How compilers affect dependency resolution in Spack

Package Management Devroom at FOSDEM 2018

Brussels, Belgium

Todd Gamblin
Center for Applied Scientific Computing, LLNL
@tgamblin

Feburary 3, 2018

LLNL-PRES-745770
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

LLNL is a multidisciplinary national security laboratory

- Established in 1952
- Approximately 6,000 employees
- 1 square mile, 684 facilities
- Annual federal budget: ~ $1.42B

High-Performance Computing (HPC) is in the Lab's DNA

Sequoia, a 1.5M-core Blue Gene/Q system.

Spack is a general purpose, from-source package manager

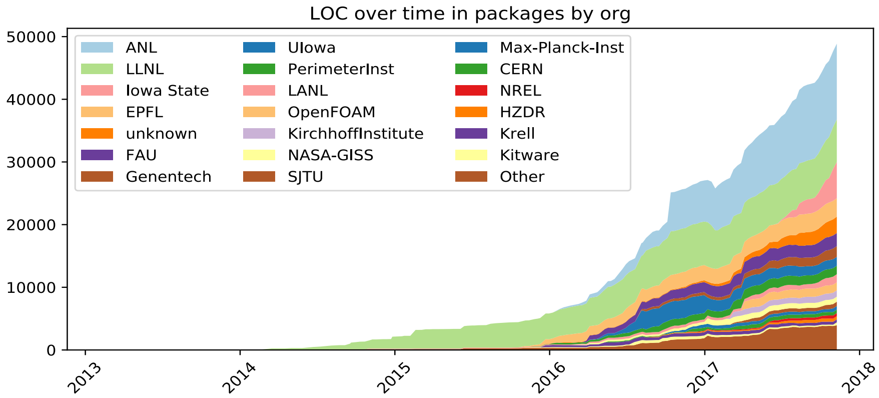
- Inspired somewhat by homebrew and nix

- 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



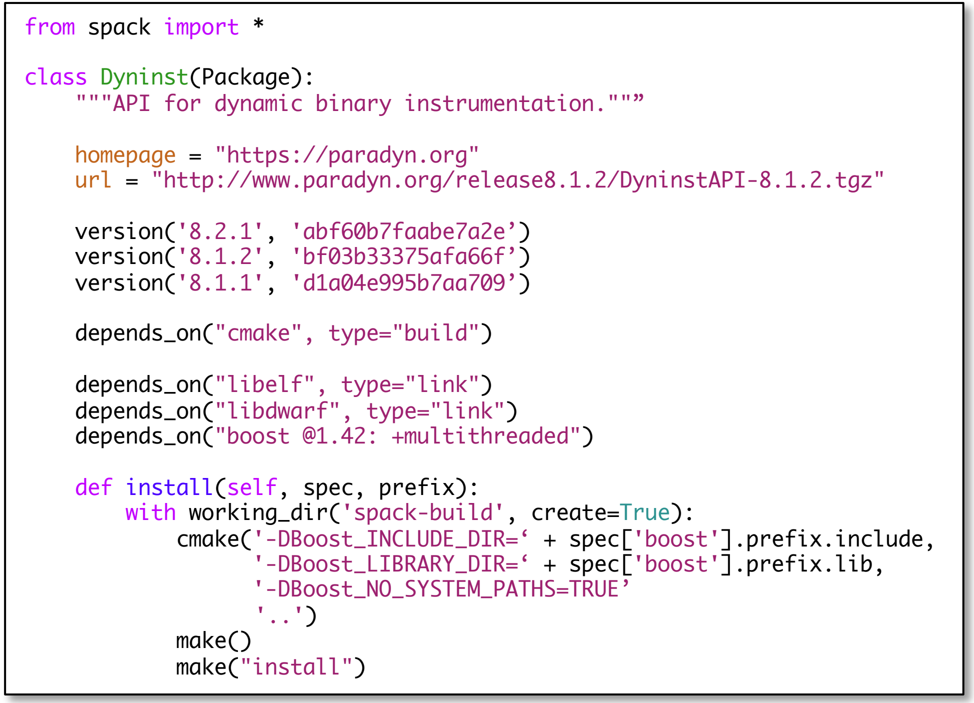
Spec CLI syntax makes it easy to install different ways

```
$ 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

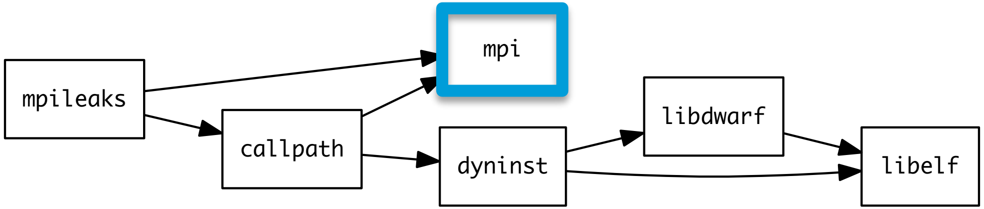
Patches, variants, resources, conflicts, etc. (not shown)

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 dependencies can be versioned:

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

dependent

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

provider

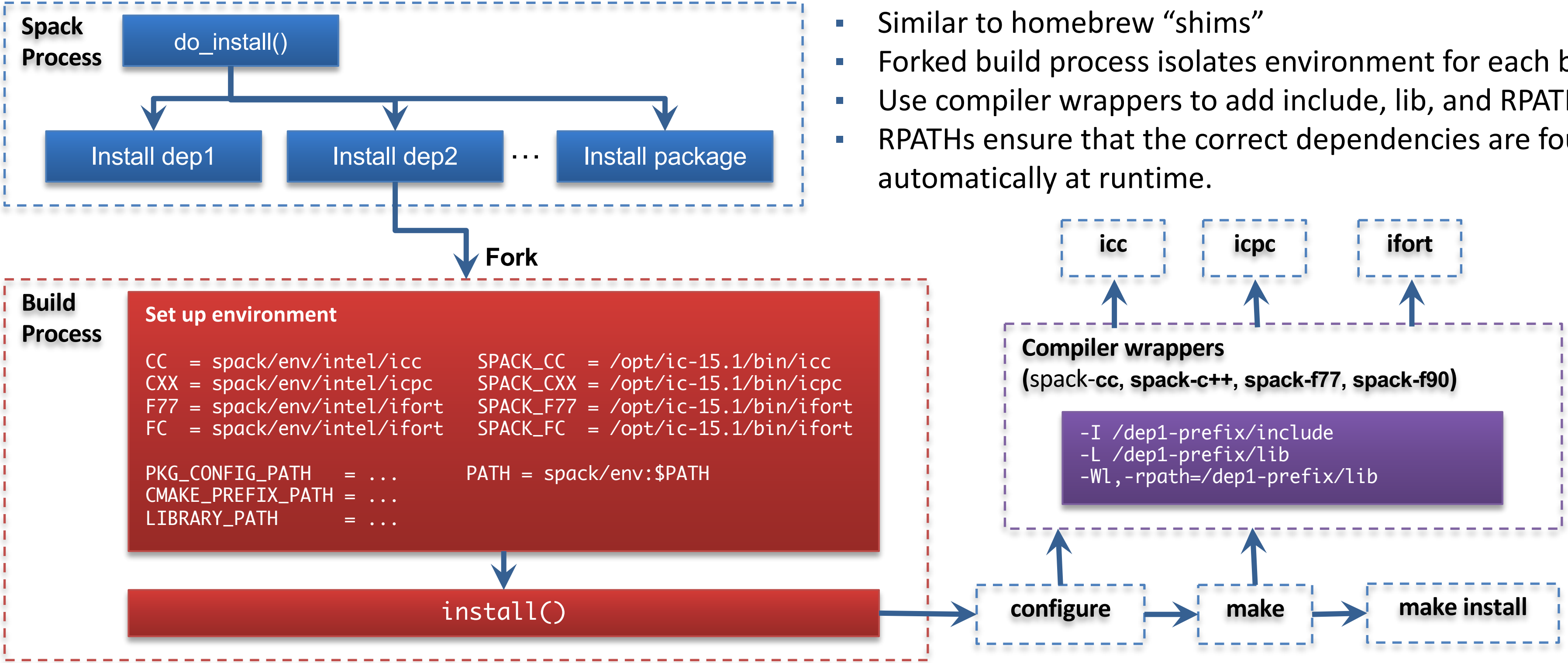
```
class Openmpi(Package):
    provides("mpi@:2.2" when="@1.6.5:")
```

provider

- 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.

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.



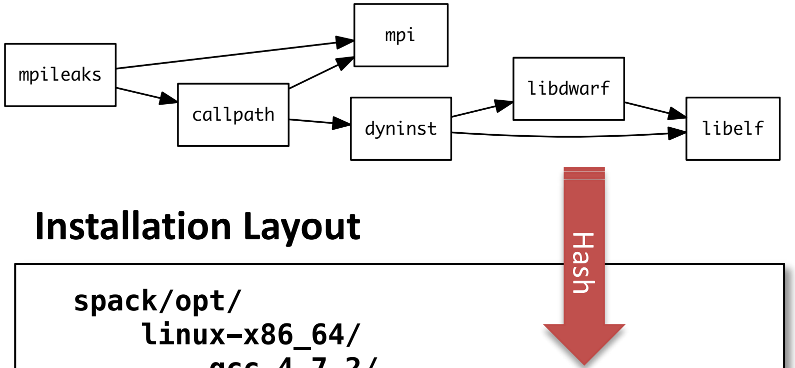
```
CC  = spack/env/intel/icc     SPACK_CC  = /opt/ic-15.1/bin/icc
CXX = spack/env/intel/icpc    SPACK_CXX = /opt/ic-15.1/bin/icpc
F77 = spack/env/intel/ifort   SPACK_F77 = /opt/ic-15.1/bin/ifort
FC  = spack/env/intel/ifort   SPACK_FC  = /opt/ic-15.1/bin/ifort

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
```

Hashes handle combinatorial software complexity.

Dependency DAG



Installation Layout

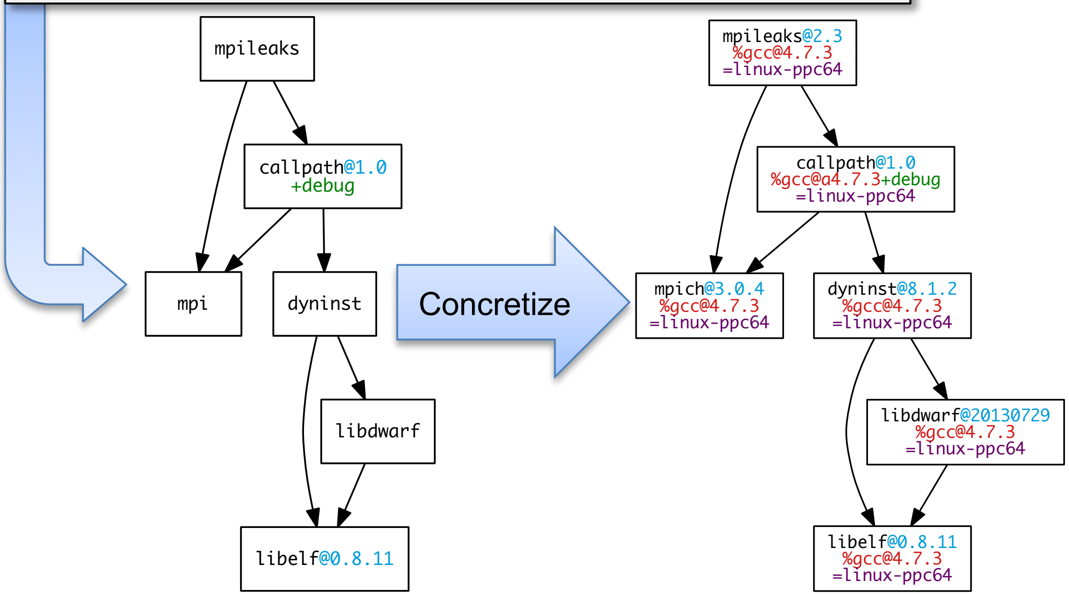
```
spack/opt/
    linux-x86_64/
        gcc-4.7.2/
            mpileaks-1.1-0f54bf34cadk/
        intel-14.1/
            hdf5-1.8.15-lkf14aq3nqiz/
    bgq/
        xl-12.1/
            hdf5-1-8.16-fqb3a15abrwx/
    ...
```

- Each unique dependency graph is a unique ***configuration***.
- Each configuration installed in a unique directory.
  - Configurations of the same package can coexist.
- Hash of 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.

- Solves for more than package/version, but similar to other resolvers
- Dependencies need to be a DAG
- Different dependency types:
  - **Build**: tools run at build time
  - **Link**: things linked with
  - **Run**: things invoked at runtime
- Only one instance of any dependency can be in the concrete DAG.
- Nodes can have different compilers

Why one configuration of a package per DAG?

- Languages like Javascript have support for multi-versions in a DAG
  - (most?) native linkers do not

- You *can* link an executable with libraries that depend on two different versions of, say, libstdc++

- You don't want to do that:
  - First one in which a function is called is loaded (this is a nasty race case)
  - If ABI is different, you'll get a fatal error when the second function version is called

- In general, we can't have two versions of one library *in the same process space*

Why aren't compilers proper dependencies?

They should be, but…

1. We want to mix compilers in one DAG
   - Can't do this with our restriction
   - Dependency model flattens compilers
2. We needed to auto-detect vendor compilers
   - Often required for fastest builds
   - Needed an expedient way to use what's available

```
$ spack compilers
==> Available compilers
-- gcc ---------------------------------
gcc@4.9.3    gcc@7.2.0

-- clang -------------------------------
intel@17.0.1
```

compilers.yaml

```yaml
compilers:
- compiler:
    modules: []
    operating_system: ubuntu14
    paths:
      cc: /usr/bin/gcc/4.9.3/gcc
      cxx: /usr/bin/gcc/4.9.3/g++
      f77: /usr/bin/gcc/4.9.3/gfortran
      fc: /usr/bin/gcc/4.9.3/gfortran
    spec: gcc@4.9.3
- compiler:
    modules: []
    operating_system: ubuntu14
    paths:
      cc: /opt/intel/17.0.1/bin/icc
      cxx: /opt/intel/17.0.1/bin/icpc
      f77: /opt/intel/17.0.1/bin/ifort
      fc: /opt/intel/17.0.1/bin/ifort
    spec: intel@17.0.1
- ...
```

Auto-generated by searching environment

Why do HPC people care about compilers so much?

1. HPC people want to use fancy compilers for high performance

2. On many machines, this requires cross-compiling for the compute nodes
   - Xeon Phi, Blue Gene, etc.

3. Some packages require compiler features, e.g.:
   - OpenMP versions
   - Language levels/verisons (C, C++, and Fortran have this)
   - CUDA
   - etc.

All of these pose some challenges for the dependency model

Fancy vendor compilers

Advantages:

- Intel compiler gets better performance and vectorization than gcc
- Similar for Cray, PGI, XL compilers

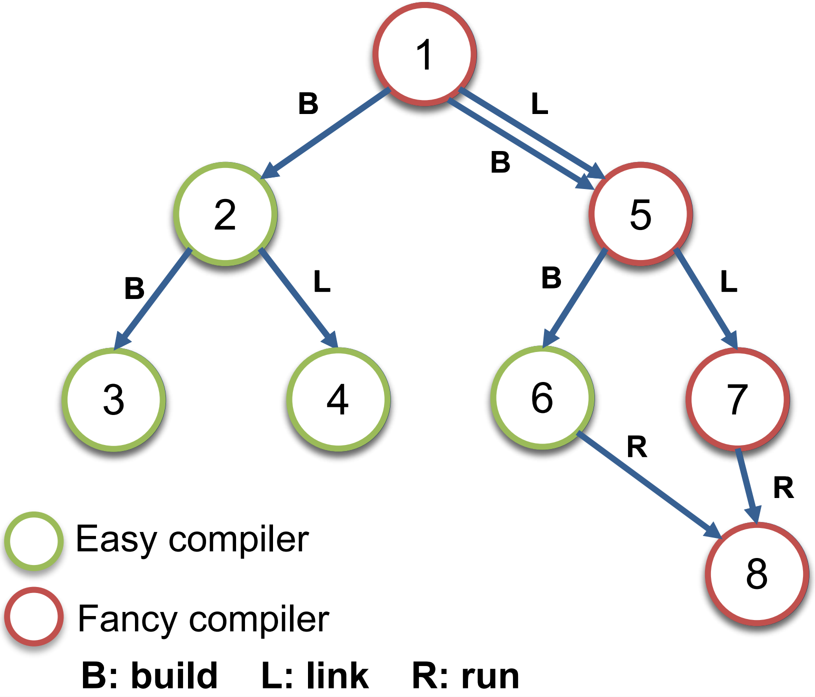
Issues:

- Fancy compilers tend to be hard to work with
  - At least most of the CLI options are consistent these days
- Most OSS projects don’t test with them, so builds are fraught with peril
  - Things like CMake (and its dependencies) don’t always build with XL
  - Typically no reason to build these with anything but the system compiler
  - No performance benefit for vectorizing build tools

How do we handle this?

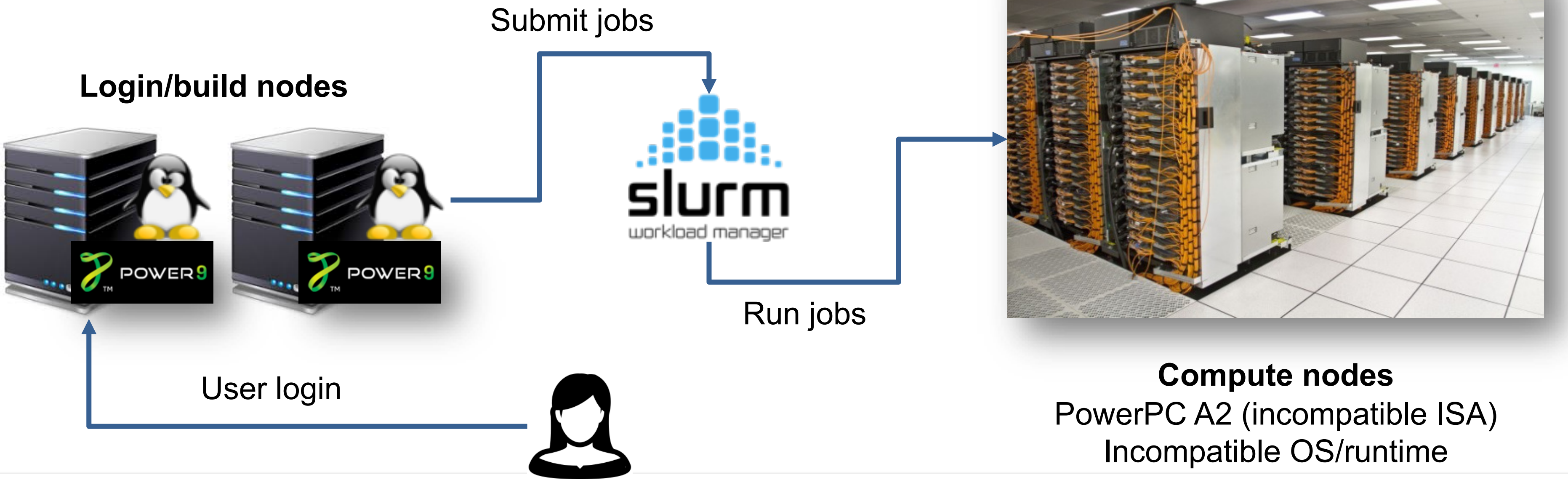
- We want to:
  - Build build dependencies with the "easy" compilers
  - Build rest of DAG (the link/run dependencies) with the fancy compiler
- Works well for porting most scientific codes
  - Results in consistent compilers within processes
- What we actually do is run the concretizer separately for 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
```



Cross-compilation

- Why cross-compile?
  - Your machine has Xeon Phi processors or maybe it’s a BlueGene/Q
  - You need to run a cross-compiler to build for the compute nodes



Submit jobs

Login/build nodes

Run jobs

User login

Compute nodes
PowerPC A2 (incompatible ISA)
Incompatible OS/runtime

Why not build natively on the compute nodes?

- In many cases, building on the compute nodes is very slow
  - There are 72 Atom cores on a Knights Landing (Xeon Phi)
  - Each is only 1.4 Ghz
  - Typically only talk to network filesystem (diskless nodes)

- Many tools (like compilers) are not ported to the compute node
  - Compute node uses a stripped down OS (e.g., BG/Q)
  - Maybe you don’t have that many licenses for your fancy compiler!

- Generally you want to build on the machines with the big cores
  - Fast Xeon front-end nodes
  - Power8/Power9 nodes

How do build dependencies work with cross-compiles?

- Recall some of the dependency types:
  - **Build**: tools run at build time.
  - **Link**: things the package links with
  - **Run**: things the package invokes at runtime.

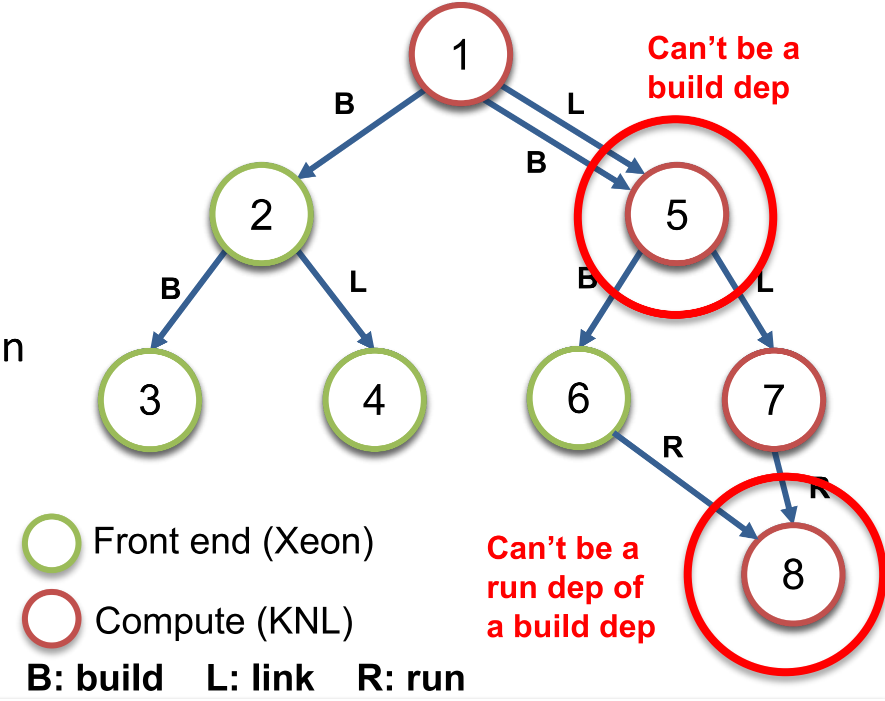
- Well, now you have an issue:
  - you need your build dependencies built for a the **architecture where you’re building**
  - Sometimes you can get away with cheating (build everything for the compute arch)
    - Depends how close the compute OS and ISA are to the build nodes

- We can use our build dependency trick here, but it’s a bit more complex

Cross compiling and dependencies

- Suppose you have a dependency that is both a build dependency AND a link dependency.
- Build dependency trick definitely helps
- Could previously share the fancy compiler version
  - But now you can't b/c the compute version won't run in the build environment

```
spack install pkg1 target=knl
```



Can't be a build dep

Can't be a run dep of a build dep

Front end (Xeon)

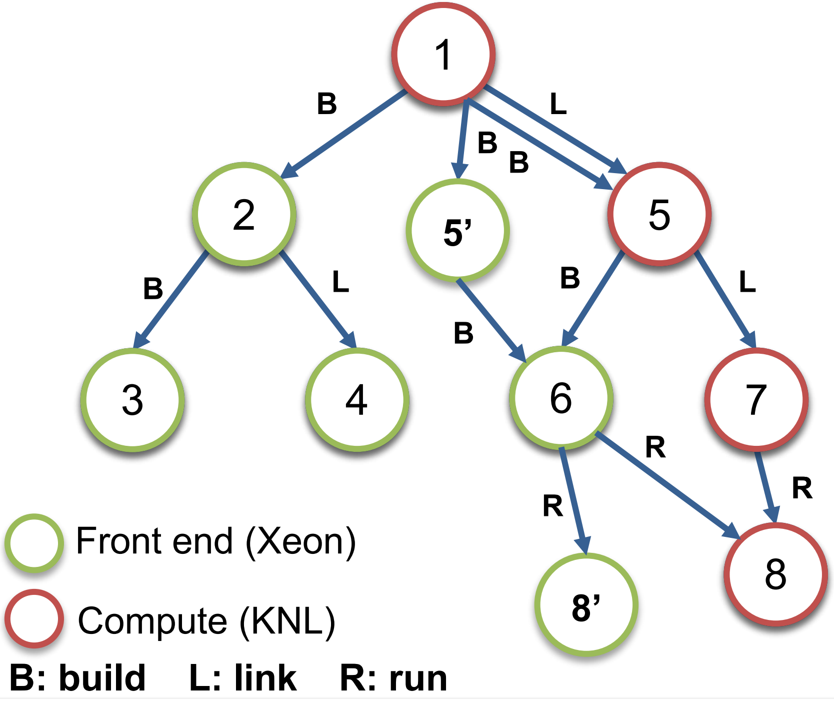
Compute (KNL)

B: build L: link R: run

Cross compiling and DAG splitting

- We can solve this problem by building two versions of the conflicting libraries
  - Need to relax our DAG constraint
- It’s ok to split these because the build and run environments are separate process spaces
  - not actually going to ever cause a race in ld.so
- Now there are 2 versions of 5 and 8, though
  - We’d like to minimize redundant versions

```
spack install pkg1 target=knl
```



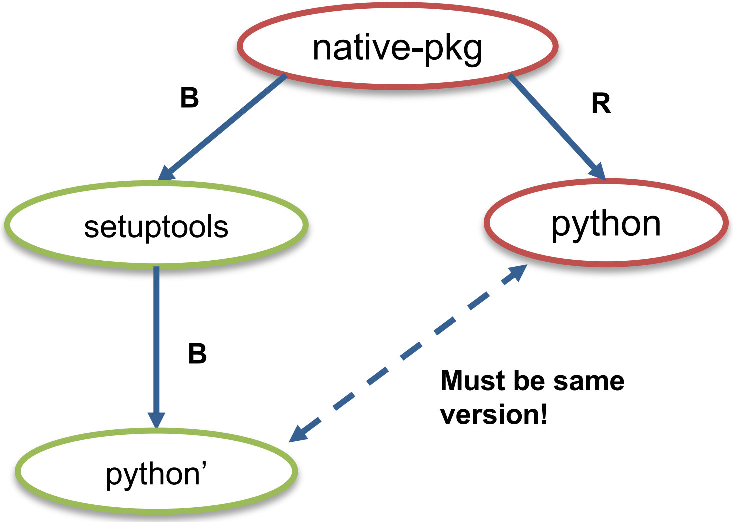
Interesting cross-architecture constraints

- Python doesn't really understand cross-compilation
  - you might not think it'd need to

- Setuptools is a **build tool** that **adds code** to the installed package
  - generated code is python version-dependent.
  - now you need front-end python and back-end and compute python to be the same version
  - this constraint spans two parts of your DAG!

- Your build env is *not* entirely separate from your run env
  - Can't just do independent resolution for build dependencies!

- This is one reason we're moving to SAT for dependency resolution
  - Easier and more general to express constraints like this



Last issue: Compiler dependencies

- Compilers are still a special case in Spack
  - Represented as attributes of nodes, not as dependencies

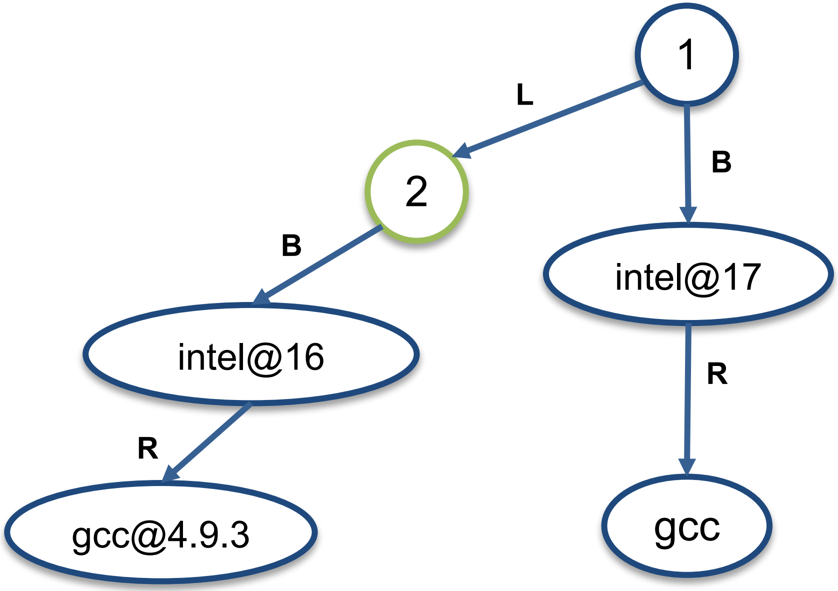
- Two issues:
  1. Compiler can't *provide* virtual dependencies like packages
  2. Compilers can't easily have their own dependencies

- We'd like packages to be able to *depend on* C++11, C++17, OpenMP 4.5, etc.
  - Requires compiler to provide C++ and OpenMP as virtual deps

- Some compilers actually *depend on other compilers!*
  - Intel compilers rely on gcc to provide libstdc++
  - Verison ranges need to match for this to work properly!
  - Coordinating this is a constant source of user frustration at HPC centers

Compiler dependencies

- Suppose we build a simple C++11 package with the Intel compiler
  - We'll model it as a build dependency
  - Easy enough to represent this
- Suppose we want to reuse an already-installed package built with an older Intel compiler version (that isn't available anymore)
  - With our relaxed constraint, build dependencies allow us to mix compilers
- But how can we ensure that the libstdc++ implementations are consistent?
  - ld.so race!



○ Already-installed dependency

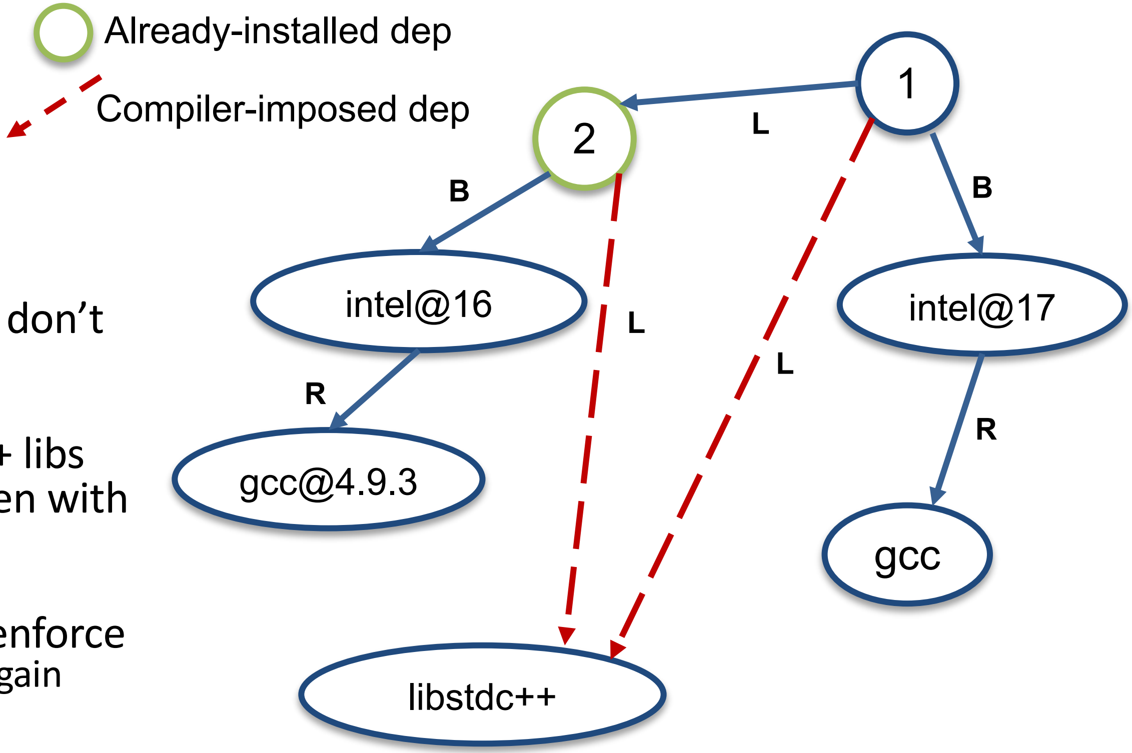
So what is a compiler anyway?

- A compiler is a build dependency that **IMPOSES** a link dependency on a DAG

- Each compiler has “hidden” dependencies
  - These are proper runtime libraries, so we need to model them like they are

- New plan:
  - Still model compilers as build dependencies
  - Bring out libstdc++ and other libraries from compilers as link dependencies *of the thing being built*
    - Ensures consistency across each DAG
  - We’ll Still “normalize” or “flatten” the (hidden) link dependencies in the DAG

New compiler dependency model



- Now consistency is enforced via link dependencies from 1 and 2
- If the libstdc++ versions from 1 and 2 don't match, then this won't resolve
- If they do, then we know that the C++ libs are compatible and can build this, even with the old dependency.
- We currently use some heuristics to enforce
  - Moving to SAT makes a lot of sense, again

Summary

- Working out constraints for compiler integration isn't easy

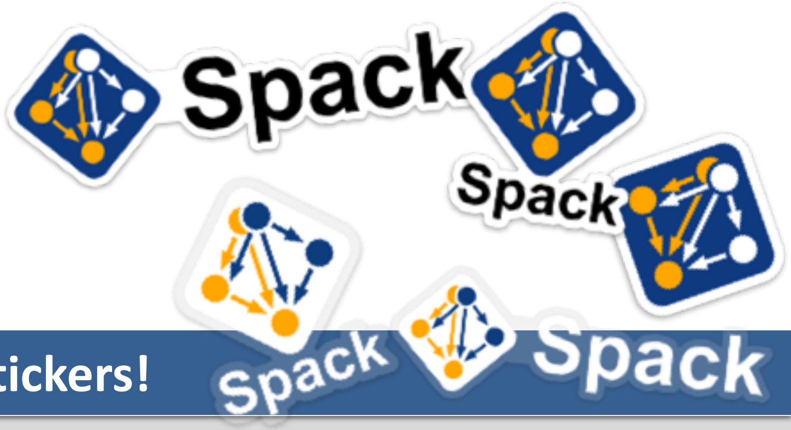
- Weird things can happen
  - build/link/run environment distinctions
  - Architecture distinctions
  - Constraints that manifest in strange ways across seemingly separate parts of the DAG
    - Setuptools
    - Stdlib compatibility

- We are aiming to automate this part of build configuration
  - Better automate experimentation with build options
  - Lower cost of supporting multiple compilers for code teams

- Working on bringing out this new compiler model with the new dependency resolver (concretizer) in Spack this year.

Spack

https://spack.io



Come and get Spack stickers!

Lawrence Livermore
National Laboratory