Using Spack to Manage Software on Cray Supercomputers

Mario Melara[1], Todd Gamblin[2], Gregory Becker[2], Robert French[3], Matt Belhorn[3],
Kelly Thompson[4], Peter Scheibel[2] Rebecca Hartman–Baker[1]
[1]*National Energy Research Scientific Computing Center,* [2]*Lawrence Livermore National Laboratory,*
[3]*Oak Ridge National Laboratory,* [4]*Los Alamos National Laboratory*

***Abstract*—HPC software is becoming increasingly complex. A single application may have over one hundred dependency libraries, and HPC centers deploy even more libraries for users. Users demand many different builds of packages with different compilers and options, but building them can be tedious and error-prone. Support teams cannot keep up with user demand without better tools. Spack is an open-source package manager created at Lawrence Livermore National Laboratory (LLNL). Packages in Spack are *templates* for many combinatorial build configurations. They provide fine-grained control over a build's directed acyclic graph (DAG). Spack is used by HPC centers, developers, and users to quickly deploy software. This paper describes our integration of Spack with the Cray Programming Environment (CrayPE). We describe changes we made to Spack's support for environment modules and machine architectures, as well as preliminary results of the rollout of Spack on NERSC's Cray XC40 system (Cori).**

***Index Terms*—software stack management, package manager**

I. Introduction

CRAY supercomputers provide cutting-edge compute power to users in many scientific fields. It is important for HPC support teams to provide these users with optimized software. Providing such software can be difficult and time consuming due to the complexity of advanced architectures and of scientific software. For example, the quantum chemistry package *CP2K*[1] requires 26 dependencies and multiple configuration variants that make it difficult to build from source. However, source distributions are the *de facto* packaging medium for supercompter software, so consultants spend hours and days fighting build and compilation issues rather than supporting for scientific application development directly.

NERSC maintains two Cray supercomputers, Edison and Cori, each of which has a large multi-versioned software stack. On Edison, NERSC supports 250 packages including its own developed software, vendor-provided tools, and third-party packages. These systems support several different compilers and programming environments: *Intel*, *GNU*, *Cray*, and *PGI*. As of 2016, there are nearly 7,000 users on NERSC systems with more than 800 projects [2], [3]. With these needs in mind, NERSC investigated various package management tools.

We discovered Spack [4], [5], the Supercomputing PACKage manager, a tool developed at Lawrence Livermore National Laboratory. It draws inspiration from the popular macOS package manager *Homebrew*[6], and Nix, a so-called "functional" package manager [7], [8]. Spack packages are templates that describe how to build packages. Using package files, users of Spack can control the combinatorial configuration space of a package and its dependencies. Package files also also provide a way to distribute software across platforms. Users can interface with the command line and provide a specification of their build using a specialized *spec* syntax.

To run on Cray machines, Spack must integrate with the Cray Programming Environment (CrayPE). This required several modifications to Spack to support CrayPE modules, compiler wrappers, and environment variables. Previous versions of Spack had run mostly on Linux clusters and macOS laptops. With a growing need for Spack on Cray machines, NERSC and LLNL collaborated to port Spack to the Cray environment. With our modifications, compiler usage on Cray machines is simple, and it mirrors the way Spack is used on other systems. To enable this, we added additional features to handle the complex environment and architecture.

In this paper we describe the functionality of Spack, the challenges we faced while compiling in the Cray environment, and the adaptations NERSC and LLNL made to Spack for our Cray machines. In addition, we review other package managers and common problems between these tools, as well as our future plans for enabling such tools. Our goal is to allow users to install optimized packages as easily on supercomputers as they do on their laptops.

II. The Supercomputing PACKage manager

Spack is a flexible and powerful tool that automates the installation of complex scientific software. Originally developed for use at Lawrence Livermore National Laboratory, Spack has seen an increase in usage among development teams across many sites. At the core of Spack are package templates, which describe the *ways* a package can be built, and a specialized *spec* syntax, which allows users to query and designate *specific* builds from the combinatorial space of configurations. Its dependency model also supports versioned, ABI-incompatible interfaces like MPI through *virtual* dependencies. Together with Spack *specs*, these features provide a simple way to manipulate package builds, and they hide build complexity from the user by providing a unified interface to many different build systems. In the following sections, we provide more details on Spack's functionality. For still more details, see the online Spack documentation [9].

Spec type	Spec Symbol	Example
package name	N/A	`python`
version	`@`	`python @2.7.13`
compiler	`%`	`python @2.7.13 %gcc`
architecture	`arch=`	`python @2.7.13 %gcc arch=cray-cnl6-haswell`
variant	`+` / `-`	`python @2.7.13 %gcc +tk arch=cray-cnl6-haswell`
compiler flags	`[ld|cpp|c|cxx|f] flags=`	`python @2.7.13 %gcc +tk arch=cray-cnl6-haswell cflags=-02`

TABLE I
MEANING AND EXAMPLE SYNTAX FOR PACKAGE SPEC CONSTRAINTS.

```
from spack import *


class Libelf(AutotoolsPackage):
    """libelf lets you read, modify or create ELF object files"""

    homepage = "http://www.mr511.de/software/english.html"
    url = "http://www.mr511.de/software/libelf-0.8.13.tar.gz"

    version('0.8.13', '4136d7b4c04df68b686570afa26988ac')
    version('0.8.12', 'e21f8273d9f5f6d43a59878dc274fec7')

    provides('elf')

    def configure_args(self):
        return ["--enable-shared",
                "--disable-dependency-tracking",
                "--disable-debug"]

    def install(self, spec, prefix):
        make('install', parallel=False)
```

Fig. 1. Simple example of a `package.py` file. This package subclasses `AutotoolsPackage`, Spack's built-in support for autotools builds. `install()` can be left out, but is overridden here to force a sequential build. The package uses the `provides` directive to specify that it is provides the *virtual package* `elf`. i.e., It can be used with packages that depend on `elf`.

A. Spack Packages

A Spack package repository consists of a number of `package.py` files in per-package directories. Each `package.py` file is a template that describes the combinatorial space of the package, as well as the commands needed to build the package. Package files are similar to Homebrew "formulae"; both allow for specification of all potential variants, dependencies, and configuration options that can be used. Each `package.py` file contains its own subclass of Spack's `Package` class, which controls the fetching, extracting and installation of a package. To allow for better control over the different build phases, Spack can autodetect the build system of package (i.e. Makefiles, Autotools, Python, R) from a sample release tarball (or other archive). It uses this to create a boilerplate `package.py` file. Figure 1 shows an example.

B. Templated, Combinatorial Packages

Traditional package managers and port systems do not handle combinatorial versioning well. Typically, they manage a single package ecosystem, with one version of each package in a software stack. Large, multi-user HPC sites, however, require multiple different versions of the same software, often with different configurations and dependencies. Spack's ability to parameterize packages allows for fine-grained control over a software stack through specs and packages. The spec syntax allows package template authors developers to easily query the requested build configuration and to conditionally add dependencies and configuration options. Package authors are passed a `spec` object, which describes the directed acyclic graph (DAG) of a package's dependencies and their configurations.

1) Package Directed Acyclic Graph: To understand how Spack parameterizes build configurations, requires knowledge of Spack's core data structure, the *spec* DAG. Consider the `libelf` package. Metadata in the *Libelf* class describe the package's dependencies, versions, and variants. This tells Spack how packages are related to each other, and what dependencies are needed in which configuration. Using this information, Spack recursively inspects the class definitions for dependencies, and it constructs a DAG of these dependencies. Spack iteratively solves for a configuration that satisfies version requirements and conditional dependencies. It uses heuristics to try to generate ABI compatible packages, but if the user specifies something that is *not* ABI compatible it will comply with that request. Spack supports installation of multiple packages with arbitrarily many different configurations, and it uses its own hashing scheme to identify unique DAGs. Within a DAG, there can be only one instance of a particular package (i.e., two configurations of the same package cannot be linked together in a single build).

2) Spec Syntax: Users can manipulate and constrain a package DAG by using a simple *spec* syntax on the command line. This syntax is expressive enough to allow users to configure and customize their software stack to their liking, but concise enough that users don't have to know the entire structure of the DAG. Here is a simple example:

```
spack install mpileaks
```

This tells Spack to install the `mpileaks` package, without regard for its configuration. We call `mpileaks` an *abstract* spec, i.e., it is only a partial description of a build. Table I shows increasingly complex examples of specs. In each case, a new constraint is added and the requested build becomes more specific. Spack uses a *concretization* algorithm to fill in unknown configuration details with sensible defaults. It converts an *abstract* spec to one that is *concrete*. This allows the user to specify only the constraints that matter to them, rather than an entire, detailed build specification.

The architecture spec in Table I has been recently developed by NERSC and LLNL. Briefly, it consists of a `platform-os-target` triple. Cray machines are considered to be a single *platform*, but different nodes in the system may have different operating systems and target hardware. For example, the login node on a Cray machines may run SuSE Linux, while the compute nodes run a different OS and may have different types of processors. Spack automatically

detects the architecture spec for the machine it runs on, but it allows users to build for other OS's and targets (assuming it is possible to do so). A user can choose a target on the command line using an architecture spec, and the architecture descriptor is part of a spec, so packages can discover what architecture they are built for. We discuss the architecture spec in detail in section IV-B.

The combination of package template files and specs allows teams fine-grained manipulation of a software stack. Because of this, the HPC teams at NERSC are no longer stuck with just a single stack with a single package version; they can support many configurations, versions, and optimizations. We believe that the freedom Spack affords developers, to optimize and build new configurations easily, is very important in the HPC environment. HPC users need to be able to experiment with potentially many different builds of their packages to achieve the best performance.

C. Automation of Package Builds

After concretization, the spec DAG is concrete, and Spack can build the DAG recursively, starting from the leaves. For each package in the DAG, Spack fetches the source, creates a build stage in the filesystem, and calls the package's `install()` method. As shown in Figure 1, `install()` takes a `spec` argument. Because all specs passed to `install()` are *concrete*, package authors need not perform complex searches or queries of the underlying system in order to build. Concretization handles the work of assembling a build configuration and packagers simply translate a concrete DAG into build commands.

During the install phase, Spack injects its own compiler wrappers into the build system. These wrappers are used for adding include (`-I`), library (`-L`), and `RPATH` (`-Wl,-rpath`) flags for dependencies. They also add compiler flags specified through the command line or through a package file. The wrappers ensure that packages can automatically find their dependencies. On Cray, we modified Spack to also load the appropriate modules for the CrayPE within the build environment. Spack will set the `CC`, `CXX`, `F77`, `FC` environment variables to point to *its* wrappers, and it sets `SPACK_CC`, `SPACK_CXX`, `SPACK_F77`, `SPACK_F77` the location of the real compiler executables. The compiler wrappers use these variables to call the wrapped compilers.

D. Modules

After a successful installation, Spack auto-generates environment module files in several formats. Current support for modules includes TCL, Lmod, and Dotkit. If shell support has been activated, Spack can manage the loading and unloading of module files using the spec syntax, as well as dependency modules. However, current management of modules is slow and it is often better to use the *–shell* flag to `spack module` to generate faster shell code. Modules can also be configured as we described in the following section.

E. Configuration

Spack has a collection of `yaml` files with parameters for customizing modules, compilers, external packages, and various other aspects of Spack. Most of these configurations are not necessary for basic functioning; one goal of the Spack team is for the tool to work immediately "out of the box." Spack's configuration system supports nested scopes that allow customization at the global, site-specific, or user level. There is also support for platform-specific settings for HPC teams that support multiple platforms. Here we will briefly describe some configuration files and settings possible under Spack.

1) External Package Support: Often, Spack users want to build with packages that are already present on the system, rather than compile a new one. To allow for this, users can register a package in Spack's *packages.yaml* configuration file. Each entry in this file maps a spec to a path to the prefix of an external package installation. Spack can then treat that package as any other Spack built package and automate linking and dependency handling. Spack trusts the user to provide valid spec metadata – typically at least a package and a version are required, but it is recommended to add compiler, architecture, and variants to `packages.yaml` specs. Spack does its best to work specs in `packages.yaml` into the build during concretization.

In addition to externals, the `packages.yaml` configuration file also allows users to specify concretization preferences for packages that Spack builds. Packages can default to a specific compiler and compiler version, package version, and variant. One thing to note is that once a package is declared an external, it does not mean that Spack can only use that external package. If a more recent version of a package is made available, Spack will begin compiling and using that package. However, if a package version is considered stable and preferred over a more recent experimental version, that version `can` be pinned in the configuration file. This allows support teams some flexibility in how they configure their software stack.

2) Modules Configuration: The *modules.yaml* configuration file allows users to customize how Spack generates module files. Module systems like Lmod, dotkit, and TCL modules can be activated or deactivated. This is useful if an HPC site does not support one or more of the supported module systems. Since different HPC sites have different naming schemes for their modules, Spack allows users to configure the naming schemes of modulefiles to model their own by using a special spec format [10]. Users can also prevent the generation of modulefiles on a per package basis. Spack can generate module files that mimic those in the CrayPE, but there is currently no way to make Spack's modules support the `module swap` functionality that Cray's modules ship with. This is because the swapping logic requires that all modules be known at generation time, and we would need to update Cray's modules to fully support swapping with Spack's. We settle for working load/unload functionality.

3) Compiler Configuration: Spack stores any data about compilers into a *compilers.yaml* file. By default, this file is auto-generated by inspecting the user's *$PATH* environment variable and environment modules. This file can be further

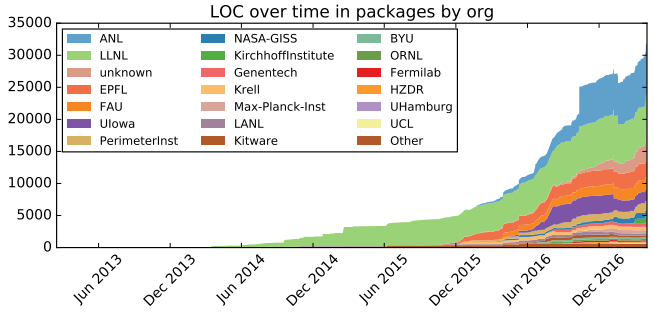

Fig. 2. Contribution of packages has increased. 75% of packages are contributed by outside sources.

customized with additional compilers and with default compiler flags, as well as with specific module dependencies for each compiler. For example, *Intel* compilers require `gcc` header files, so Intel depends on the `gcc` module. Paths specified in `compilers.yaml` are activated during installation as described in Section II-C

4) RPATH linking: Spack's configuration options allow its users to maintain multiple software stacks using package templates and the *spec* syntax. Each package configuration is installed in its own prefix, and users can access these packages using modules. However, Spack installations do not rely on the `LD_LIBRARY_PATH` set by modules to find their dependencies. Rather, packages are able to peacefully coexist without environmental dependencies due to Spack's use of `RPATH` for linking. An `RPATH` is a library search path stored by the compiler (in this case by Spack's compiler wrappers) in the generated binary. Using `RPATH`, a package will always find its link dependencies, regardless of the user's environment.

F. The Spack Open Source Community

Perhaps the most notable aspect of Spack is its growing user community. Spack's package repository supported only 300 packages when it was first presented at the Supercomputing conference in November of 2015 [4], but it has grown to over 1,300 packages the 18 months since. Figure 2 shows the rate of increase. We can see that in November of 2015, nearly all Spack packages were developed by LLNL. One year later, over 75% of the lines of code in packages is contributed by outside organizations. Spack now has over 130 unique contributors, and 30-50 pull requests are merged into the git repository each week. It is used at many of the largest DOE laboratory computing centers in the US, and it has become a booming community for HPC application developers and facility support staff.

III. The Cray Programming Environment

Cray architectures differ significantly from commodity Linux clusters. Recently, NERSC has acquired the *Cori* Cray-XC40 machine. Cori is ranked the 5th most powerful computer in the world as of November 2016, according to the Top 500 list [11]. It includes 2,004 Intel Xeon *Haswell* and 9,300 Intel Xeon Phi *"Knight's Landing"* (KNL) nodes. NERSC also maintains the older Edison XC30 machine. Building software for machines like this is complex as it requires support for heterogeneous architectures. On both machines, the login (front-end) and compute (back-end) nodes use different chipsets. We must therefore compile differently for *at least* these two machines. On Cori, the situation is even more complex: we must also consider the dual architectures (Haswell and KNL) of the compute nodes, Over 250 unique software packages are maintained by NERSC staff on these systems, and support staff frequently need to build them with multiple compilers. On a commodity cluster, we typically only expect a single architecture and compiler, and we typically build and run on exactly the same type of node.

As mentioned, Cray machines come with the Cray Programming Environment (CrayPE), a collection of modules and libraries that simplifies, to some extent, the process of buliding for multiple architectures. However, the CrayPE is not a package manager and it does not automate builds. It is designed with a human in mind. Its interface is environment modules. Without a programmatic API, automating tasks in this environment is difficult. This section provides a brief overview of the Cray Programming Environment (CPE) and the mechanisms it uses to control builds.

A. TCL Environment Modules

Most Cray systems come with TCL modules to control the environment. Modules allow system administrators or vendors to specify sets of environment changes in *module files*. Users can simply type `module load <name>` or `module unload <name>` to enable or disable a particular module. This eliminates the need for users to set any environment variables explicitly, e.g. adding executable directories to *$PATH* or library directories to *$LD_LIBRARY_PATH*. Cray provides users with a default set of site-specific modules as shown in Figure 3. Cray's modules are pre-compiled and optimized for the machine. Some are libraries or tools, such as the *udreg* and *atp* modules. We focus here on the `PrgEnv` and target modules.

1) Programming Environment Modules: Loading a programming environment, or `PrgEnv`, module will add a set of tools, math libraries, and communication libraries to the user's environment. This includes *cray-mpich, cray-fftw, and cray-libsci*, to name a few. `PrgEnv` modules also affect the loaded compiler and associated build toolchain. Cray ensures that libraries in the `PrgEnv` modules are consistent, compatible, and optimized. Examples of programming environment modules are:

- Intel Programming Environment
- GNU Programming Environment
- Craype Programming Environment

B. Cray Compiler Wrappers

Aside from modules, the main interface to the CrayPE is Cray's compiler wrappers. Cray provides (`cc`, `CC`, `ftn`) for `C`, `C++`, and `FORTRAN` languages. Cray's environment modules affect the behavior of the compiler wrappers, including the libraries they link to and the flags passed to the compilers

```
Currently Loaded Modulefiles:
 1) modules/3.2.10.5                          12) xpmem/2.1.1_gf9c9084-2.38
 2) nsg/1.2.0                                 13) job/2.1.1_gc1ad964-2.175
 3) intel/17.0.2.174                          14) gni-headers/5.0.11-2.2
 4) craype-network-aries                      15) alps/6.3.4-2.21
 5) craype/2.5.7                              16) rca/2.1.6_g2c60fbf-2.265
 6) cray-libsci/16.09.1                       17) atp/2.0.3
 7) udreg/2.3.2-7.54                          18) PrgEnv-intel/6.0.3
 8) ugni/6.0.15-2.2                           19) craype-haswell
 9) pmi/5.0.10-1.0000.11050.0.0.ari           20) cray-shmem/7.4.4
10) dmapp/7.1.1-39.37                         21) cray-mpich/7.4.4
11) dvs/2.7_2.1.67_geb949b6-1.0000.eb949b6.2.32
```

Fig. 3. The default modules available on Cori (Cray-XC40). Loading and unloading of these modules modifies the Cray Programming Environment.

to optimize for specific back-end architectures. The link statements are long and complicated but hidden from the user. One can see how long these statements are by passing the *-craype-verbose* flag to the compiler wrapper. Cray recommends compilation of code through their compiler wrappers, and invoking the underlying *Intel*, *GNU*, or *CrayPE* compiler directly will result in incorrect linkage on Cray systems, and code that can only be executed on the login node [12].

1) Compiler Modules: Loading a *PrgEnv* module is the primary way that users swap compilers on Cray machines, but this will load a *default* version of each compiler. Most HPC sites contain a number of different versions of the same compiler. To fine-tune the compiler version, Cray also offers separate compiler modules to load specific versions. This is important for Spack, as Spack allows users to specify very specific compiler versions using specs. In the CrayPE, the compiler module is the interface between the user and the compiler. We must be conscious of *both* the `PrgEnv` module and the compiler modules, as only the compiler modules corresponding to the loaded `PrgEnv` can affect the behavior of Cray's compiler wrappers. For example, one can load a *GNU* compiler module under the *Intel* programming environment, but it does not affect compilation with Cray's wrappers. We must *first* load the `GNU PrgEnv` module, *then* the correct compiler module.

2) Target Modules: Cray machines use a heterogeneous architecture that separates nodes into front-end and back-end nodes. The front-end nodes run a Linux operating system and are used to compile code and manage data, while the back-end nodes run a specialized version of the Linux kernel and are used for HPC workloads. It is necessary to cross compile all code for the back-end nodes using the Cray compiler wrappers on the front-end nodes. Cray provides target modules that specify the back-end architecture to optimize for. As mentioned, Cori has two types of backend nodes: Haswell and KNL. By swapping target modules on the front-end nodes, users can generate optimized code for either processor.

C. Cray and Third-Party Software

On every system, Cray provides optimized software libraries. This software includes custom-built versions of MPI (`cray-mpich`), I/O libraries such as NetCDF (`cray-netcdf`) and HDF5 (`cray-hdf5`), and advanced math libraries such as PETSc (`cray-petsc`) and Trilinos (`cray-trilinos`). In addition, Cray provides a collection of numerical routines called the Cray Scientific Library (`cray-libsci`), which provides libraries for BLAS, LAPACK, and SCALAPACK. Cray LibSci is loaded by default under the `GNU` and `Cray` Programming Environments, and the `Intel` programming environment uses Intel's own optimized Math Kernel Library (`mkl`).

IV. Spack on Cray

Spack is designed to be a portable build interface and to allow packages to be built on many different platforms using the same commands. Spack's configuration typically comes from files or from the command line, while on Cray machines the configuration is heavily based on environment settings. Before Spack would work as expected on Cray systems, a number of modifications were required. Spack needed better support for modules, better support for heterogeneous hardware, better support for additional compiler wrappers (beyond its own), and better support for static linking. We describe these modifications in this section. As of Spack v0.10, these features are available to all Cray users.

A. Module Support

CrayPE is fundamentally a module-based system, and this is the interface that Cray supports for its developers. Modules are required to set the compiler, MPI implementation, and math libraries, and target for each build. As such, to build on Cray, Spack needed to be able to make use of Cray's modules. Spack's build environment is designed to be sandboxed, and Spack actively *unsets* user environment variables that may affect the build adversely. This includes many variables that are set by the CrayPE modules.

Modules are easy for users but difficult as a programmatic interface. Fundamentally, environment modules are shell files that are *sourced* to execute within the current shell. However, Spack is implemented in Python, and it cannot execute shell code directly. While Spack can execute module code in a subshell, the subshell has no effect on Spack's own environment, so launching a subshell, loading a module, and returning to Python has no effect on Spack's build process.

Fortunately, the *module* command is a wrapper around the *modulecmd* executable, which can generate code for a variety of scripting languages. `modulecmd` takes an argument to indicate the language for which it should generate code. As `modulecmd` supports Python, we can have Spack call `modulecmd` to generate Python code that sets the environment as a shell would, and we can `eval` the Python code from within Spack's Python interpreter. The generated Python code *directly* modifies `os.environ` and other globals in Python, but Spack sandboxes each build in its own subprocess, so we can safely evaluate all `modulecmd` code in each package's build subprocess without corrupting Spack's global state. Using this mechanism, we were able to add the capability to "load" and "unload" modules to Spack.

B. Architecture Support

To support heterogeneous architectures, we needed to update several aspects of Spack's architecture support. Prior to

version 0.10, Spack encoded the architecture in each spec as a single string. This was derived from LLNL's long tradition of providing a `SYS_TYPE` environment variable on each of its systems. The variable contained an identifier that correlated loosely to binary compatibility across machines. The original `architecture` field in specs was similar; it mainly provided a way to separate packages built on one platform from those built on another.

Heterogeneous machines and cross-compilation require the build system to have a notion of *both* the front-end (login) and back-end (compute) targets. Build dependencies like CMake must be built for the front-end architecture, and link dependencies like math libraries must be built for the back-end. Further, as Spack relies to some extent on system libraries like `glibc`, the operating system stack on the front-end and back-end nodes is significant for differentiating incompatible binaries. We need to know, for example, whether an installation relies on the SuSE system libraries of the front-end nodes, or on the CNL OS on the backend nodes. Finally, as Spack *must* run on the front-end nodes (where users build packages), we need a way to make it aware of the available front-end and back-end nodes for the entire platform.

Given this system model, we introduced the concepts of a `platform`, `os`, and `target` to Spack. Internally, we introduced classes to represent each of these concepts, an we added logic to Spack's `Spec` class to allow more detailed architecture descriptors. The new descriptors are `platform-os-target` triples. The *platform* is auto-detected and indicates the type of the overall system. Spack supports platforms like `linux` for commodity Linux, `darwin` for macOS, `bgq` for BlueGene/Q machines, and `cray` for Cray machines. Each platform *may* have multiple operating systems (`os`) and targets (`target`) associated with it, and the platform knows which is the default front-end and back-end OS. Further, different operating systems have different *compiler* detection strategies, and Spack knows which *compilers* are associated with which OS.

The scheme has proven sufficiently general to model a range of systems, from Linux, to macOS, to Cray, to Blue Gene/Q, and it allows us to quickly add new machines to Spack. Making concept of front-end nodes explicit in Spack has enabled us to rework Spack's concretization logic to compile build dependencies (like CMake, make, or their dependencies) for the front-end machine where they must run, while still compiling back-end libraries for optimized compute nodes.

Our architecture support also gives the user fine-grained control over builds. To compile for a given architecture, they can specify the entire architecture string using, `arch=cray-cnl6-haswell`, or they can specify the operating system and target independently with `os=cnl6` or `target=intel-mic`. We also added default frontend and backend targets and operating systems so that users would not have to be intimately familiar with the Cray machine to build on it. Users can request specific operating systems or targets using `os=frontend` or `os=backend`. Targets are similarly available on the command line with `target=backend` or `target=backend`. With these additions, Spack is, to our knowledge, the only package manager with a dependency model that supports heterogeneous machines to this degree.

```
packages:
  mpich:
    buildable: false
    compiler: []
    modules:
      mpich@7.3.2%intel@17.0.0.098: cray-mpich/7.3.2
      mpich@7.4.1%cce@8.4.4: cray-mpich/7.4.1
      mpich@7.4.1%gcc@6.1.0: cray-mpich/7.4.1
    paths: {}
    providers: {}
    version: []
  python:
    buildable: false
    compiler: []
    paths:
      python@2.7.12%gcc@6.1.0: /global/common/software/python
      python@2.7.12%intel@17.0.0.098: /global/common/software/python
      python@2.7.12%cce@8.4.4: /global/common/software/python
```

Fig. 4. To link with cray packages we use Spack's configuration file packages.yaml. We can declare a package to be located via path or by module. If the package is declared as a module, then Spack will find the directory using 'module avail'. Spack's compiler wrappers will then link the package.

C. External Packages

Support for building with external packages already existed in Spack prior to version 0.10. As described in Section II-E1, Spack will normally build any Spec in a concrete DAG, but using a `packages.yaml` file, we can specify that Spack should rely on existing installations of certain packages and *not* build them. Originally, it could only do this using a package prefix, and it would use the prefix to determine where its compiler wrappers should point their `-I`, `-L`, and `-Wl,-rpath` arguments.

To support Cray provided packages, we added the ability to record a *module* for an external package instead of an installation prefix path. If an external package has a module listed in the external packages configuration file instead of a path, Spack will load the corresponding module and use the information in the module to link against the package if it is a link dependency. Now Spack can support *either* path-based or module-based externals in a single build. An example of a record of an external module can be seen on Figure 4.

D. Improved Compiler Detection and Wrapper Handling

With the new platform detection algorithm and support for modules, Spack could now fully support loading and unloading of Cray compilers and programming environments. However, Spack was Spack was originally designed to build using system-provided compilers directly, and not through custom wrappers. Its compiler *detection* logic searched the user's `$PATH` for compilers, and it stored the compiler metadata into a *compilers.yaml* configuration file. Spack did not check for Cray compiler modules. Moreover, the Cray compilers must be called through Cray's wrappers.

We improved compiler detection methods by allowing Spack to parse the output of the `module avail` command and create a compiler configuration by pointing to the Cray compiler wrappers. We also associated a compiler with an operating system and target to help distinguish between compilers found via the two different methods as shown in Figure 5. After detecting the available compilers on a system, we next pointed Spack to use the Cray compiler wrappers.

```
- compiler:
    environment: {}
    extra_rpaths: []
    flags: {}
    modules: []
    operating_system: sles12
    paths:
      cc: /usr/bin/gcc-4.8
      cxx: /usr/bin/g++-4.8
      f77: null
      fc: null
    spec: gcc@4.8
    target: x86_64
- compiler:
    environment: {}
    extra_rpaths: []
    flags: {}
    modules:
    - PrgEnv-gnu
    - gcc/6.3.0
    operating_system: cnl6
    paths:
      cc: cc
      cxx: CC
      f77: ftn
      fc: ftn
    spec: gcc@6.3.0
    target: any
```

Fig. 5. New addition of operating system and target to compilers. Users can specify front-end or back-end architecture by specifying either target or os on spec.

Cray discourages using the direct compiler binary since it only allows for packages to compile against the front-end. We modified the `compilers.yaml` metadata stored on Crays to point to the Cray compiler wrappers. This was done by adding any compiler found via modules to point `cc`, `cxx`, `f77`, and `fc` to `cc`, `CC`, `ftn`, and `ftn`, respectively, *instead* of the actual compiler name. This avoids bypassing Cray's compiler wrappers.

Spack still has its *own* compiler wrappers, which we still use in Spack builds. The only difference is that previously, Spack's wrappers would call the selected compiler directly, and now Spack's wrappers call Cray's wrappers, which eventually call the module-loaded compiler. Now, not only can Spack automatically handle linking of dependencies but it can also use Cray's own lib and include flags. This process is shown in Figure 6.

E. MPI and External Vendor Packages

In the next section we will use the Message Passing Interface (MPI) as an example of a package that must be configured as external on Cray systems.

The Message Passing Interface (MPI) is a parallel programming model widely used in high performance computing software. Cray provides a Message Passing Toolkit (MPT) that includes `cray-shmem` and `cray-mpich`. To use the MPI compilers, the `cray-mpich` module must be loaded and the environment variables of `MPICC`, `MPICXX`, `MPIFC`, and `MPIF77` should be be set to point to the Cray compiler wrappers. Since users should not install their own versions of MPI on Cray machines, MPI should be configured to always use mpich when on a Cray platform, and mpich should be listed in the external packages configuration file as having a module "cray-mpich." The logic in handling the MPI environment variables lies in its package file. By listing mpich as a preferred package, and registering `cray-mpich` as the external, we can use the MPI compiler through the cray wrappers. This logic is shown in Figure 7.

F. Static and Dynamic Linking

The GNU linker can be configured to use either static or dynamic linking by default. Depending on the default, the user must supply either a `-static` or a `-dynamic` flag to enable the non-default mode. Most systems default to *dynamic* linking and require developers to pass `-static` to the linker when they want to do a static link, but the CrayPE defaults to *static* and developers must explicitly pass the `-dynamic` flag.

Most build systems assume that linkers will be dynamic by default, and many even hard-code this behavior and lack settings to change the default sense of the linker flags. This means that many build systems simply fail in the Cray environment, as they *cannot* pass the necessary `-dynamic` flag when it is needed. To standardize the environment and prevent linking errors, Spack sets the environment variables `$CRAYPE_LINK_TYPE` to `dynamic`. This causes the Cray linker to behave as most packages expect, and increases portability across the Spack software stack. Note that this does *not* mean that we have to build all of our packages dynamically on Cray machines. It simply means that, to get a static build, we need to pass the flag, which is what most common open source build systems already support.

1) Future Cray Support: We have highlighted most of the changes we made to Spack to support Cray machines. We are working towards improving Spack Cray support on an ongoing basis, and our roadmap for Spack version 1.0 includes improved support for external packages, improved module support, and front-end compilation of build dependencies.

V. Spack at NERSC and ORNL

Once we implemented the necessary changes and features needed to control the Cray programming environment, NERSC decided to test out using Spack to manage installs on our Cray-XCs. We will go over the preliminary results of our experiences using Spack at NERSC. Also included is usage of Spack at Oak Ridge National Laboratory.

A. NERSC

Here we present our workflow at NERSC as well as the packages we have provided to users using Spack.

1) Pseudo-user SWOWNER: To handle installs on NERSC systems, a dedicated pseudo-user account called *swowner* was created for NERSC staff to log in and compile NERSC-provided software packages. This user has special write and read access for installing software under a global shared directory. Once a user has logged in as *swowner* they can install packages either via a bash script or using Spack.

2) Spack Workflow: To allow for site-specific changes, we forked the Spack LLNL repo into a NERSC github account. To keep LLNL upstream changes separate from NERSC changes, we created a git branch called *nersc-spack* that can be used to change any core Spack code prior to merging upstream. Our

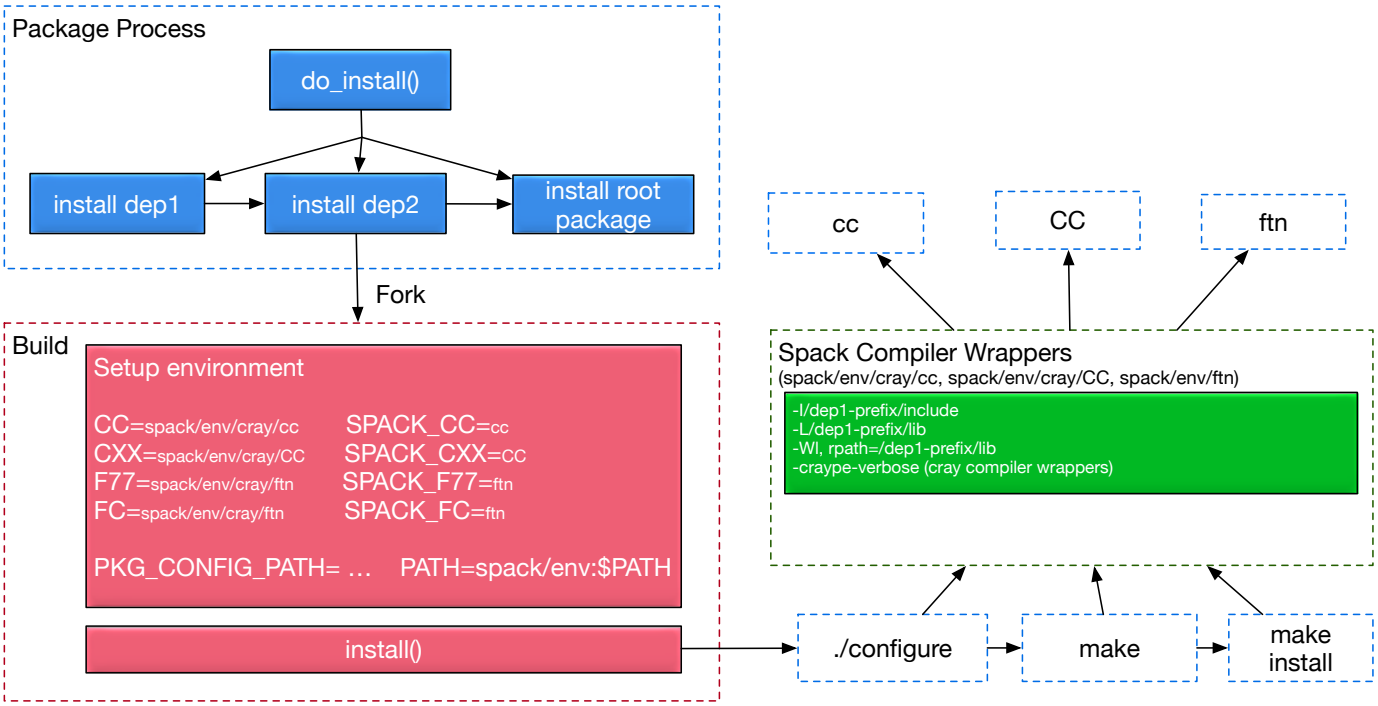

Fig. 6. Spack compiler wrappers. In this figure we describe the process in which Spack installs packages. Each dependency is installed by Spack prior to the root package. When a package is built, an environment specific to that build is forked ensuring that no package environments collide. Once the install procedure is executed the Spack compiler wrappers point to the cray wrappers (cc, CC, ftn).

```
def setup_dependent_environment(self, spack_env, run_env, dependent_spec):
    # On Cray, the regular compiler wrappers *are* the MPI wrappers.
    if 'platform=cray' in self.spec:
        spack_env.set('MPICC',  spack_cc)
        spack_env.set('MPICXX', spack_cxx)
        spack_env.set('MPIF77', spack_fc)
        spack_env.set('MPIF90', spack_fc)
    else:
        spack_env.set('MPICC',  join_path(self.prefix.bin, 'mpicc'))
        spack_env.set('MPICXX', join_path(self.prefix.bin, 'mpic++'))
        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)
```

Fig. 7. Handling of the MPI in Cray and other platforms. The mpich package.py file contains the logic for pointing the MPI to the correct executables.

reasoning behind this is to have changes to either core Spack code or the package repository immediately available to our consultants rather that submit a pull request. Consultants create private repositories using Spack's repository feature which creates a unique namespace for Spack packages. Creating a private repository is useful for consultants to avoid using a broken package file.

3) Spack Packages Provided: We provide a very limited stack of software on our systems. Currently, we are testing our workflow in using a single Spack instance. We are able to install some packages such as *abinit*, *wannier90*, and *boost* libraries, but most of our production software consists of smaller libraries such as: openssl[13], gmp[14], mpfr[15].

B. Oak Ridge National Laboratory

ORNL houses Titan, Eos, Gaia-c3, and Gaia-c4. All except for Titan are Cray-XCs with the Gaia supercomputers being Cray-XC40s and Eos being a Cray-XC30. Their Titan supercomputer is an older Cray XK7 model. They currently maintain a minimal software stack of Spack-built packages on their Crays. Most of their Spack packages are maintained on their non-Cray supercomputers.

1) Managing Spack Installs: Currently, a single Spack instance is present per host on Titan and Eos. On Gaia, there exist two Spack instances per host. This however will soon change as multiple Spack instances will exist per each host. Plans are to have a Spack instance dedicated to building for the login nodes and base OS compiler and a second that uses builds from the first instance to assist with building scientific software for the computer nodes. All software installs are made via a continuous integration system, where an install is made in a developmental location to prevent concretization issues or inconsistencies with builds. A single Spack instance is used amongst the support teams to install. Once an installation is shown to work, the CI system deploys modulefiles in a CrayPE consistent manner to general users' `$MODULESPATH`.

All software built by Spack is available to users. Many of the staff members at ORNL maintain private Spack instances for testing and personal use. Similar to NERSC, ORNL has future plans to provide Spack as a module to its users. However, the availability will only be granted to a select few users and development teams. Users will be able to take advantage of Spack's database introspection functions.

2) Development Teams: Development teams at ORNL are free to use their own instance of Spack to install third party

software in their own private locations. However, any software that is made available to general users must go through their CI system to keep track of what is installed.

C. Spack Results and Limitations

We found Spack to have a lot of potential in our software maintenance but lacked production stability with packages. It has powerful features such as database querying and cross-compilation using the architecture spec. During our time using Spack, we found it a bit rigid when it comes to modifying modulefiles and the installation tree. Most HPC sites have their own conventions and hierarchy of directories for their packages. Spack unfortunately does not currently allow for customization of trees (though this is a project already underway). We also found support for installing with compilers from different platforms was needed. At times Spack can get confused with it's *compilers.yaml* configuration and use compilers from another system, or output an error message that the compiler could not be found. We also found Spack unable to recognize already installed dependencies. Spack tends to re-download dependencies from a software stack even if it has already been downloaded as a dependency from another package. This could lead to a cluttering of low-level dependency libraries. At Oak Ridge, Spack is used to maintain 7 packages out of 193 on Titan, and 5 packages out of a total of 73 on Eos. Currently, main usage of Spack is seen in Oak Ridge Leadership Computing Facility (OLCF) and National Climate-Computing Research Center (NCRC) machines. Another limitation of Spack involves the lack of testing of stable packages on Cray supercomputers. Most testing is done in a Linux container environment that lacks the complexity of a Cray environment. Though these limitations have slowed down adoption at NERSC, work is being made to improve package stability on Cray machines as well as development to move Spack towards a beta release.

VI. Other Package Managers

Other package managers have been developed at other sites to support Cray supercomputers. In this section we will compare three package managers with Spack: *EasyBuild*, *Maali*, and *Smithy*.

A. EasyBuild

EasyBuild is a tool for building and installing scientific software. It is developed and maintained by HPC support teams at Ghent University (Belgium) and has been in development since 2009 and was publicly in 2012[16]. EasyBuild is a collection of Python modules that interact with each other and can pick up additional Python modules needed for building and installing software packages specified via specification files. EasyBuild consists of three different parts: a framework, install procedures called **easyblocks**, and specification files called *easyconfig* files. Together these help automate and build packages on HPC systems.

Easybuild is Spack's main competitor in the HPC space, and both systems are notable for having active developer communities that span large numbers of HPC sites.

B. Maali

Maali is a package manager developed at Pawsey Supercomputing Centre. It is a lightweight automated system used for managing optimized scientific libraries. Maali consists of a set of **bash** scripts that read like a template file. These scripts help automate the Autoconf process of `configure`, `make`, and `make install`. It has also been developed to handle CMake and Python builds. Maali depends on system-level configurations to set environment variables that can be used for building and defining the build environment[17].

C. Smithy

Smithy is a package manager written in Ruby that was first developed at Oak Ridge National Laboratory. It borrows heavily from the MacOS package manager *Homebrew*. Its main purpose was to handle software stacks specifically for Cray supercomputers. It uses "formulae", which, like Spack's package files, describe the build process.[18].

D. Key differences

This section compares how each package manager handles software installations with a specific emphasis on the Cray supercomputing environment.

1) Managing Dependencies: Both Spack and EasyBuild contain logic for managing dependencies. In Spack, dependency resolution is handled dynamically, for each build, with its *concretization* algorithm. Spack works with an in-memory model of the DAG and exposes DAG semantics to packages at build time. The same `package.py` file can be used to build many different versions and configurations of a package. Once a DAG is concretized, Spack will install each dependency on the DAG according to its spec. EasyBuild uses either an `easyconfig` file or modules to determine which dependencies are available. If no dependencies are available via modules it will search through a user-configured path for other `easyconfig` files and install each dependency. Easybuild's dependency structure is static, that is, easyconfig files do not use any templating, and new packages for new platforms and versions require new config files. Further, while Spack maintains a database of installed packages and tracks installed dependencies, EasyBuild does not track dependencies after a packages is installed. At this time Smithy and Maali do not support any dependency resolution; it is entirely up to the user to manage resolving installation of dependencies for complex packages.

2) Compilers: All package managers that support the Cray environment are able to interact with compilers through modules. EasyBuild uses compiler toolchain modules to help setup the build environment prior to installation. Adjustments were made to provide support to the CrayPE compiler toolchains for GNU, Intel, and Cray Compiling Environment (CCE)[19]. To use such toolchains requires the user to simply enter the desired toolchain into an `easyconfig` file and EasyBuild will use that compiler and load the appropriate modules. Spack can autodetect compilers by searching the `$PATH` and available Cray modules. Spack also differentiates between front-end and back-end compilers making it easy to swap compilers in and

out of builds, while re-resolving dependency DAGs with each change. To use such compilers one can either set the target in the spec or the operating system. EasyBuild and Spack both load the appropriate `PrgEnv` and `compiler` modules for a given toolchain. Smithy and Maali also interact with compilers via the module system, but Smithy requires that the **build_name** of a formula match the required compiler toolchain name. Once the name has been queried, the appropriate modules will be loaded. Maali requires an environment module be set to the desired compiler toolchain before building, so the user must handle interaction with the Cray environment manually.

Aside from Spack, each package manager requires a dedicated file or specification of a compiler. Although this is not terribly difficult, it does add complexity to the process, since many files are needed for each programming environment. Spack can build its `compilers.yaml` file automatically, and detected compilers can plug seamlessly into package templates.

3) Generation of module files: Smithy, Maali, EasyBuild, and Spack all support module file generation. EasyBuild integrates well with the Cray modules. During an install, module files are generated and integrated to the modules environment. These modules are automatically available to the user and EasyBuild as well. Spack can also auto-generate module files and allows for configuration of module file naming scheme and manipulation of environment variables. Smithy and Maali both create module files but the information contained in them is hard coded. Easybuild and Spack both have integration with Lmod and TCL modules.

VII. Common issues installing on Cray

Our implementation of Spack on Cray machines revealed some common issues that were also encountered by others who have attempted to produce automated build systems for Cray, e.g., the Easybuild team [19].

a) Clean build environment: Both EasyBuild and Spack have difficulty obtaining sanitized build environment on Cray machines. Cray provides a default list of modules that each contain an important set of optimized libraries and compiler flags. To avoid accidentally linking or using the wrong package during some builds requires *unloading* modules that are *not* part of Cray's standard list. We expected that the **module purge** command would be able to create a "factory" environment, but instead it breaks the module system. After a `module purge`, the default environment cannot be reconstructed without logging out of the system and then logging in again. This triggers the system init scripts, which add environment settings that are *outside* the control of the module system, but required to load a proper environment. Reconstructing a proper CrayPE environment from a purged state is sensitive to the order in which modules are loaded and to the particular *site* where the Cray system is hosted. This is because Cray allows sites to heavily customize their installations, and there is no "factory" setting for a Cray. We believe that it is important to be able to recover a factory setup when `module purge` is run, so that we can create a consistent build environment that is portable across Cray sites. The lack of such a capability impedes our ability to collaborate with other sites on Spack packaging.

b) Integration with Cray modules: Another common problem is integration of modules with the Cray programming environment. Making `module swap` behave properly for `PrgEnv` modules is difficult, as it requires lengthy conditional logic. Easybuild adds custom logic to their modules, but Spack does not yet add this. Whenever a user loads a specific module in a programming environment, the respective package will be loaded. Although this can be resolved with adding a conditional, we think it would be best if Cray provided a hook for modulefiles so that whenever a user switches a programming environment, that package will be switched with the appropriate one.

c) Linker behavior: As mentioned in Section IV-F, the default behavior of the linker on Cray machines is incompatible with many common build systems. We believe it would be easier to port many open source packages to Cray if the default linker mode were more similar to most commodity Linux/UNIX environments.

d) Compiler wrapper names: Cray uses a nonstandard set of names for its compiler wrappers. Specifically, it sets their names to `cc`, `CC`, and `ftn`. While the Cray names are simple and straightforward for humans writing basic makefiles, they can cause portability issues, because some build systems (including the widely used `libtool` component of `autotools`) are sensitive to the compiler name and use it to deduce the correct compiler and link flags to use during the build. Porting packages to Cray may require patching the affected build system or adding special compiler flags.

Spack's compiler wrapper methods have another beneficial side effect for some builds. Spack ensures that during a build, its compiler wrappers are *named* similarly to the selected compiler. That is, if we are compiling with `gcc`, we ensure that the compiler wrapper scripts are called `gcc`, `g++`, `gfortran`, etc., and if we are compiling with the Intel compiler, we similarly use the standard `icc`, `icpc`, and `ifort`. This makes porting some packages to Cray *easier* when using Spack than they would be without it.

VIII. Conclusion

As a result of our collaboration with LLNL, various other Cray sites have shown interest in using Spack for their software management. For example, Los Alamos National Laboratory has show interest in using Spack and has contributed packages that have been built on Cray. Although NERSC has not fully depended on Spack for software management, our current experience has been satisfactory with the Spack package manager. NERSC is currently working on stabilizing Spack on both of our Crays. Some limitations of Spack such as custom module file generation and detection of external packages are being worked on, and it is our hope that they will be released for either version 1.0 of Spack or for future releases. Currently, Spack is running tests on the *cDASH* platform, and NERSC plans to run nightly tests on our machines to ensure stable packages. In addition, we are planning on providing Spack as a module for users to install their software. We found that Spack works well for users wanting to install their own configured package and to install

older less supported software. We also believe that Spack would be suitable for user-defined containers, such as our own Shifter[20]. Since Shifter will contain lightweight Linux images, we believe that Spack with it's powerful capabilities to provide different software stacks would work very well in a container environment.

ACKNOWLEDGMENT

This work used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. ORNL research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725. Work at Lawrence Livermore National Laboratory was performed under the auspices of the U.S. Department of Energy under Contract DE-AC52-07NA27344.

REFERENCES

[1] J. Hutter, M. Iannuzzi, F. Schiffmann, and J. VandeVondele, "cp2k: atomistic simulations of condensed matter systems," *Wiley Interdisciplinary Reviews: Computational Molecular Science*, vol. 4, no. 1, pp. 15–25, Jan. 2014. [Online]. Available: http://onlinelibrary.wiley.com/doi/10.1002/wcms.1159/abstract

[2] NERSC, "Usage Demographics," Apr. 2016. [Online]. Available: http://www.nersc.gov/about/nersc-usage-and-user-demographics/usage-demographics/

[3] ——, "Number of NERSC Users and Projects Through the Years," Apr. 2016. [Online]. Available: http://www.nersc.gov/about/nersc-usage-and-user-demographics/number-of-nersc-users-andjavascript:void(0);-projects-through-the-years/

[4] Todd Gamblin, Matthew P. LeGendre, Michael R. Collette, Gregory L. Lee, Adam Moody, Bronis R. de Supinski, and Scott Futral, "The Spack package manager: bringing order to HPC software chaos," *Supercomputing 2015 (SC'15)*, vol. 00, p. 12, Nov. 2015.

[5] G. Becker, P. Scheibel, M. P. LeGendre, and T. Gamblin, "Managing Combinatorial Software Installations with Spack," in *Second International Workshop on HPC User Support Tools (HUST'16)*, Salt Lake City, UT, November 13 2016.

[6] Rmi Prvost, "Homebrew." [Online]. Available: https://brew.sh/

[7] 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)*, ser. LISA '04. Berkeley, CA, USA: USENIX Association, 2004, pp. 79–92. [Online]. Available: http://dl.acm.org/citation.cfm?id=1052676.1052686

[8] E. Dolstra and A. Löh, "NixOS: A Purely Functional Linux Distribution," in *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '08. New York, NY, USA: ACM, 2008, pp. 367–378. [Online]. Available: http://doi.acm.org/10.1145/1411204.1411255

[9] Todd Gamblin, "Spack Documentation Spack 0.10 documentation," 2013. [Online]. Available: https://spack.readthedocs.io/en/latest/

[10] T. Gamblin, "Modules Spack 0.10 documentation." [Online]. Available: https://spack.readthedocs.io/en/latest/module-file-support.html/using-module-files-via-spack

[11] H. Meuer, E. Strohmaier, J. Dongarra, and S. Horst. (2009) "Top500 Supercomputer Sites". [Online]. Available: http://www.top500.org

[12] Cray Inc, "Cray Programming Environment User's Guide," Mar. 2013. [Online]. Available: http://docs.cray.com/books/S-2529-111/S-2529-111.pdf

[13] "Openssl: Cryptography and ssl/tls toolkit." [Online]. Available: https://openssl.org

[14] "Gmp:arithmetic without limitations." [Online]. Available: https://gmplib.org

[15] "Gnu mpfr library." [Online]. Available: https://mpfr.org

[16] K. Hoste, J. Timmerman, A. Georges, and S. D. Weirdt, "EasyBuild: Building Software with Ease," in *Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, ser. SCC '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 572–582. [Online]. Available: http://dx.doi.org/10.1109/SC.Companion.2012.81

[17] R. C. Bording, Christopher Harris, and David Schibeci, "Using Maali to Efficiently Recompile Software Post-CLE Updates on a CRAY XC System," 2015, p. 7.

[18] A. Digirolamo, "smithy - build, test, and install software with ease." [Online]. Available: http://anthonydigirolamo.github.io/smithy/smithy.1.html?3

[19] P. Forai, K. Hoste, G. Peretti-Pezzi, and B. Bode, "Making Scientific Software Installation Reproducible On Cray Systems Using EasyBuild," May 2016.

[20] "Shifter: User Defined Images." [Online]. Available: http://www.nersc.gov/research-and-development/user-defined-images/