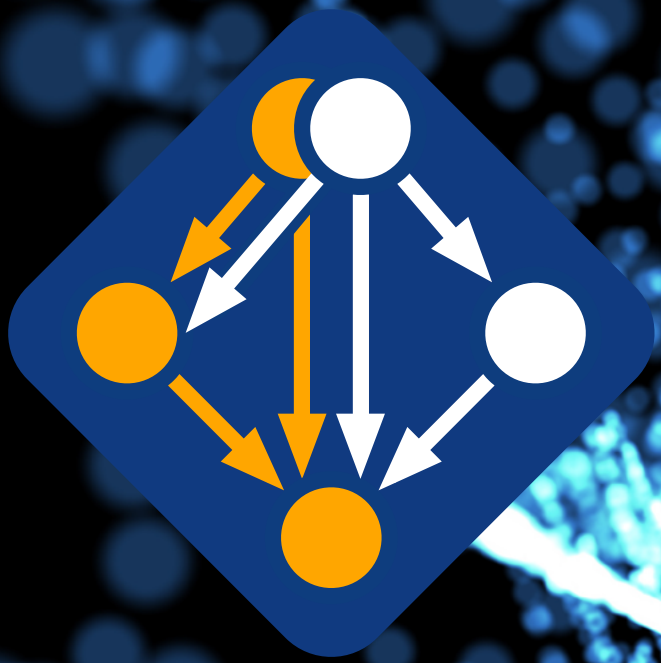
Lawrence Livermore National Laboratory

7000 East Avenue
Livermore CA 94550

Contact
Todd Gamblin

Spack: A Package Manager for HPC Systems



Prepared for:

2019 R&D 100 Award Entry

LLNL-ABS-777858

Prepared by LLNL under Contract DE-AC52-07NA27344.

Spack: A Package Manager for HPC Systems

1. PRODUCT/SERVICES CATEGORIES

A. Title

Spack

B. Product Catagory

Software/Services

2. R&D 100 PRODUCT/SERVICE DETAILS

A. Primary submitting organization

Lawrence Livermore National Laboratory

B. Co-developing organizations

Argonne National Laboratory; Columbia University; École Polytechnique Fédérale de Lausanne; Fermi National Accelerator Laboratory; Iowa State University; Kitware, Inc.; NASA Goddard Institute for Space Studies, Center for Climate Systems Research; National Energy Research Scientific Computing Center; Perimeter Institute; University of Hamburg; University of Illinois at Urbana–Champaign; University of Iowa

C. Product brand name

Spack

D. Product Introduction

This product was introduced to the market between January 1, 2018, and March 31, 2019.

This product is not subject to regulatory approval.

E. Price in U.S. Dollars

$1.5M (U.S. dollars) average in annual development costs; open source (free to users)

F. Short description

Spack is an open source software package management tool for scientific computing. It simplifies and accelerates building, installing, and customizing complex software stacks. Spack unifies software deployment for laptops, clusters, and supercomputers, enabling a community of thousands of users to share and leverage over 3,200 scientific software packages.

G. Type of institution represented

Government or independent lab/institute

H. Submitter's relationship to product

Product developer

I. Photos

Attached inline

J. Video

https://youtu.be/D0p5xpsboK4

3. PRODUCT/SERVICE DESCRIPTION

A. What does the product or technology do?

Spack automates the process of building and installing scientific software on laptops, high-performance clusters, and supercomputers. Modern scientific software combines libraries written in many programming languages and is deployed on diverse computing architectures. To achieve the best performance in these environments, developers build software directly from source code. This allows compilers to optimize code for the unique hardware it will run on. However, building software by hand is tedious and error-prone. Many codes comprise tens or hundreds of independently developed libraries. Downloading, building, and installing all of these programs is too great of a barrier for most scientists, so high-performance computing (HPC) facilities employ teams of experts to install and manage scientific software. Spack automates the build workflow without sacrificing software performance or flexibility. It reduces deployment time for large software stacks from weeks to hours, and it enables end users and developers to install software without the aid of specialized staff.

Spack consists of three key components: (1) a command-line tool, which allows users to build software packages on demand; (2) a repository of over 3,200 templated package recipes; and (3) a *specification language* with which users can customize builds.

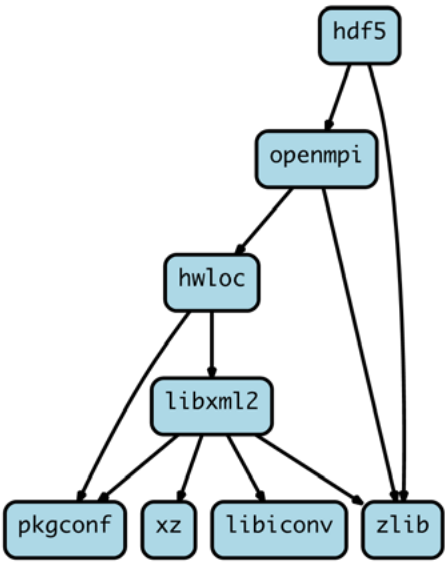


Figure 1: The hdf5 library, with 7 dependencies.

Spack's Command-Line Tool

To use Spack, simply download it from GitHub. At the command line, the user can run `spack list` to view and query the list of available packages. When they find the package they want (e.g., `hdf5`, a library for scientific data analysis), users can invoke `spack install hdf5` to begin the installation process. Spack downloads source code for `hdf5`'s seven dependency libraries (Figure 1), builds them, installs them, and then does the same for `hdf5` itself. Thus, the `hdf5` library is automatically built and optimized for the user's platform, and the user can write code that uses it.

To accomplish the same tasks manually, the user would need to read `hdf5`'s documentation, learn about its dependencies, find *their* websites, and download source code for all of them. The user would then manually configure, build, and install all of these dependencies along with the `hdf5` library. Each package may use a different build system, and each build system is likely to require its own package-specific parameters and options.

Worse, packages may require specific *versions* of their dependencies, or they may require dependencies to use build-time options to enable specific features. If a user installs the wrong version, they will discover after the fact that the package's dependents are incompatible with it, and they will have to rebuild. Likewise, if the package is configured in a way that conflicts with dependents' requirements, it will need to be rebuilt. Even if the build process is executed perfectly, HPC users expect to be able to use packages with many different compilers and MPI (message passing interface) implementations, so the whole process must be repeated for each unique compiler/MPI combination. Moreover, when developers build packages with many options by hand, they often forget exactly how different packages were configured. Without detailed record keeping, it can be extremely difficult to diagnose issues with the software, or to try new configurations systematically (e.g., when tuning for performance).

Building all configurations of even a small package like `hdf5`—with relatively few dependencies—can take a very long time. Larger packages like `rminer` (Figure 2) can have over 150 dependencies, and it is not reasonable to install a package like this without the on-demand automation that Spack provides via simple commands.

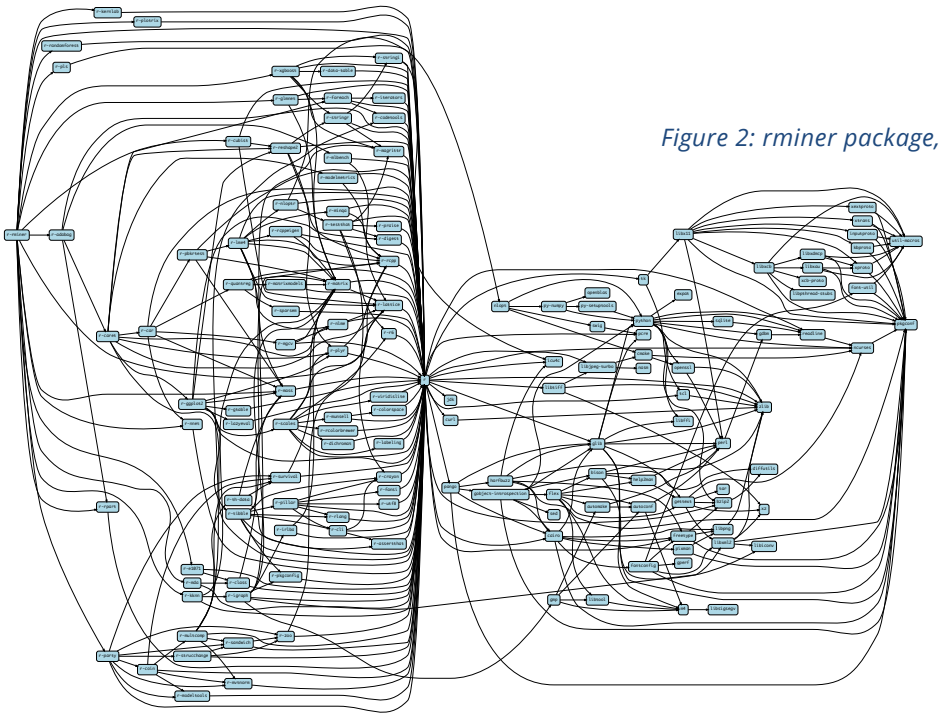


Figure 2: rminer package, with 150 dependencies.

Spack's Package Repository

Spack is able to build complex packages automatically thanks to a large community of contributors. At the time of this submission, Spack has over 400 contributors who add recipes—or *package files*—to its built-in package repository. The community has been able to grow so large because Spack makes packages easy to write. Package files are written in Python, which is extremely popular and widely known in the HPC community. On top of Python, Spack provides its own embedded domain-specific language (DSL), which is *templated* so that package scripts can support many configurations. This allows packages to be simple yet flexible enough for HPC users.

Figure 3 shows an example package file for a Lawrence Livermore National Laboratory (LLNL) deterministic transport mini-application called Kripke. There are only 18 lines of code in this package, but it allows Kripke to be built with different MPI versions, different compilers, and other parameters. The directives in this file tell Spack which versions of Kripke are available, which build options ("variants") are possible, and which dependencies (i.e., other packages) are required to build Kripke. The `cmake_args` and `install` methods contain parameterized build instructions that describe how to interact with the HPC build system and install the package when users request different configurations. Here, the user-specified values for the `openmp` and `mpi` variants are passed as parameters to Kripke's build system. While most Spack packages can be short like this one, logic in the `install` method can be arbitrarily complex if needed. This allows Spack to handle the many nonstandard package build systems in HPC.

```
class Kripke(CMakePackage):
    """Kripke is a simple, scalable, 3D Sn deterministic particle
       transport proxy/mini app.
    """
    homepage = "https://computing.llnl.gov/projects/co-design/kripke"
    url      = "https://computing.llnl.gov/projects/co-design/download/kripke-openmp-1.1.tar.gz"

    version('1.2.3', md5='485e6dcdf7cf5d76dca3dd8177b9f430')
    version('1.1',   md5='7fe6f2b26ed983a6ce5495ab701f85bf')

    variant('mpi',    default=True, description='Build with MPI.')
    variant('openmp', default=True, description='Build with OpenMP enabled.')

    depends_on('mpi', when='+mpi')
    depends_on('cmake@3.0:', type='build')

    def cmake_args(self):
        return [
            '-DENABLE_OPENMP=%d' % ('+openmp' in self.spec),
            '-DENABLE_MPI=%d' % ('+mpi' in self.spec),
        ]

    def install(self, spec, prefix):
        # Kripke does not provide install target, so we have to copy
        # things into place.
        mkdirp(prefix.bin)
        install('../spack-build/kripke', prefix.bin)
```

Figure 3: Package for LLNL's Kripke code.

Other package management tools do not have Spack's level of customization; they typically only allow a single configuration of each package—for instance, only the default configuration and default version. Spack's package templates specify the *possible* ways to build a package, and users build packages in the manner that suits their needs without needing to learn the configuration details of every package. When Spack installs packages, it still records detailed provenance about all configuration parameters, so that builds can be queried, diagnosed, and reproduced later, if needed.

Spack's Specification Syntax

While Spack package files are written in Python, users of the `spack` *command-line tool* do not need to understand Python or HPC build systems to leverage the power of Spack's package repository. For users, Spack provides an expressive specification language that allows users to instantiate package templates. The language is *telescoping*—that is, it allows users to opt for default installations in simple cases while also allowing them full control over the build process if they require more complicated configurations. The specification language is designed to be simple—it is not a full programming language, and while *packagers* will need some experience with Python, end users of the Spack tool do not need to know Python at all.

Figure 4 gives a series of examples. As described above, if a user wants to install the standard `hdf5` library, they simply run the command `spack install hdf5`. Spack will choose the latest versions of `hdf5` and its dependencies, and the user does not need expertise in these details. If the user wants a particular version of the `hdf5` library and/or the gcc compiler, they can indicate this in the command line—for instance, `spack install hdf5 %gcc@7.2.0`. Spack's syntax also accommodates other build options such as enabling additional features that are disabled by default (`+fortran`), specifying values for compiler flags (`cflags=`), and designating a target build architecture (`target=`). If a user wants to see a list of available versions and options for hdf5, they can simply run `spack info hdf5`—they do not need to be familiar with `hdf5`'s build system or to have any prior experience with the package.

Spack syntax	Meaning
`hdf5`	Install the `hdf5` package
`hdf5@1.10.5`	Install `hdf5` with specific version 1.10.5
`hdf5@1.10.5 %gcc&7.2.0`	Install `hdf5 1.10.5` using the gcc compiler version 7.2.0
`hdf5@1.10.5 %gcc&7.2.0 +fortran`	Install `hdf5 1.10.5` with `gcc 7.2.0` and Fortran support enabled
`hdf5@1.10.5 cflags="-03 -fast"`	Install `hdf5 1.10.5` with specific compiler flags
`hdf5@1.10.5 target=haswell`	Install `hdf5 1.10.5`, optimized for Intel Haswell chips
`hdf5@1.10.5 ^mpich@3.2 %gcc@7.2.0`	Install `hdf5 1.10.5`, and link with the `mpich` MPI implementation at version 3.2, built with `gcc 7.2.0`

Figure 4: Spack's package specification (spec) language.

Spack's syntax is expressive enough that these options can be applied to dependencies in addition to the root package. For example, many implementations of MPI exist, and Figure 1 shows that one of `hdf5`'s dependencies is `openmpi`. A user can easily choose to build `hdf5` with a *different* MPI implementation (e.g., mpich) by running `spack install hdf5 ^mpich`. The `^` character here can be read as "depends on". The user can specify the version, compiler, variants, and other options for `mpich` on the command line using the same syntax as for `hdf5`. Moreover, the user does not have to specify constraints for *all* of `hdf5`'s dependencies, only the ones they care about. Thus, while the user has *control* over the entire package specification, they do not need to be familiar with the software package's complete dependency graph to install it successfully. Users can limit their concern to the details they care about. Spack handles the rest.

Spack's specification language is useful not only for single commands; it also enables the configuration of large software stacks to be versioned and managed in special `spack.yaml` files. In the examples in Figure 4, each command installs a single package and its dependencies. Spack allows multiple specifications to be bundled together in a single `spack.yaml` file (Figure 5), which can be checked into version control so that developers can collaborate on common configurations. The entire contents of this file can be built and installed reproducibly with a single spack install command, and Spack ensures that each specification and its dependencies can coexist in the same environment. This feature is useful at large HPC centers, where thousands of packages may need to be installed at once.

```
spack:
  specs:
   - hdf5 @1.8.16
   - openmpi fabrics=libfabric
   - nalu
```

Figure 5: A `spack.yaml` file enables users to concisely express an entire software stack with specific configurations.

Such configuration flexibility is also useful for building containers. Container solutions like Singularity, Docker, and Charliecloud (winner of a 2018 R&D 100 Award) are becoming increasingly popular for bundling HPC applications. Using a single `spack.yaml` file and a single Dockerfile (Figure 6), a user can build a container image with many packages by writing only a few lines of code. Without Spack, the same scripts would require hundreds or thousands of lines of script code to build all of the components, making the container recipes hard to maintain. Spack simplifies the container workflow and allows developers to collaborate easily on a common software stack.

```
FROM spack/centos:7

WORKDIR /build
COPY spack.yaml .
RUN spack install
```

Figure 6: A simple Dockerfile that uses `spack.yaml` to install a large number of packages. To build such a stack manually would require hundreds of lines of script code to download and install each package.

Impact and Availability

From laptops to small clusters to the world's largest supercomputing sites, the need to rapidly build and deploy software stacks is widespread. Todd Gamblin created the first prototype of Spack to automate the many tedious software builds he and LLNL colleagues were forced to do manually. Development quickly became a grassroots effort as others began to use it. Today, Spack is widely available as open source software. Features and improvements are regularly being added to Spack by a broad community of contributors. The most recent release (version 0.12) has added key improvements that have led to broad adoption at top-tier HPC centers and influential HPC organizations.

Figure 7: ORNL's Summit supercomputer, ranked #1 on the Top500 list of the world's fastest supercomputers.

Spack allows software packages comprising hundreds of dependency libraries to be built and installed quickly and reliably, which has led to its adoption by a number of prominent code teams, supercomputing centers, and software development communities. For example, Oak Ridge National Laboratory (ORNL) replaced an existing deployment process with Spack for their #1-ranked Summit supercomputer (Figure 7). ORNL deploys over 1,300 software packages for users of the system, and building all of this software used to require roughly 2 weeks of work. With Spack, the process is now automated and can be completed in 12 hours. Summit's entire stack can now be redeployed overnight.

ARES is an LLNL multi-physics code, used in mission-critical inertial confinement fusion (ICF) simulations. ARES relies on a 46-package software stack (Figure 8). Spack has enabled the team to rapidly test this stack with new compilers and configurations in preparation for new HPC platforms and environments before they arrive. The developers now build and test 36 different configurations of ARES nightly. Before Spack became available, this volume and range of testing were not possible to execute automatically, and the development team waited to test new compilers until they were needed. Now, because of Spack, the additional testing is essentially free. The work of porting the code to a new machine, which used to take weeks, now only takes 3 hours thanks to Spack's automation and ARES's increased robustness from testing. The ARES lead build engineer stated, "It is inconceivable how we would handle the growing number of interdependencies between frequently updated library versions, GPU interfaces, and compiler versions without Spack."

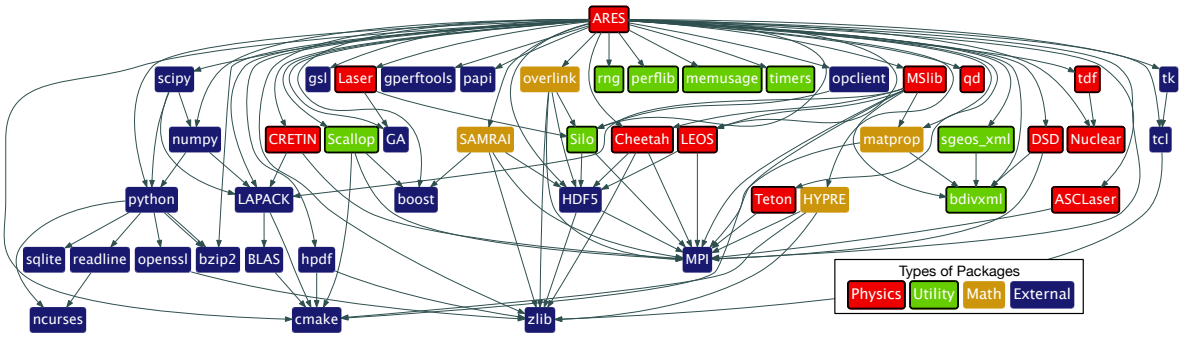


Figure 8: LLNL's ARES multi-physics code and its 46 dependency libraries (plotted using Spack).

Spack is now used for software deployment on 6 of the top 10 supercomputers in the world[1]. It has also been adopted and the standard deployment tool of several national-level projects. The U.S. Exascale Computing Project (ECP) is a $300M/year effort tasked with building a capable software stack for future exascale supercomputers. Within ECP, Spack is used to coordinate a hierarchy of software releases for multiple supercomputing platforms. ECP is a collaboration of over 1,000 researchers from 17 national laboratories, and ECP staff use Spack for both local and ECP-wide software releases. Spack will eventually be

1 According to the *Top500 list of world's fastest supercomputers at* https://top500.org.

used to deploy over 90 different software products within ECP to HPC centers across the U.S. Department of Energy (DOE), and it is critical to the success of the ECP core mission.

In Japan, Spack has been selected as the de-facto deployment tool for the upcoming Fugaku supercomputer, formerly known as Post-K. This nearly billion-dollar, national-level project will deliver the first supercomputer based on the new ARM scalable vector extensions (SVE) instruction set. Spack was chosen for this project due to the ease with which Fujitsu and RIKEN staff were able to use and adapt Spack for ARM platforms. Its templated package recipes made software deployment on this new architecture an extremely easy task compared to other systems that required package recipes to be entirely rewritten.

Even outside of HPC centers, Spack has had tremendous impact. The high-energy physics (HEP) community, based at Fermilab in the U.S. and CERN (the European Organization for Nuclear Research) in Switzerland, builds software infrastructure that is comparable in complexity to DOE's. In this community, Spack has been adopted as an end-to-end development tool. Developers use it locally to build their dependencies, and administrators use it project-wide to deploy software to HEP's worldwide network of compute clusters. The HEP community has been so pleased with Spack's capabilities that they have contributed significant features to the tool—they believe it is not only useful but also a worthy long-term investment.

These examples indicate an exciting trajectory. Since its initial open source release, Spack's capabilities, use cases, and community engagement have increased by orders of magnitude. In particular, the 2018 release (version 0.12; see Product Comparison below) enables reproducible builds with improved concretization and lockfiles, features more packages and configurations than ever before, and future-proofs Spack's growth via a permissive license—a significant effort that required buy-in from hundreds of contributors. These combined enhancements convinced leadership at ECP, Fujitsu, and RIKEN to adopt Spack for HPC software deployment.

Worldwide Adoption and Outreach

Spack's adoption is not confined to just the high end of the HPC market. Its design makes it usable by scientists on laptops, workstations, and small clusters in addition to the elite institutions of the world. Indeed, from-source builds are difficult to manage, but they have long been the standard way of distributing HPC software. Spack is the only tool that gives users the full flexibility and power of building by hand along with the automation required to make it quick and easy. Spack's specification syntax, coupled with templated packages, allows users to rapidly build a large set of package configurations, to quickly test and converge on the fastest configuration, and to build existing package recipes in new environments. Because of its ease of use, Spack's adoption has been widespread, and we have striven to lower the barriers to contribution and to encourage community participation in the project.

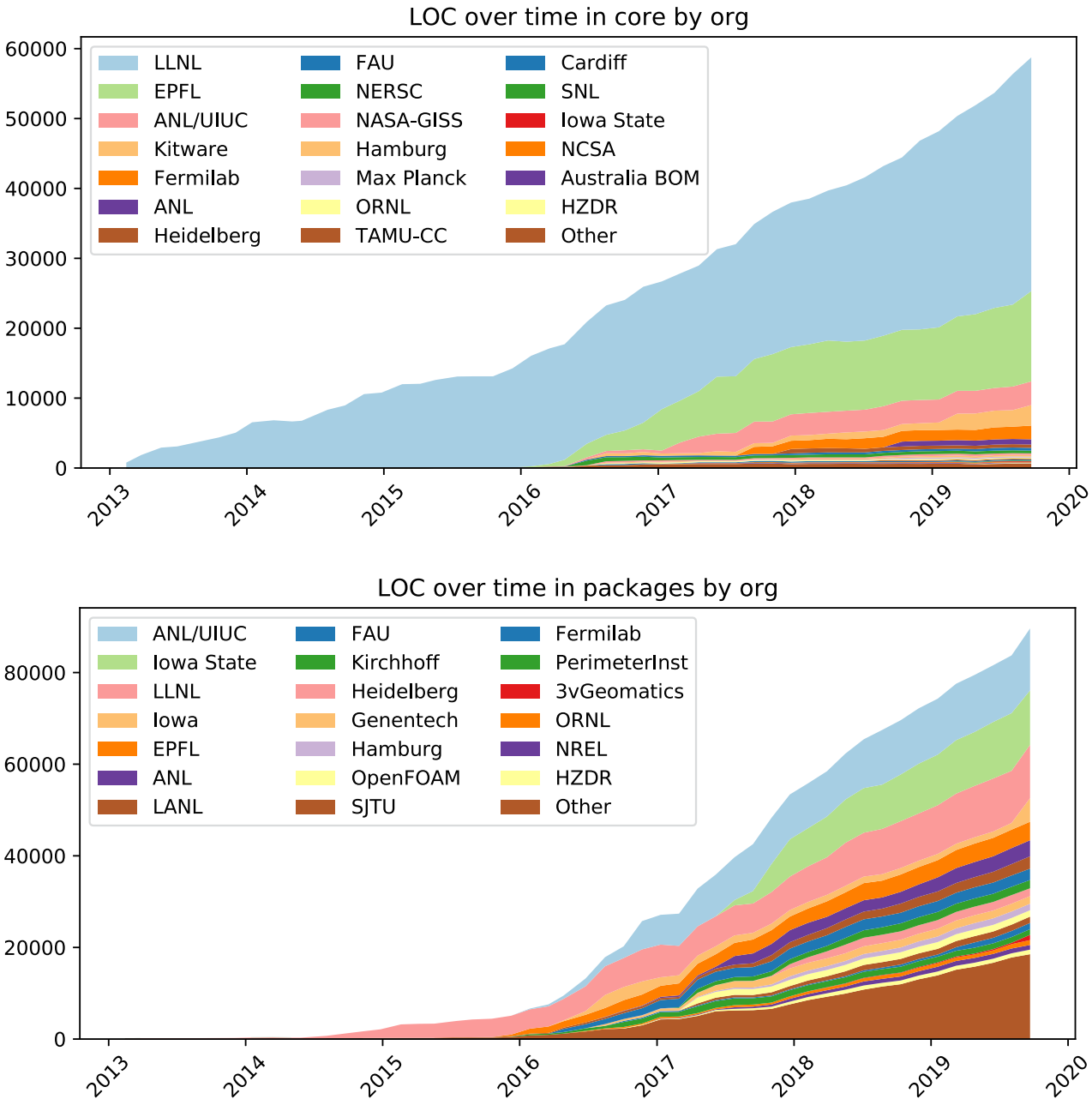


Figure 9: Contributions to Spack over time, by organization. Top: contributions to the Spack core tool; Bottom: contributions to Spack's package repository. (LOC = lines of code.)

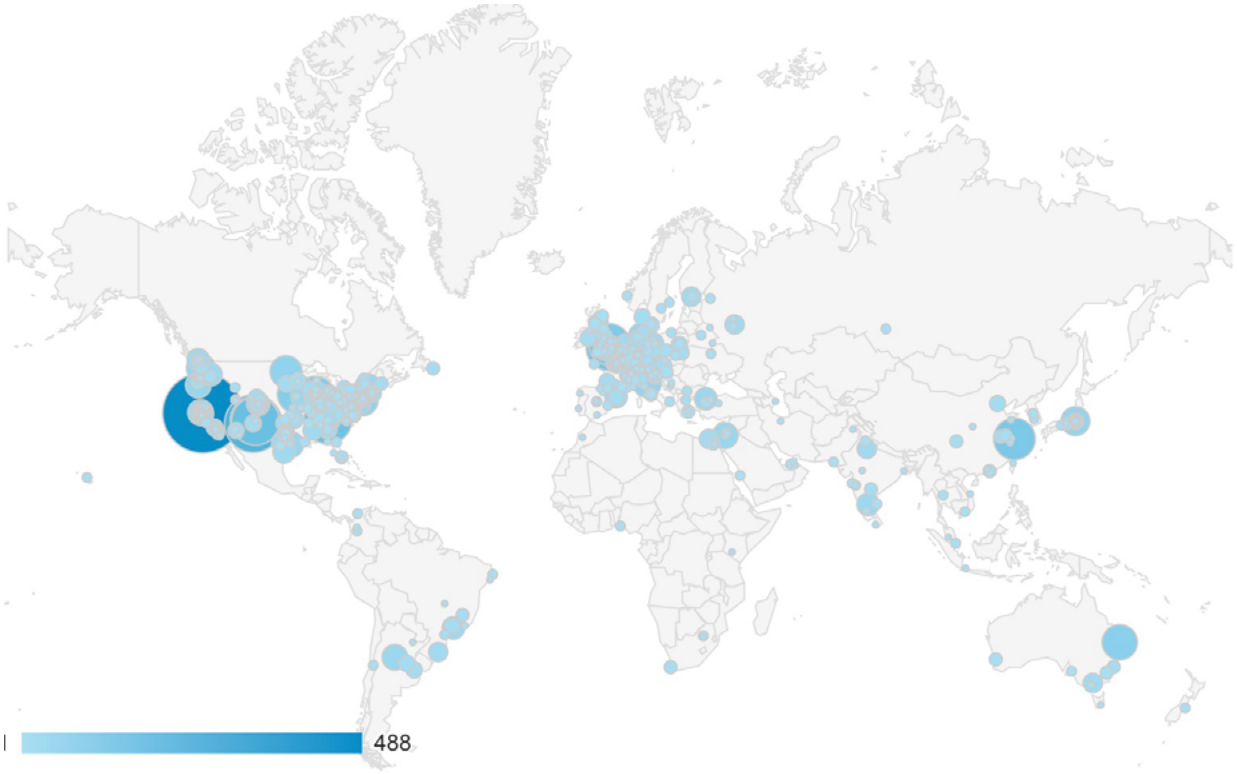


Figure 10: Unique users on Spack's documentation website in a single month. At the end of 2018, Spack's documentation was visited by over 1,100 users per month.

Figure 9 shows the total lines of code in the Spack tool and in its package repository, grouped by organization. Spack now has over 400 contributors from over 100 organizations. Though the project started at LLNL, the majority of contributions to Spack's packages now come from organizations including other laboratories, academia, and industry. LLNL still maintains the bulk of the core tool with a number of close collaborators. Spack's features, innovative design, and simple package DSL have enabled us to crowd-source package maintenance solutions and support this thriving community.

In addition to its core contributors, Spack has thousands of end users all over the world. In December 2018, over 1,100 users browsed Spack's documentation site. Figure 10 shows their locations around the globe. At the time of writing, this number had grown to over 2,000 monthly active users. Spack's worldwide traction is a testament to its usefulness within the HPC field and beyond. Spack has filled a longstanding gap in the scientific software ecosystem and saved users countless hours of tedious manual configuration and iteration.

Community success relies in a large part on user outreach and education, which has always been crucial to Spack's development. The leadership team travels to user sites around the world (such as to RIKEN,

Figure 11: LLNL's Spack developers met with the RIKEN team in Japan in April 2019.

shown in Figure 11), offering hands-on instruction and step-by-step tutorials. Spack workshops and "birds of a feather" gatherings are ubiquitous at several premier international HPC conferences, including Supercomputing and ISC High Performance (see the tutorials list under References below). A dedicated Twitter account (@spackpm) and a continually active Slack chat instance (spackpm.slack.com) provide additional ways to connect with the Spack community.

B. How does the product operate?

When a user invokes `spack install hdf5`, Spack takes a number of steps behind the scenes to ensure that `hdf5` is properly installed. Essentially, Spack must look at the user's request, determine *what* to build, and then build it. At first glance, this seems trivial, but the devil is in the details. User specifications are vague, in that there are *many* possible configurations of hdf5, and Spack must find a valid one. Moreover, Spack allows many combinations of packages to be built, but most operating systems (OS) and build tools are configured so that, by default, the *system* versions of packages are preferred over customized versions. These settings can easily creep into builds and cause incompatibilities, and they make it very difficult for one user to get the same results as another when installing a piece of software. Spack is designed to deliver a one–two punch: (1) concretization of specifications and (2) isolated, reproducible package builds.

Concretization

Earlier, we discussed Spack's two main user-facing pieces: the *specification* syntax, which allows users to express their requirements when they install packages, and *package files*, which provide templated recipes to build package specifications. We described how a user could quickly type a command using the specification syntax, and have it built using instructions from package files. We mentioned that users need only provide the details that they care about, and Spack would handle the rest. The heart of Spack and the key component that binds specifications and packages together is the *concretizer.* Put simply, the concretizer is an algorithm that converts *abstract* specifications from users into complete, *concrete* specifications that can be built.

The concretization process is shown in Figure 12. A user invokes `spack install hdf5@1.10.5 ^mpich@3.2`. The specification from the user is converted to a directed acyclic graph with a node for each of `hdf5`'s dependencies. Some nodes (hdf5 and mpich) have constraints. In this case, the constraints limit which versions the packages can be built with (1.10.5 and 3.2, respectively). These are the user's requirements from the input specification. Spack constructs a separate graph that encodes constraints from the package files. It then intersects these constraints package by package and checks each set of constraints for inconsistencies. Inconsistencies can arise if, for example, the user inadvertently requests two versions of the same package, or if a package requires (for compatibility reasons) a different version than the user requested. Likewise, if the package and the user specified different compilers or variants for particular packages, Spack will stop the build process and notify the user of the conflict.

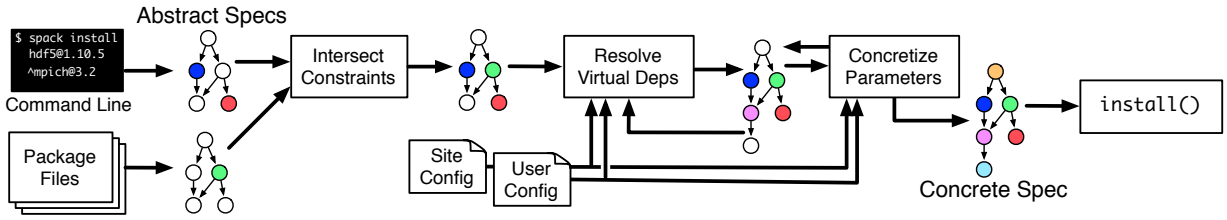


Figure 12: Spack's concretization process.

Assuming the intersection succeeds, Spack generates a single graph with the merged constraints of the user and the package files. Essentially, the user's request has now been combined with the collective knowledge of Spack's contributors. The next part of the concretization algorithm is iterative. If any node in the graph is a *virtual* dependency, Spack resolves it to a suitable provider of the virtual interface, either by creating a new node for the implementation, or by redirecting edges to an existing provider in the graph. When Spack has a choice of which provider, version, variant, or compiler to use, it consults user and site preferences to determine the "best" possible choice. This may introduce

new dependencies to the graph, so we repeat this process until the graph no longer changes. Once all parameters for all nodes in the graph have been set, we call the dependency graph *concrete*.

Concretization is Spack's way of filling in free variables in the package configuration. When we designed Spack, we found that users who built software devoted a large amount of code to searching for software and configuration information on the host system in order to fill in details like this. This complex code was error-prone and would often find subtly incompatible libraries or unusable versions of software installed on the system. Spack's concretization process takes over this task and ensures that all such decisions are made *before* the package starts building. This key design aspect of Spack enables us to rapidly grow our contributor base. Because of concretization, logic in Spack packages can be much simpler because it only *translates* a concrete specification to build instructions; complex configuration decisions are left to Spack.

The concretization process enables another of Spack's key benefits: reproducibility. As explained earlier, `spack.yaml` files (Figure 5) enable an entire software stack to be specified in a single file. When the specifications in these files are concretized, Spack generates an additional `spack.lock` file that records the *output* of the concretization algorithm. This contains specific version and platform configuration information for all specifications in the software stack. A user can easily leverage this `spack.lock` file to reproduce a software stack *exactly* as another user built it. The `spack.lock` file effectively "locks" the versions, build options, and optimization choices produced when the first user ran the concretization algorithm.

Isolated, Reproducible Builds

Spack's internal graph model lets it represent arbitrary software configurations, and concretization allows it to generate a complete configuration from a partial specification provided by the user. Spack also provides the infrastructure needed to *build* these arbitrary configurations. This is not trivial: Most build systems and OS provide defaults that steer compilers and other tools towards the default versions of libraries and programs used in a build. Because of this, when users build software by hand, it is easy to accidentally use incompatible versions of libraries and tools. Spack takes a number of measures—including RPATH linking and compiler wrappers—to ensure that the build environment is clean, so that two users building the same concrete specification in different environments are sure to get the same result.

RPATH Linking

When one library needs to call code from one of its dependencies, dependency libraries are found by looking in default locations set by either the OS or the user. These settings are designed for systems

where only *one* version of a given library will be installed, and they can result in incorrect code in HPC environments, where the same library may need to be built in different, incompatible ways for different applications. For example, a parallel mesh partitioner may use a parallel version of the `hdf5` library, while a different application that uses the mesh may use a sequential version of `hdf5`. These two `hdf5` versions are incompatible, so one code cannot use the `hdf5` version preferred by the other. This means we cannot run the mesh partitioner in the same environment with the application, so they cannot be used together in a common workflow.

To avoid this conflict, Spack ensures that all installed libraries "know" which dependencies they were built with using a technique called RPATH linking. An RPATH is a special location embedded in a library or executable code that tells the OS where to look for dependencies. Each package installed with Spack has RPATHs for all of its dependencies, so when it runs, it will *always* know where to find the right libraries, and users cannot accidentally run Spack-built programs with conflicting code. Whether managing tens or hundreds of dependencies, this failsafe solution is extremely useful, as the OS does not know about user-installed dependencies, and users cannot be expected to remember and configure the locations of hundreds of library versions.

Compiler Wrappers

To ensure that RPATHs are set on executables and libraries when they are created, and to enable different compilers to be easily swapped into a build, Spack uses a special *wrapper script* in its build environment. This feature prevents a number of errors. First, many default environment settings from the OS and the user can creep into a build environment. Certain variables like `LD_LIBRARY_PATH`, `CFLAGS`, and `CC`, which are commonly set by users, can cause a build to inadvertently change its behavior by using the wrong libraries, the wrong compiler flags, or the wrong compiler. Spack clears these and other variables from the user environment before it builds, and in place of the real compilers, it injects its own wrapper scripts. When the build calls the wrapper script as though it were the real compiler, Spack adds explicit search paths and RPATHs for its *own* libraries to the command line before calling the real compiler with the full set of arguments. This process overrides any environment settings and ensures that Spack packages are built as intended. It also removes part of the burden of configuration from package authors, as they can leave the configuration of most building and linking options to the compiler wrappers. Users do not need to write code to add extra arguments to compilers because Spack has already done so.

In addition to helping isolate builds, Spack's compiler wrappers allow the user to inject compiler optimization flags into its builds. This means users can rapidly iterate on optimization settings for their software stack as well as on other configuration options.

C. Product Comparison

There are many software package management tools on the market, both in and outside the scientific computing space. Package management systems have existed since the late 1990s, when products like RPM and APT were created to manage dependencies among binary packages in Linux distributions. These and other systems are present in nearly all Linux distributions to this day. While revolutionary at the time, these package managers assume that users have root privileges on the system; however, users are usually only allowed unprivileged user accounts on nearly all high-performance clusters. This alone makes these tools unusable in HPC environments. Spack can operate in a completely unprivileged environment. System package managers also assume that software is only installed in a single location on the system, which prevents them from installing multiple configurations of the same package. Further, because they manage only binary packages, the software distributed with these systems is built without architecture-specific optimizations in order to be as portable as possible. This prevents them distributing software that takes advantage of high-performance hardware, like the new hybrid architectures found at many top-tier supercomputing centers.

Only a few package managers seriously target the scientific computing space: Gentoo, Nix/Guix (two very similar systems), Conda, EasyBuild, and Spack. Spack's key advantage over all of these tools is its flexibility and ability to build arbitrary configurations from scratch. This allows users to build any packages they like, at any version, and to optimize these packages for the machine. Users do not need to know any programming to do this; they need only download the Spack tool and learn its specification syntax. Spack's flexibility also allows it to target a much broader set of users than do other tools. It is usable by non-programmers, developers, and administrators, while these other tools have limitations that limit them to one or another of these groups.

Gentoo is derived from the traditional system package managers described above, but additionally it allows users to build from source to optimize for specific hardware. Gentoo's sub-project Gentoo Prefix can be run in unprivileged mode. However, Gentoo Prefix does not support true multi-version installation, as it limits software installation to a single location (the "prefix"). Gentoo Prefix is entirely self-contained, which means that creating a Gentoo environment on an HPC system can take 12 hours or more. To support multi-version installation, a user would need to install several such environments, which is not tractable and wastes space. In contrast, a user can install their first package within minutes of downloading Spack.

Nix and Guix are so-called "functional" package managers, allowing packages to be installed in arbitrarily many configurations. They use similar directory isolation schemes to Spack, but both tools require root access to the machine to build packages. Further, they do not support a templated package syntax or anything like Spack's specification language. Users must learn functional programming languages like Nix's custom expression language and Guile, which are not widely known or used in the HPC community

or the general programming community, and each "package file" in these systems corresponds to a single configuration. Building new versions, changing configuration options, or swapping in compilers is not feasible. Further, these tools do not integrate well with the optimized MPI or compiler packages on HPC systems. Adapting to these systems would require rewriting packages, while Spack can swap in a new compiler or configuration easily due to its templated packages.

Conda is a binary package installation system rapidly gaining traction in the scientific computing community. It targets desktop scientific environments, *mostly* in the Python ecosystem. While Conda packages a large range of scientific software, it does not build packages from source. Packagers must build Conda packages separately from the tool itself. They can do this by submitting to a build farm in the cloud, but this makes Conda ill-fitted for developers, who need to manage dependency versions and configurations at very fine granularity. It also makes it difficult for Conda packages to be optimized for the machines they run on. Conda's mainline packages include limited machine-specific optimization, and special "channels" are available with GPU versions of some packages. However, making packages from different Conda channels work together is difficult because the binaries are built in incompatible ways. Porting to new types of machines with Conda is difficult because Conda packages must be built with an entirely separate tool. Further, Conda does not provide any integration with the high-performance MPI implementations used on HPC systems, so its packages typically do not use the resources available on HPC machines and, therefore, perform poorly.

Spack's main competitor in the HPC space is EasyBuild. EasyBuild is a software management system designed for HPC administrators to simplify software deployment. However, EasyBuild does not offer the flexibility of package installation that Spack does. Like Nix and Guix, each EasyBuild package configuration requires a configuration file, and no templating feature exists. EasyBuild fixes versions for its dependencies, so if users want to generate new versions of packages on demand, they must edit not just one but potentially tens or hundreds of configuration files to change dependency versions across the whole software stack. Indeed, EasyBuild supports around 1,700 software packages (compared to Spack's 3,200), but it requires over 7,000 configuration files to support a more limited number of configurations of these packages.

Using different compilers and MPI versions within EasyBuild also requires generating a new, full stack of configurations. Essentially, EasyBuild makes the *users* perform the concretization process that Spack has automated. EasyBuild has only limited support for external packages; it allows externals to be included in builds if they are loadable via an *environment module*, but Spack allows essentially any external package to be integrated with builds, even if no module is available. EasyBuild has managed to make inroads among HPC administrators, but it has not appealed to developers and end users the way Spack has.
Even among administrators, EasyBuild is typically used among small cluster administrators who are

happy with settings close to the default configuration. Spack has made inroads both with small cluster administrators and among developers and administrators of high-end machines, where quickly customizing the stack is necessary and highly valued.

In addition to the features already mentioned, Spack is designed to be easy to *install*. Because Spack is often the first tool that users download in order to install other packages, it is important that Spack itself have very few external dependencies. We have therefore gone to great lengths to include all of *Spack's* needed dependencies in the Spack distribution. All the user needs to do is clone a single repository and have Python installed (along with some basic Unix utilities provided in all major distributions), and Spack is ready to run immediately. It does not require compilation, nor does it rely on environment modules to do its builds. EasyBuild, on the other hand, requires that the user have the environment module system and several other dependencies installed in advance. These requirements are common in HPC environments but make installation of EasyBuild difficult on laptops or other workstations. Further, EasyBuild requires that the user download not one but *four* distinct repositories and set numerous environment variables before it can be used.

Spack is often compared to the environment module system, which allows administrators to easily expose a large number of package configurations to users. Environment modules are popular on HPC systems because they let users easily load different versions of packages into their environment. However, environment modules are not a package management system—they only manage the user's environment *after* packages are installed. There is no build support, common naming scheme, or central repository of environment modules; facilities have historically written their own modules by hand after installing packages. Spack and EasyBuild both support automatically generating module files for all installed packages, and eliminate the need for facilities to maintain these files manually.

Among competing package managers, Spack is the only tool that offers reproducible lock files (`spack.lock`, Figure 6). This feature is becoming increasingly popular among language-specific package managers (e.g., npm, cargo, bundler, pipenv), which are geared toward developers. These tools deal only with locking package *versions* and only within a single language's software ecosystem. No existing tool provides this configuration-locking functionality for multiple compilers, multiple languages, and multiple build configurations like Spack. Moreover, no other tool provides this functionality for C, C++, or Fortran—the key languages used in high-performance programming. These languages have many different implementations and no standard build system, yet Spack provides a common configuration language for all of them and is the first system to allow builds to be reproduced with a workflow as simple as those for language-specific tools.

Finally, Spack's license is a major advantage over many of its competitors. While all of the systems described here are free and open source software, only Spack and Conda use *permissive* licenses that

permit nearly unrestricted use of the software. Spack is dual licensed under either the Apache-2.0 *or* MIT license. RPM, APT, Gentoo, Guix, and EasyBuild all use the GNU General Public License (GPL), which means that organizations that build on them *must* release the source code for any software that uses the package manager. Nix uses the Lesser GPL (LGPL), which requires that organizations release the source code only for changes to Nix itself. As a result, many companies are wary of integrating these tools into their software stacks, as license compliance can be murky when integrating GPL-licensed code into proprietary systems. Spack is licensed to be easily used by as broad an audience as possible. This was a strategic choice; we aim to be vendor-friendly so that Spack can be easily integrated into high-end HPC systems. We believe our licensing strategy enables us to grow Spack's contributor base as rapidly as possible, and this will ultimately result in better open source software regardless of whether some contributors choose to build proprietary solutions on top of Spack.

C. Competitors

Product	Manufacturer	License
Spack	Spack Project	Apache 2.0 or MIT
RPM	Red Hat	GPL-2.0
APT	Debian Project	GPL-2.0
Gentoo	Gentoo Foundation	GPL-2.0
Nix	Nix Project	LGPL-2.1
Guix	GNU Project	GPL-3.0
Conda	Anaconda, Inc.	BSD-3-clause
EasyBuild	HPC University of Ghent	GPL-2.0
Lmod	Texas Advanced Computing Center (TACC)	MIT
Environment Modules	INRIA Bordeaux	LGPL-2.1

D. Comparison summary

Feature	Spack	RPM/APT	Gentoo	Nix/Guix	Conda	EasyBuild
Users and systems						
Target audience	Users, developers, admins	Admins, developers	Admins, developers	Admins, developers	Users	Admins
Platform support	Linux, MacOS, Cray	Linux	Linux	Linux, MacOS	Linux, MacOS, Windows	Linux, Cray
Usable without root privilege	✓	✗	✓	✗	✓	✓
Installation requirements	Python	Included with Linux distribution	Many-hour build	Download binary	Download binary	Extensive, difficult to install
Multi-configuration installation	✓	✗	✗	✓	✗	✓
Build process and configuration						
Command-line build specification syntax	✓	✗	✗	✗	✗	✗
Reproducible lockfiles	✓	✗	✗	✗	✗	✗
Build from source	✓	✗	✓	✓	✗	✓
Optimized builds	✓	✗	✓	✓	Limited	✓
RPATH linking	✓	✗	✗	✓	✗	Optional
Compiler wrappers inject optimization flags	✓	✗	✗	✗	✗	✗
Easily swap compilers with wrappers	✓	✗	✗	✗	✗	✗
Flexible concretization	✓	✗	✗	✗	✗	✗
Optional dependencies	✓	✗	✓	✗	✗	✗
Ease of changing stack	✓ Single command	✗ Re-write packages	✗ Re-write packages	✗ Re-write packages	✗ Re-write packages	✗ Re-write packages
Package repository						
Number of packages	3,200+	10,000+	10,000+	10,000+	5,000+	1,700+
Package recipe language	Python DSL	RPM SPEC format	RPM-like	Nix expressions Guile (Guix)	RPM-like	Python
Templated packages	✓	✗	✗	✗	✗	✗
Binary packages	✓	✓	✓	✓	✓	✗
Relocatable binaries	✓	✗	✗	✗	✓	✗
Integrate external system packages	✓	✗	✗	✗	✗	Via Modules only

E. Limitations

One benefit of Spack's large and varied community is that opportunities always arise to learn about and address new use cases. Having a global user base also means Spack's limitations come to the foreground quickly, so we are able to prioritize and test new features as development evolves. Currently Spack has one primary limitation: The concretizing feature uses heuristics instead of a full backtracking solve for NP-hardness—a class of nondeterministic polynomial problems resolved by Spack's decision algorithms. In other words, dynamic dependency resolution can occasionally fail to find a solution when one is possible, and sometimes Spack may rebuild new versions of packages when it does not need to. Fortunately, in most cases, the current concretization algorithm is sufficient, and we are reworking the algorithm for our next release. Spack's efficient handling of the overall work of building and installing a complex scientific software stack saves users many hours of tedious labor, and the user's effort to overcome this limitation manually is minor. When the concretizer fails to find a solution, a user can simply supply more constraints in Spack's specification language to help it find the desired configuration.

4. SUMMARY

Building complex software is a challenge even for seasoned professionals. Next-generation HPC architectures will only increase the complexity and dependencies of scientific software. This in turn will increase demands on software deployment time and configurations. As a software package manager for high-performance scientific computing applications, Spack is easy, versatile, and scalable. Spack's main goal is to simplify the process of managing scientific software for administrators, developers, and end users alike. To do this, Spack automates the build process for scientific software packages and allows users to easily download, install, and manage packages with hundreds of dependencies. It automates the build workflow so that users can focus on their scientific work and speeds up installation and manages multiple configurations efficiently. Spack is ready to run immediately after download, and users benefit from the combined knowledge of the Spack community. Spack's flexibility and large community have made achievements at major supercomputing centers possible. Spack originated at LLNL, which continues to support it. Once Spack became available as open source, other organizations have contributed to its ongoing development.

5. CONTACT INFORMATION

Principal investigator from each of the submitting organizations:

Name: Todd Gamblin
Title: Computer scientist, Spack PI
Organization: Lawrence Livermore National Laboratory
Phone: 925-422-9319
Email: gamblin2@llnl.gov

Name: Adam Stewart
Title: PhD student
Organization: Argonne National Laboratory/ University of Illinois at Urbana–Champaign
Phone: 607-972-5364
Email: adamjs5@illinois.edu

Name: Massimiliano Culpo
Title: Software developer
Organization: École Polytechnique Fédérale de Lausanne
Phone: 41-21-69-31994
Email: massimiliano.culpo@epfl.ch

Name: Patrick Gartung
Title: Programmer analyst
Organization: Fermi National Accelerator Laboratory
Phone: 630-840-3832
Email: gartung@fnal.gov

Name: Levi Baber
Title: Research IT director
Organization: Iowa State University
Phone: 515-294-2907
Email: baber@iastate.edu

Name: Aashish Chaudhury
Title: Technical leader
Organization: Kitware, Inc.
Email: aashish.chaudhary@kitware.com

Name: Elizabeth Fischer
Title: Associate research scientist
Organization: NASA Goddard Institute for Space Studies, Center for Climate Systems Research/ Columbia University
Phone: 212-678-5581
Email: elizabeth.fischer@columbia.edu

Name: Mario Melara
Title: Computer systems engineer
Organization: National Energy Research Scientific Computing Center
Phone: 925-858-0436
Email: mamelara@lbl.gov

Name: Erik Schnetter
Title: Research technologies group lead
Organization: Perimeter Institute
Phone: 519-569-7600 x7032
Email: eschnetter@perimeterinstitute.ca

Name: George Hartzell
Title: Bioinformatics, software engineering, and dev-ops
Organization: Independent consultant
Email: hartzell@alerce.com

Name: Michael Kuhn
Title: Scientific computing staff
Organization: University of Hamburg
Phone: 49-40-460094-108
Email: michael.kuhn@informatik.uni-hamburg.de

Name: Glenn Johnson
Title: HPC architect
Organization: University of Iowa
Email: glenn-johnson@uiowa.edu

Media and public relations person who will interact with R&D's editors regarding entry material:

Name: Connie Pitcock
Title: Business Development and Marketing Associate
Organization: Lawrence Livermore National Laboratory
Phone: 925-422-1072
Email: pitcock1@llnl.gov

Person who will handle banquet arrangements for winners:

Name: Todd Gamblin
Title: Computer scientist, Spack PI
Organization: Lawrence Livermore National Laboratory
Phone: 925-422-9319
Email: gamblin2@llnl.gov

6. AFFIRMATION

By submitting this entry to R&D Magazine you affirm that all information submitted as a part of, or supplemental to, this entry is a fair and accurate representation of this product. You affirm that you have read the instructions and entry notes and agree to the rules specified in those sections. For more information, please call 973-920-7032 or email rdeditors@advantagemedia.com

7. REFERENCES

Peer-Reviewed Papers

1. Mario Melara, Todd Gamblin, Gregory Becker, Robert French, Matt Belhorn, Kelly Thompson, Peter Scheibel, and Rebecca Hartman-Baker. Using Spack to Manage Software on Cray Supercomputers. In Cray Users Group (CUG 2017), Seattle, WA, May 7–11, 2017.
2. Gregory Becker, Peter Scheibel, Matthew P. LeGendre, and Todd 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.
3. Todd Gamblin, Matthew P. LeGendre, Michael R. Collette, Gregory L. Lee, Adam Moody, Bronis R. de Supinski, and W. Scott Futral. The Spack Package Manager: Bringing Order to HPC Software Chaos. In Supercomputing 2015 (SC15), Austin, TX, November 15–20, 2015. (22% acceptance rate.)

Presentations and Meetings

4. Todd Gamblin, Gregory Becker, Michael Kuhn, and Massimiliano Culpo. Spack Community BoF. In ISC High Performance 2019, Frankfurt, Germany, June 18, 2019.
5. Massimiliano Culpo. Spack: A Package Manager for Scientific Software. In EasyBuild User Meeting, Ghent, Belgium, February 1, 2019.
6. Todd Gamblin, Gregory Becker, Matthew P. LeGendre, and Peter Scheibel. Spack Roundtable Discussion. In Exascale Computing Project 3rd Annual Meeting, Houston, TX, January 16, 2019.
7. Todd Gamblin, Adam J. Stewart, Johannes Albert von der Gonna, Marc Perache, Matt Belhorn, and Veronica Vergara. Spack Community BoF. In Supercomputing 2018 (SC18), Dallas, TX, November 13, 2018.
8. Todd Gamblin. Decluttering HPC Software Chaos with SPACK. In SIAM Conference on Parallel Processing for Scientific Computing (PP18), Minisymposium on Productive Programming using Parallel Models, Tools and Scientific Workflows, Tokyo, Japan, March 7, 2018.
9. Todd Gamblin, William Scullin, Matt Belhorn, Mario Melara, and Gerald Ragghianti. Spack State of the Union. In Exascale Computing Project 2nd Annual Meeting, Knoxville, TN, February 6–8, 2018.
10. Todd Gamblin. Binary Packaging for HPC with Spack. In Free and Open source Software Developers' European Meeting (FOSDEM18), Brussels, Belgium, February 4, 2018. LLNL-PRES-745747.
11. Todd Gamblin. How Compilers Affect Dependency Resolution in Spack. In Free and Open source Software Developers' European Meeting (FOSDEM18), Brussels, Belgium, February 3, 2018. LLNL-PRES-745770.
12. Todd Gamblin. Recent Developments in Spack. In EasyBuild User Meeting, Amsterdam, The Netherlands, January 30, 2018.
13. Todd Gamblin. Tutorial: Managing HPC Software Complexity with Spack. In HPC Knowledge Meeting (HPCKP17), San Sebastian, Spain, June 16, 2017. 2 hours.
14. Massimiliano Culpo. Spack: A Package Manager for Supercomputers, Linux, and MacOS. In HPC Advisory Council Swiss Conference, Lugano, Switzerland, April 10–12, 2017.

Tutorials

15. Todd Gamblin, Gregory Becker, Massimiliano Culpo, Mario Melara, Peter Scheibel, and Adam J. Stewart. Managing HPC Software Complexity with Spack. In Supercomputing 2019 (SC19), Denver, CO, November 18, 2019. Half day (to appear).
16. Levi Baber, Adam J. Stewart, Gregory Becker, and Todd Gamblin. Managing HPC Software Complexity with Spack. In Practice and Experience in Advanced Research Computing (PEARC19), Chicago, IL, July 31, 2019. Half day (to appear).
17. Todd Gamblin and Gregory Becker. Spack Tutorial. In 1st Workshop on NSF and DOE High Performance Computing Tools, Eugene, OR, July 10, 2019. Full day (to appear).
18. Todd Gamblin, Gregory Becker, Michael Kuhn, and Massimiliano Culpo. Managing HPC Software Complexity with Spack. In ISC High Performance 2019, Frankfurt, Germany, June 16, 2019. Half day.
19. Gregory Becker. Managing HPC Software Complexity with Spack. In HPC Day, Frankfurt, Germany, June 13, 2019. Goethe-Universitat. Half day.
20. Todd Gamblin, Gregory Becker, Peter Scheibel, Matt Legendre, and Mario Melara. Managing HPC Software Complexity with Spack. In Exascale Computing Project 3rd Annual Meeting, Houston, TX, January 14, 2019. Full day.
21. Todd Gamblin, Gregory Becker, Massimiliano Culpo, Peter Scheibel, Matt Legendre, Mario Melara, and Adam J. Stewart. Managing HPC Software Complexity with Spack. In Supercomputing 2018 (SC18), Dallas, TX, November 12, 2018. Full day.
22. Todd Gamblin, Gregory Becker, Peter Scheibel, Matt Legendre, and Mario Melara. Managing HPC Software Complexity with Spack. In Exascale Computing Project 2nd Annual Meeting, Knoxville, TN, February 6–8, 2018. Half day.
23. Todd Gamblin, Gregory Becker, Massimiliano Culpo, Gregory L. Lee, Matt Legendre, Mario Melara, and Adam J. Stewart. Managing HPC Software Complexity with Spack. In Supercomputing 2017 (SC17), Salt Lake City, UT, November 13, 2017. Full day.
24. Gregory Becker, Matt Legendre, and Todd Gamblin. Spack for HPC. Livermore Computing, Lawrence Livermore National Laboratory, Livermore, CA, April 6, 2017. Half day.
25. Todd Gamblin, Massimiliano Culpo, Gregory Becker, Matt Legendre, Greg Lee, Elizabeth Fischer, and Benedikt Hegner. Managing HPC Software Complexity with Spack. In Supercomputing 2016 (SC16), Salt Lake City, UT, November 13, 2016. Half day.

8. APPENDIX

Support letters (separate files):

- Exascale Computing Project
- Fermi National Accelerator Laboratory
- Fujitsu
- Los Alamos National Laboratory
- Oak Ridge Leadership Computing Facility
- RIKEN Center for Computational Science

Additional supporting information:

- Spack website: https://spack.io
- Spack video: https://youtu.be/D0p5xpsboK4
- Spack code repository: https://github.com/spack
- Spack user documentation (including tutorial): https://spack.readthedocs.io/
- Spack on Twitter: https://twitter.com/spackpm

Date: April 22, 2019

James F. Amundson
Head, Scientific Computing Division

Scientific Computing Division
P.O. Box 500
Kirk Road and Pine Street
Batavia, Illinois 60510-5011 USA
Office: 630.840.2430
Mobile: 630.488.6910
amundson@fnal.gov

Subject: Spack for the R&D 100 Awards

To whom it may concern,

I am writing to express my wholehearted support for the Spack entry in the R&D 100 Awards. I am currently the head of the Scientific Computing Division at Fermilab, the U.S. DOE laboratory for accelerators and high energy physics (HEP). Here at the lab we are both preparing for the next generation of neutrino experiments with the DUNE detector and Long-Baseline Neutrino Facility and, at the same time, preparing to support the CMS experiment at CERN's High Luminosity Large Hadron Collider. Between the two experiments we will be storing multiple exabytes of data and writing millions of lines of code to process that data.

Spack is providing the backbone of the software infrastructure we are building up to support these experiments. Given my field's tendency to write its own software to solve every problem, just the fact that we are using any infrastructure from the broader scientific community is amazing in itself. It is only because Spack is an extraordinarily complete solution to complicated packaging problems that we have agreed to adopt this tool.

Particle physicists have a tendency to create very complicated sets of interdependent software packages with strict versioning requirements. They also like to be able to keep multiple versions of the packages around simultaneously. Spack is the first tool we've used that manages to tame the complexity we have created while allowing us to tap into the expertise of the broader scientific computing community. Spack's technical achievement is that it solves the difficult technical problems as a part of its core design. Equally impressive is the fact that the Spack team has managed to build a community that has encouraged cooperation among notoriously individualistic computational science people.

I find Spack to be highly worthy of the R&D 100 Award, and I hope you will, too.

Sincerely,

James F. Amundson
Head, Scientific Computing Division
Office of the CIO

Kouichi Hirai
Vice President, Software Development Division
Next Generation Technical Computing Unit
Fujitsu Limited
Tokyo, Japan

To the R&D 100 Evaluation Committee:

I am writing to endorse Spack for a 2019 R&D 100 award. I am Vice President of the Software Development Division in the Next Generation Technical Computing Unit at Fujitsu Limited—Japan's leading information and communication technology company. My division develops system software for supercomputers. Recently we decided to advise our customers to use Spack when deploying open source scientific software on their supercomputing systems.

Fujitsu provides Arm processor–based supercomputing systems to our customers, including the Japanese research institute RIKEN, whose upcoming Post-K system (the successor to the K supercomputer) we have designed. Many of our customers' users rely on open source software (OSS) to conduct research and analyze data. However, most OSS is built and tested primarily for x86 architectures. To best serve our customers and to help their users achieve their goals more efficiently, it is important for our business to be able to run OSS on both x86- and Arm-based systems.

Adapting OSS to an Arm-based system requires a large amount of effort, so it is essential to have an efficient mechanism to share knowledge regarding OSS builds. We carefully compared open source package management tools, and we were convinced that Spack was the best solution for our purposes. With Spack we can easily share OSS package recipes and support complex build systems. Spack is fully extensible, and its community has a high degree of participation and continuously evolves Spack's capabilities. We will continue to contribute to the Spack community and encourage our customers to use Spack with confidence.

Sincerely,

Dr. Gabriel Rockefeller
Scientist and Team Co-Lead
Lagrangian Applications Project
Los Alamos National Laboratory
Los Alamos, New Mexico, USA

To the R&D 100 Evaluation Committee:

I am writing to endorse Spack for a 2019 R&D 100 award. I co-lead the Lagrangian Applications Project at Los Alamos National Laboratory, and my team is tasked with the development of a suite of multiphysics codes used at high performance computing (HPC) sites across the National Nuclear Security Administration (NNSA) laboratories. I primarily work on the FLAG hydrodynamics code. My team and our collaborators have adopted Spack to deploy our codes, and it has dramatically improved our day-to-day workflow. We began using Spack to manage just a few libraries, and it has rapidly become the standard deployment tool for all of our HPC software. Spack allows us to leverage efforts across projects where work was previously duplicated, and it has enabled us to replace error-prone processes with repeatable, automated workflows.

Building and linking HPC codes like ours can be extremely challenging, as teams depend on many common libraries but require them to be configured in slightly different ways. Before we integrated Spack into our workflow, we attempted to share library installations among projects on our HPC systems, but the process was unreliable and error-prone. Because there was no way to understand the full configuration of installed libraries (e.g., whether particular features of a library were enabled or disabled, and which versions of lower-level dependencies had been chosen), misconfigurations could happen easily. For example, in one instance, a code was linked with multiple incompatible versions of the *same* library, which can lead to nondeterministic behavior. Concern over issues like this led many teams to duplicate effort and rebuild their own library versions to ensure consistency. It was clear we needed a better system to ensure reliable, correct execution.

We were inspired to adopt Spack during a presentation at the Department of Energy's Workshop on Exascale Technologies. Spack allows us to manage shared dependency installations effectively. Its build model ensures that linked libraries are built consistently and that only one version of any library can ever be included in a build. Other tools lack the expressiveness and ability to handle the subtle complexities of HPC packages. Spack stores more metadata about packages, so subsequent builds *know* whether they can reuse existing installations. Additionally, Spack's custom package language allows builds to be described in a general, cross-platform way, so we can easily leverage work on package recipes across code teams.

Today, we use Spack to manage all 29 of FLAG's dependencies, which we can share with collaborators on other code teams. Some of our dependencies are open source libraries for which we no longer need to create recipes ourselves because we rely on contributions by LLNL and others to the mainline Spack repository. Likewise, we have contributed recipes for some

LANL software to Spack for the benefit of other NNSA laboratories. Because Spack's recipe format is well defined and expressive, we can tell new teams who wish to integrate their libraries with our code *exactly* what to do: If they simply write a Spack recipe and tag their releases in a predefined way, we can easily bring their code into our stack without ever having to learn to build it ourselves. Spack has enabled us to nearly eliminate duplication of effort in our build workflow.

In addition to facilitating collaboration, the level of automation possible with Spack has helped us tremendously. Thanks to Spack, we now deploy our dependencies on more HPC systems than ever before—including systems built around x86, POWER, and ARM processors—using multiple compilers and MPI (Message Passing Interface) implementations. With Spack we also build virtual machines that contain our full development environment, and team members are able to reproduce and test with their own full versions of the FLAG stack. Developers can easily conduct experimental work outside the main FLAG workflow, giving us a head start on future containerized HPC deployments.

None of these recent productivity gains would be possible without a tool like Spack that can describe and manage the entire deployment tree for our code. Spack's concretization, query capabilities, and ability to concisely describe package requirements allow us to collaborate and automate much more effectively than we could otherwise. Spack has made the Lagrangian Applications Project more efficient by transforming the way we develop, deploy, and release software to users.

Sincerely,

Gabriel Rockefeller

OAK RIDGE National Laboratory

PO Box 2008
Oak Ridge, TN 37831
(865) 574-1841 | ashley@ornl.gov

To the R&D 100 Evaluation Committee:

I am writing to endorse the Spack Package Manager for a 2019 R&D 100 award. I lead the User Assistance and Outreach Group at the Oak Ridge Leadership Computing Facility (OLCF). My group is responsible for deploying scientific software on OLCF's Summit supercomputer, currently ranked #1 in the world (as of the November 2018 list at Top500.org). The Summit software ecosystem contains over 1,300 package installations, which need to be built, tested, installed, and updated frequently. Prior to implementing Spack at the OLCF, it generally took about two weeks to deploy a new software stack on our large systems. Using Spack, the OLCF is able to deploy a new software stack in less than 12 hours, which not only reduces the amount of resources required for this task, but it also enables the OLCF to respond more quickly to user requests. Software installation would present a much heavier maintenance burden on my team without this tool.

OLCF's software stack traces its roots to the Titan and Jaguar Supercomputers, which were predecessors of our latest Summit machine. On those systems, the facility was also faced with the problem of deploying large numbers of software packages for users. For many years, we used a manual but structured process called "SWTools", but the scheme was not flexible or automated enough to accommodate the sheer number of versions we wanted to deploy. There was no regular naming scheme or query capability, and software and module files had to be written by hand.

To replace SWTools, one of our employees developed an automated tool, Smithy, for software installation. Unfortunately, the Smithy tool was complex and took a significant amount of development time to maintain, and it did not work well as we needed in a collaborative environment. Perhaps more importantly, the tool did not handle versioning or dependency management well, and it was still not able to manage all of the complexities of our multi-compiler, multi-version, multi-library software installation. After the Smithy developer left the team, we began evaluating other tools that could better serve our needs.

We investigated a few options, but Spack was the clear winner. The fact that many other HPC centers were beginning to adopt Spack meant that we wouldn't necessarily have to build everything ourselves, and we could utilize community builds for some packages. The OLCF has benefited from the excellent documentation and training provided by the Spack development team, and as a result we are now able to share the effort of software deployment across our entire team, and do not have to rely on one person to carry this load. Spack has given us a much better tool in which to operate our center.

This capability was crucial for our center; the software environment on a new machine is constantly in flux, and we need a tool that can keep up with changes from the machine vendor and from facility staff. For example, during acceptance testing, we were receiving new software builds from the vendor on a regular basis to address known issues. In large part due to Spack, my team was able to rapidly deploy software on Summit when the machine arrived at our center and continue redeploying new software updates as needed. This would have been a much more burdensome process without the use of Spack. We have also started utilizing Spack to begin initial

deployment of the Exascale Computing Project (ECP) software stack on Summit. Once again without Spack, this effort would have also required additional human resources to manage these package installations.

The OLCF has greatly benefited from our adoption of Spack, and we plan to continue to leverage the use of this tool for many systems to come.

Sincerely,

Ashley Barker
Group Leader, User Assistance and Outreach
Oak Ridge Leadership Computing Facility
Oak Ridge National Laboratory

Fumiyoshi Shoji
Director of Operations and Computer Technologies Division
RIKEN Center for Computational Science
Kobe, Japan

To the R&D 100 Evaluation Committee:

I am writing to endorse the Spack package management tool for a 2019 R&D 100 award. I am Director of Operations and Computer Technologies Division at RIKEN, Japan's largest and most comprehensive research institution. Specifically, I work on the Flagship 2020 project, a joint effort between RIKEN and Fujitsu to build the Post-K supercomputer, the successor to Japan's current K supercomputer. Recently we decided to adopt Spack as the official deployment tool for open source scientific software on the Post-K system.

Modern scientific software relies on a broad foundation of publicly available open source software (OSS) packages. However, the multitude of available dependency libraries presents problems; building and managing hundreds of dependencies can be a monumental task. On supercomputers, the problem is even more prominent, as many packages must be rebuilt frequently when the system MPI implementation and compiler are updated. Moreover, on bleeding-edge systems like Post-K, which will use a new type of Arm processor, each package in a software stack must be ported and tuned to run efficiently on the new hardware.

We selected Spack after carefully comparing its features with those of competing solutions. In our tests, Spack was easily adaptable for Arm systems like Post-K. Its templated package language allows many existing package recipes to be reused on new platforms, and its internal dependency model is versatile enough for cross-compiled HPC environments. Our team was able to add support for new compilers easily, and the existing command line tools worked smoothly and intuitively. Spack was ready to work immediately after download; the tool itself required no complicated installation.

The most significant advantage Spack affords us is its extensive library of package recipes. We can regard Spack as a knowledge repository; over 400 contributors have already added packages to its repository, and we can avoid reimplementing the same fixes and patches by building on this existing work. The packages are written for use on new machines, which allows us to collaborate effectively within our group of operators, software deployers, and users at RIKEN, as well as externally with the Post-K team at Fujitsu. This capability also connects us with developers across the broader HPC community.

Manually deploying all of the open source packages that users want to run on Post-K would be extremely difficult. Spack automates this process. Post-K software deployment will be much faster as a result, and users of the machine will have access to many more software packages than would be possible without Spack.

It is satisfying to be able to give the Spack project my highest recommendation. I hope this information proves helpful.

Sincerely,