Solving MPI integration problems with Spack

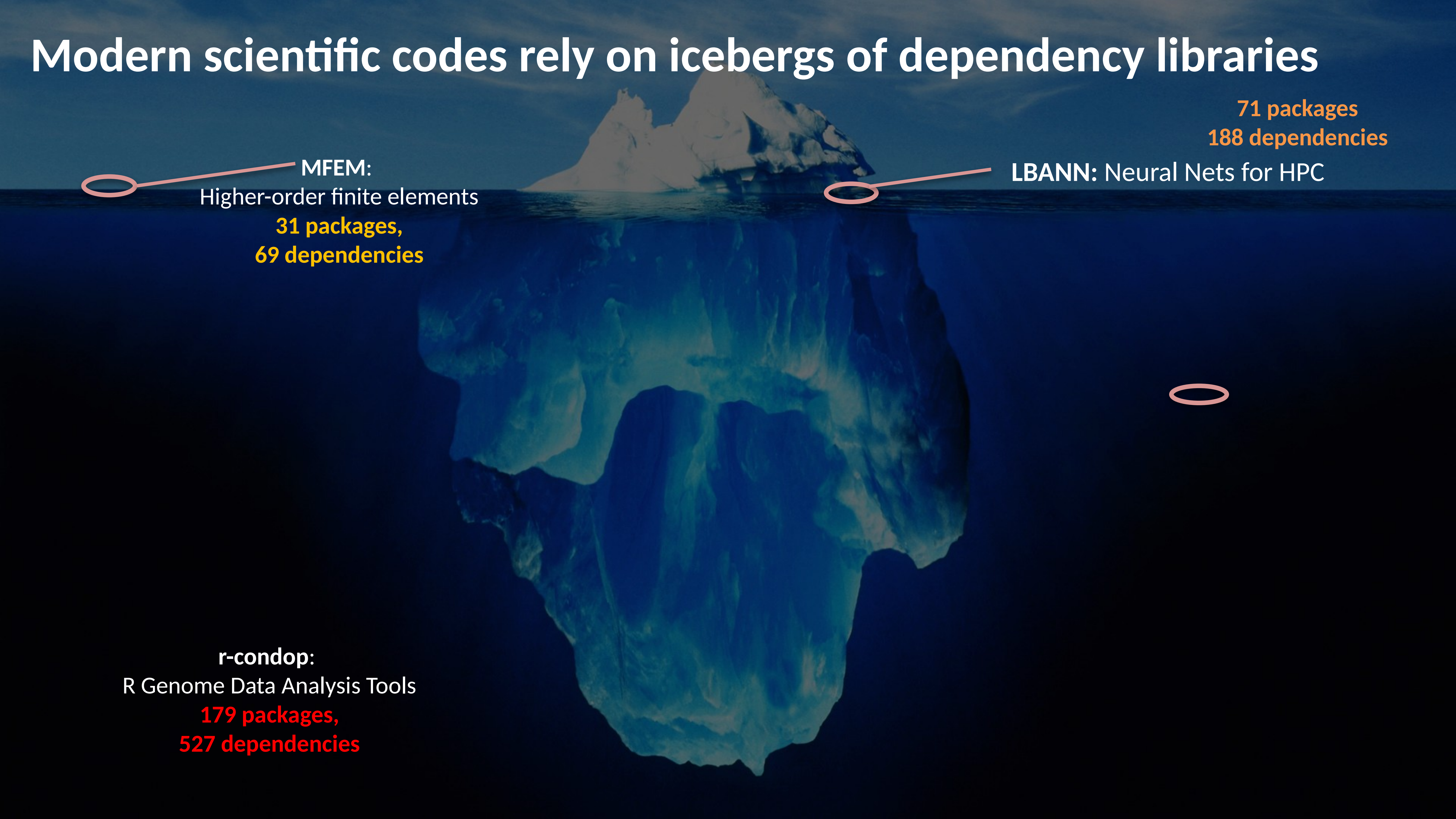
MUG 2022

Greg Becker

August 24, 2022

LLNL-PRES-839068

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

The HPC software space is immense

- Not much standardization in HPC: every machine/app has a different software stack
- Sites share unique hardware among teams with *very* different requirements
  - Users want to experiment with many exotic architectures, compilers, MPI versions
  - All of this is necessary to get the best ***performance***
- Example environment for some LLNL codes:

70 third party packages

x

4 MPI versions
IntelMPI MVAPICH2
OpenMPI MPICH

x

Multiple Platforms
x86_64 aarch64 PPC64LE

x

Up to 7 compilers
Intel GCC XLC Clang
Cray NVHPC rocmcc

x

Oh, and 2-3 versions of each package

= >**10,000** combinations

We want an easy way to quickly sample the space, to install configurations on demand!

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

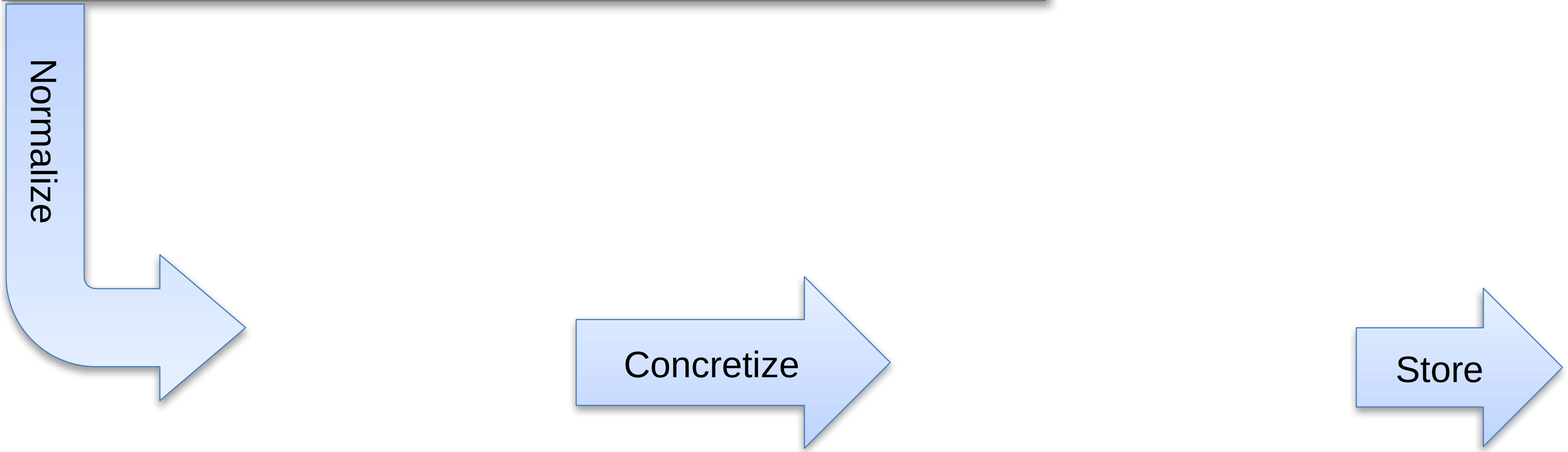
```
$ spack install mpileaks                                unconstrained
$ spack install mpileaks@3.3                              @ custom version
$ spack install mpileaks@3.3 %gcc@4.7.3                     % custom compiler
$ spack install mpileaks@3.3 %gcc@4.7.3 +threads              +/- build option
$ spack install mpileaks@3.3 cppflags="-O3 –g3"      set compiler flags
$ spack install mpileaks@3.3 target=cascadelake      set target microarchitecture
$ 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
  - Full control over the combinatorial build space

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

```
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
- boost:
    arch: linux-x86_64
    compiler:
      name: gcc
      version: 4.9.2
    dependencies: {}

    hash: teesjv7ehpe5ksspjim5dk43a7qnowlq
    variants: {}

    version: 1.59.0
```

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

Spack packages are *parameterized* using the spec syntax

Python DSL defines many ways to build

```
from spack import *

class Kripke(CMakePackage):
    """Kripke is a simple, scalable, 3D Sn deterministic particle transport 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):
        mkdirp(prefix.bin)
        install('../spack-build/kripke', prefix.bin)
```

Base package (CMake support)

Metadata at the class level

Versions

Variants (build options)

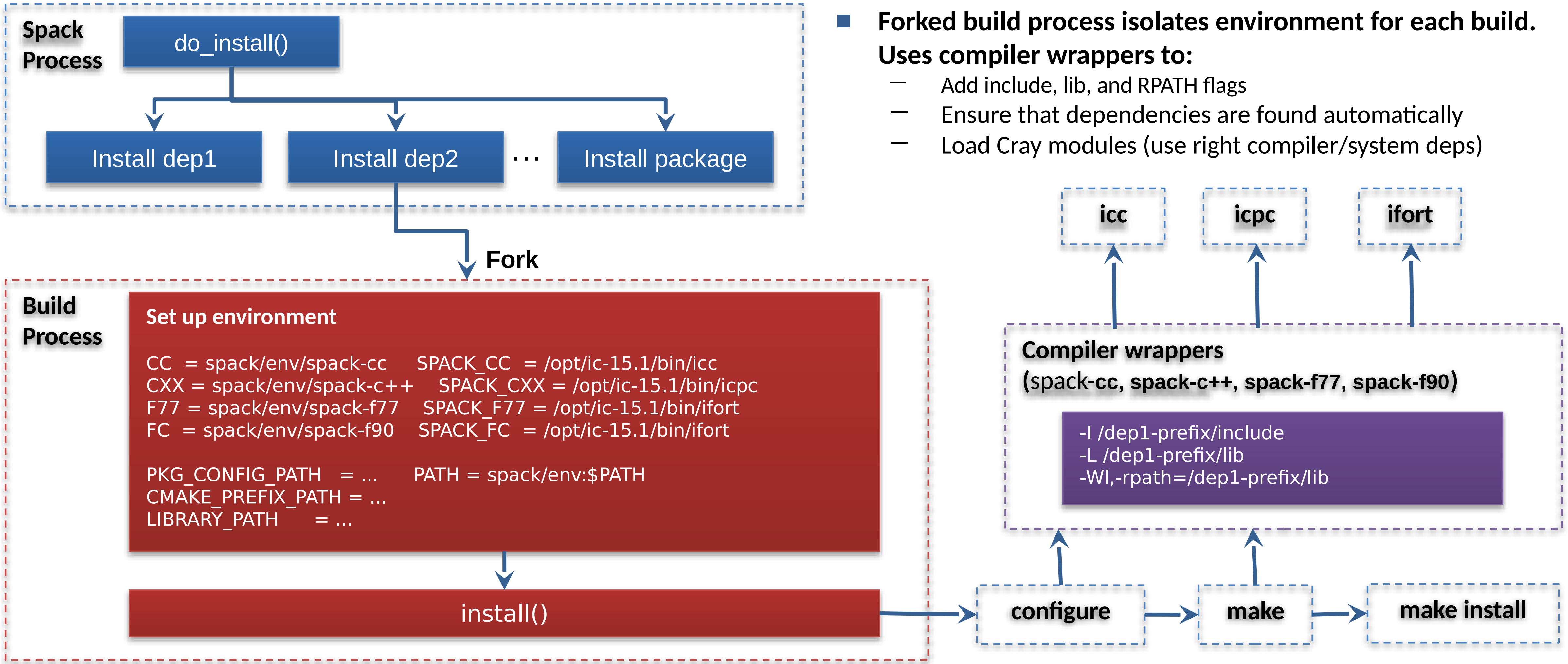
Dependencies (same spec syntax)

Install logic in instance methods

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

***One* package.py file per software project!**

An isolated compilation environment allows Spack to easily swap compilers



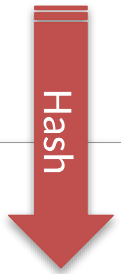
```
CC  = spack/env/spack-cc      SPACK_CC  = /opt/ic-15.1/bin/icc
CXX = spack/env/spack-c++     SPACK_CXX = /opt/ic-15.1/bin/icpc
F77 = spack/env/spack-f77     SPACK_F77 = /opt/ic-15.1/bin/ifort
FC  = spack/env/spack-f90     SPACK_FC  = /opt/ic-15.1/bin/ifort

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

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

Hashing allows us to handle combinatorial complexity

Dependency DAG

Installation Layout



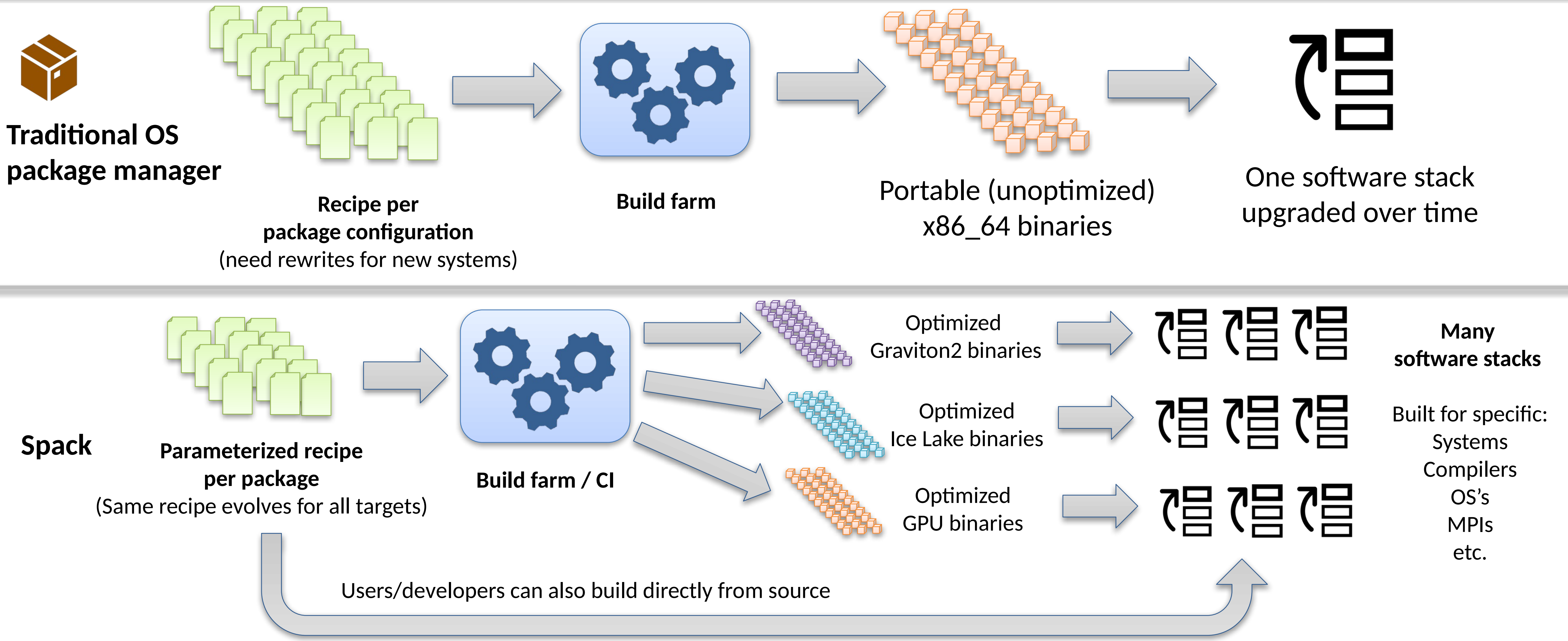
```
opt
└── spack
    ├── linux-rhel7-skylake
    │   └── gcc-8.3.0
    │       ├── mpileaks-1.0-hc4sm4vuzpm4znmvrfzri4ow2mkphe2e
    │       ├── callpath-1.0.4-daqqpssxb6qbfrztsezkmhus3xoflbsy
    │       ├── openmpi-4.1.4-u64v26igxvxyn23hysmklfums6tgjv5r
    │       ├── dyninst-12.1.0-u64v26igxvxyn23hysmklfums6tgjv5r
    │       ├── libdwarf-20180129-u5eawkvaoc7vonabe6nndkcfwuv233cj
    │       └── libelf-0.8.13-x46q4wm46ay4pltriijbgizxjrhbaka6
```

- Each unique dependency graph is a unique ***configuration***.
- Each configuration in a unique directory.
  - Multiple configurations of the same package can coexist.

Hash of entire directed acyclic graph (DAG) is appended to each prefix.

Installed packages automatically find dependencies
- Spack embeds RPATHs in binaries.
- No need to use modules or set LD_LIBRARY_PATH
- Things work *the way you built them*

Spack binary packages model full provenance


Traditional OS package manager
Recipe per package configuration
(need rewrites for new systems)
Build farm
Portable (unoptimized) x86_64 binaries
One software stack upgraded over time
Spack
Parameterized recipe per package
(Same recipe evolves for all targets)
Build farm / CI
Optimized Graviton2 binaries
Optimized Ice Lake binaries
Optimized GPU binaries
Many software stacks
Built for specific:
Systems
Compilers
OS's
MPIs
etc.
Users/developers can also build directly from source

Relocating Binaries

1. Make a copy of all the files that come with an installed package in Spack *except metadata*.
2. Normalize all relative paths in the new copy
3. Change all RPATHs in the ELF binaries to reflect new location and links to new dependencies.
4. Recreate any symlinks, update any paths in text files, etc.
5. Replace any hardcoded paths in binaries.
6. Install files and finish install as usual

Spack handles ABI-incompatible, versioned interfaces like MPI

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

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

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

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

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

Environment Setup from Dependencies

- All MPI implementation packages make MPICC available in the environment at build for their dependents
  - Set MPICC environment variable, etc.
  - Set MPICH_CC, etc. to Spack compiler wrappers
  - Set mpicc, etc. paths on spec object.
- Similar conventions for other common dependencies
  - Python package sets \`python\` on spec
  - Python package sets \`PYTHONPATH\` environment variable
  - Cmake package sets \`cmake\` on spec

```
def setup_dependent_environment(self, spack_env, dependent_spec):
    spack_env.set('MPICC',  join_path(self.prefix.bin, 'mpicc'))
    spack_env.set('MPICXX', join_path(self.prefix.bin, 'mpicxx'))
    spack_env.set('MPIF77', join_path(self.prefix.bin, 'mpif77'))
    spack_env.set('MPIF90', join_path(self.prefix.bin, 'mpif90'))

    spack_env.set('MPICH_CC', spack_cc)
    spack_env.set('MPICH_CXX', spack_cxx)
    spack_env.set('MPICH_F77', spack_f77)
    spack_env.set('MPICH_F90', spack_fc)
    spack_env.set('MPICH_FC', spack_fc)

def setup_dependent_package(self, module, dependent_spec):
    self.spec.mpicc  = join_path(self.prefix.bin, 'mpicc')
    self.spec.mpicxx = join_path(self.prefix.bin, 'mpicxx')
    self.spec.mpifc  = join_path(self.prefix.bin, 'mpif90')
    self.spec.mpif77 = join_path(self.prefix.bin, 'mpif77')
    self.spec.mpicxx_shared_libs = [
        join_path(self.prefix.lib, 'libmpicxx.so'),
        join_path(self.prefix.lib, 'libmpi.so')]
```

Spack virtual packages can share test infrastructure

```
from spack.package import *


class Mpi(Package):
    """Virtual package for the Message Passing Interface."""

    homepage = "https://www.mpi-forum.org/"
    virtual = True

    def test(self):
        for lang in ("c", "f"):
            filename = self.test_suite.current_test_data_dir.join("mpi_hello." + lang)

            compiler_var = "MPICC" if lang == "c" else "MPIF90"
            compiler = os.environ[compiler_var]

            exe_name = "mpi_hello_%s" % lang
            mpirun = join_path(self.prefix.bin, "mpirun")

            compiled = self.run_test(compiler, options=["-o", exe_name, filename])
            if compiled:
                self.run_test(
                    mpirun,
                    options=["-np", "1", exe_name],
                    expected=[r"Hello world! From rank \s*0 of \s*1"],
                )
```

Virtual package
(No install method

Test method
for all providers

This system will be expanded to include interface information and safeguards

Each package can also include its own particular tests

We can configure Spack to build with external software

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

packages.yaml

```
packages:
  mpi:
    buildable: False
    paths:
      openmpi@2.0.0 %gcc@4.7.3 arch=linux-rhel6-ppc64:
        /path/to/external/gcc/openmpi-2.0.0
      openmpi@1.10.3 %gcc@4.7.3 arch=linux-rhel6-ppc64:
        /path/to/external/gcc/openmpi-1.10.3
      ...
```

/path/to/external/gcc/openmpi-2.0.0

Spack prunes the DAG when adding external packages.

We frequently want to swap in a new MPI

- Running against a system MPI
  - OpenMPI package maintainers tell Spack that OpenMPI 4.0.7 is ABI-compatible with OpenMPI 4.1.2
  - OpenMPI 4.1.2 satisfies all symbols present in the 4.0.7 version.
  - Therefore, users will know that software built against OpenMPI 4.0.7 will run against OpenMPI 4.1.2, regardless of the symbols used.
  - Even if versions are the same, we need to relocate the package to use the external

- Running in a container
  - User built their application with MPICH in a container
  - needs to run with MVAPICH2 from the host for performance
  - bind-mount host MPI into the container

- How can we deploy this in Spack?
  - We need to model the provenance
  - We need to modify the packages on disk

We need three things to make binary swapping possible in Spack

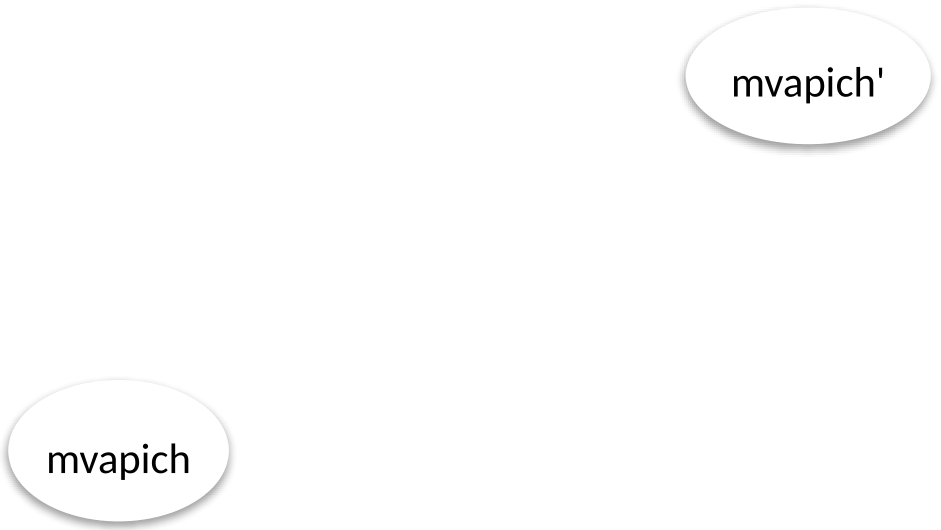
1. ***New deployment and metadata model***
   - Splicing
     - Need to be able to swap one dependency for another
     - Need to avoid losing provenance and *preserve build metadata* even when *deployment is different*
   - Rewiring
     - Need to be able to relocate package RPATH's, shebangs, etc. to point to new dependency
     - Use patchelf, binary rewriting, rewriting symlinks, etc. on installation as part of relocation
     - Rewiring is fundamentally similar to binary relocation, no need to dwell on it

2. ***New ABI information in packages***
   - Specified with DSL by user
   - Tells you *what swaps are safe*

3. ***Solver changes***
   - Solver needs to know about ABI constraints
   - Find safe configurations

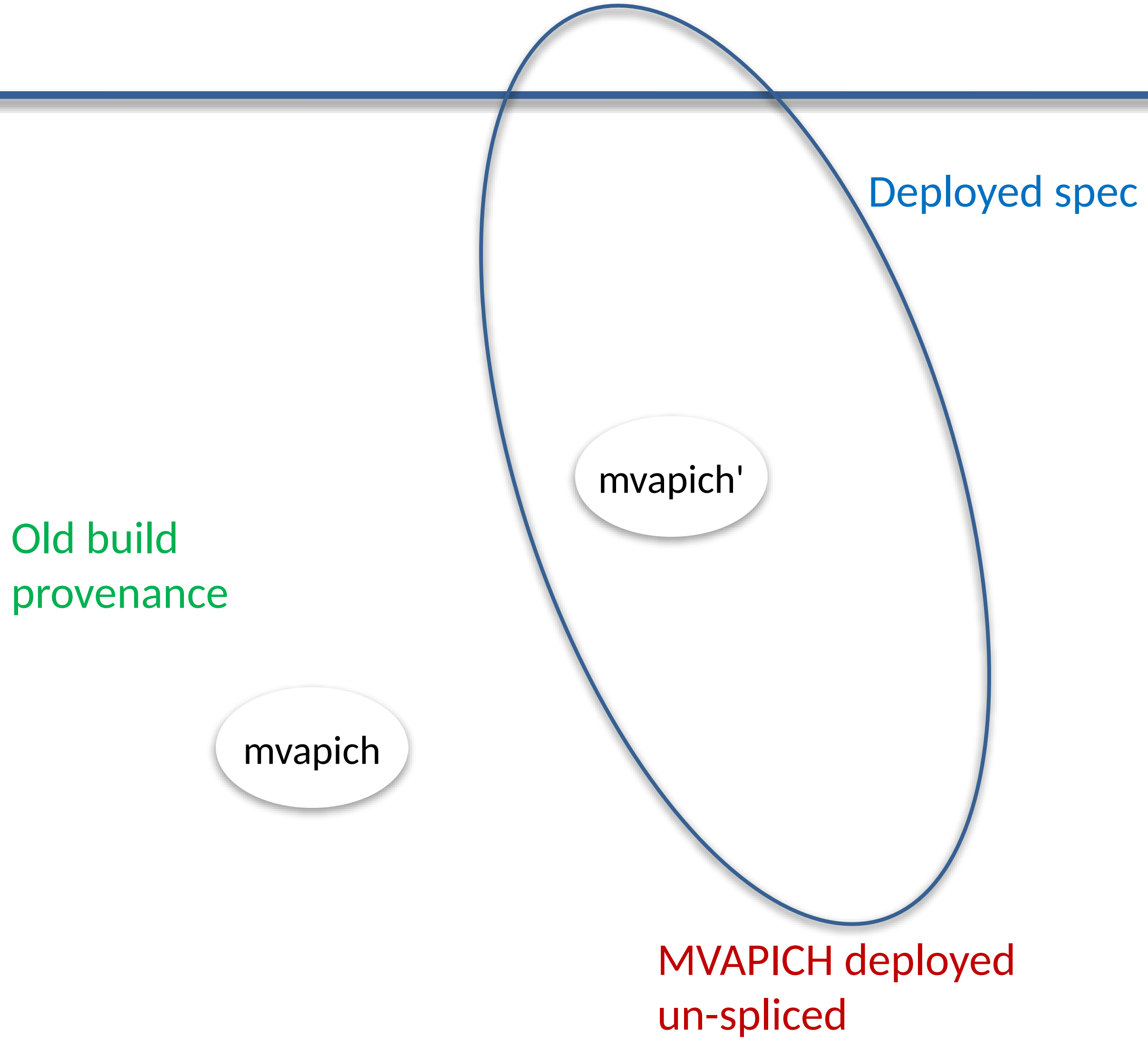
Splicing: a new deployment model for Spack

- A binary of trilinos has already been built and will be deployed on a system with its own MVAPICH installation (in green).
- We need to use this system-installed MVAPICH (in red).
- We we don’t want to totally rebuild trilinos.
- So the system-installed MVAPICH is spliced into the DAG



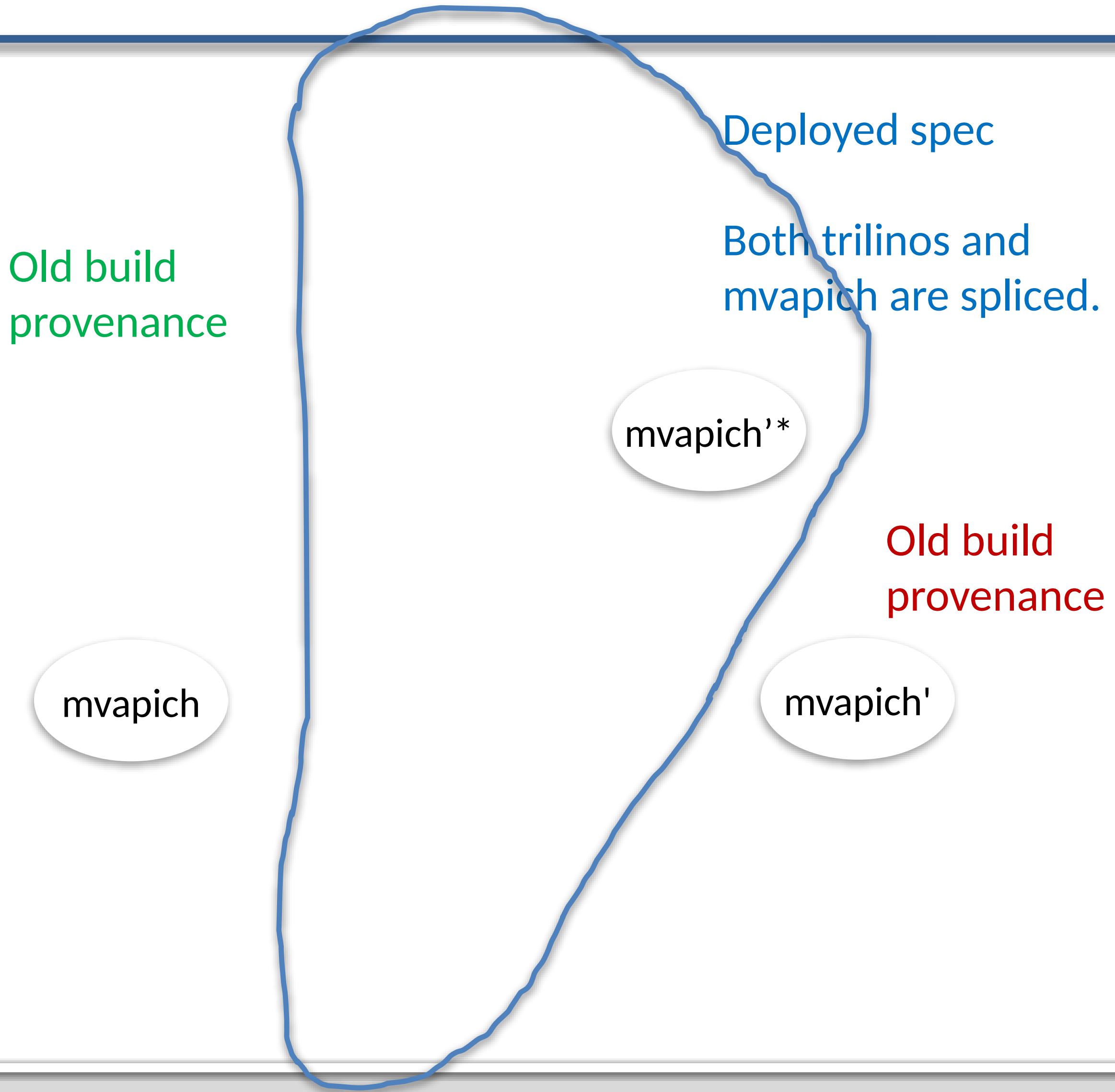
Splicing MPI

- Trilinos* installation uses the the system-installed **MVAPICH**.
  - Different MPI than it was built with
  - RPATHs from trilinos install now point at the new MVAPICH
- Black arrow is a “build_spec”
  - Metadata recording original build graph
  - Records original build information
  - Can be used to check ABI compatibility later
- Trilinos now *also* uses the system-installed zlib’ that MVAPICH depended on
  - We call this a ”transitive” splice



An Intransitive Splice

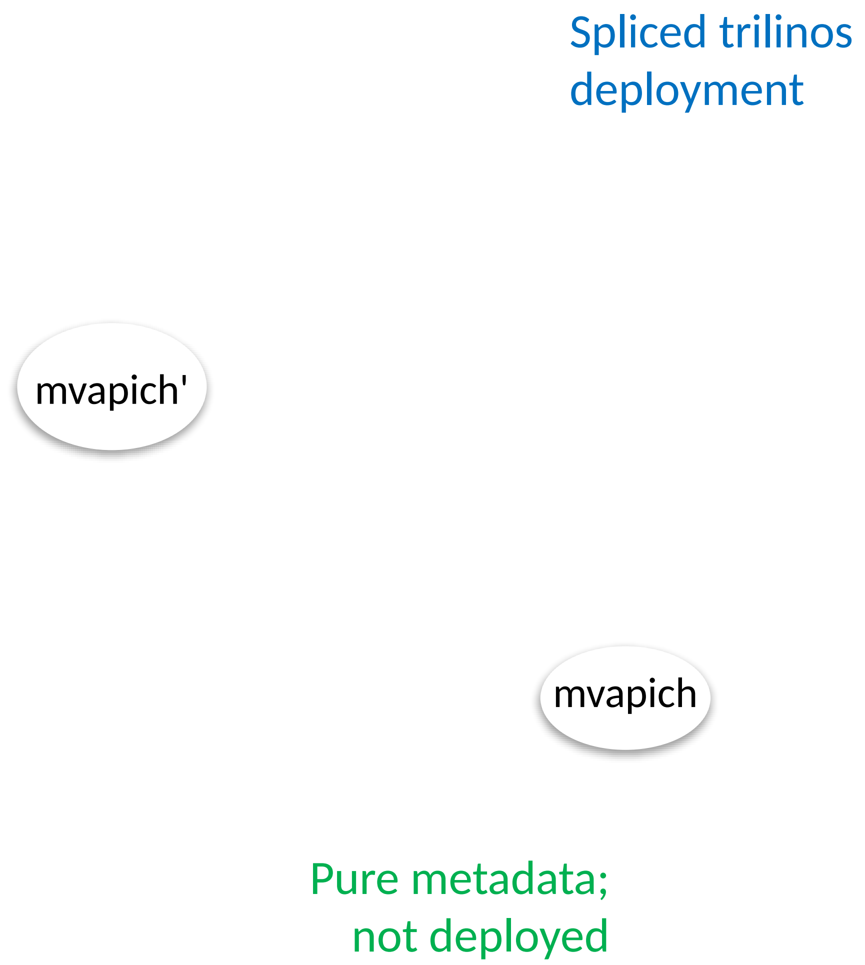
- In some cases, the version of a dependency in the root spec is the one which satisfies both specs constraints
- We support this use case with the intransitive splice, where only the spliced dependency is brought in.
- Our trilinos* installation now depends on a new installation of MVAPICH based on the system build, but *always* uses the zlib that came with the original trilinos binary distribution.



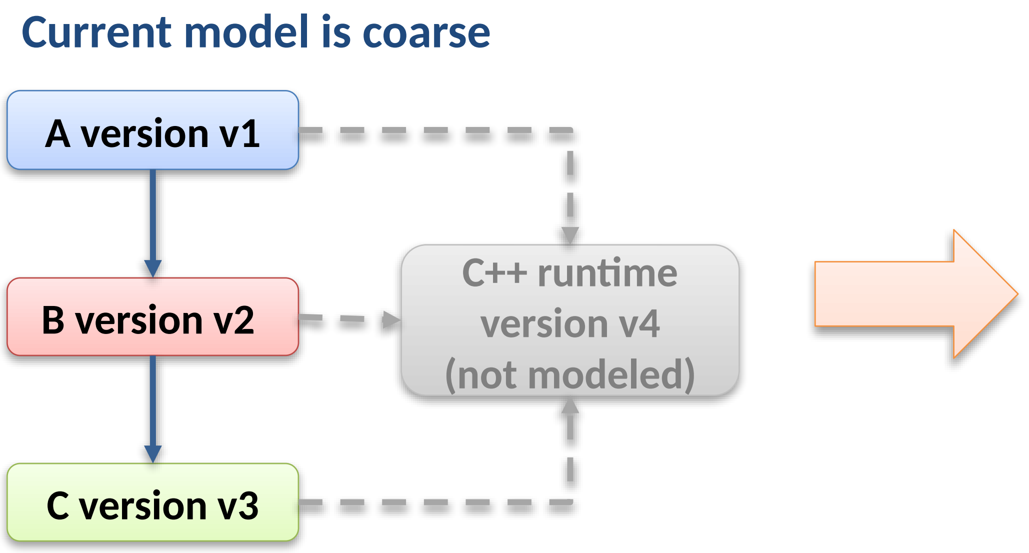
ABI constraints and splicing

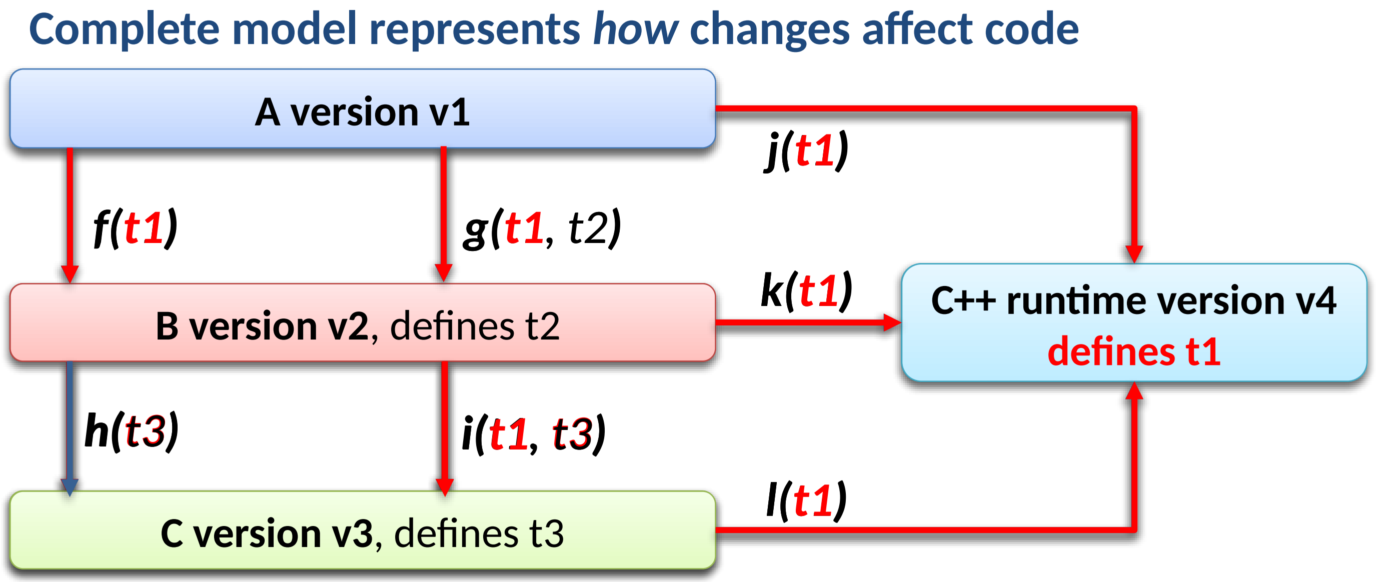
- We will reduce the Spec for each package to its ABI-relevant attributes
  - This will require per-package logic
    - What changed?
    - What’s relevant?
    - (eventually) What’s used?

- For each *deployed* edge A □ B:
  - Check whether abispec(B) satisfies abispec(A)[B]
    - Includes DSL information from packages:
      - Version constraints
      - Enabled sub-APIs
      - Compiler flags
      - etc.



Future goal: Build fine-grained compatibility models that cover functions, data types, and other aspects of ABI





- We will model libraries at call granularity:
  - Entry calls
  - Exit calls
  - Data type definitions & usage
- We will model runtime libraries behind compilers
  - C++, OpenMP, glibc
  - GPU runtimes
- We will model changes in the graph
  - "If h(t3) changes, is B still correct?
  - "If C changes, what needs to be rebuilt?"
  - We will model semantics of interfaces

This model will allow us to solve for compatibility, so we can find usable packages and splices