SAND2019-12405C

Implementing a Common HPC Environment in a Multi-User Spack Instance

Carson Woods
vzl837@mocs.utc.edu
SimCenter
University of Tennessee at Chattanooga
Chattanooga, Tennessee

Matthew L. Curry
mlcurry@sandia.gov
Sandia National Laboratories
Albuquerque, New Mexico

Anthony Skjellum
Tony-Skjellum@utc.edu
SimCenter
University of Tennessee at Chattanooga
Chattanooga, Tennessee

ABSTRACT

High performance computing is highly dependent on validated, tested environments that are tuned for specific hardware to yield good performance. Spack enables users to construct their own software stacks, but does not provide the benefits of a centrally curated environment. We implemented new features in Spack to support co-existing, system-wide deployments and user Spack environments, as well as a Spack-enabled global environment with these. This improved the global environment's portability and the efficiency of Spack use on a production system.

CCS CONCEPTS

• **Software and its engineering** → **Runtime environments**; *Software configuration management and version control systems*; *Software creation and management*.

KEYWORDS

Spack, high performance computing, environments, package management, programming

ACM Reference Format:
Carson Woods, Matthew L. Curry, and Anthony Skjellum. 2019. Implementing a Common HPC Environment in a Multi-User Spack Instance. In *Proceedings of HPCSYSPROS19.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Computing environments are critically important to high performance computing (HPC). Often, individual packages are tuned for performance for specific architectures. Centrally curated software environments have both advantages and disadvantages. They are useful because they provide a common set of validated software that users can leverage. But, they are also cumbersome because of their static, minimal coverage of potential software dependencies. A more flexible approach that allows users to build their own software stacks according to their individual needs is desired.

Spack is a package manager for HPC software [1]. Unlike other package managers, Spack installs each package from source by default and offers users complete control over how packages are installed with optional configuration flags and package variants. It also has the ability to create highly portable software stack definitions that can be easily modified or distributed. Spack easily solves the problem of customizable, per-user software environments; however, it does not provide any of the benefits of a central, curated environment (i.e., verification, security reviews, compatibility checks, etc.). Spack also has the challenge that CPU time, disk space, and user effort is duplicated among many users for commonly needed software.

In this work, we demonstrate a multi-level Spack extension that allows a system to provide multiple Spack instances working in concert across multiple privilege levels. In effect, this allows an administrator to create a system-wide Spack installation with curated packages for users, while allowing users to create dependent Spack installations that can leverage the system-level packages. This feature allows for the best of both worlds—A centrally curated software stack that works seamlessly with a user's own Spack instance, allowing the user access to high quality software configurations without the inefficiency of duplicating an environment.

In the following, first we describe the Advanced Tri-Lab Software environment including what it contains and the background for its development. Then, we describe Spack's feature set and its shortcomings as well as common alternatives to Spack. Third, we cover how we implemented multi-user functionality in Spack, as well as detailing how to use it in example scenarios. We describe how we implemented the ATSE environment in Spack and measured its impact on time and disk space saved by comparing installations of Trilinos with and without ATSE installed. Then we discuss the challenges faced when building these features and running this test. Finally, we draw conclusions about our work and discuss possible future contributions to the Spack project to improve on what we have achieved thus far.

2 BACKGROUND

We consider background areas that motivate and enable this work: Advanced Tri-Lab Software Environment, Spack itself, Spack environments, and, finally, related work.

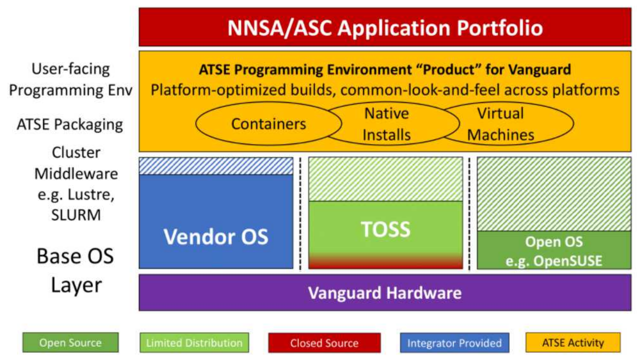
Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

2.1 Advanced Tri-Lab Software Environment

The Advanced Tri-Lab Software Environment (ATSE) is a scientific computing environment being developed as a part of the Vanguard program at Sandia National Laboratories. The Vanguard program is intended to "[expand] the high-performance computing ecosystem by evaluating and accelerating the development of emerging technologies in order to increase their viability for future large-scale production platforms" [3]. As a part of this program, ATSE was developed to provide a portable computing environment. Astra, a large Arm-based system at Sandia, was used as a proving ground for the environment.

ATSE was designed to be a complete software and runtime environment that could be used irrespective of the underlying operating system. One of its deployment targets is the Tri-Lab Operating System Stack (TOSS), a Red Hat Enterprise Linux derivative [4]. While TOSS offers some of the packages that are included in ATSE, different versions were installed so that users would be able to rely completely on the ATSE environment [6]. Figure 1 shows how ATSE and the underlying operating system interact. Currently ATSE consists of a variety of compilers, parallel processing libraries, virtual machines, containers, etc. [5]. ATSE contains no system-specific programs or packages and is subsequently not dependent on TOSS or any other specific operating system. Because ATSE is completely separate from the underlying operating system—which may have restrictive licensing or distribution limits—it becomes possible to share ATSE with researchers in external organizations.

Figure 1: ATSE provides a common environment over a variety of operating systems.


NNSA/ASC Application Portfolio
User-facing Programming Env
ATSE Programming Environment "Product" for Vanguard
Platform-optimized builds, common-look-and-feel across platforms
ATSE Packaging
Containers
Native Installs
Virtual Machines
Cluster Middleware e.g. Lustre, SLURM
Base OS Layer
Vendor OS
TOSS
Open OS e.g. OpenSUSE
Vanguard Hardware
Open Source
Limited Distribution
Closed Source
Integrator Provided
ATSE Activity

On Astra, the ATSE environment has been implemented with an infrastructure based on the Open Build Service [8], which generates RPMs from SPEC files defining the configurations for each package. These RPMs are installed via Yum, a package manager distributed with Red Hat Enterprise, CentOS, and SUSE distributions of Linux. These customized SPEC files contain the specific tuning necessary for each package to achieve good performance. While the RPM-based strategy has been effective, Spack promises to improve cross-OS portability and external distribution of configurations.

2.2 Spack

An alternative to an RPM-based environment is Spack, an open source package manager for high performance computing maintained by Lawrence Livermore National Laboratory [1]. Spack is a source-based package manager, which gives it some key advantages for HPC users. For instance, users have fine-grain control over the configuration (or spec) for each package before it is installed, allowing for deep optimization of each package. Spack also allows for installing multiple versions and/or configurations of a package concurrently on a Spack instance, which can be problematic for software distributed as RPMs.

Note that, when discussing Spack in this paper, an "installation" refers to the Spack program files (e.g., the `Spack` script that a user invokes). In contrast, a Spack "instance" refers to a set of installed packages and configuration. For typical Spack use, a Spack instance exists in a user's `.spack` directory and can be referenced by any Spack installation.

2.3 Spack Environments

Spack includes support for a feature called environments. Spack environments are highly portable configuration files that can be used to define a set of packages that adhere to certain specifications on any machine. It only takes a single YAML or JSON file to define a complete environment and can be shared and installed on any instance of Spack. This feature allows users to create software stacks that are highly portable while still maintaining a high amount of specificity. This is a departure from traditional HPC environment implementations.

Using Spack environments, a user can define an environment to contain certain packages with various compilers and configuration options in a lock file. This file can be shared with anyone and is used by another instance of Spack to recreate the environment. An environment created from a lock file will be guaranteed to be identical to the original environment used to create it. Spack's lock files are extremely brittle so slight manual changes are prone to break the environment. Additionally, this configuration file is generated by Spack and it is subsequently not easily readable.

Because of this, Spack supports a human readable and easily modifiable format, `spack.yaml` files. These files are relatively more loose definitions of a software stack and are more likely to work across different systems and system architectures. The `spack.yaml` files also allow a user to define extra parameters such as upstreams, mirrors, repos, compilers, etc. in addition to the specifications of the packages that need to be installed.

Together, these configuration files form a complete Spack environment; and, the full environment can be generated from either file. The `spack.lock` file has a purpose beyond simply defining an environment. It defines the concretized[1] specification for how every package will install and how the dependencies for each package will be related to other packages. In addition to having a high level of portability, multiple Spack environments can exist simultaneously. This allows a user to activate the given environment that they

[1]Spack's concretizer determines how packages depend on each other and determines compatibility. When an environment or package specification is determined to be valid it is considered "concretized."

want to use, rather than locking them into a single environment's configuration.

While there are some limitations to this process, this process makes it much easier to share and collaborate when environments can be easily shared and modified. The primary limitation on the portability of Spack environments arises because there might need to be changes made to the environment when shared based on hardware limitations. Generally speaking, it is a straightforward solution to the problem of environment portability. Making modifications to an environment is expected when moving between systems. Modifying SPEC files is time consuming and requires users learning how packages in the shared environment interact together. By using Spack environments instead, this barrier can be circumvented. Providing a single file that is easily readable makes it much easier to understand the layout of the environment. Additionally, if changes need to be made to individual packages, then a user only has to append the desired configuration to the end of a package and the Spack concretizer will ensure that the dependencies support that configuration automatically.

2.4 Related Work

EasyBuild [2] offers similar functionality to Spack inasmuch as it allows for automatic dependency resolution and the potential for single command package installs. However, it differs in a few key ways. EasyBuild allows for more flexibility when configuring packages, because any configuration flag can be passed directly to the installation via the command line. Spack, by comparison, only supports configuration options that are in the `package.py` file for a given package. Because it was designed to be highly readable, Spack's configuration flags and environment files are often more user friendly and quicker to understand than EasyBuild's counterparts. However, for certain packages, some options may not be available without modifying the package file.

ATSE's RPM-based infrastructure is derived from that of OpenHPC. OpenHPC is a collection of packages that site system administrators can install on Red Hat Enterprise Linux, CentOS, or OpenSUSE to create HPC systems [7]. OpenHPC integrates with existing package managers such as Yum to add repositories with the packages a user might want to use. OpenHPC's RPM-based distribution limits the configuration options for each package, since optimizations cannot be included after the RPMs are built. This can prevent machine-specific optimizations from being used, limiting available performance.

3 IMPLEMENTATION

When starting this project, Spack did not have significant multi-user support. Without multi-user support, each user would have to be provided the same `spack.yaml` file that contains the specifications for the environment and for each change or revision, new `spack.yaml` files would have to be issued to users. Additionally with every revision, a user would have to rebuild their environment, wasting both time, CPU cycles, and disk space. This requires more user involvement than traditional implementations and could lead to misconfigured environments if an install failed. Because of this it was important for us to implement a shared mode for Spack so that multiple users can have access to the same environment.

Spack initially was designed with a single user in mind and so had limited support for multiple users. The closest pre-existing feature to a multi-user mode was the ability to chain instances of Spack together. This allowed packages and environments in one instance of Spack to be made visible in a second instance of Spack that was pointed at the `$spack/opt/spack` directory[2] of the first instance. Unfortunately, this feature had some drawbacks that made it an incomplete solution for what we needed for our work. If the upstream instance of Spack was in a location with restricted write permissions, there was no way to deal with permission disparities between users when installing, uninstalling, and accessing packages. Additionally, because of how Spack handled its internal database of installed packages, it could not differentiate between upstream packages and those in the user's instance of Spack. Because of this, if a package was located upstream in a user-writable directory, Spack would be able to uninstall it as though it were installed locally. Because of these limitations, we expanded Spack's upstream capabilities so that multiple users could use the same instance of Spack without deviating from the fundamental way that Spack operated.

With the advent of shared mode, there are now two locations that can be used to install software: The default location, and the upstream location. The default install location is `$HOME/.spack`, which is unique for each user. The upstream location is within `$spack/`, which is determined by the location of the Spack instance. For example, if a shared instance of Spack is installed at `/opt/spack`, the upstream location is located within that directory as well, and can be made read-only to all non-upstream owners.

Spack's shared mode is an extension of the upstream feature. By including a globally accessible upstream, every user of a Spack instance has access to globally installed packages. Spack previously lacked a straightforward way to install or uninstall packages from upstream Spack instances. We added support for installing and uninstalling from the global upstream by running Spack's install command with either `-g` or `--global`. This tells Spack to target the global upstream rather than the user specific install location. The same process is used from uninstalling from the global upstream. Running `spack <install/uninstall> <-g/--global> <spec>` will allow system administrators to install packages that should be available to all users. If the Spack instance is located in a restricted write-access location, a user would need to run the install/uninstall command with appropriate privileges. In settings where administrators are changing frequently, by setting ownership of the Spack installation to a specific group of users, administrators could be added or removed from this user group readily.

While adding support for installing and uninstalling from the global upstream, we also added support for targeting any upstream that a user might have specified. Because upstreams can be unique to a specific user we wanted users to be able to target any upstream that they might want to interact with, in addition to the included global upstream. In order to interact with a desired upstream, a user can specify `-u/--upstream <upstream_name>` in the install and uninstall command. This command can also be used to target the global upstream via `--upstream global`. However, it is simpler to use `--global` for that process.

[2] `$spack` is defined as the top level directory of a user's Spack installation.

Additionally, because Spack supports storing multiple environments, different versions of ATSE could exist simultaneously. This would allow for administrators to develop a new version of ATSE without removing access to the original version. Users could also choose which version of ATSE that they prefer to use, and administrators could offer a range of versions of the environment to maximize compatibility with user's workload.

3.1 Usage

Using Spack to administer an environment is not dramatically different from a traditional RPM-based environment. It is much easier to install new packages and configurations; however, the methods users and system administrators use to interact are nearly identical. An ideal use case for the global upstream would be as follows:

(1) System administrators installs Spack to a system wide location.
(2) System administrator installs packages and environments that they want to be accessible to all users using `spack install/uninstall --global <spec>`
(3) Users are provided a module file for loading this system wide instance of Spack or they are pointed to `$spack/shared/spack/setup-env.sh`.
(4) A user uses the module file to set up their Spack environment.
(5) Users can install packages and load Spack environments. When they install packages they are installed to `$HOME/.spack/` rather than the system wide Spack install directory.
(6) Users can use Spack as normal and can install packages as needed. System Administrators can easily manage the packages and available environments. Multiple environments and multiple versions of the same environment can coexist without conflict.

Similarly, a use case for the `--upstream` toggle would be as follows:

(1) Users load their instance of Spack.
(2) Users define their own upstream configurations for various Spack instances that they want to interact with an use as upstreams. These definitions should be created in `$HOME/.spack/upstreams.yaml`. See Listing 2 for an example of the default upstreams.yaml.
(3) Users can install packages and load environments. When they install packages they are installed to `$HOME/.spack/opt/spack` rather than the system wide Spack install directory.
(4) Users can install and uninstall packages at their targeted upstreams through `spack <install/uninstall> <-u/ --upstream` upstream_name> <package_spec>. If the user needs elevated permissions in order to install or uninstall from a specific upstream they would need to run the command with appropriate permissions.

4 EVALUATION

The process for converting ATSE from a Yum-based solution to a Spack-based environment was done in stages and faced several challenges during this process.

Inside a `spack.yaml` package spec section, we defined all of the packages included in ATSE along with the configuration flags. Certain packages, such as slurm, compilers, etc., are provided by ATSE RPMs; however we excluded them from the Spack implementation of ATSE because a user would not normally build these components themselves. We consider these packages to be system infrastructure. See Listing 1 for the ATSE environment's `spack.yaml` file that can be used to generate the ATSE environment.

We also provide a `spack.lock` file for the ATSE environment, however the expected use case is different. We would expect a user to use `spack.lock` on a system where the `spack.yaml` would not have to be changed in order to have a working environment. This would reduce the install time, while achieving the same results as though `spack.yaml` were used. If a user anticipates needing to make changes to ensure compatibility, then the `spack.yaml` is the configuration file to use.

4.1 Adding Packages

The first step of converting ATSE into a Spack-based environment was to add the packages into a `spack.yaml` file. Spack allows for adding packages to an environment easily through the `spack add <spec>` command.

One of the challenges that we encountered when adding packages is that not all packages that we required were available through Spack by default. We used two different approaches to circumvent this issue. The first solution involved creating packages in our own repository and configuring Spack to use them. Spack supports users adding custom packages that use any of a few predefined build tools such as autotools, cmake, etc. This allowed us to define our own packages when Spack did not already include them. These custom repositories can be used to add packages or override default installation options used by Spack. Custom repositories take precedence over the built-in repository. We include an ATSE repository in our distribution that contains all of the packages that the ATSE environment needs to install with non-default configurations and patches.

The second solution is an extension of the solution just described. To improve portability, we wanted to reduce reliance on custom repositories such as the one we provided. To do this, we submitted pull requests to the Spack repository to have non-included packages added as standard Spack packages. In addition to adding packages we have also fixed bugs in a variety of package installation instructions to improve how packages and their multiple variants are handled. Using these strategies, we were able to add all packages in ATSE to the environment.

4.2 Refining Package Specs

For many different packages, the ATSE environment uses custom configurations to optimize performance. Spack supports these configurations through flags that can be specified at an environment level or at install time. In order to match configurations in the existing implementation of ATSE, when possible, we enabled variants on certain packages in the `spack.yaml` environment file. One challenge with this process is that not every package installation profile in Spack had all of the configuration flags that ATSE needed. To overcome this issue, we again used the same two-tiered approach that we used to overcome missing packages. We made changes to our local repository to temporarily override this shortcoming for certain packages. We also opened pull requests to have these variations in packages added to the default Spack repository.

4.3 Resource Comparison

To quantify the resource savings enabled by our Spack extension, we installed Trilinos and measured the difference in install times and in space savings when installed on Astra. In both upstream and downstream installs Spack's default, install behavior was unchanged. Spack installed one package at a time, but used parallel threads within each build process. The experiment was split into two steps.

Figure 2: Trilinos Disk Space Usage Comparison

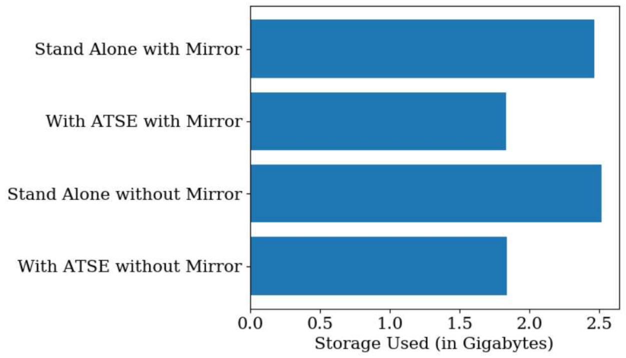
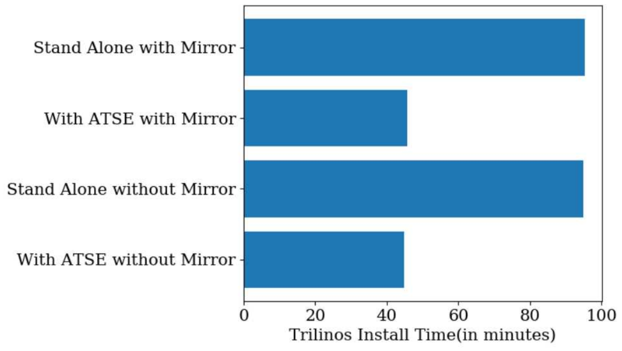


Figure 3: Trilinos Install Time Comparison



For the first stage we started with an empty Spack instance and installed Trilinos by running `time spack install trilinos`. We ran the install step five times. Each iteration we measured the size of the `.spack` directory before and after. Additionally, after each install we uninstalled every package that had been installed and cleared the staging area to emulate a completely clean installation. As shown in Figure 2 and Figure 3 respectively, we found that by installing Trilinos and its dependencies from scratch we used 2.464 gigabytes of disk space and it took 95m 22.859s to install on average.

For the second stage, we prepared a similar process, but this time we allowed Spack to leverage an upstream installation of ATSE packages. We repeated the same process as before five times. However, we only uninstalled Trilinos and dependencies that were not included in ATSE. Specifically, we uninstalled GLM, Hypre, Matio, Netlib-Scalapack, Mumps, and Trilinos each time. The Trilinos dependency graph is listed in Listing 3. This yielded a disk usage of 1.834 gigabytes and only took 45m 45.343s on average to install.

In the test, that involved installing all packages from scratch, we were using a mirror to store the packages that are found in ATSE. We later reran both tests without the mirror. With the mirror disabled and with ATSE pre-installed, we found that installing Trilinos used 1.837 gigabytes of storage and took an average of 44m 48.357s to install. Without a mirror and without ATSE pre-installed, the Trilinos install used 2.514 gigabytes of disk space and took an average of 94m 46.8216s to install. The nearly identical installation time resulted because of Spack's ability to stage archives of previously downloaded files which makes those packages available to future installs by a single user. Because Spack stages each package within a user's `.spack` directory, users would not benefit from other user's previously staged packages.

These space and time savings are for a single instance of Trilinos. Because the upstream packages are still consuming disk space, the space savings are only leveraged once there is more than a single user relying on the same upstream package. Every time a user attempts to install Trilinos with ATSE installed they would save an additional average of 629 megabytes of disk space as well as approximately 50 minutes of build time; this performance is driven by not having to reinstall all of the dependencies.

5 DISCUSSION

The primary challenge with building ATSE in Spack was the lack of previous testing on Arm-based systems. Because of their current rarity in production environments, many of the packages used in ATSE did not install correctly without modification. To overcome this, pull requests were made to the Spack repository to improve Arm compatibility.

Additionally, limitations with Spack's detection of compilers made installing certain packages fail without manually informing Spack about compiler modules. Spack environments do allow for modules to be specified in advance of the environment being installed. However, this would cause problems if used on any other machine than the original machine that the environment was developed on. Different module naming schemes are common, requiring manual updating of this setting for each site. This puts the onus for specifying the correct module for the compiler on the user or system administrator.

Package availability is another limitation of this system. Because packages are often managed by Spack's community of open source contributors rather than package developers, the number of packages is rapidly outpacing the number of contributors. This means that future versions and package configurations are potentially not added immediately or at all. In some instances, package installation recipes would be non-functional and would require manual bug fixes, or else a newer version of a package would install differently than its predecessor.

In the event that a version of a package does not exist in Spack's internal database, Spack tries to extrapolate a download link using the known download link and the version number. However, this strategy is not guaranteed, and the checksum safety features will no longer function. Additionally, if a package were to change dramatically, it is possible that newer versions would not install without dramatic changes. This is not necessarily an immediate drawback; however obscure packages could suffer from a lack of maintenance, which could leave system administrators responsible for maintaining the packages they need. This does not differ too much from how traditional RPM environments are managed, but it removes some of the convenience of using Spack to manage the environment.

6 CONCLUSION

Traditionally, computing environments are built using tools that make portability and modification challenging and time consuming. The Spack package manager makes managing environments much easier. In addition to improving how environments are managed, Spack provides straightforward mechanisms for sharing environments with other users and sites. Additionally, once shared, users are able to easily modify any part of the environment to suit their needs and better match the system on which they are working.

For Spack to efficiently manage software environments for a large HPC system, multiple users need to be able to share a common base environment. For this reason, we implemented multi-user support in Spack; this makes Spack suitable for systems with many users sharing a single environment. Using these new features, multiple users can use a single instance of Spack as they normally would, while also being able to access a shared set of packages representing a system-wide computing environment. Under this system, disk space usage and install time for other packages can be significantly reduced.

6.1 Future Work

Despite ATSE being fully implemented in Spack, there are still many improvements that could be made. The extra steps required in getting certain compiler configurations to work mentioned in Section 5 could be streamlined. If Spack automatically detected the required module file and loaded it, rather than forcing the user to manually specify it, the ATSE environment would have a much greater chance of compiling without requiring tweaking or modifications by the user.

Currently, no usage metrics are collected by Spack. However, since central environments can now be managed by Spack, it could be be useful to locally collect installation data from users within an environment. In a shared environment setting, administrators would be able to determine if multiple users were installing a package or using a configuration that was not inlcuded in the shared environment. Making this change would allow for a better informed shared environment with packages that better represent their user's needs.

Finally, protections need to be implemented for when user's rely on a package as a dependency that is located upstream. It is reasonable to assume that a package could be removed or changed upstream which is being used as a dependency of a downstream package. In a traditional Spack installation, there are protections to avoid uninstalling a package that is relied upon as a dependency. By extending these safeguards to packages installed between various upstreams, it could greatly reduce potential confusion experienced by some users. Currently, the best way to avoid these isues while also updating packages and configurations is to simply make concurrent versions of the shared environment available to users.

7 ACKNOWLEDGEMENTS

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

This work was also supported in part by the National Science Foundation under Grants Nos. CCF-1822191, CCF-1821431. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

A REPRODUCIBILITY

A.1 spack.yaml

The following is the `spack.yaml` file required to get the full ATSE environment in Spack. It lists all included packages and their configurations.

Listing 1: ATSE `spack.yaml` file

```
# This is a Spack Environment file.
#
# It describes a set of packages to be installed, along with
# configuration settings.
spack:
  # add package specs to the `specs` list
  specs:
  - binutils@2.31.1+libiberty
  - charliecloud@0.9.6
  - autoconf@2.69
  - automake@1.16.1
  - cmake@3.12.2
  - git@2.19.2
  - libtool@2.4.6
  - ninja@1.8.2
  - valgrind@3.15.0~ubsan
  - lua-luaposix@33.2.1
  - hdf5@1.10.5+cxx+fortran
  - bzip2@1.0.6
  - hwloc@1.11.11
  - netcdf-cxx4@4.3.0
  - netcdf-fortran@4.4.5
  - metis@5.1.0
  - openblas@0.3.4 threads=openmp
  - boost@1.68.0
  - papi@5.7.0
  - pmix@2.1.4
  - numactl@2.0.12
  - xz@5.2.4
  - zlib@1.2.11
  - openmpi@3.1.4+legacylaunchers+pmi schedulers=slurm
  - fftw@3.3.8~double simd=neon
  - superlu@5.2.1
  - qthreads@1.14
  - scotch@6.0.6
  - netcdf@4.6.3+parallel-netcdf
  - tau@2.28.1+openmp+papi+pdt cflags="-fPIC"
      cppflags="-fPIC"
  - pdt@3.25
  - cgns@3.4.0+fortran+int64
  - yaml-cpp@0.6.2
  - osu-micro-benchmarks@5.6.1
  - ucx@1.5.1
  - singularity@3.2.1
  - mpip@3.4.1
  - powerapi@1.1.1
  - parmetis@4.0.3
  mirrors: {}
  modules:
    enable: []
  repos: []
  packages:
    slurm:
      paths:
        slurm: /usr
      buildable: false
      version: [17.11.12]
      providers: {}
      modules: {}
      compiler: []
    go:
      paths:
        go: /usr
      buildable: true
      version: []
      providers: {}
      modules: {}
      compiler: []
  config: {}
  upstreams: {}
```

A.2 upstreams.yaml

The following is an example of the upstreams.yaml file. This is the default file that defines the global upstream.

Listing 2: Default upstreams.yaml

```
upstreams:
  global:
    install_tree: $spack/opt/spack
    modules:
      tcl:    $spack/share/spack/modules
      lmod:   $spack/share/spack/lmod
      dotkit: $spack/share/spack/dotkit
```

A.3 Trilinos

The following is the concretized dependency graph for the Trilinos package.

Listing 3: Trilinos spec

```
trilinos@12.14.1
        ^boost@1.70.0
            ^bzip2@1.0.6
            ^zlib@1.2.11
        ^glm@0.9.7.1
        ^hdf5@1.10.5
            ^openmpi@3.1.4
                ^hwloc@1.11.11
                    ^libpciaccess@0.13.5
                    ^libxml2@2.9.9
                        ^libiconv@1.15
                        ^xz@5.2.4
                    ^numactl@2.0.12
        ^hypre@2.16.0
            ^openblas@0.3.6
        ^matio@1.5.13
        ^metis@5.1.0
        ^mumps@5.2.0
            ^netlib-scalapack@2.0.2
        ^netcdf@4.7.0
        ^parmetis@4.0.3
        ^suite-sparse@5.3.0
```

A.4 Where to Find Our Work

By cloning our fork of the Spack repository all software needed to reproduce the experiment can be obtained through Spack.

The fork of Spack can be found at: https://github.com/carsonwoods/spack and the pull request to merge our feature into Spack can be found at: https://github.com/spack/spack/pull/11871.

REFERENCES

[1] Todd Gamblin, Matthew P. LeGendre, Michael R. Collette, Gregory L. Lee, Adam Moody, Bronis R. de Supinski, and W. Scott Futral. 2015. The Spack Package Manager: Bringing Order to HPC Software Chaos. *Supercomputing 2015 (SC '15)* (November 2015). LLNL-CONF-669890.

[2] Markus Geimer, Kenneth Hoste, and Robert McLay. 2014. Modern Scientific Software Management Using EasyBuild and Lmod. *HUST 2014* (2014).

[3] Sandia National Laboratories. [n. d.]. Vanguard Program. https://vanguard.sandia.gov. Accessed: 2019-08-20.

[4] Lawrence Livermore National Laboratory. [n. d.]. TOSS: Speeding Up Commodity Cluster Computing. https://computing.llnl.gov/projects/toss-speeding-commodity-cluster-computing. Accessed: 2019-07-31.

[5] James H. Laros III, Kevin T. Pedretti, Simon Hammond, Michael Aguilar, Matthew L. Curry, Ryan E. Grant, Robert Hoekstra, Ruth Klundt, Stephen Monk, Jeffry Ogden, Stephen L. Olivier, Randall Scott, Lee Ward, and Andrew J. Younge. 2018. FY18 L2 Milestone #8759 Report: Vanguard, Astra, and ATSE — an ARM-based Advanced Architecture Prototype System and Software Environments. *Sandia National Laboratories* (September 2018). https://doi.org/10.2172/1470822

[6] Kevin Pedretti, Jim H. Laros III, and Si Hammond. 2018. Vanguard Astra: Maturing the ARM Software Ecosystem for U.S. DOE/ASC Supercomputing. ExaComm 2018. (June 2018). SAND2018-7066 C.

[7] Karl W. Schulz, C. Reese. Baird, David Brayford, Yiannis Georgioum, Gregory M. Kurtzer, Derek Simmel, Thomas Sterling, Nirmala Sundararajan, and Eric Van Hensbergen. 2016. Cluster Computing with OpenHPC. *HPCSYSPROS 2016* (November 2016).

[8] Open Build Service. [n. d.]. Open Build Service Homepage. https://openbuildservice.org/. Accessed: 2019-08-20.