The Spack Package Manager: Bringing Order to HPC Software Chaos

Supercomputing 2015 (SC15)
Austin, Texas
November 18, 2015

Todd Gamblin, Matthew LeGendre, Michael R. Collette, Gregory L. Lee, Adam Moody, Bronis R. de Supinski, and Scott Futral
What is the production environment for HPC?

- Someone’s home directory?
  - Environments at large-scale sites are very different.
- Which MPI implementation?
- Which compiler?
- Which dependencies?
- Which versions of dependencies?
  - Many applications require specific dependency versions.

**Real answer:** there isn’t a single production environment or a standard way to build.
HPC software is becoming increasingly complex

- 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:

  - 48 third party packages
  - Up to 7 compilers: Intel GCC XLC Clang PGI Cray Pathscale
  - 3 MPI versions: mvapich mvapich2 OpenMPI
  - 3-ish Platforms: Linux BlueGene Cray
  - Oh, and 2-3 versions of each package

  \[ \times \times \times \times = \sim 7,500 \text{ combinations} \]

We want an easy way to quickly sample the space, to build configurations on demand!
Most existing tools do not support combinatorial versioning

- Traditional binary package managers
  - RPM, yum, APT, yast, etc.
  - Designed to manage a single stack.
  - Install *one* version of each package in a single prefix (/usr).
  - Seamless upgrades to a *stable, well tested* stack

- Port systems
  - BSD Ports, portage, Macports, Homebrew, Gentoo, etc.
  - Minimal support for builds parameterized by compilers, dependency versions.

- Virtual Machines and Linux Containers (Docker)
  - Containers allow users to build environments for different applications.
  - Does not solve the build problem (someone has to build the image)
  - Performance, security, and upgrade issues prevent widespread HPC deployment.

How do HPC sites deal with combinatorial builds?

- HPC software is typically installed manually in a directory hierarchy.
  - Hierarchy often doesn’t give all needed information about a build.
  - Sites can run out of unique directory names quickly.

<table>
<thead>
<tr>
<th>Site</th>
<th>Naming Convention</th>
</tr>
</thead>
<tbody>
<tr>
<td>LLNL</td>
<td>/usr / global / tools / $arch / $package / $version</td>
</tr>
<tr>
<td></td>
<td>/usr / local / tools / $package-$compiler-$build-$version</td>
</tr>
<tr>
<td>Oak Ridge</td>
<td>/ $arch / $package / $version / $build</td>
</tr>
<tr>
<td>TACC</td>
<td>/ $compiler-$comp_version / $mpi / $mpi_version / $package / $version</td>
</tr>
</tbody>
</table>
Environment modules can help, but are hard to get right.

- **Advantages:**
  - Swap different library versions dynamically, in a shell.
  - Abstracts a lot of environment complexity from the user.

- **Disadvantages:**
  - Users must typically remember to load the same module that they built with.
    - Easy to load wrong module and break code.
  - Many sites and vendors deploy extremely brittle, inconsistent modules.
  - Module systems do not build software; they only change the environment.

```bash
$ module avail
--------------------------- /opt/modules/modulefiles ---------------------------
acml-gnu/4.4               intel/12.0                   mvapich2-pgi-ofa/1.7
acml-gnu_mp/4.4            intel/13.0                   mvapich2-pgi-psm/1.7
acml-intel/4.4             intel/14.0(default)         mvapich2-pgi-shmem/1.7...

$ module load intel/13.0
$ module load mvapich2-pgi-shmem/1.7
```
Spack handles combinatorial software complexity.

### Dependency DAG

- 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*

### Installation Layout

- Spack handles combinatorial software complexity.

```plaintext
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-fqb3a15abwx/
...
```
`spack list` shows what packages are available

```
$ spack list
  == 243 packages.
activeharmony  coreutils  ghostscript  leveldb  libarchive  llvm  netcdf  ppl  py-pychecker  qt  thrift
adepit-utils   cppcheck  git        libarchive  libcircle  llvmlld  netlib-blas  protobuf  py-pycparser  gthreads  tk
apex           cram       glib       libdrm      libdwarp   lmu       netlib- blas  py-basemap  py-pyelftools  R  tmux
arpack          cscope     glibglobal  libelf     libevent   lwgrp     ompss     py-biopython  py-pygments  ravel  tmuxinator
asciidoc       cube       gmp        libffi      libevent   lwnum     omp0-openmp py-cffi  py-pylint  readline  trilinos
atk             czmq       gnome       libffi      libevent   lwnum     omp0-openmp py-cffiy  py-pypar   rose   uncrustify
atlas           dbus       gnutils     libffi      libevent   lwnum     omp0-openmp py-epdocy  py-pygt   ruby    util-linux
autoconf        docbook-xml gperf       libgcrypt  memxases  openmpi  py-genders  py-pyside  samtools  scalasca  vim
automated       doxygen    gperftools  libgcrypt  memxases  openmpi  py-gnuplot  py-python  samtools  scalasca  vtk
automake        dri3proto  graphlib    libgpg-error mesa      opensssl  py-grnxplot  py-python  scorep    scorch  wget
bear            dtcmap     graphviz    libgjson-c  m6tos     ofTf2      py-h5pyy  py-pyzt  scorep    scorch  wx
bib2xhtml       dyninst    gtkplus     libgmg     mpc        pang0     py-h5pyy  py-pyzt  scorep    scorch  wx
binutils        elfutils   harfbuzz    libgmonitor hdft5      libnbc     py-lockfile py-mako    scipy     snappy  yasm
bison           extrae     hfwloc       libpcaccess mpfr       paraver   py-matplotlib py-setuptools  spinn0le  zbermq
boost           exuberant-ctags hfwloc       libpcaccess mpfr       paraver   py-matplotlib py-setuptools  spinn0le  zbermq
bowtie2         fish       hfwloc       libpcaccess mpfr       paraver   py-matplotlib py-setuptools  spinn0le  zbermq
boxlib          flex       hfwloc       libpcaccess msbhash   paraview  py-matplotlib py-setuptools  spinn0le  zbermq
bzip2           flux       hfwloc       libpcaccess msbhash   paraview  py-matplotlib py-setuptools  spinn0le  zbermq
cairo           fontconfig icu         libpsodium  mnsic     parmetis  py-mock     py-mpi4py  py-sip    sqlite  zbermq
callpath        freetype   ImageMagicck libtool     libunwind  musteur1  py-mypandas  py-mypfib  py-six     sundials  zbermq
chroot          gasnet     isl         libunwind  musteur1  py-mypfib  py-mypandas  py-mypfib  py-six     sundials  zbermq
clg             gcc        jdk         libuid     mvpapiч2  pixman    py-mypandas  py-mypfib  py-six     sundials  zbermq
clang           gdk-pixbuf jpeg        libxcb      nasm      pkgr-config py-pexpecpy  py-pypaf  tau
clouo           geos       launchmon   libxml2     ncdu      pmgr-collective py-pil     quhull    the_silver_searcher
cmake           gflags     lcms        libxshfence ncrurses   postgresql py-pymw
```

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

- 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!

- Syntax abstracts details in the common case
  - Makes parameterization by version, compiler, and options easy when necessary

```bash
$ 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 =bgq = cross-compile
```
Spack Specs can constrain versions of dependencies

- Spack ensures *one* configuration of each library per DAG
  - Ensures ABI consistency.
  - User does not need to know DAG structure; only the dependency *names*.

- Spack can ensure that builds use the same compiler, or you can mix
  - Working on ensuring ABI compatibility when compilers are mixed.

$ spack install mpiLeaks %intel@12.1 ^libelf@0.8.12
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 mpi leaks ^mvapich@1.9
  $ spack install mpi leaks ^openmpi@1.4:
  ```

- Let Spack choose MPI version, as long as it provides MPI 2 interface:
  
  ```
  $ spack install mpi leaks ^mpi@2
  ```
Spack packages are simple Python scripts.

```python
from spack import *

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

    homepage = "https://paradyn.org"

    version('8.2.1', 'abf60b7fa0e7a2e', url="http://www.paradyn.org/release8.2/DyninstAPI-8.2.1.tgz")
    version('8.1.2', 'bf03b3375af066f', url="http://www.paradyn.org/release8.1.2/DyninstAPI-8.1.2.tgz")
    version('8.1.1', 'd1a04e995b7aa709', url="http://www.paradyn.org/release8.1/DyninstAPI-8.1.1.tgz")

    depends_on("libelf")
    depends_on("libdwarf")
    depends_on("boost@1.42:"")

    def install(self, spec, prefix):
        libelf = spec['libelf'].prefix
        libdwarf = spec['libdwarf'].prefix

        with working_dir('spack-build', create=True):
            cmake('...',
                  '-DBOost_INCLUDE_DIR=%s' % spec['boost'].prefix.include,
                  '-DBOost_LIBRARY_DIR=%s' % spec['boost'].prefix.lib,
                  '-DBOost_NO_SYSTEM_PATHS=TRUE'
                  **std_cmake_args)
            make()
            make("install")

        @when('@:8.1')
        def install(self, spec, prefix):
            configure('--prefix=' + prefix)
            make()
            make("install")
```

Metadata

Versions and URLs

Dependencies

Patches, variants (not shown)

Commands for installation

Access build config through the `spec` parameter.

Dependencies in Spack may be optional.

- The user can define named variants:

  ```python
  variant("python", default=False, “Build with python support")
  depends_on("python", when="+python")
  ```

- And use them to install:

  ```bash
  $ spack install vim +python
  $ spack install vim -python
  ```

- Dependencies may be optional according to other conditions: e.g., gcc dependency on mpc from 4.5 on:

  ```python
  depends_on("mpc", when="@4.5:")
  ```

- DAG is not always complete before concretization!
Concretion fills in missing configuration details when the user is not explicit.

**User input:** abstract spec with some constraints

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

**spec.yaml**

```
spec:
  - mpileaks:
      arch: linux-x86_64
      compiler:
        name: gcc
        version: 4.9.2
      dependencies:
        adept-utils: kszrtkpbzac3ss2ixcjkcorlaybnptp4
        callpath: bah5f4h4d2n47mgycej2mtrnrivvxy77
        mpich: aoa61fj23lyjmqmadbeekpejcltyz7t3
      hash: 3hryhjaks7plgyzn5ptgyes7sghyprujh
      variants: {}
      version: 1.0
  - adept-utils:
      arch: linux-x86_64
      compiler:
        name: gcc
        version: 4.9.2
      dependencies:
        boost: teesju7ehpe5ksspjin5d43a7qnowlq
        mpich: aoa61fj23lyjmqmadbeekpejcltyz7t3
        hash: kszrtkpbzac3ss2ixcjkcorlaybnptp4
      variants: {}
      version: 1.0.1
  - boost:
      arch: linux-x86_64
      compiler:
        name: gcc
        version: 4.9.2
      dependencies: {}
      hash: teesju7ehpe5ksspjin5d43a7qnowlq
      variants: {}
      version: 1.59.0
...
```

**Detailed provenance is stored with the installed package**
When underspecified, concretization chooses a value based on user/site preferences.

Concretization must add new dependencies in response to constraint updates.

Current algorithm is greedy, will not backtrack once a decision is made.
  - Can fail to find a build that satisfies a query, but has not happened for current packages.
  - Really needs a full constraint solver (coming soon!)
Spack builds each package in its own compilation environment

- Forking build process isolates environment for each build.
- Compiler wrappers add include, lib, and RPATH flags
  - Ensure that dependencies are found automatically

### Build Process

- **Set up environment**
  - `CC = spack/env/spack-cc`
  - `CXX = spack/env/spack-c++`
  - `F77 = spack/env/spack-f77`
  - `FC = spack/env/spack-f90`
  - `PKG_CONFIG_PATH = ...`
  - `CMAKE_PREFIX_PATH = ...`
  - `LIBRARY_PATH = ...`
  - `PATH = spack/env:$PATH`

- **Compiler wrappers (cc, c++, f77, f90)**
  - `icc`
  - `icpc`
  - `ifort`

- **Forking**
  - Build process isolates environment for each build.

- **Compiler wrappers add include, lib, and RPATH flags**
  - Ensure that dependencies are found automatically

- **do_install()**
  - Install dep1
  - Install dep2
  - ... Install package

- **Install**
  - `install()`

- **Build Process**
  - `do_install()`
  - Fork
  - `configure`
  - `make`
  - `make install`
  - `-I /dep1-prefix/include`
  - `-L /dep1-prefix/lib`
  - `-Wl,-rpath=/dep1-prefix/lib`

Build automation allows tedious work to be leveraged.

- Spack enables teams to share work.
  - Archives common library build recipes.
  - Prevents duplication of build effort.
  - We can share builds among LC, code teams, and users.

- Patches allow rapid deployment of bug fixes
  - App team porting a library may not own its repo.
  - Library teams may not have time to fix issues quickly.
  - Code teams can fix quickly, then feed back changes.

- Python allowed quick adoption by code teams.
  - Many app developers already know Python.
  - Spec syntax provides extra expressiveness.
### Use Case 1: Managing combinatorial installations

```bash
$ spack find
-> 183 installed packages.
  -- linux-x86_64 / gcc@4.4.7 ----------------------------------
  ImageMagick@6.8.9-10 glib@2.42.1 libtiff@4.0.3 pango@1.36.8 qt@4.8.6
  SAMRAI@3.9.1 graphlib@2.0.0 libtool@2.4.2 parmetis@4.0.3 qt@5.4.0
  adept-utils@1.0 gtkplus@2.24.26 libxcb@1.11 pixman@0.32.6 ravel@1.0.0
  atk@2.14.0 harfbuzz@0.9.37 libxml2@2.9.2
  boost@1.55.0 hdf5@1.8.13 libvm@3.0 py-dateutil@2.4.0 readline@6.3
  cairo@1.14.0 icu@54.1 metis@5.1.0 py-ipython@2.3.1 scotch@6.0.3
  callpath@1.0.2 jpeg@9a py-metis@3.0.4 starpu@1.1.4
  dyninst@08.1.2 libdwarf@20130729 ncurses@5.9 py-numpy@1.9.1 stat@2.1.0
  dyninst@08.1.2 libelf@0.8.13 ocr@2015-02-16 py-pytz@2014.10 zxml@5.2.0
  fontconfig@2.11.1 libffi@3.1 openssl@1.0.1h py-six@1.9.0
  freetype@2.5.3 libmg@2.0.2 of凋@1.12.5Salmon python@2.7.8
  gdk-pixbuf@2.31.2 libpng@1.6.16 oftf@2.1.4 quhull@1.0
  -- linux-x86_64 / gcc@4.8.2 ----------------------------------
  adept-utils@1.0.1 boost@1.55.0 cmake@5.6-special libdwarf@20130729 mpich@3.0.4
  adept-utils@1.0.1 cmake@5.6 dyninst@8.1.2 libelf@0.8.13 openmpi@1.8.2
  -- linux-x86_64 / intel@14.0.2 ----------------------------------
  hwloc@1.9 mpich@3.0.4 starpu@1.1.4
  -- linux-x86_64 / intel@15.0.0 ----------------------------------
  adept-utils@1.0.1 boost@1.55.0 libdwarf@20130729 libelf@0.8.13 mpich@3.0.4
  -- linux-x86_64 / intel@15.0.1 ----------------------------------
  adept-utils@1.0.1 callpath@1.0.2 libdwarf@20130729 mpich@3.0.4
  boost@1.55.0 hwloc@1.9 libelf@0.8.13 starpu@1.1.4
```

- **spack find** shows all installed configurations
  - Multiple versions of the same package are ok.

- Packages are divided by architecture/compiler.

- Spack also generates module files.
  - Don’t have to use them.
Using the Spec syntax, Spack can restrict queries

```
$ spack find mpich
==> 5 installed packages.
-- linux-x86_64 / gcc@4.4.7
    mpich@3.0.4
-- linux-x86_64 / gcc@4.8.2
    mpich@3.0.4
-- linux-x86_64 / intel@14.0.2
    mpich@3.0.4
-- linux-x86_64 / intel@15.0.0
    mpich@3.0.4
-- linux-x86_64 / intel@15.0.1
    mpich@3.0.4
```

- Querying by package name retrieves a subset
The Spec syntax doubles as a query language to allow refinement of searches.

```bash
$ spack find libelf
==> 5 installed packages.
-- linux-x86_64 / gcc@4.4.7 -------
  libelf@0.8.12  libelf@0.8.13
-- linux-x86_64 / gcc@4.8.2 -------
  libelf@0.8.13
-- linux-x86_64 / intel@15.0.0 -------
  libelf@0.8.13
-- linux-x86_64 / intel@15.0.1 -------
  libelf@0.8.13
```

**Query versions of libelf package**

**List only those built with Intel compiler.**

```bash
$ spack find libelf %intel
-- linux-x86_64 / intel@15.0.0 -------
  libelf@0.8.13
-- linux-x86_64 / intel@15.0.1 -------
  libelf@0.8.13
```

**Restrict to specific compiler version**

```bash
$ spack find libelf %intel@15.0.1
-- linux-x86_64 / intel@15.0.1 -------
  libelf@0.8.13
```
Users can query the full dependency configuration of installed packages.

```
$ spack find callpath
==> 2 installed packages.
-- linux-x86_64 / clang@3.4 -------- callpath@1.0.2
-- linux-x86_64 / gcc@4.9.2 --------
```

```
$ spack find -dl callpath
==> 2 installed packages.
-- linux-x86_64 / clang@3.4 -------- callpath@1.0.2
-- linux-x86_64 / gcc@4.9.2 --------
```

```
xv2clz2  callpath@1.0.2
ckjazss  ^adept-utils@1.0.1
3ws43m4  ^boost@1.59.0
ft7znm6  ^mpich@3.1.4
qnuet3   ^dyninst@8.2.1
3ws43m4  ^boost@1.59.0
65rddu   ^libdwarf@20130729
cj5p5fk  ^libelf@0.8.13
cj5p5fk  ^libelf@0.8.13
g65rddu  ^libdwarf@20130729
cj5p5fk  ^libelf@0.8.13
cj5p5fk  ^libelf@0.8.13
ft7znm6  ^mpich@3.1.4
```

Expand dependencies with `spack find -d`

- Architecture, compiler, and dependency versions may differ between builds.

Use Case 2: Package Views for HPC Center Installs

Many users like to navigate a readable directory hierarchy
  – Spack’s combinatorial package space is large and can be hard to navigate

Spack can generate a coarser tree view of symbolic links
  – View is a projection from the higher-dimensional Spack space
  – Some names may conflict, but spec syntax allows us to express preferences to guide view creation.

Many interpreted languages have their own mechanisms for modules, e.g.:

- Require installation into interpreter prefix
- Breaks combinatorial versioning

Spack installs each Python package in its own prefix

"Activating" links an extension into the interpreter directory on demand
- Supports .egg, merging .pth files
- Mechanism is extensible to other languages
- Similar to virtualenv, but Spack allows much more build customization.
ARIES is a 1, 2, and 3-D radiation hydrodynamics code

Spack automates the build of ARES and all of its dependencies
  — The ARES configuration shown above has 47 dependencies
ARES has used Spack to test 36 different configurations

- Nightly builds of ARES are shown at right.
  - 4 code versions:
    - (C)urrent Production
    - (P)revious Production
    - (L)ite
    - (D)evlopment

- Learning Spack and porting all libraries took a single developer 2 months, half-time.

- Previously, the team was only able to automate its development Linux builds.
  - Spack enabled thorough testing of many more configurations
  - Testing with Spack helped find compilation issues when using Clang compiler.

- Spack is helping the team port to LANL’s new Trinity (Cray XC-40) machine
Related work

- **OS package managers**
  - Don’t handle combinatorial builds
  - Single compiler; single stable version of pkg.
  - Allow smooth upgrades and predictable user experience.

- **Gentoo Prefix**
  - Based on Gentoo Linux: builds from source, installs into common prefix
  - Common prefix limits multi-compiler and multi-version support.

- **Nix (from NixOS)**
  - Allows many separate configurations
  - Packages are cryptographically hashed.
  - Multi-compiler, version support is limited
  - No virtual dependencies
  - No syntax for parameterization.

- **EasyBuild (HPC U. Ghent)**
  - Requires a file per configuration of software
    - 3300 config files for 600 packages (!)
  - Limited command line interface
  - Limited DAG and dependency analysis

- **Hashdist**
  - No spec syntax, more package file and profile editing required, less composable.
  - Compiler/architecture support is limited

- **Smithy (ORNL), Maali (Pawsey)**
  - No dependency management; only install automation

Many new feature developments are in progress

- **Current:**
  - Lmod hierarchy integration
  - External dependencies
    - Autodetect system MPI and other packages
  - Custom compiler flag injection
  - XML Test output (JUnit)
    - Each dependency exposed as test case
  - Better Cray environment integration

- **Planned:**
  - Use compiler wrappers to apply tools to large codes
    - Klocwork, thread sanitizers, etc.
  - Dependencies on compiler features (C++11, lambdas, OpenMP versions)
  - Automatic ABI checking & upgrading

The Spack project is growing rapidly.

- Spack is flexible enough for HPC needs
  - From single users of small clusters, to large code teams on top-10 supercomputers.

- Spack is starting to be used in production at LLNL
  - Build, test, and deployment by code teams.
  - Build research projects for students, postdocs.

- Spack has a rapidly growing external community.
  - NERSC is working with LLNL on Cray support for Cori.
  - Argonne/IIT cluster challenge project.
  - Kitware contributing ParaView builds & features.
  - INRIA using Spack to package MORSE numerical software
  - Users and contributors at EPFL, U. Oregon, Sandia, LANL.

Get Spack!


Builds share as many dependencies as possible

- May add space overhead compared to an LD_LIBRARY_PATH based system
- Safer than modules or LD_LIBRARY_PATH since the user cannot get deps wrong
  - Installations always run they way they are built.
- Above shows mpileaks built with mpich, then openmpi
  - Dotted packages must be rebuilt.

Concretization time is reasonable even for large packages.

- Fixed-point concretization algorithm scales quadratically

- Spack graphs are small, even for the largest packages
  - Thousands of dependencies are unlikely, even in multi-million line code bases.
  - Using a proper constraint solver will speed this up.
Compiler wrappers incur some overhead

- Extra script layer requires some overhead
- Spack’s decision to build in tmp filesystem improves more than script overhead hurts.

Future direction: Dependencies on compiler features

Profusion of new compiler features frequently causes build confusion:
- C++11 feature support
- OpenMP language levels
- CUDA compute capabilities

Spack could allow packages to request compiler features like dependencies:

```
require('cxx11-lambda')
require('openmp@4:')</code>
```

Spack could:
1. Ensure that a compiler with these features is used
2. Ensure consistency among compiler runtimes in the same DAG.
Future direction: Compiler wrappers for tools

- **Automatically adding source instrumentation to large codes is difficult**
  - Usually requires a lot of effort, especially if libraries need to be instrumented as well.

- **Spack could expose Klocwork, Scalasca, TAU, etc. as “secondary” compiler wrappers.**
  - Allow user to build many instrumented versions of large codes, with many different compilers:
    
    ```
    spack install application@3.3 %gcc@4.7.3 +tau
    ```

- **Spack packages provide a general interface to build details.**

- **LLNL PRUNER debugging tool is looking into this.**
  - Uses LLVM for instrumentation; needs to cover all libraries.
Future direction: Automatic ABI checking

- We’re starting to add the ability to link to external packages
  - Vendor MPI
  - OS-provided packages that are costly to rebuild

- External packages are already built, so:
  - Can’t always match compiler exactly
  - Can’t always match dependency versions exactly

- Need to guarantee that the RPATH’d version of a library is compatible with one that an external package was built with
  - Allows more builds to succeed
  - Potentially violates ABI compatibility

- Looking into using libabigail from RedHat to do some checking at install time.