Dependency DAG





Installation Layout

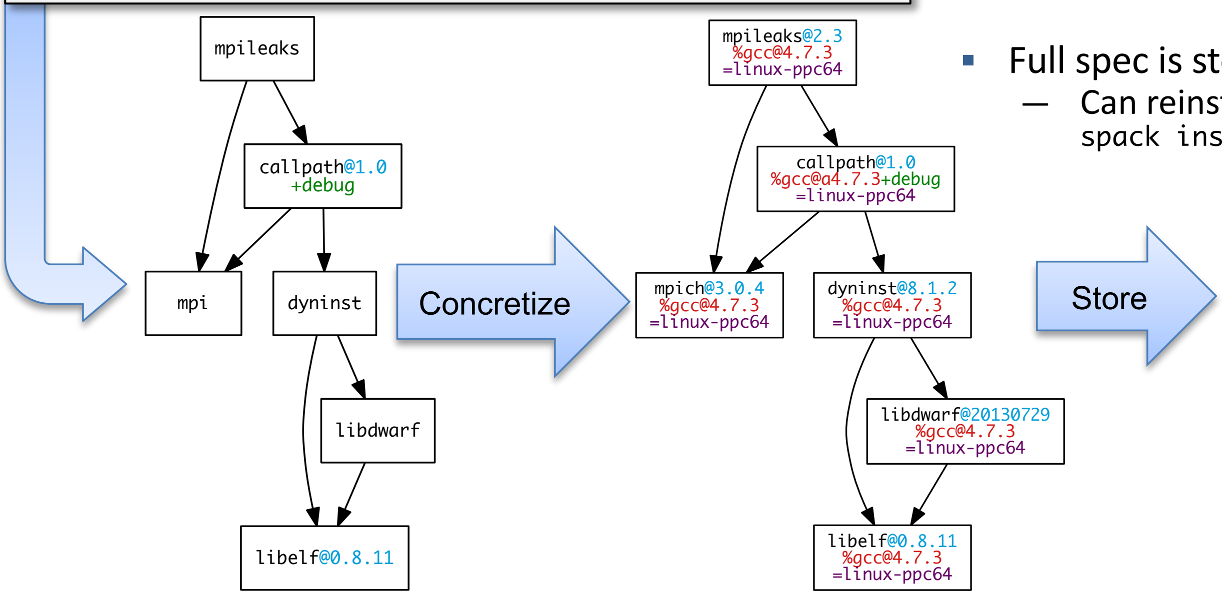
```
spack/opt/
    linux-rhel7-x86_64/
        gcc-4.7.2/
            mpileaks-1.1-0f54bf34cadk/
        intel-14.1/
            hdf5-1.8.15-lkf14aq3nqiz/
    bgq/
        xl-12.1/
            hdf5-1-8.16-fqb3a15abrwx/
    ...
```

- Each unique dependency graph is a unique ***configuration***.
- Each configuration installed in a unique directory.
  - Configurations of the same package can coexist.
- **Hash** of directed acyclic graph (DAG) metadata is appended to each prefix
  - Note: we hash the metadata, not the artifact.
- Installed packages automatically find dependencies
  - Spack embeds RPATHs in binaries.
  - No need to set LD_LIBRARY_PATH
  - Things work *the way you built them*

Spack's dependency model centers around "concretization"

User input: *abstract* spec

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



Abstract, normalized spec with some dependencies.

Concrete spec is fully constrained and can be built.

- Similar to other dependency resolvers, but solves for more than just package and version.
- Full spec is stored in a file in the installation directory
  - Can reinstall same build with:
    `spack install -f spec.yaml`

spec.yaml

```
spec:
- mpileaks:
    arch: linux-x86_64
    compiler:
      name: gcc
      version: 4.9.2
    dependencies:
      adept-utils: kszrtkpbzac3ss2ixcjkcorlaybnptp4
      callpath: bah5f4h4d2n47mgycej2mtrnrivvxy77
      mpich: aa4ar6ifj23yijqmdabeakpejcli72t3
    hash: 33hjjhxi7p6gyzn5ptgyes7sghyprujh
    variants: {}
    version: '1.0'
- adept-utils:
    arch: linux-x86_64
    compiler:
      name: gcc
      version: 4.9.2
    dependencies:
      boost: teesjv7ehpe5ksspjim5dk43a7qnowlq
      mpich: aa4ar6ifj23yijqmdabeakpejcli72t3
    hash: kszrtkpbzac3ss2ixcjkcorlaybnptp4
    variants: {}
    version: 1.0.1
...
```

Detailed provenance is stored with the installed package

Source installs are great, but they're slow

- Most people prefer using a binary package manager
  - Binary packages typically use portable code
  - Binary installs are typically a lot slower than what you get from building from source

- We'd like to have the best of both worlds:
  - Optimized buids for specific machine models (skylake, haswell, ivy bridge, etc.)
  - Binary packages available without having to build from source

- What's needed?
  1. Binary packaging capability
  2. Metadata describing architecture-specific builds
  3. Good dependency resolution to select optimized or generic versions of packages

https://github.com/spack/spack/releases/

We recently released Spack v0.11

- 2,178 packages (up from 1,114 a year ago)

- Big features for users:
  - **Relocatable binary packages (spack buildcache)**
  - Full support for Python 3
  - Improved module support; custom module templates using jinja2

- Many improvements for packagers:
  - Multi-valued variants
  - Test dependency type
  - Packages can patch their dependencies (not just themselves)

- Many speed improvements (to Spack itself)

Binary packaging in Spack v0.11

- Spack v0.11 has a new `spack buildcache` command:

```
spack buildcache create <spec>      # create a new binary package
spack buildcache list               # list available binaries
spack buildcache install            # install a binary package (specifically)
```

- Typically, install is not needed; you can just do:

```
spack install --use-cache           # prefer binaries if available
```

- We don't enable binaries by default yet
  - We'll make –use-cache default when we start hosting stable binaries

- Thanks to our collaboration with Fermilab, CERN, and Kitware for this feature!

How to make a binary

1. Set up GNU PG for binary signing
   - Unsigned binaries can be created but are discouraged

```
spack gpg create "Todd Gamblin" tgamblin@gmail.com   # create a new signing keypair
spack gpg init                                             # trust initial keypair
```

2. Install something

```
spack install m4     # install m4
...
spack find
==> 2 installed packages.
-- darwin-elcapitan-x86_64 / clang@8.0.0-apple ------------------
libsigsegv@2.11     m4@1.4.18
```

3. Run spack buildcache create on that thing

```
spack buildcache create -d /path/to/mirror m4     # create a binary package in mirror
```

- Binaries and metadata for the package and its dependencies will be in: `/path/to/mirror/build_cache/`

Binary mirror structure

```
mirror/
└──build_cache
    ├──darwin-elcapitan-x86_64
    |    └──clang-8.0.0-apple
    |      ├──libsigsegv-2.11
    |      |    └── darwin-elcapitan-x86_64-clang-8.0.0-apple-libsigsegv-2.11-5rjv56uui3crfmsgsqgqab52yfhr6t4b.spack
    |      └──m4-1.4.18
    |           └── darwin-elcapitan-x86_64-clang-8.0.0-apple-m4-1.4.18-eqbzieloqiwxfe4drksmekvfqia7mqbu.spack
    ├── darwin-elcapitan-x86_64-clang-8.0.0-apple-libsigsegv-2.11-5rjv56uui3crfmsgsqgqab52yfhr6t4b.spec.yaml
    ├── darwin-elcapitan-x86_64-clang-8.0.0-apple-m4-1.4.18-eqbzieloqiwxfe4drksmekvfqia7mqbu.spec.yaml
    └── index.html
```

- Binaries go in build_cache/<platform>/<compiler>/<pkg-version>

- Metadata for all packages is (currently) kept at the top level
  - We'll need to index these files eventually

- build_cache subdirectory sits inside of a Spack mirror directory
  - Makes it easy to add binaries to an existing source mirror

- This structure is very easy to host in something like S3, a web server, or a shared filesystem

Pointing Spack to a mirror

- Spack v0.11 has a new `spack buildcache` command:

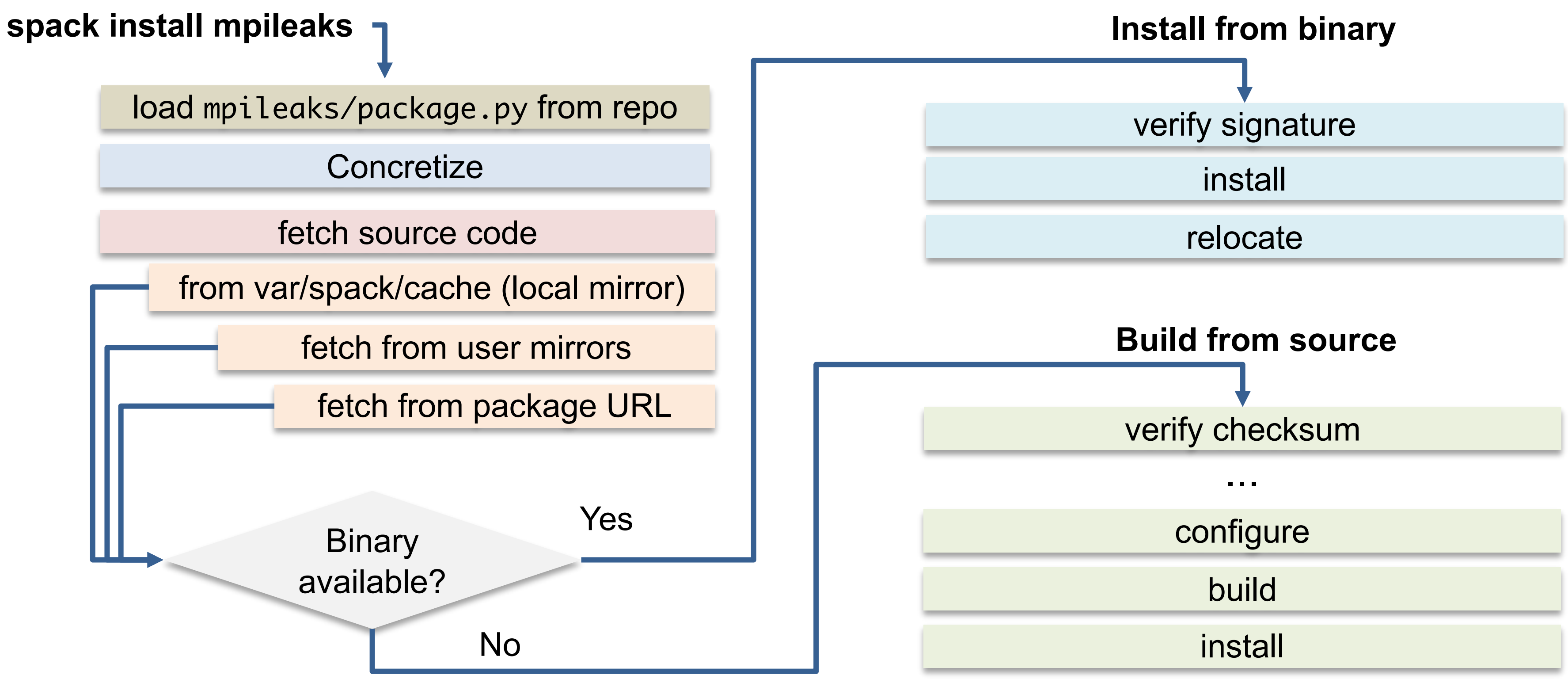
```
$ spack mirror add mypkgs https://example.spack-mirror.com/mirror
```

- You can verify that it worked by looking at what mirrors are configured:

```
$ spack mirror list
mypkgs    https://example.spack-mirror.com/mirror
```

- Mirrors can contain source tarballs and binaries
  - Detailed info in docs on `mirrors.yaml`

How fetching works in spack



What's in a Spack binary package?

```
$ tar tzf darwin-elcapitan-x86_64-clang-8.0.0-apple-m4-1.4.18-eqbzieloqiwxfe4drksmekvfqia7mqbu.spack

darwin-elcapitan-x86_64-clang-8.0.0-apple-m4-1.4.18-eqbzieloqiwxfe4drksmekvfqia7mqbu.tar.gz
darwin-elcapitan-x86_64-clang-8.0.0-apple-m4-1.4.18-eqbzieloqiwxfe4drksmekvfqia7mqbu.spec.yaml
darwin-elcapitan-x86_64-clang-8.0.0-apple-m4-1.4.18-eqbzieloqiwxfe4drksmekvfqia7mqbu.spec.yaml.asc
```

- The binary is just a tarball

- Contains:
  1. Another tarball of the installed prefix
  2. The spec.yaml:
     - describes the build (Spack metadata)
     - Contains a special entry with the checksum of the source tarball (maps spack hash to SHA256)
  3. A signature
     - tells us we can trust the spec.yaml

Why do we checksum source but sign binaries?

- Other systems provide checksums for sources and binaries in their package files
  - e.g., homebrew "bottles"

- In Spack, the number of binaries associated with a source tarball can be very large!
  - We could have thousands of binaries for the same source:
    - Different flags, different build options etc.
  - Each of these would have a different Spack

- Putting checksums for all of these in the package files:
  - Would add a lot of extra bytes to a package repository
  - Is unmaintainable
  - Means that we have to update package.py files whenever we update a mirror

- With signing, the client can trust one or several keys and verify a large number of packages with a small number of public keys

What’s relocation?

- When Spack creates a binary package, it traverses the installation directory and examines the files
  - Uses the file command
- It records the files that need to be relocated after installation:
  - Libraries with RPATHs
  - Shell scripts with #! Lines
- After installation, Spack:
  - rewrites RPATHs with patchelf (Linux) or install_name_tool (macOS)
  - Rewrites #! lines to point to the Spack installation on the installing machine
- Install is faster b/c we record the needed relocations at package creation time

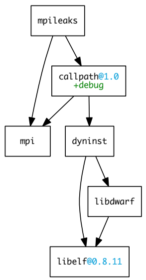
Installation Layout

```
spack/opt/
    linux-rhel7-x86_64/
        gcc-4.7.2/
            mpileaks-1.1-0f54bf34cadk/
                bin/
                      mpileaks-run
                lib/
                      libmpileaks.so
```

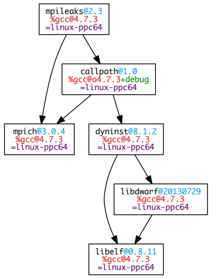
- We try to make root-relative RPATHs when possible, but don’t always get everything.
- We also rewrite RPATHs to directories *within* the spack root fi the install machine uses a different layout.
- We are not currently relocating compiler runtime paths
  - We should. This is work in progress

How do we decide which binaries to fetch?

Current

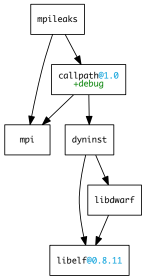


Concretize



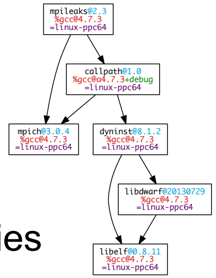
Fetch exact hashes, if available

In progress



Download available specs from mirror

Concretize



Prefer available binaries in concretization

- We currently only fetch binaries if they satisfy the exact hash result of concretization
- This doesn't leave a lot of room for change in the system
  - Small changes in Spack mean having to build from source again
  - Works best on a stable release
- We are working on a new concretizer that will consider available binary specs
  - Doing this better requires a backtracking SAT solve

Spack can ship optimized binaries

- The Spack architecture descriptor currently includes:
  - **Platform:** cray, mac, linux, bgq
    - Meant to represent a family of machines with potentially many OS/target combinations
  - **OS:** rhel6, rhel7, ubuntu14, elcapitan, sierra, centos6, centos7, etc.
  - **Target:**
    - Generic: x86_64, ppc64le, etc.
    - Specific: haswell, ivybridge, knl, power8, power9, etc.

- Some triples:
  - darwin-sierra-x86_64
  - darwin-elcapitan-x86_64
  - cray-cnl6-knl
  - cray-cnl6-haswell

- These architecture descriptors are part of the binary metadata
  - If we can fetch an index of available packages first, we can be picky about what binaries we want
  - This is a core spec parameter in Spack, not just a naming convention for packages

Detecting optimized architectures

- Currently Spack will only use an optimized architecture descriptor on Cray
  - We get the architecture name from the Cray Programming Environment
  - We can know whether we're building for Haswell, Broadwell, KNL, etc.

- We have work in progress that detects these names for Intel, AMD, Power, and ARM hardware (looking at available info in /proc/cpuinfo, etc.)
  - We're planning to shift to a model where we use the specific descriptor by default
  - We would still allow a user to set preferences to build generic if they want
  - Important for CERN and Fermi collaborators who run heterogeneous clusters

- Once this is done, we do plan to make arch-specific binaries available.

Some issues with optimized binaries

- Architectures like ARM don’t lend themselves to concise descriptors.
  - We may need to be more fine-grained here

- We may need to add a more fine-grained architecture descriptor that just exposes the instruction sets available on the machine
  - E.g., instead of “haswell”, put “sse4.2, avx, avx2, etc.”
  - These attributes may actually be easier for packagers and maintainers to use

- We need a setting on the user side (e.g. in packages.yaml) that lets them choose what the minimum architecture is for compatibility
  - This seems easier to do with well known system names

Tuning for optimized binaries may be tricky

- Most compilers give you two knobs to control architecture-specific tuning:
  `-march=[generic|native]`
  `-mtune=[generic|native]`

  - If you tune generic for an older architecture, it will run fine on that architecture *and* on newer architectures.
  - If you tune native, you'll get code that *only* runs fast on the specific architecture, and may not perform as well on future chips.

- In Spack, we might want policies like this:
  - "Generic tuning, with code no later than sandy bridge", e.g. if sandy bridge is the older architecture on a heterogeneous cluster
  - "Native tuning, just for this machine", e.g., if we know we want to optimize for a long-lived, homogeneous cluster.

Building a primary binary mirror for Spack

- We’re currently setting up build automation to create binaries for:
  - All default package configurations in releases (result of `spack install` <name>)
  - Selected other slices of those configuration spaces, e.g.:
    - x MPI versions
    - x Compilers
    - x OS’s
    - x large-scale DOE machines (Cori, Theta, Titan, Summit, etc.)

- Once this is done, we’ll also continuously build packages for the develop branch as PRs come in
  - We’ll need to determine when to purge old builds and binaries
  - Depends on analytics

- We are currently planning to host binaries in S3
  - but Jfrog/bintray sounds interesting. Maybe we should talk to them.

Summary

We built relocatable binary packaging into Spack

Current projects:

1. Binary build infrastructure
2. Better concretization to support optimized binaries
3. Compiler library relocation

Shooting for September to have all of this done.

Spack

https://spack.io

Come and get Spack stickers!

Lawrence Livermore
National Laboratory