LLNL-CONF-701389

Managing Combinatorial Software Installations with Spack

G. B. Becker, G. T. Gamblin, P. J. Scheibel, M. P. LeGendre

August 26, 2016

HPC User Support Tools Workshop
Salt Lake City, UT, United States
November 13, 2016 through November 13, 2016

Managing Combinatorial Software Installations with Spack

Gregory Becker, Peter Scheibel, Todd Gamblin, Matthew LeGendre
Lawrence Livermore National Laboratory, Livermore, California 94551
{becker33, scheibel1, tgamblin, legendre1}@llnl.gov

***Abstract*—HPC centers deploy a wide variety of software for scientific applications, but the complexity of building scientific applications makes package management increasingly difficult. Users demand combinatorial versions of packages, but site administrators may need to perform in-place upgrades for security and for bug fixes.**

This paper describes an extension of the *Spack* package manager that allows HPC centers to navigate a compromise between fully combinatorial versioning and a stable, upgradable software stack. Spack provides a set of templated packages that can be deployed in arbitrarily many combinatorial build configurations. We introduce subspaces, an extension of Spack's versioning system that allows HPC sites to choose an arbitrary combinatorial complexity for packages they deploy. Subspaces allow us to use a single Spack package to generate binary packages for systems such as RPM. Using subspaces, support staff can configure the degree of combinatorial versioning exposed to the user. This capability enables an intuitive and flexible user environment that can be leveraged across multiple HPC sites.

I. Introduction

The LClong (LC) center manages high performance computing resources at Lawrence Livermore National Laboratory (LLNL), from small test systems to the 17 petaflop Sequoia supercomputer. Across these systems, LC supports over 3,000 users, who require a wide variety of software packages to be installed and maintained. In addition to these users, LC works in conjunction with other National Nuclear Security Agency (NNSA) laboratories—Los Alamos National Laboratory (LANL) and Sandia National Laboratories (SNL)—to support a common software environment across all NNSA commodity clusters. Work is underway to unify the user environments across these systems. The diversity of required packages is likewise growing, as modern scientific applications used by a larger user base rely on an increasing number of numerical and system libraries.

The set of potential build configurations for a software package in high performance computing (HPC) is combinatorial in size. Package versions, compilers, compiler versions, message passing interface (MPI) implementations, and configure options are just several of the combinatorial build choices, and we can view each build as a point in this "build space." Typical operating system (OS) distributions provide only a part of the build space: a stable set of packages that comprise the system software. These packages are upgraded in place over time, transparently to users. Software in such an OS is typically installed in a common prefix, so users benefit from upgrades immediately. However, OS package managers typically have no support for combinatorial versioning or selecting from multiple configurations of the same package. Application software that relies on custom compilers, MPI implementations, and optimized numerical libraries is typically installed by hand to support many different points in the build space, though automated tools for handling this task are growing in popularity [16, 18, 20, 21].

To meet user demands, HPC sites need to support much of the full combinatorial build space, but they must also provide a stable foundation of system software using the deployment mechanism of the base OS. Further, administrators of live systems often need to upgrade packages in place (e.g., for security updates). Full combinatorial versioning is useful because it allows sites to quickly meet users' needs by deploying *exactly* the software they wish to use. However, if the site installs a new version of the software in a new location, users typically need to recompile, and they may have to do so frequently. Our goal is to provide a compromise between a stable, upgradable software stack and full combinatorial versioning that is flexible enough to support the differing needs of our users.

Spack [16, 21], is a package management tool developed at LLNL. It supports building and deploying combinatorially versioned software in user space. Spack gives users control over the entire directed acyclic graph (DAG) of dependencies. Using this, users can deploy arbitrary build configurations, with different package versions, MPI implementations, compilers, and build options. We leverage Spack to build arbitrary configurations, and we extend it with the concept of *subspaces*: naming schemes that project Spack's build space into a lower-dimensional configuration space. Subspaces allow administrators to generate arbitrary simplified views of packages for users, to generate system RPM packages from Spack packages, and to balance transparent, in-place upgrades (e.g. for hot fixes or security updates) with recompilation of new versions. In this work, our main contributions are:

- *Package subspaces*: customizable projections of Spack's combinatorial build space into lower-dimensional spaces;
- Techniques to manage in-place upgrades within subspaces;
- A methodology for automatically generating many upgradable RPM packages from a single subspace; and
- An implementation of these techniques in Spack.

Together, these techniques comprise a deployment solution that allows subspaces of fully combinatorial packages to be managed with traditional package- and environment management tools. Spack itself is policy-free, but Spack subspaces allow HPC administrators to *choose* the granularity of packages to deploy. For example, with only minor configuration changes, Spack can generate RPMs for the same package built with each version of the same MPI implementation, only for *major* versions of the MPI implementation, or a single package for *all* versions of the same MPI implementation. To our knowledge, this is the only system to provide this fine-grained control over combinatorial builds. The remainder of this paper describes our techniques in detail. We also describe how this flexibility, in concert with existing tools, facilitates the support of commodity Linux users across the NNSA tri-laboratory facilities.

II. The Tri-Lab Computing Environment

A. The Tri-lab Operating System Software Stack

Since 2007, the three NNSA laboratories (LLNL, LANL, and SNL) have used common hardware and operating system stacks across their commodity clusters. The Tri-lab Operating System Software Stack, or *TOSS* [8], is a fully functional cluster operating system distribution based on Red Hat Enterprise Linux (RHEL). TOSS is maintained by the tri-labs; it centralizes processes for bug tracking, packaging, maintenance and configuration management, saving costs across the NNSA laboratories. TOSS adds additional system packages to RHEL that are needed for HPC clusters. It builds off of the prior *CHAOS* effort at LLNL [2, 17], which began in 2001 to build a commodity OS stack at LLNL.

B. System Software vs. User Environment

While TOSS covers system packages and some basic mathematical libraries, it does not include all of the parallel scientific software used by many applications. Historically, the tri-labs have done manual installs of this software at each site. In 2016, with the deployment of new RHEL7/TOSS3-based clusters, the tri-labs are attempting to unify the user environment in addition to the OS stack. This new environment is called the *Tri-lab Computing Environment (TCE)*, and it adds package support to TOSS for multiple compilers, MPI implementations, math libraries and other tools.

The main distinction between TOSS packages and TCE packages is one of versioning. TOSS uses RPM for its package management, and it is based on a stock RHEL7 operating system. TOSS packages typically install into reserved system locations such as `/usr`, and there is only *one* version of a particular software package installed in a TOSS instance at a time. TCE, on the other hand, is designed to allow many versions and configurations of packages to exist simultaneously. A TCE environment may have an installation of GCC 6.1 for users interested in advanced C++14 features, and a version of GCC 4.9 for users interested in backwards compatibility.

Managing multiple versions of software is challenging for many reasons documented in the literature [13, 16, 18, 20]. Most operating systems are designed to support a single, stable software stack with a fixed version for each package. On multi-user HPC systems, conflicting user demands, desire to experiment with many configurations, and lack of software engineering resources on small teams all necessitate multi-versioned installations. At the tri-labs, TCE is required to run on TOSS, and packages must be deployed using RPMs. This paper describes the techniques we have used to implement a common mechanism for multi-version installations in TCE.

C. RPATH vs. LD_LIBRARY_PATH

One major problem with multi-version installations is that each compiled binary must be able to find its dependencies at runtime. If there are multiple versions of dependencies installed on a system, either the user must take action to set the `LD_LIBRARY_PATH` environment variable (or some other linker configuration variable), or the binary must be generated with this information embedded so that it can find *its own* dependencies. This is typically done by embedding `RPATH` entries in the binary. In the TCE environment, we rely on the `RPATH` in installed packages in order to find dependencies. This allows users to run binaries without setting `LD_LIBRARY_PATH` or any other special action. We find this approach much less error-prone in a multi-version, multi-user environment.

D. Environment Modules

Environment modules automate the process of setting environment variables (`LD_LIBRARY_PATH`, `MANPATH`, etc.) necessary to run software packages. There are many existing systems, the oldest of which is simply called *modules* and based on the Tool Command Language (TCL). The Cray computing environment, along with many clusters, use TCL modules. Loading and unloading TCL modules can alter the user's `PATH` and other, package-specific environment variables. LC has historically used an internally developed tool called `dotkit` [3] to provide many of the same features offered by TCL modules. With TCE, we have given up `dotkit` in favor of Lmod [22], a feature-rich Lua-based module system developed at the Texas Advanced Computing Center (TACC). It is backward-compatible with TCL modules and adds the ability to create hierarchies of modules. Lmod can show concise views of *only* modules that are compatible with those currently loaded in the user environment, simplifying navigation on systems with many installed modules. For example, only packages built with the GCC compiler are shown when the GCC module is loaded.

While the package installations are unified among TCE sites, environment modules are not. Rather, we install modules separately and allow each site to customize its modules. Sites can thus publish the same set of packages in a different manner, for backward-compatibly with existing job submission scripts.

III. The Spack Package Manager

Before going into the details of our TCE deployment strategy, we first give an overview of Spack [16, 21]. Spack was originally created to solve three aspects of building user-space HPC software. First, it is designed to manage the increasing combinatorial set of build configurations. Users demand that arbitrary versions of software be installed, and application developers must build many versions for testing their code with many different versions, compilers, and configurations. Second, it can be difficult to build HPC software, which often ships with immature or missing build systems [13, 16, 18, 20, 23], and automating the manual package configuration process can save considerable time. Finally, is often difficult to reproduce builds of HPC software across different environments. Spack provides a way to re-run the same automated build logic and to leverage builds designed by others.

```
1  class Hdf5(Package):
2      """A library and file format for storing and managing data."""
3      homepage = "http://www.hdfgroup.org/HDF5/"
4      url = "http://www.hdfgroup.org/releases/hdf5-1.8.13.tar.gz"
5
6      version('1.10.0', 'bdc935337ee8282579cd6bc4270ad199')
7      version('1.8.13', 'c03426e9e77d7766944654280b467289')
8
9      variant('mpi', default=False, description='Enable MPI support')
10
11     depends_on("mpi", when='+mpi')
12     depends_on("zlib")
13
14     def install(self, spec, prefix):
15         args = ["--prefix=%s" % prefix,
16                 "--with-zlib=%s" % spec['zlib'].prefix]
17
18         if '+mpi' in spec:
19             args.append('--enable-parallel')
20
21         configure(*args)
22         make()
23         make("install")
```

Fig. 1: Simplified Spack package for the `HDF5` library.

A. *Package files as templates*

Figure 1 shows a simplified Spack package file for the `HDF5` library. Each package is a simple Python file containing a class with metadata and an `install()` method. The `install()` method contains the code necessary to build the package and to install it into a designated prefix directory. The metadata includes properties such as a `homepage`, a `url` where we can download the package, and `version`, `variant`, and `depends_on` directives.

In a Spack package, metadata directives provide information about combinatorial choices in the build. Version directives allow Spack to download and checksum different source archives for different versions of the same package. Variants are boolean options that a user can set on a package. `depends_on` directives describe dependencies, which are names of other packages required to build this one. Here, The `mpi` dependency is optional because it has a `when` clause. `hdf5` will only depend on `mpi` when the `mpi` variant is set (i.e., to `+mpi`). The compiler and platform information are also parameters of a Spack build, but the mechanisms for handling these are not shown here

Spack supports the concept of *virtual dependencies*, where multiple packages may provide the same interface. In this example, `mpi` is a virtual dependency. Packages that depend on MPI can link with any MPI implementation (e.g. `openmpi` or `mpich`). We say that OpenMPI and MPICH *provide* `mpi`. Every virtual dependency must be replaced with a provider before a spec is ready to be installed.

The `install()` method can build package configurations with arbitrary values for each build parameter. Package authors check the values of the parameters inside the method. For example, on line 18 in the figure, there is a check for `+mpi` variant. If `mpi` is enabled, extra arguments are passed to `configure`. The package is effectively a template for building many different instantiations of the same package, and this is what makes building combinatorial software stacks manageable. Rather than managing a package file per configuration, with redundant build information spread over may files, users can maintain the bulk of their code in one place, with platform- and configuration-specific cases mixed in. This allows users to write very general package files.

B. *Spec DAGs*

To generate different configurations of the same package, Spack has robust infrastructure for managing the directed acyclic graph (DAG) of dependencies for each installation. Spack calls the graph a *spec*, short for *build specification*, as it tells the package file how to build. The spec DAG is passed as a parameter to the `install()` function, and the package author is responsible for translating this specification into the commands needed to perform the installation.

Figure 2 shows some of the build configurations that Spack can generate from the HDF5 package in Figure 1. When a user first invokes a command to install `hdf5`, the spec DAG is an *abstract*, or underspecified, graph, as in Figure 2a. Only the name and immediate dependencies of packages are known, and specific build parameters have not yet been assigned to nodes. We only see the two immediate dependencies of `hdf5`: `zlib` and `mpi`.

Figures 2b, 2c, and 2d show possible *concrete* specs for the HDF5 package. We call a spec concrete when each node in the DAG has a version, build parameters, a compiler, and dependencies assigned; and no dependency in the DAG is virtual. We can see from these examples that the space of build configurations is large. Figure 2b represents HDF5 version 1.8.15 built *without* the optional MPI dependency and with GCC version 4.5. Figure 2c builds HDF5 version 1.10.0 with the MPICH MPI implementation and GCC version 4.4.7. Finally, Figure 2d uses HDF5 version 1.18.15, with OpenMPI. Most of the DAG is built with the Intel compiler version 13.1, but the `zlib` dependency is built with GCC 4.4.7.

1) Specs and Concretization: Usually, when users install packages, they have only an abstract idea of what is to be installed. For example, a user may know that they want a parallel HDF5 implementation, but users likely do not know exactly what libraries HDF5 depends on, or which build options will work best on their system. Spack offers a simple, recursive syntax for specs so that users can request builds with only the constraints they are concerned with.

The simplest spec is simply a package name. For example, a request for `hdf5` is completely unconstrained. Users can add additional constraints using sigils. For example, to request a particular version or version range, users can request, e.g.:

```
hdf5@1.18
hdf5@1.10:1.18
```

To request a particular compiler, a user might request:

```
hdf5 %gcc@4.5
```

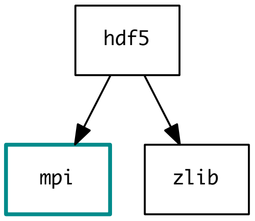
And to request variants, the user can use `+` or   to indicate whether they should be enabled or disabled:

```
hdf5+mpi
hdf5~mpi
```

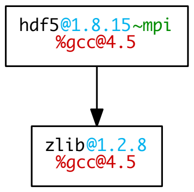
The syntax can be recursively applied to dependencies using a caret:

```
hdf5@1.10 ^zlib@1.2
```

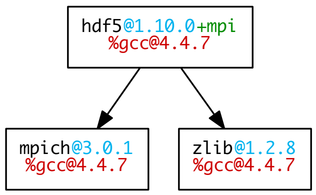
Each of the specs above will produce a partially constrained DAG. Spack fills in missing constraints using a process called



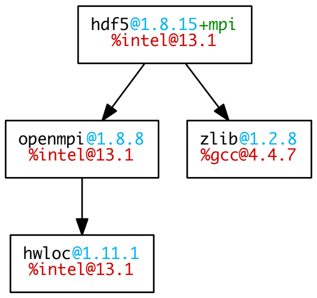
(a) Abstract `hdf5` with virtual `mpi` dependency.



(b) `hdf5` with `gcc` 4.5, without `mpi`.



(c) `hdf5` built with `gcc` 4.4.7, with `mpich`.



(d) `hdf5` built with `icc` 13.1 and `openmpi`.

Fig. 2: Possible build specs (DAGs) for the HDF5 package.

concretization, which manages conflicting constraints and fills in default values for build parameters using site- and user-provided policies. Spack effectively relieves the burden of searching for site preferences; package authors are guaranteed that the spec passed to `install()` is concrete, and they need not clutter their build logic with code to manage combinatorial parameters. Spack handles this burden and package authors can simply translate a concrete spec to a build.

C. *Installation Layout and Hashing*

Spack allows a combinatorial set of build configurations to coexist together in the same installation by installing each package into its own prefix. To do this, Spack guarantees each unique build configuration a unique directory name within the install tree. Since the build configuration is a complex graph, it is not possible to directly encode the configuration as a directory name without creating overlong file names, so Spack uses a hashes to produce a suitable identifier. Spack first writes the spec DAG into a YAML-formatted file, and then applies a cryptographic hashing algorithm to this file. We call the resulting hash the *DAG hash*, and Spack appends it to the names of Spack installation prefixes to ensure uniqueness.

Spack ensures that each package knows where to find its dependencies by adding `RPATH` entries to the binaries and libraries that it builds. `RPATH`s ensure that a binary will run and find the dependencies it was built with, regardless of the user's environment.

D. *External Packages*

Dependencies in Spack can be satisfied either by Spack built packages or by external installations. Configuration files at the Spack installation level and at a per-user level allow the user or maintainer to register system packages and other pre-installed packages with their configurations expressed in Spack spec syntax. From these files Spack can determine whether an externally installed software package satisfies a particular dependency and will link against external installations when possible.

E. *Upgrading Spack packages*

Spack allows many build configurations of packages to exist together within the same directory layout, and it provides a hashing scheme that enables the build space to be arbitrarily large. However, the DAG hash is not always the friendliest way to present an installation layout to users. Users who build against a Spack package with one DAG hash continue to rely on the specific hash long after the package is built, and it is not easy for a system administrator to upgrade a large number of hashed packages for to fix security issues or bugs. Further, installing new versions of Spack packages with minor configuration differences results in new directories being installed. This is good for reproducibility, but upgrading user software that relies on Spack packages is less transparent because it can require a recompile. Effectively, there is a trade-off between reproducibility and upgradability. In the following sections, we describe an extension to Spack that allows users to navigate this trade-off.

IV. Subspaces

As discussed in section III, Spack allows us to specify and build any point in the build space. We extend Spack with the concept of *subspaces*. A subspace is a projection from the build space into a lower-dimensional space. Subspaces allow us to support upgrade-in-place, and they allow administrators to deploy binary versions of Spack packages using the package manager of the host OS.

As discussed in section II, RPM is the package manager for TCE. In RPM, packages are distinguished by name, and only a single package of a given name can be installed at a time. Subspaces provide a naming scheme that enables combinatorially versioned packages to be installed with RPM and coexist on the system.

A. *Defining Subspaces*

Spack subspaces are subsets of the full build space described in I. They are defined by a subset of package configuration variables. Recall that these variables comprise the dimensions of the full build space—similarly, the subset of these variables comprises the dimensions of the subspace. Each unique tuple of parameter values is a unique point in the subspace, and configurations with the same subspace parameters target the same installation prefix. Typically, we add a dimension to the subspace when a deployer anticipates maintaining multiple installations of the package that differ in terms of the

```
foo:
  subspaces:
    subspace1:
      name: {NAME}-{VERSION}
      prefix: /usr/tce/tools
    subspace2:
      name: {NAME}-{VERSION}-{COMPILER}
      prefix: /usr/tce/tools
bar:
  subspaces:
    FooCompilerSpace:
      name: {NAME}-{COMPILER}
      prefix: /usr/tce/tools
    MySubspace:
      name: {NAME}-zlib-{DEP:zlib:COMPILER}
      prefix: /usr/tce/tools/my_dir
```

Fig. 3: An example subspace configuration file

variable. For example, if the deployer anticipates maintaining installations for different versions of the package, they include the version in the subspace. If they anticipate maintaining different versions built with different MPI implementations, they include the name of the MPI implementation. If a change in some build parameter should *not* require a new prefix, the deployer can simply leave the parameter out of the subspace dimensions, and changes to that parameter will simply result in an in-place upgrade.

Subspaces are defined by *subspace descriptors*, which are template strings that map a package configuration to a string. The string can contain a static ID (arbitrary text) and placeholders for arbitrary build parameters from a Spack spec. We can use this string to project a spec DAG in the Spack build space down into the lower-dimensional subspace by simply substituting the values of the variables into corresponding placeholders in the descriptor template. If projecting two specs results in the same subspace point, the two specs will share an installation directory.

As mentioned, a subspace descriptor is simply a template for turning package configurations into names. Each package may have multiple subspaces associated with it. An example configuration file for subspaces is in figure 3. Note that package `foo` has two subspaces, one of which maintains the compiler as part of the package name, while the other does not. Multiple subspaces can be used to support different groups of users with different package needs, to support particularly pathological use cases, or to continue support for an outdated subspace.

The subspace descriptor can contain arbitrary static text (so long as it is valid in an RPM name) along with a set of substitution variables which are understood by Spack. In addition to the name and version of a package mentioned above, these include the compiler, compiler version, and Spack variants enabled when building the package. Each of these details can also be queried recursively for a dependency of the package. For virtual dependencies like MPI, the variable resolves to the chosen implementation.

B. Instantiating Subspaces

An RPM name is derived from a pairing of a concretized Spack package spec and a subspace. The configuration details for the package are substituted into the subspace descriptor. Any package configuration detail omitted from the subspace descriptor does not affect the name, so two different configurations that match in terms of the subspace descriptor's variables will project to the same RPM name.

Under the subspaces in figure 3 as an example, a build of package `foo` at version 5 would project to the name `foo-5` under subspace1, regardless of the other variables on package `foo`. However, under subspace2, `foo` at version 5 compiled with the GCC compiler projects to a different name than `foo` at version 5 compiled with the Intel compiler.

Figure 4 provides an example of possible name projections for the HDF5 [14] package compiled with MPI support. We assume for the example that it is desirable to maintain separate HDF5 installations for different MPI implementations to ensure compatibility with libraries using each implementation. The upper subspace is labeled as `bad` because it projects the configurations of HDF5 dependent on OpenMPI [15] and MPICH [19] to the same RPM name. For a different package, such as zlib, which has no MPI dependency, this may be an appropriate subspace choice.

V. RPM Generation

Leveraging subspaces to generate sufficiently-unique names for combinatorially-versioned software, we can use RPM to install HPC software to the user environment. We extend Spack to generate RPM SPEC files and manage the installation of those builds to the user environment under TCE.

Spack's RPM functionality supports the generation of RPMs which install to package-specific prefixes. The actual build for a package is run under a Koji build system using Spack, and Koji packages the build results into an RPM. Spack-generated RPMs do not currently include module files, although support was added recently for automatic generation of lmod modules and this may be used in the future for RPMs.

A. Build mechanism

Koji is a tool that uses Mock to build packages in a sandboxed environment (a chroot). Sandboxing the build ensures that the environment is "clean": that nothing is installed and no environment variables are set that could change the behavior of the build. Koji builds packages and associates them with a "tag", which corresponds to an RPM repository. An RPM SPEC can require any previously built RPM SPEC with the same tag, and Koji will automatically pull in that dependency.

Spack is invoked twice in the process of building an RPM. First, Spack generates an RPM SPEC from the concretized Spack spec and writes the associated RPM file. Spack names the package according to the subspace descriptors in Spack configuration files. As part of this process Spack generates RPM dependencies based on the package DAG. Spack also packages itself along with the RPM file and associated property files into a centrally managed repository. For TCE, this repository

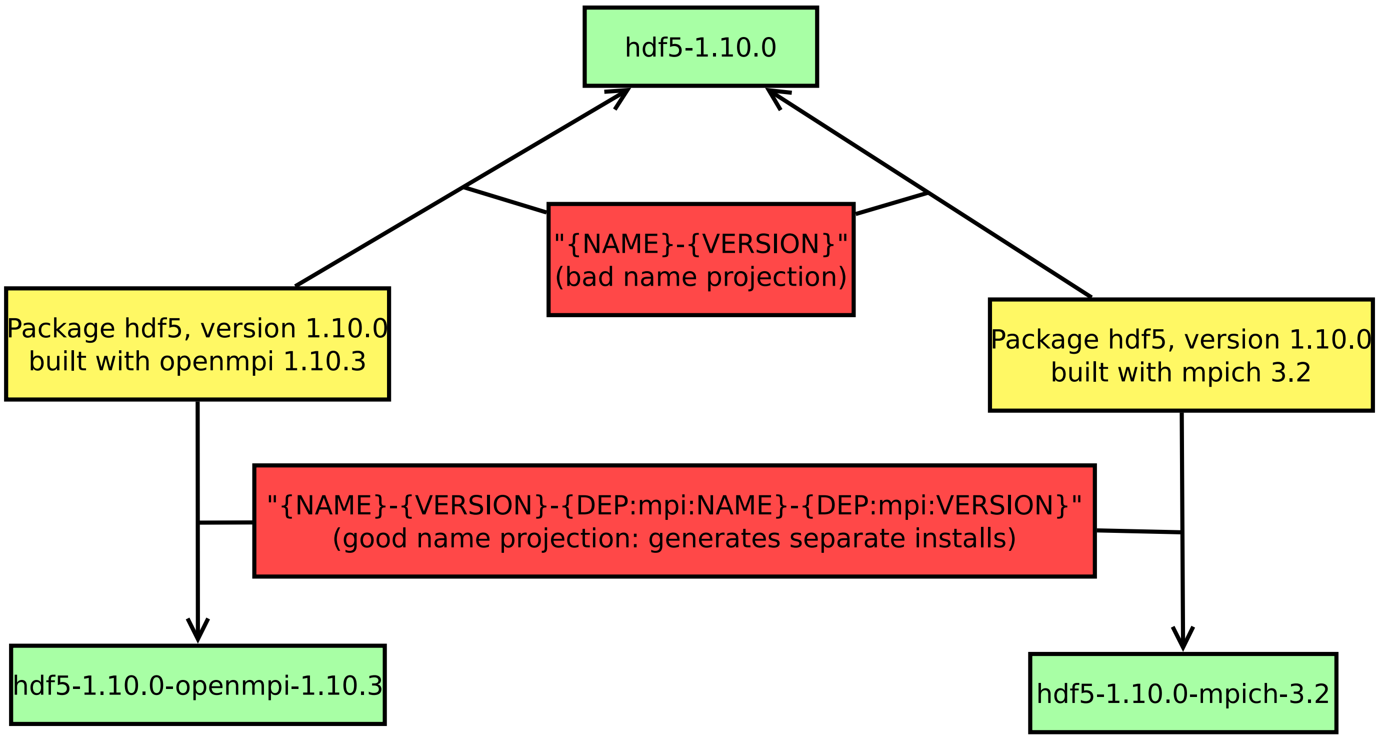

hdf5-1.10.0
"{NAME}-{VERSION}"
(bad name projection)
Package hdf5, version 1.10.0
built with openmpi 1.10.3
Package hdf5, version 1.10.0
built with mpich 3.2
"{NAME}-{VERSION}-{DEP:mpi:NAME}-{DEP:mpi:VERSION}"
(good name projection: generates separate installs)
hdf5-1.10.0-openmpi-1.10.3
hdf5-1.10.0-mpich-3.2

Fig. 4: An example of good and bad subspace usage. Yellow boxes are Spack package specs, red boxes are subspace descriptors, and green boxes are RPM names.

is managed by tosspkg, a variant of fedpkg. Second, the Koji build system executes the build as described in the RPM SPEC file. This includes calling Spack to execute the actual build process.

To support running in a sandboxed Mock environment without access to the install location of the packages it builds, SPACK supports optional redirection of installs. For many packages this is handled automatically through support for DESTDIR option in build systems such as CMake.

Spack has no dependencies other than python 2 at version 2.6 or higher, so it is easily supported in the Mock environment for building system software.

Although Spack typically automatically downloads source files during installation and validates them against a checksum for the requested version of a package, source files can be saved in a cache within Spack for build environments that do not support internet access.

B. Building and referencing dependencies

With a mapping of package name to subspace, a Spack DAG can be mapped to a set of RPM SPECs. The Spack RPM logic then applies the dependency information in its Spack DAG to set up the RPM dependencies (e.g. the "requires" tag). Figure 5 provides an example which shows the mapping of a Spack DAG to a tree of RPM dependencies given a particular subspace configuration.

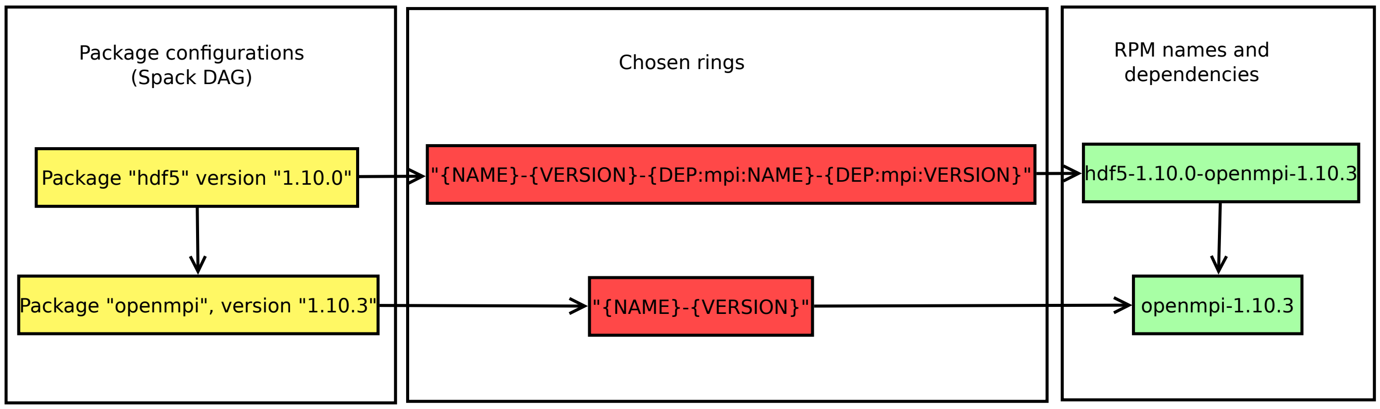
Spack RPM generation uses Spack's external packages functionality to locate dependencies. The Spack RPM SPEC generation step produces a build-specific packages.yaml configuration that enables Spack to find the packages built from RPM SPECs and use them to satisfy dependencies of future packages. Figure 6 shows the configuration file generated by the installation shown in figure 5.

Once Spack can locate the dependency packages, it will automatically set up RPATHs to these packages as part of its build, as Spack does for user-space builds as discussed in section III.

C. How to manage updates

As is discussed in section IV, from an RPM system perspective two package configurations that project to the same name are considered different releases of the same RPM. Since the installation prefix is determined by the RPM SPEC name, all RPM SPECs which depend on a given package (have the original package in their "requires" field) will transparently use any updates to that package. This is important for updates that require automatic migration to the newer version, such as security patches.

When a deployer anticipates updating an RPM package with respect to a given package configuration variable (e.g. minor version), they can omit the variable from the subspace descriptor for that package. For elements of configuration which commonly contribute to package names this creates an explicit trade-off between stability and upgradability: for example if the deployer omits the package version from the name projection, they can replace a package installation with any other version; they will only make this choice if they anticipate that new versions can be used transparently in place of old ones. However, to generate packages with simple names, users will typically omit all but a few variables from the projection, for example the major versions of dependencies. Forcing all updates that change variables omitted from the


Package configurations
(Spack DAG)
Chosen rings
RPM names and
dependencies
Package "hdf5" version "1.10.0"
"{NAME}-{VERSION}-{DEP:mpi:NAME}-{DEP:mpi:VERSION}"
hdf5-1.10.0-openmpi-1.10.3
Package "openmpi", version "1.10.3"
"{NAME}-{VERSION}"
openmpi-1.10.3

```
packages:
  hdf5:
    subspace1:
      name: '${NAME}-${VERSION}-${DEP:mpi:NAME}-${DEP:mpi:VERSION}'
      prefix: /usr/tce/packages
  openmpi:
    openmpiSubspace:
      name: '${NAME}-${VERSION}'
      prefix: /usr/tce/packages
```

Fig. 5: An example subspace configuration file and the RPM DAG generated for HDF5 under said configuration.

name to be treated as upgrade-in-place therefore creates a tension between name-simplicity and stability. It is anticipated that an additional stricter projection can be used to fix variables which the deployer prefers to omit from the name but does not anticipate updating the package with respect to. Modules, as discussed in section VI-A, provide an interface for using simpler names without the trade-off in stability.

When Spack initially creates an RPM it tracks a number of properties to determine whether a future configuration that maps to the same RPM differs from the installed version; in many cases it can automatically determine when to update an RPM. Tracked properties include the Spack spec excluding details related to dependencies, the dependencies managed as RPMs by Spack, and those dependencies exposed to Spack for which a system dependency is preferred; changes in these properties are simple to detect—for example Spack specs implement an equality operation.

Future work will involve including the associated package.py contents, a hash of the source, and the contents of all applied patches in the tracked properties. For a given Spack package file Spack will construct a hash which includes the particular build logic executed for the chosen configuration. This will ensure that Spack tracks not only the user-specified properties of the build, but also those which are an artifact of choices made by the Spack package maintainer. Until this work is complete, the RPM update logic creates a new RPM release for the top-level package every time a new build is requested, as we cannot guarantee that two builds are identical.

```
packages:
  openmpi:
    buildable: false
    paths:
      openmpi@1.10.3%gcc@4.8.5:
        /usr/tce/packages/openmpi/openmpi-1.10.3
```

Fig. 6: An example configuration file generated by to track installed packages and their properties

When Spack detects that an RPM should be updated, it is capable of automatically generating an updated RPM SPEC for the package. This includes incrementing the release number, adding a changelog entry, and updating "requires" appropriately.

VI. Usability Features

Previous sections discussed building and installing packages into a complex and multi-dimensional build space. While this is an important component of the overall problem, we also need mechanisms for end-users to navigate and select packages from this complex space. We could not expect end-users to navigate and use our build space without guidance.

There are several competing goals we maintained while building the end-user accessibility layer:

- New users should get reasonable defaults when they first log into a system. Compilers and MPI implementations should work out of the box without needing special configurations or flags. They should be able to easily navigate and select additional packages when needed. We generally expect these users would work with default versions of packages and upgrade to new versions as the center upgrades its package installs.
- Advanced users, such as dedicated build engineers for large applications, should be able to select a limited set of packages they have validated against their applications.

Their build systems typically only use specific versions of selected packages, and their upgrade cycle is desynced from the center's.

- System administrators want to control the visibility of packages they install. Some packages may be deployed as early-access or deprecated, where the system administrator does not want to advertise the packages existence, but still want it to be accessible to select end-users.

A. Modules

The three NNSA laboratories host thousands of users across their systems, and there is naturally a wide range of software packaging expertise across the user base. The majority of our users are not experts in the idiosyncrasies of packaging, and we need to provide them with a system that hides the complexities of package management. We've chosen a mixture of compiler wrappers and the Lmod module system from TACC to do so.

Lmod is a natural fit for the combinatorial multi-dimensional packaging space we deployed. Similar to other module systems such as Cray Modules [6] and Dotkit, Lmod provides simple commands for loading and managing packages in a user's environment. For example, a user could issue an Lmod command to load the Intel compiler version 16.0.3, and the appropriate paths would be added to their PATH and MANPATH environment variables.

Unlike some other module systems, Lmod understands the package hierarchy. It can limit the slice of the package space visible to users and prevent them from loading incompatible packages. For example, the Dotkit module system does not understand the relationship between MPIs, compilers, and packages. A user could ask Dotkit to load OpenMPI, the GCC compiler, and the HDF5 package into their environment and get an HDF5 compatible with MPICH and an OpenMPI built with the Intel compiler. Dotkit is not aware of the relationships between packages, and does not know that none of these would work together. On the other hand, Lmod can be configured to understand the relationships between packages. If a user asks for OpenMPI, GCC, and HDF5 Lmod understands that it should load the OpenMPI built with GCC and the HDF5 built with GCC and OpenMPI. If a user loads GCC and OpenMPI modules, then queries what modules are available, Lmod will only show modules compatible with GCC and OpenMPI.

The Lmod hierarchy is configurable, and for TCE systems we mimicked the TACC recommended configuration. Users first select a compiler at the top-level of the Lmod hierarchy, which reveals the available MPI implementations built with that compiler. Selecting an MPI then reveals the available packages built with that MPI and compiler. When users first log in they are given a site-chosen MPI and compiler by default, though they can easily change these with Lmod commands. Some packages are installed in the compiler and MPI independent parts of the hierarchy, such as GIT version control software and debuggers like TotalView. Some packages are installed the compiler-specific but MPI-independent part of the hierarchy, such as the Boost C++ libraries. Some packages are installed in the compiler and MPI specific part of the hierarchy, such as HDF5. Spack is able to generate Lmod files in a configurable hierarchy, which helps manage the installation of the module files.

B. Compiler Wrappers

One of our goals is to make the TCE software environment predictable and reproducible. Based on our experiences working with end-users we believe that a frequent class of bugs comes from users mixing up libraries, runtimes, and compilers between iterative builds and between build– and runtime. For example, a user may have two terminal windows open on a screen, one loaded with the environment for GCC 4.8.5 and one loaded with the GCC 4.9.3 environment. If they switch between windows in the middle of a build (perhaps after the build was interrupted by a compilation error), they may get hard-to-interpret linking errors from mixing compilers or strange faults at runtime. Perhaps more commonly, users may build against one runtime environment, such as with the MPICH MPI implementation, then try to run with a different runtime loaded, such as the OpenMPI MPI implementation.

We cannot completely save users from these scenarios—a determined user will be able to mix compilers together regardless of any safeguards we put in place. But we can put in place mechanisms that try to prevent these scenarios from occurring in common usage patterns. A general guideline we have tried to follow is that the behavior of a package should be independent of the user's environment at runtime. The most common place this comes up is with library search path and guaranteeing that an application that is built against a library only runs with that library. An application that builds against MPICH and GCC 4.9.3, for example, should always run with the MPICH and GCC 4.9.3 runtimes, even if the OpenMPI package is loaded in the environment.

We enforce this through compiler wrappers and the RPATH [12] mechanism. RPATH is a mechanism by which a compiler can embed a library search path into a library or executable. When the library is loaded or executable is run the system's dynamic linker will inspect that embedded search path and load libraries from it before loading libraries that may be found via the user's environment (such as LD_LIBRARY_PATH). We install compiler wrapper scripts for each MPI/compiler combination, and we use these scripts to automatically add RPATHs to user's binaries for every TCE library they link against. We also build TCE's own libraries and packages with RPATHs so that users do not need to concern themselves with finding the transitive dependencies for our TCE libraries. In addition to allowing new users to ignore many details of library loading, the RPATHs and compiler wrappers also allow advanced users to ignore the Lmod system, if they so choose. Some advanced users want to limit their packages to only build with certain compilers or MPIs and not be susceptible to the multi-terminal problem mentioned above. By specifying their compilers using exact paths to wrapper scripts, the advanced users can guarantee that only that MPI/compiler combo is being used to build their code.

It is worth noting that the use of RPATH in packaging is controversial. Policies in the Debian community [7] about RPATH can be summarized with the statement "RPATH

Considered Harmful," and we have heard similar sentiments from other Linux distributions. We acknowledge some of these objections, such that RPATH can make it difficult for a user to replace system packages with their own installations, particularly when a system library location (such as /usr/lib64) is added to an RPATH. However, in a multi-compiler and multi-library-version environment such as TCE we believe that RPATH's benefits outweigh these problems. Typical users do not build their own MPIs or compilers, and the few who may do not expect them to be drop-in replacements.

We also use our compiler wrappers to hide other idiosyncrasies and common bugs in compilers. One common example comes from how the Intel compiler handles GCC compatibility. When run the Intel compiler will search for a version of GCC in the user's PATH and generate object files that are compatible with that GCC. This works well for systems with a single GCC installed, as the Intel compiler always finds the same GCC. However, this can cause issues when multiple GCC versions are installed in parallel. A user building an application with the same version of the Intel compiler may get very different builds because they loaded a different GCC compiler into their environment—a non-intuitive side effect. We avoid this by having our compiler wrappers lock the Intel compiler to a specific version of GCC, and only allow it to be changed by users who explicitly request a different version.

C. Package Visibility

Package deployers also need fine-grained control over what packages users can see. For example, a package deployer may want to install a new version of the TotalView debugger, but they want to test it before advertising it to users. Some software testing can be done in a controlled environment away from users, but some testing requires "friendly" users testing at full scale. While test packages must be available to these users, other users should not accidentally use untested software.

Another element of package visibility concerns is default package naming. In section VI-A, we discussed the default packages that users should find in their environment. For some packages, these defaults differ substantially from the package names supplied by Spack subspaces. For example, there should be a package "python" in the users path, even though multiple versions of python should be supported. We support this model through symbolic linking of a default choice for certain packages to the install location chosen by Spack with our subspaces extension. For example, by default "python" may be a symlink to "python-2.7.12" if the subspace descriptor for python is `{NAME}-{VERSION}`.

We support both fine-grained control over package visibility and default packages by splitting the RPM for each package into three separate installations. The "base" RPM SPEC installs the package, but does not add the package's modules to the default MODULE_PATH environment variable, nor does it effect default packages. The "public" RPM SPEC adds the package's modules to the MODULE_PATH environment variable. The "default" RPM SPEC updates the symbolic links for the package name so that the base name is linked to the package whose default RPM was installed. For example, if a package deployer installs the RPM SPEC named "python-3.3.2-default," then the symbolic link from "python" to "/usr/tce/tools/python-2.7.12" is removed and replaced with a link from "python" to "/usr/tce/tools/python-3.3.2," assuming the same subspace as above. Because we RPATH all dependencies automatically, changing the default for a package does not change the behavior of user code transparently (because RPATHs follow symbolic links and include the true path, rather than the provided path, in the compiled code).

The deployment interface supported by splitting the RPM SPEC into three parts also supports seamless deprecation of outdated package installations. To deprecate a package, we remove the public interface of that package by uninstalling the "public" RPM SPEC for that package. This removes the package from the module layout of the user environment, but leaves the software in place so as not to break previously compiled user code with RPATHs to the deprecated package. Advanced users who still require the package can still link against it, but new users will not find it accidentally.

Future work will include implementing testing and deprecated environments within Lmod. This will enable users to easily add the entire set of deprecated packages, or of packages installed for testing, to their module path if they so desire.

VII. Related Work

Recently, many tools have emerged to manage HPC software installations in user space, including Smithy [9], Maali [1], EasyBuild [18, 20], and Spack [16, 21]. All of these arose from efforts of HPC center support staff to reduce the complexity of installing software. Smithy and Maali do not include robust dependency management capabilities; they check to ensure that prerequisite modules are installed but users must still know what packages their software depends on and ensure that prerequisites are built consistently with their dependents. While these tools do address the problem of scripting the build process, they do not provide any management of the relationships between packages and do not fully address the combinatorial build problem.

EasyBuild and Spack have both developed broad communities outside the institutions of their creators, and they are used at many HPC sites. EasyBuild was designed from the start to manage a software stack for HPC users. It can support combinatorial versioning of some packages through a *toolchain* mechanism, but its dependency mechanisms are not as general or flexible. Each software stack in EasyBuild must be fully specified through *easyconfig* files, and changes to the version of one package in a hierarchy of easyconfigs often require updating that package's version across *all* easyconfigs in the stack. The EasyBuild distribution currently has over 6,500 easyconfigs for just over 1,000 software packages.

In contrast, Spack currently supports 670 packages, but it uses a single `package.py` file per package, and choice points in the build space are expressed as parameters to avoid redundant configuration. With the subspaces introduced in this paper, creating a new software stack from package templates is a matter of creating a subspace descriptor. This flexibility allows stacks with different combinatorial properties to be generated with little effort. Like Spack, EasyBuild has support for generating RPMs, but only for configurations already packaged with easyconfigs. It does not support the customizable versioning and upgrade strategies that subspaces provide.

In addition to the HPC-specific package management tools mentioned above, a number of sites have attempted to use so-called *functional* package management tools such as Nix [4, 10, 11] and its GNU analog, Guix [5]. These tools build packages in an isolated `chroot` environment, so offer greater reproducibility than Easybuild or traditional Spack usage. However, unlike these tools, Spack and Easybuild provide direct support for interfacing with the module system of HPC machines when performing a build; in particular this is the only reliable way to build on Cray machines.

Spack's DAG hashes and versioning system were inspired by Nix's hashing scheme. Nix and Guix both allow arbitrary combinations of package to coexist on a system using a similar mechanism. However, Spack builds on this system by offering more fine-grained control over DAG components through optional and virtual dependencies. Spack's templated build parameters, spec syntax, and concretization allows users more flexibility to build combinatorial versions of the same package, a task frequently needed on HPC systems.

VIII. Conclusion

HPC software management is a complex task, and it requires support staff to manage a combinatorial set of packages to satisfy users. While tools exist for managing and deploying combinatorial builds, they often require frequent package rebuilds, and it can be difficult to provide in-place upgrades of site-provided packages for fully combinatorial deployments. At the tri-labs, to integrate with our production TOSS environment, we need a deployment solution that allows for in-place upgrades and that interfaces with package managers such as RPM.

In this paper, we extended Spack, a combinatorial package management tool, to provide these capabilities, and we adapted our environment to support this new deployment workflow. We introduced the concept of *subspaces*, which allow support staff to project the combinatorial Spack build space down to a lower-dimensional, upgradable set of install locations, and we introduced a simple configuration methodology using *subspace descriptors*. Our solution allows support staff to choose an arbitrary degree of combinatorial complexity for their deployment, and to trade off fine-grained DAG hashing and reproducibility for in-place upgradability and reuse. To our knowledge, no other system provides this level of flexibility for generating binary packages. We integrated this technique with our local RPM deployment environment.

In addition to the packaging techniques mentioned above, we described the ways in which we have integrated our deployment workflow into the tri-lab compute environment. This included techniques for generating and managing Lmod modules, as well as compiler wrappers that simplify for users the task of working in a mixed-compiler environment on our newest commodity clusters.

Acknowledgments

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory, Contract DE-AC52-07NA27344. LLNL-CONF-701389.

References

[1] R. C. Bording, C. Harris, and . D. Schibeci). Using maali to efficiently recompile software post-CLE updates on a Cray XC system. In *Proc. Cray User Group Meeting (CUG2015)*, 2015.

[2] R. L. Braby, J. E. Garlick, and R. J. Goldstone. Achieving Order Through CHAOS: The LLNL HPC Cluster Experience. In *The 4th International Conference on Linux Clusters: The HPC Revolution*, San Jose, CA, June 24-26 2003. Linux Cluster Institute.

[3] L. Busby and A. Moody. The Dotkit System. http://computing.llnl.gov/?set=jobs&page=dotkit.

[4] B. Bzeznik. I'm using Nix! In *Bull User Group Meeting (BUX 2016)*, March 22 2016.

[5] L. Courtés and R. Wurmus. Reproducible and user-controlled package management in hpc with gnu guix. In *Workshop on Reproducibility in Parallel Computing (RepPar)*, Vienna, Austria, August 25 2015.

[6] Cray Inc. Optimizing Applications on the Cray X1 System. http://docs.cray.com/books/S-2315-52/html-S-2315-52/b8umksmg.html.

[7] Debian. Rpath issue. https://wiki.debian.org/RpathIssue.

[8] T. D'Hooge. TOSS: Speeding Up Commodity Cluster Computing. http://computation.llnl.gov/projects/toss-speeding-commodity-cluster-computing.

[9] A. DiGirolamo. The Smithy Software Installation Tool. http://github.com/AnthonyDiGirolamo/smithy, 2012.

[10] E. Dolstra, M. de Jonge, and E. Visser. Nix: A Safe and Policy-Free System for Software Deployment. In *Proceedings of the 18th Large Installation System Administration Conference (LISA XVIII)*, LISA '04, pages 79–92, Berkeley, CA, USA, 2004. USENIX Association.

[11] E. Dolstra and A. Löh. NixOS: A Purely Functional Linux Distribution. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, pages 367–378, New York, NY, USA, 2008. ACM.

[12] U. Drepper. How to write shared libraries. December 10 2010. https://www.akkadia.org/drepper/dsohowto.pdf.

[13] P. F. Dubois, T. Epperly, and G. Kumfert. Why Johnny Can't Build. *Computing in Science and Engineering*, 5(5):83–88, Sept. 2003.

[14] M. Folk, A. Cheng, and K. Yates. Hdf5: A file format and i/o library for high performance computing applications. In *Proceedings of Supercomputing*, volume 99, pages 5–33, 1999.

[15] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, et al. Open mpi: Goals, concept, and design of a next generation mpi implementation. In *European Parallel Virtual Machine/Message Passing*

Interface Users' Group Meeting, pages 97–104. Springer, 2004.

[16] T. Gamblin, M. P. LeGendre, M. R. Collette, G. L. Lee, A. Moody, B. R. de Supinski, and W. S. Futral. The Spack Package Manager: Bringing order to HPC software chaos. In *Supercomputing 2015 (SC'15)*, Austin, Texas, November 15-20 2015. LLNL-CONF-669890.

[17] J. E. Garlick and C. M. Dunlap. Building chaos: an operating system for livermore linux clusters, 2002.

[18] M. Geimer, K. Hoste, and R. McLay. Modern Scientific Software Management Using EasyBuild and Lmod. In *Proceedings of the First International Workshop on HPC User Support Tools*, HUST '14, pages 41–51, Piscataway, NJ, USA, 2014. IEEE Press.

[19] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel computing*, 22(6):789–828, 1996.

[20] K. Hoste, J. Timmerman, A. Georges, and S. De Weirdt. EasyBuild: Building Software with Ease. In *High Performance Computing, Networking, Storage and Analysis, Proceedings*, pages 572–582. IEEE, 2012.

[21] Lawrence Livermore National Laboratory. Spack. http://github.com/LLNL/spack.

[22] R. McLay. Lmod: Environmental Modules System. https://www.tacc.utexas.edu/research-development/tacc-projects/lmod.

[23] G. Wilson, D. A. Aruliah, C. T. Brown, N. P. C. Hong, M. Davis, R. T. Guy, S. H. D. Haddock, K. Huff, I. Mitchell, M. D. Plumbley, B. Waugh, E. P. White, and P. Wilson. Best Practices for Scientific Computing. *CoRR*, abs/1210.0530, 2012.