Title
Software Deployment Process at NERSC: Deploying the Extreme-scale Scientific Software Stack (E4S) Using Spack at the National Energy Research Scientific Computing Center (NERSC)

Permalink
https://escholarship.org/uc/item/5zh5z08q

Authors
Siddiqui, Shahzeb
Shende, Sameer

Publication Date
2022-03-01

DOI
10.2172/1868332

Peer reviewed
Software Deployment Process at NERSC: Deploying the Extreme-scale Scientific Software Stack (E4S) Using Spack at the National Energy Research Scientific Computing Center (NERSC)

Shahzeb Siddiqui (shahzebsiddiqui@lbl.gov), Sameer Shende (sameer@cs.uoregon.edu)

National Energy Research Scientific Computing Center
Lawrence Berkeley National Laboratory
Berkeley, CA 94720

Report No. LBNL-2001458

Office of Advanced Scientific Computing Research
Office of Science
US Department of Energy

March 2022
This document was prepared as an account of work sponsored by the United States Government. While this document is believed to contain correct information, neither the United States Government nor any agency thereof, nor the Regents of the University of California, nor any of their employees, makes any warranty, express or implied, or assumes any legal responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by its trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof, or the Regents of the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof or the Regents of the University of California.
# Table of Contents

**Motivation**  
4  
**Background**  
5  
**The Journey of Deploying Software**  
Step 0 - Determine which system to deploy your software  
7  
Step 1 - Acquire Spack Configuration  
8  
Step 2 - Preparing Spack Configuration  
9  
Step 3 - Module Generation  
14  
Step 4 - Deployment Script  
16  
Step 5 - User Documentation  
17  
Step 6 - Give back to the community  
18  
**Recent Developments**  
Building E4S on Perlmutter  
20  
Automation  
22  
MPI Support  
25  
Container-based deployment of E4S  
26  
Testing E4S Post Deployment  
28  
**Conclusion**  
How to Get Involved  
33  
**Acknowledgement**  
34  
**Bio**  
35
Motivation

One of the many benefits of using a high-performance computing (HPC) system at a Department of Energy (DOE) Office of Science (SC) HPC facility is the large number of software products, built and optimized for the system. The HPC center staff and HPC vendors provide optimized software such as libraries and even full scientific applications, ready to be used by users as building blocks to accelerate scientific discovery. Behind each provided packaged software module are a large number of decisions - which compiler, optimizations, variants/options - to build the software on the target system. And, even before the software gets deployed, the software must be developed, tested, and maintained, including deprecating old versions and ensuring compatibility across versions. The software lifecycle is complex and is further convoluted by a web of interdependencies on other software.

In an HPC environment, the software lifecycle is even more complicated and requires a community to address the many challenges. The Extreme-Scale Scientific Software Stack (E4S) is a community effort supported by the Exascale Computing Project (ECP) to provide an ecosystem of open source software packages for developing, deploying and running scientific applications on HPC platforms. E4S provides unified and consistent deployment through a collection of Spack packages which can be used by users, a development team, or site administrators at an HPC facility. E4S is a flexible software stack for HPC systems that enables an end-user to install a subset of E4S packages for their development purpose, a software development team to install their software product and integrate E4S spack builds into their CI/CD process, or a site-administrator to install E4S on bare-metal or container system-wide for all users.

The utility of E4S extends beyond providing ready-made recipes for building some or all of E4S on a particular machine. E4S releases can be used to identify reasonable default versions of software packages that are known to be interoperable. Recipes for building on a system can be leveraged as a starting point for similar or next-generation systems. For existing supported systems, E4S can be used to quickly learn how to add a new package to a deployment not previously supported on the system. Much of the value of E4S is not in specific recipes for specific systems, but rather in the general improvement of interoperability of packages that comprise the software ecosystem, or documenting where versions of packages are known not to be interoperable, as well as in documenting when certain packages are not yet ported to a platform.

In 2021, National Energy Research Scientific Computing Center (NERSC) at Lawrence Berkeley National Laboratory released its first deployment of E4S/20.10 on the Cori supercomputer using the spack package manager. NERSC wants to leverage E4S to provide an advanced, performant, stable and supported HPC environment to its users. By being at the forefront of E4S deployment and testing, NERSC is able to provide feedback to ECP Software Technology teams with build failures during deployment so they can be fixed in future versions.

NERSC has since also deployed E4S on its newest supercomputer, Perlmutter, a Cray Shasta supercomputer composed of 1,536 GPU-accelerated nodes with of AMD EPYC 7713 (Milan) CPUs and NVIDIA A100 GPUs, a 35 PB all-flash Lustre scratch file system and the HPE Cray Slingshot 10 high speed interconnect. Perlmutter is currently (2022) being augmented with 3,072 dual-socket CPU only nodes updated to the Slingshot 11 network.
Even with software seemingly packaged and delivered with a bow like E4S, an actual system-wide deployment is complicated and requires many decisions to make for site-specific customizations, e.g. each system supports multiple compilers, compiler versions and MPI providers that impact how software is deployed. Here, we describe the steps and lessons learned to deploy the E4S software stack at NERSC to help users navigate their E4S deployment. The lessons learned can also guide future developers of packaged community software on development-to-deployment requirements.

SINCE THE INITIAL E4S DEPLOYMENT, NERSC CONTINUES TO DEPLOY NEW VERSIONS OF E4S ON CORI AND PERLMUTTER AND PROVIDES THE LATEST INFORMATION ON THE E4S - NERSC DOCUMENTATION PAGES.

Background

E4S is a collection of 100+ top-level scientific software packages needed for scientific computing in high-performance computing (HPC) environments. E4S member packages must demonstrate compatibility with the E4S community policies, including a production quality spack-based build and installation procedure. The Department of Energy Office of Science (DOE SC) ASCR Facilities (NERSC, OLCF and ALCF) are expected to build and deploy E4S on the pre-exascale systems, which helps to ensure a consistent programming environment for users across facilities.

The HPC centers interested in deploying E4S on their facility system(s) should consider how it aligns with their overall software update strategy, which takes into consideration planned system-wide upgrades that may require a rebuild of the full software stack. Leveraging planned disruptive events can minimize the overall system downtime for users. Deploying the entire E4S stack requires installing 500+ software packages, including software dependencies, using a single compiler. Installation scales linearly as one introduces additional compilers to build E4S. However, not all packages need to be installed and HPC centers should take the time to determine which packages are beneficial to their user community. At NERSC, we install a subset of the total E4S software stack system-wide, and of course users can install and configure individual E4S software packages on their own.

In close collaboration with the ASCR facilities, the E4S team created a well-defined release strategy that specifies a spack commit or branch along with a list of packages as part of the E4S release. The E4S team is committed to quarterly releases with new versions of each package. A release will contain spack configuration (spack.yaml) and reference commit, branch or tag of spack project to build E4S that will be available on GitHub at https://github.com/E4S-Project/e4s. E4S adopted the Calendar Versioning scheme (e.g. 22.02), with YY.MM format to indicate the year and month for E4S release. For more details on this discussion, please see https://github.com/E4S-Project/e4s/issues/2.

E4S also provides architecture-specific container releases and GPU-based images in Docker and Singularity image format. The E4S Download page describes how individual users may also download and install E4S without any system-level privileges. These deployment options provide flexibility to maximize user productivity. In general, users should consider the performance and portability trade-offs between using containers or building E4S with Spack, targeting the specific architecture.
The Journey of Deploying Software

E4S is released quarterly, however facilities may choose to install it bi-annually or annually. Although E4S is intended to be easily deployed on systems, in practice, deploying E4S system-wide can be complicated, at least initially, involving a trial-and-error process of determining which components work on the target system. Each successive deployment can leverage the previous recipes and should shorten the time to deployment.

FIGURE 1. E4S software deployment process for the Office of Science HPC facilities (NERSC, OLCF, ALCF)
In Figure 1, we outline the E4S software deployment process at the ASCR facilities. The release begins with E4S providing a reference spack commit/branch along a reference `spack.yaml` that ASCR facilities will acquire when building E4S on their system. The reference spack.yaml was built on the University of Oregon HPC system and it’s worth noting that simple copy/paste won’t work, since site-specific customizations are needed to take into account differences in system architecture, available compilers, and operating system. Each facility will typically need to port the spack configuration for their system.

During the porting process, the facility will determine which packages to install, select their preferred compilers and specify package preference to optimize for their system. In an ideal world, with no build errors, building the E4S stack via `spack install`, which will build all packages from source, requires a few hours of build time depending on the size of the chosen packages. In practice, the build time can take substantially longer to address/debug any software build failures. We leverage Gitlab to automate the entire software deployment process which allows us to analyze pipeline logs once the stack is built and focus on any build failures. Without automation, one would have to run these steps manually on a terminal.

**Lesson Learned: Planning ahead based on site-specific resources and your community needs**

Deploying a software stack as large as E4S customized for a site can typically take up to 6-8 weeks from inception depending on the facility deployment process. At NERSC, the process is complete once software is deployed as modules supplemented with user documentation. A significant portion of the software deployment process is troubleshooting build errors that may arise from using different compilers and MPI implementations, and working towards a full stack build with no errors. To build the entire E4S stack, the number of software packages can be up to 500 including third party dependencies. At NERSC, we are working on cutting down this deployment process to 2-3 weeks by automating the deployment process and reducing the size of the E4S software stack we provide to end-users. With each version, we expect to leverage existing software patches to reduce the total time.

**Step 0 - Determine which system to deploy your software**

The deployment process begins by determining which system(s) to install E4S and understanding the architecture of the systems. At NERSC, we have two production systems, Cori and Perlmutter that are our target systems for deploying E4S. For the first half of 2021, our focus was deploying E4S on Cori while Perlmutter was getting ready for initial acceptance.

Cori has two primary system partitions, Intel **Haswell** and **KNL**, along with Haswell login nodes. Currently, we build E4S on a Login node that is tuned to target the Haswell architecture which is compatible with the KNL nodes, although it is not optimized for the architecture. Given the significant time to build E4S for each node architecture, we decided to deploy E4S such that it will provide the greatest impact to our users and help us understand the deployment process for building E4S in preparation for Perlmutter.
Cori does not have any GPUs, therefore we were able to skip any E4S packages that require GPUs which typically require CUDA as a software dependency. Shown below is an overview of the Cori system architecture which can be found at https://docs.nersc.gov/systems/cori/.

![Figure 2. NERSC Cori system partitions](image)

### System Specification

<table>
<thead>
<tr>
<th>System Partition</th>
<th>Processor</th>
<th>Clock Rate</th>
<th>Cores Per Node</th>
<th>Threads/Core</th>
<th>Sockets Per Node</th>
<th>Memory Per Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>Login</td>
<td>Intel Xeon Processor E5-2698 v3</td>
<td>2.3 GHz</td>
<td>32</td>
<td>2</td>
<td>515 GB</td>
<td></td>
</tr>
<tr>
<td>Haswell</td>
<td>Intel Xeon Processor E5-2698 v3</td>
<td>2.3 GHz</td>
<td>32</td>
<td>2</td>
<td>128 GB</td>
<td></td>
</tr>
<tr>
<td>KNL</td>
<td>Intel Xeon Phi Processor 7250</td>
<td>1.4 GHz</td>
<td>68</td>
<td>4</td>
<td>96 GB (DDR4), 16 GB (MCDRAM)</td>
<td></td>
</tr>
<tr>
<td>Large Memory</td>
<td>AMD EPYC 7302</td>
<td>3.0 GHz</td>
<td>32</td>
<td>2</td>
<td>2 TB</td>
<td></td>
</tr>
</tbody>
</table>

### Step 1 - Acquire Spack Configuration

We acquire the spack configuration from E4S, such as the e4s/21.02_spack_configuration, which was released in Feb 2021. This usually means copying the content of spack.yaml and storing this in a git repository in NERSC gitlab server https://software.nersc.gov. The E4S team recommends using a per-release tagged (e.g., e4s-22.02) branch of Spack that has been validated with the E4S Spack configuration, and one that maintains release-specific patches as bugs are reported and fixed.

Some HPC facilities may choose to have a fork of spack in order to build E4S, which allows them to update the spack source code to apply their preferred changes without relying on the upstream branch. If one wants to maintain a fork, then one needs to have a deep understanding of the spack source code, or use a tagged branch, when troubleshooting builds. We don't maintain a fork of spack because this further complicates the deployment process to keep the fork in sync with upstream.

**Lesson Learned:** You can’t win them all - not all packages we planned to install are actually installed each release.

During the spack builds, we discover build errors for certain software packages that did not support our preferred compilers and runtime libraries on the target platform. First we try to troubleshoot the build error by analyzing the build log to resolve the issue and if we require further assistance we contact the spack community via slack or directly reach out to the package maintainer. If we need more visibility into the issue we would report this to the spack issue tracker: https://github.com/spack/spack/issues. In the future, we intend to report E4S build issues at the E4S issue tracking system at
https://github.com/E4S-Project/e4s/issues, a one-stop site that will curate all E4S issues and coordinate with individual package maintainers.

Someone from the spack community will apply a fix for our issue as a pull request to the spack develop branch. However, we don’t change our version of spack provided by E4S to satisfy build errors, instead we will defer these build errors in the next version of E4S. In the event of build failures, we try our best by experimenting with different build options to address build error, if all else fails we skip the build and document the issue. This process can be improved in the future as well.

Step 2 - Preparing Spack Configuration

Compiler Definition
The Cori system supports a wide range of compilers and versions, but for a large software stack deployment, we need to be more selective and determine the compilers we want to use for building E4S. On Cori, we select Intel and GCC as our preferred compilers. We have several versions of Intel and GCC compilers installed that are accessible via modules. We select one version of the compilers (a stable and widely used version) and define a compiler stanza in the spack configuration relevant for our system.

Shown below is our compiler stanza for the e4s/21.02 deployment. The compiler specs intel@19.1.2.254 and gcc@10.1.0 are the compilers used to build E4S. We use Cray compiler wrappers cc, CC, ftn and Fortran wrappers, respectively. The modules section informs what modules should be loaded when using the chosen compiler. On Cori we have PrqEnv-intel and PrqEnv-gnu modules which are Cray provided modules in order to use Intel and GCC compilers that use Cray PE wrappers.

We plan to stick to one compiler version for gcc and intel compiler when building the E4S stack on Cori even though we may have several compilers installed on the system. In future deployments, we will incorporate Cray compilers (PrqEnv-cray) into the compiler suite when building E4S to broaden our software support across the three compilers.

Lessons Learned: The compiler selection will impact which packages can be built.

It’s generally a good idea to use the gcc compiler since it is widely supported by the open-source community and most packages will be built successfully. You should target multiple compilers and versions when building a software stack, for instance we noticed that a few packages fail to build with gcc/10.1.0 but when we try a different gcc version they were built successfully.

If your HPC system is running the Cray Programming Environment, it is a good idea to use the Cray provided...
compilers which are accessible via modules PrgEnv-gnu, PrgEnv-intel, PrgEnv-cray, and PrgEnv-nvidia. If your system supports NVIDIA GPUs, then it would make sense to use the NVIDIA HPC SDK (nvhpc) compiler.

Package Selection

Although E4S has 100+ top-level software packages, not all are used by the NERSC user community. To avoid software bloat and reduce the amount of work in deployment, we need to determine which packages get installed along with the preferred compiler. In addition, each software package has a set of build options called variants that can be configured in the spack configuration (spack.yaml). The package versions that are provided by E4S typically are the latest versions of the software for the given spack release and we copy these versions provided in E4S release.

The variants are selected by inspecting each package via spack info to see applicable build options suitable for our system which requires insight into the specific system software stack. On Cori, generally we enable support for openmp and mpi when applicable. In some cases we take variants provided by E4S and copy them in our spack configuration. For example, when installing tasmanian we build the package as follows: tasmanian@7.3 +blas +fortran +mpi +python +xsdkflags. Each spack package comes with several variants along with default values for each variant. Shown below are the available variants for the tasmanian package.

![FIGURE 4. List of variants for Tasmanian spack package](image)

We define a definition name e4s_intel and e4s_gcc to map spack packages that will be installed with gcc and intel compiler. We skip some packages for various reasons, for instance we don’t want openmpi installed in the E4S stack. Packages like parallel-netcdf and python extensions that start with py-* are generally skipped. Some packages were skipped due to build failures.
FIGURE 5. List of spec definitions to specify which spack packages to install

```
definitions:
- intel_compiler: ['inteldir1.2.2.54']
- gcc_compiler: ['gcc@0.1.0']
- e4s_intel:
  - adios@2.7.1 +hdf5
  - zoal@0.1.0
  - arborx@0.9-beta +openmp
  - bolt@2.0
  - caliper@2.5.0 +fortran
  - fake-dir@1.1006.1
  - tiles@1.4 +cinch +caliper +graphviz +tutorial
  - fiirt@2.1.0
  - gasnet@2020.3.0 +udp
  - ginkgo@1.3.0
  - gotcha@1.0.3 +test
  - hdf5@1.10.7
  - hypre@2.20.0 +mixedint +superlu-dist +openmp
  - libnm@0.1.0
  - libquokka@1.3.1
  - mercury@2.8.0 +udeg
  - mfe@4.2.0 +examples +gntls +gslib +lapack +libunwind +openmp +threadsafe +pumi +umpire
  - ninja@1.10.2
  - omegahp@0.32.5 +trlinos
  - openmp-api@0.13.2
  - papi@0.8.0.1 +example +static_tools +powercap +infiniband
  - paperclip@1.8.1
  - pfio@2.5.1 +pic
  - precice@2.2.0 +python
  - pumi@2.2.5 +fortran
  - qthreads@1.16 -hloc
  - raja@0.13.0 +tests
  - slepc@3.18.2
  - strumpack@5.1.1 +shared
  - sundials@5.7.0 +examples-cxx +hypre +klu +lapack
  - superlu@5.2.1
  - superlu-dist@6.4.0 +openmp
  - svg@4.0.2-fortran
  - tassign@7.3 +blas +fortran +mpi +python +xsdkflags
  - taub@2.38.1 +mpi +pdf
  - turbospg@1.2.3 +hdf5 +python
  - umpq@0.1.8 +tests
  - umpire@4.1.2 +fortran +numa +openmp
  - upcc@2020.10.0
  - zfp@0.5.5 +aligned +c +fortran +openmp +profile

- e4s_gcc:
  - darshan-runtime@3.2.1 +slurm
  - darshan-util@3.2.1 +bzip2
  - dyninst@10.2.1
  - legins@20.8.0
  - plasma@20.9.20
  - slate@2020.10.00 -cuda

# skipping package
# - adios@1.13.1 +bzip2 +fortran +hdf5 +netcdf
# - kokkos-kernels@2.0.0 +skl +openmp
# - kokkos@2.1.0 +compiler_warnings +deprecated_code +examples +hloc +menkind +numactl +openmp +pic +tests
# - openmpi@4.0.5 +cxx +thread_multiple_schedulers+slurm
# - parallel-netcdf@1.12.1 +burstbuffer
# - petsc@3.14.4 +X +fftw +jpeg +libpng +libyaml +menkind
# - py-jupyterhub@1.0.0
# - py-libensemble@0.7.1 +mpi +nlopt +petsc4py +scipy
# - py-petsc4py@3.14.1
# - trilinos@13.0.1
```
Lesson Learned: Building software is an art, and beauty is in the eye of the installer

It’s worth noting that variant selection is an art which sometimes comes down to the installers preference and our selection may not be consistent with the developers or users preference. There is no universal selection for each package because this selection process depends on the system architecture, the compiler, and system software stack. The variant selection is an important aspect when building packages as it impacts how a package gets installed and our selection may not be optimal for all user needs. In certain situations, we reach out to developers for recommendations on the package variants. There are situations where certain variants are mutually exclusive, for instance some package X can either support openmp or pthreads so doing package X +openmp +pthreads is not allowed.

Package Preference

Most scientific software requires MPI, BLAS, and ScALAPACK as common dependencies when installing software, but on Cori we choose not to build these from source since they are not optimized for the system and are generally provided as vendor software (cray-libsci, intel-mkl, cray-mpich). This typically requires one to specify preferences to ensure spack doesn’t use the default preference to build from source, and instead uses an alternative. In our spack configuration, we leverage cray-libsci, intel-mkl and mpich as preferences for mkl, mpi, blas and scalapack.

The spack documentation has a detailed summary on build customization which can be found at https://spack.readthedocs.io/en/latest/build_settings.html

```
packages:
  all:
    compiler: [intel@19.1.2.254, gcc@8.1.0]
    target: [haswell]
  providers:
    mpi: [mpich]
    mkl: [cray-libsci, intel-mkl]
    blas: [cray-libsci, intel-mkl]
    scalapack: [cray-libsci, intel-mkl]
  pkgconfig: [pkg-config]
```

FIGURE 7. Package Preference for compiler and spack providers

```
cray-libsci:
  buildable: false
  externals:
    - spec: cray-libsci@19.06.1@intel
      modules:
        - cray-libsci/19.06.1
```

FIGURE 8. Spack external definition for cray-libsci

Certain packages like cray-libsci, intel-mkl, mpich are provided on our system which typically requires setting a spack external to ensure spack will leverage our preferred libraries provided by Cray. In the example below we define cray-libsci as an external module which maps to modulefile cray-libsci/19.06.1 which is available on Cori.

Lessons Learned: Grab a cup of coffee and take time to read the dependency tree

During the package preference determination, we analyze output of spack concretize or run spack spec to see the dependency tree to determine if output seems reasonable. In this process, we analyze each package variant, the dependency tree such as what MPI wrapper, blas provider is used. This process is time consuming especially when one is trying to analyze output of the entire software stack, which can be hundreds of packages.
Figure 9 shows a concretized spec of hdf5, and its dependencies, we can see hdf5 +mpi is set in the concretized output which means build HDF5 with MPI support and this leads to the cray-mpich dependency which is an external package.

Spack External Recommendation

Spack provides ability to reuse software pre-installed on system via spack externals to avoid reinstalling software that will never be used. You should consult output of spack concretize --f if you are in a spack environment or spack spec --spec for one of packages to see list of dependencies. We have compiled a list of spack packages that should be external when building spack stacks on NERSC systems. You may run spack external find <spec> to update your spack.yaml however we recommend you always confirm your spack configuration with what's provided by system.

<table>
<thead>
<tr>
<th>Spack Package</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bash</td>
<td>This is GNU Bourne Again Shell which is typically found in most linux systems. You can check supported shells by cat /etc/shells. You can also check this version by running bash --version. This package should not be installed as spack package</td>
</tr>
<tr>
<td>bzip2</td>
<td>/usr/bin/bzip2, should be available on system. You can run bzip2 --version to check version. You can check if rpm is available by running rpm -q --whatprovides s (which bzip2)</td>
</tr>
<tr>
<td>caio</td>
<td>The GNU caio is program to manage archives of files. This program is available at /usr/bin/caio. This package should not be installed via spack</td>
</tr>
<tr>
<td>cray-libsci</td>
<td>cray-libsci is provided by cray programming environment. You should see available modules by running module avail cray-libsci. The cray-libsci package is typically installed in /opt/cray/pe/libsci. For instance a module like cray-libsci/23.8.12.0 should be external for package cray-libsci/23.8.12.0</td>
</tr>
<tr>
<td>cray-mpich</td>
<td>cray-mpich is MPI wrapper based on mpich provided by cray. You can find cray-mpich module file, this is typically installed at /opt/cray/pe/mpich/ on cray machines. You should set this as external for packages that depend on mpi</td>
</tr>
<tr>
<td>cuda</td>
<td>cuda is typically provided on NERSC but not as a system library (/usr/local/cuda) but usually through modules. You can find this typically in modules such as module load cuda nvcc cudai cudatools cudai. You should run nvcc --version to check version. The nvcc compiler provides typically three versions of cuda for instance cuda packages by nvcc@9.1.9 are located at /opt/nvidia/cuda_s/NVIDIA_LINUX_X86_64/21.9/cuda on Perlmutter. If you want to set a cuda@9.2 external then the path would be /opt/nvidia/cuda_s/NVIDIA_LINUX_X86_64/21.9/cuda@9.2. Sometimes you may not want cuda as external especially when one needs to install from source or you have incompatible between compiler and cuda version. For reference please see nvcc release notes. There is a more detailed breakdown of cuda compatibility requirement found at <a href="https://docs.nvidia.com/cuda/index.html">https://docs.nvidia.com/cuda/index.html</a>. Please check the cuda driver via nvidia-smi for driver compatibility</td>
</tr>
<tr>
<td>concrelit</td>
<td>This package is install on linux OS which provides common utilities like cat, ls, echo and many other utilities. You can check if concrelit is installed and its version by running rpm -q concrelit</td>
</tr>
<tr>
<td>diffutils</td>
<td>This is the GNU diffutils package which provides /usr/bin/diff. This package should be external and will most likely get picked up when building a large software stack</td>
</tr>
<tr>
<td>findutils</td>
<td>The findutils package provides find and xargs command which is provided by os. This is typically located in /usr/bin/find and /usr/bin/xargs. Please check the version of the utility or check the rpm version by running rpm -q findutils</td>
</tr>
<tr>
<td>git</td>
<td>git is typically provided on NERSC systems that can be found at /usr/bin/git. This should be an external, we don’t need spack to install multiple versions of git that user will never need.</td>
</tr>
<tr>
<td>libfabric</td>
<td>The libfabric package is provided by cray which can be searched by running module avail libfabric. This is typically installed at /opt/cray/libfabric/</td>
</tr>
</tbody>
</table>

FIGURE 10. Spack External Documentation with breakdown by spack package and description

In this process, we learned spack tries to do some interesting things like installing openssl, openssl, basic linux utilities or even a scheduler like Slurm based on concretization preferences, which are redundant or
unoptimized for Cori. These types of selections need intimate knowledge of the system stack along with analyzing output of `spack concretize` to see what gets installed. After several E4S software stack builds, we documented a list of externals shown in Figure 10 that should be set which is applicable for a NERSC system. Some of these externals may be applicable for your system.

**Step 3 - Module Generation**

Most HPC systems nowadays leverage modules to allow the user to easily interface with the software stack and provide a consistent programming environment. A modulefile is a file that configures a software package such as PATH and LD_LIBRARY_PATH to configure the user environment in order to use the software with ease. Currently, there are two module systems in use, Lmod and environment-modules which provide a module implementation that is widely used in the HPC community. In environment-modules modulefiles are written Tool Command Language (TCL), while Lmod supports both TCL and Lua modules.

Spack provides a mechanism to generate modules in TCL and Lua format based on the module system. On Cori we use environment-modules which support TCL based modules. During the module generation process, we inform spack on the format of the modulefile. We avoid hash in modules and prefer having modules in the format `{name}/{version}-{compiler.name}-{compiler.version}` which avoids module conflicts when a package like hdf5@1.10.7 is installed with both compilers. Shown below is the spack configuration for modules along with output of the spack generated modules. It's worth noting we limit our module generation to root specs and avoid generating modules for dependencies by setting `blacklist_implicits: true` which avoids explosion of modules and higher likelihood of module conflicts. The `hash_length` configures the number of hash characters to append to each modulefile. For every modulefile we set a conflict on the same name which adds a keyword `conflict` to each modulefile such that one can't load two instances at same time.

```bash
modules:
  enable:
    - tcl
  tcl:
    blacklist_implicits: true
    hash_length: 0
    naming_scheme: '{name}/{version}-{compiler.name}-{compiler.version}'
    all:
      conflict:
        - '{name}'
      environment:
        set:
          {name}_ROOT: '{prefix}'
    darshan-runtime:
      conflict:
        - 'darshan'
    darshan-util:
      conflict:
        - 'darshan'
    projections:
      all: '{name}/{version}-{compiler.name}-{compiler.version}'
```

**FIGURE 11.** Spack configuration for module generation and output of generated modules.
We provide an overarching modulefile to load the specific e4s stack version that corresponds to each release of E4S. For instance on Cori we have three versions of e4s as shown below. The modulefile will set up a spack instance used for deployment and update MODULEPATH with spack generated modules.

![FIGURE 12. E4S modulefile used for accessing E4S stack](image)

During production deployment we select the location where spack will install the software and modules which are defined using `install_tree` and `module_root`. We install software on a shared file system that is accessible on both login and compute nodes, in this case the root directory `/global/common/software/spackecp` is available for us to perform E4S deployment. We organize each e4s deployments by version to support multiple releases.

**Lessons Learned:** What's in a name? Just keep it short and sweet.

We like to keep module names as short as possible, this means we don’t include hash names during module generation which is the default behavior when spack generates modules. In our first deployment of E4S, we included hash names as shown below. This can be inconvenient for users as they have a very long modulename that they need to load such as `module load adios2/2.6.0-intel-19.1.2.254-n4dtk4qs` to load the adios2 package.

```bash
siddiq90@cori12> module av e4s
```

```bash
--------- /global/common/software/nersc/cle7/extra_modulefiles -e4s/20.10 e4s/21.02 e4s/21.05
```

Since the e4s/21.02 deployment and all future releases we don’t have any hash in modulefile naming. In the module generation step, we run `spack module tcl refresh` to build the TCL modules. During this process we can run into module conflicts which require unique module names. For example, `warpx/21.05` package has three modulefiles for the same version which require unique module names.

```bash
siddiq90@cori12> module av warpx
```

```bash
--------- /global/common/software/spackecp/e4s-21.05/modules/cray-cnl7-haswell/
```

This is because we have three instances of the warpx package that was installed with variants `dims=2`, `dims=3` and `dims=rz` so we have three unique module names to support all of the instances. Shown below are the warpx packages installed from our e4s/21.05 deployment.
If you are deploying multiple software stacks like E4S, it’s a good idea to keep your software stack behind a meta-module like e4s/21.02 as we did for our E4S deployment. With this approach, we are able to support multiple E4S deployments at same time which has the following benefits

- Minimize output of module avail at startup modules, one has to load the e4s module
- Site administrators can easily deprecate stack by removing modulefile and also adding notice in modulefile
- Avoid Name/Version conflicts in modulefile across different directories in MODULEPATH (i.e two modulefiles called gcc/9.3.0)
- Users will be forced to run module load e4s/<version> to access stack followed by loading some package (i.e module load petsc) compared to just module load petsc can cause scripts to break if site-administrators update the version or remove modulefile.

Step 4 - Deployment Script

The deployment process for e4s/21.02 was initiated through GitLab CI by defining a gitlab job in .gitlab-ci.yml. Shown in Figure 14 is the deploy job which does the production deployment. This process will clone spack into the production path and install specs from buildcache which we did in advance and generate the modulefiles. The deploy job is initiated once the entire stack can be rebuilt from source and pushed to buildcache. The gitlab configuration for e4s/21.02 can be found at https://github.com/spack/spack-configs/blob/main/NERSC/cori/e4s-21.02/.gitlab-ci.yml

![FIGURE 14. Gitlab Deployment Job for e4s/21.02](image-url)
Step 5 - User Documentation

The last step for deployment is writing user documentation for our E4S stack on NERSC systems. Our home page for E4S is [https://docs.nersc.gov/applications/e4s/](https://docs.nersc.gov/applications/e4s/) where we have a separate documentation page per E4S release. The user documentation goes through peer review and further testing to ensure documentation is accurate. Shown below is a preview of our E4S documentation at NERSC, we have a subpage with documentation for each E4S stack.

![Figure 15. NERSC E4S Documentation](image)

**Extreme-scale Scientific Software Stack (E4S)**

The Extreme-scale Scientific Software Stack (E4S) is a curated software stack from the Spack ecosystem that is continuously built and tested on several pre-exascale systems. E4S is composed of many open-source projects, including xSDK, dev-tools, math-libraries, compilers, and more. For a complete product list see E4S Product Dictionary.

E4S is shipped as a container (Docker, Singularity, Shift, CharlieCloud), a buildcache, or a Spack manifest (spack.yaml). Currently, we focus on building E4S from source using a `spack.yaml` file provided by the E4S team from [https://github.com/ES-Project/e4s](https://github.com/ES-Project/e4s).

![Note]

We install as many packages from E4S provided in `spack.yaml` as possible. Some packages were intentionally skipped such as those tied to development branches or packages built for GPUs. Some additional packages couldn’t be installed or had compilation issues with our base compiler.

**E4S Support Timeline**

This table outlines the support lifetime for each E4S version. The Release Date is marked on the day of E4S release when user documentation was live. Once E4S version has reached end of support we will remove E4S and corresponding modulefiles and user documentation. As we approach the End of Support for a particular release, we will communicate via email and modulefile will include a banner when loading the module.

<table>
<thead>
<tr>
<th>System</th>
<th>Version</th>
<th>Release Date</th>
<th>End of Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cori</td>
<td>20.10</td>
<td>Jan 15th 2021</td>
<td>Mar 16th 2022</td>
</tr>
<tr>
<td>Cori</td>
<td>21.02</td>
<td>Jan 11th 2021</td>
<td>Mar 16th 2022</td>
</tr>
<tr>
<td>Perlmutter</td>
<td>21.11</td>
<td>Jan 22nd 2021</td>
<td>Mar 31st 2022</td>
</tr>
<tr>
<td>Cori</td>
<td>22.02</td>
<td>Mar 16th 2022</td>
<td>End of Life Cori (2023)</td>
</tr>
</tbody>
</table>

Recently, we outlined a support timeline for each E4S stack in order to deprecate older stacks in preference of newer versions. This process will entail removal of modulefile, user documentation and uninstall the software stack from the filesystem. Shown below is a preview of our E4S Support Timeline.
E4S Support Timeline

This table outlines the support lifetime for each E4S version. The Release Date is marked on the day of E4S release when user documentation was live. Once E4S version has reached end of support we will remove E4S and corresponding modulefiles and user documentation. As we approach the End of Support for a particular release, we will communicate via email and modulefile will include a banner when loading the module.

We recommend users to port their application and any scripts to the newer release.

<table>
<thead>
<tr>
<th>System</th>
<th>Version</th>
<th>Release Date</th>
<th>End of Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cori</td>
<td>20.10</td>
<td>Jan 15th 2021</td>
<td>Mar 16th 2022</td>
</tr>
<tr>
<td>Cori</td>
<td>21.02</td>
<td>Jun 11th 2021</td>
<td>Mar 16th 2022</td>
</tr>
<tr>
<td>Cori</td>
<td>21.05</td>
<td>Aug 23rd 2021</td>
<td>Oct 31st 2022</td>
</tr>
<tr>
<td>Perlmutter</td>
<td>21.11</td>
<td>Jan 22nd 2022</td>
<td>Mar 31st 2023</td>
</tr>
<tr>
<td>Cori</td>
<td>22.02</td>
<td>Mar 16th 2022</td>
<td>End of Life Cori (2023)</td>
</tr>
</tbody>
</table>

FIGURE 16. E4S support timeline for each release on NERSC systems

Step 6- Give back to the community

We contribute back our spack configuration to [https://github.com/spack/spack-configs](https://github.com/spack/spack-configs) in addition we update the E4S Facility Dashboard as shown below. We want other HPC centers to contribute to this page ([https://e4s.readthedocs.io/en/latest/facility_e4s.html](https://e4s.readthedocs.io/en/latest/facility_e4s.html)) as they deploy E4S on their system so we can see where E4S has been deployed. The HPC community can benefit by seeing how other centers have deployed E4S by sharing their spack configuration.
FIGURE 17. Summary of E4S deployments at DOE facilities available in E4S Documentation

We will communicate our E4S deployment release with our NERSC and ECP user-base through NERSC weekly emails and slack channel. This way we can coordinate efforts across the various ASCR facilities and share best practices.
Recent Developments

E4S development and deployment at the ASCR facilities moves quickly. Here are details on some of the latest developments.

Building E4S on Perlmutter

We will discuss some of our experiences building E4S on Perlmutter. In Oct 2021 we started the process of building the most recent version of e4s at that time. We picked e4s/21.11 as the preferred version which was released in Nov 2021. During this period we encountered several changes to the Cray Programming Environment (CPE) over the span of 3 months with CPE 21.08, 21.10, 21.11 and 21.12. These changes impact our compiler and package preference in our spack configuration. We went through 3 rebuilds of Perlmutter E4S/21.11 over the span of three months with one rebuild being performed in April 2022.

Our compiler choice for Perlmutter is gcc and nvhpc compiler and cray-mpich as our preferred MPI provider. One of the pain points was having to update the package external for cray packages such as cray-mpich. For instance, we noticed that CPE 21.10 had cray-mpich version 8.1.10 but CPE 21.12 provided version 8.1.12. Furthermore, we also had `cuda` modules which were changed to `cudatoolkit` which were provided by NERSC staff since we didn’t have a standalone cuda module.

<table>
<thead>
<tr>
<th>cray-mpich</th>
<th>cray-mpich</th>
</tr>
</thead>
<tbody>
<tr>
<td>buildable: false</td>
<td>buildable: false</td>
</tr>
<tr>
<td>externals:</td>
<td>externals:</td>
</tr>
<tr>
<td>- spec: cray-mpich@8.1.10 %nvhcc@21.7</td>
<td>- spec: cray-mpich@8.1.12 %gcc@9.3.0</td>
</tr>
<tr>
<td>modules:</td>
<td>modules:</td>
</tr>
<tr>
<td>- cuda/11.3.0</td>
<td>- cudatoolkit/21.0.11.4</td>
</tr>
<tr>
<td>- spec: cray-mpich@8.1.10 %gcc@9.3.0</td>
<td>- spec: cray-mpich@8.1.12 %nvhcc@21.9</td>
</tr>
<tr>
<td>modules:</td>
<td>modules:</td>
</tr>
<tr>
<td>- cray-mpich/8.1.12</td>
<td>- cudatoolkit/21.9.11.4</td>
</tr>
<tr>
<td>- cuda/11.3.0</td>
<td>- cuda/11.3.0</td>
</tr>
<tr>
<td>version: [11.3.0]</td>
<td>version: [11.3.0]</td>
</tr>
<tr>
<td>externals:</td>
<td>externals:</td>
</tr>
<tr>
<td>- spec: cuda@11.3.0</td>
<td>- spec: cuda@11.4.0</td>
</tr>
<tr>
<td>prefix: /global/common/software/nersc/oscl3/cuda/11.3.0</td>
<td>prefix: /opt/nvidia/hpc_sdk/Linux_x86_64/21.9/cuda/11.4</td>
</tr>
<tr>
<td>modules:</td>
<td>modules:</td>
</tr>
<tr>
<td>- cuda/11.3.0</td>
<td>- cudatoolkit/21.9.11.4</td>
</tr>
</tbody>
</table>

**FIGURE 18.** Package preference for cray-mpich and cuda on Perlmutter for CPE 21.10 and 21.12

Cray provides an NVHPC compiler provided by NVIDIA which typically comes with 3 versions of cuda in the same distribution. We ended up writing modulefiles for each cuda version mapping to the NVHPC compiler. For instance `cudatoolkit/21.3_10.2` refers to cuda version 10.2 from the NVHPC 21.3 compiler.
Most recently, we removed older versions of cudatoolkit modulefile and changed the modulefile name format to exclude nvhpc version. These changes impacted our spack build and any system changes like changes to modulefile need to be synced with our spack configuration. In addition we recently added NVHPC 21.11 with intent of removing 21.9 in near future, so we are now planning to rebuild with the latest compiler. Our initial deployment used gcc@9.3.0 compiler and nvhpc@21.9 however, gcc/9.3.0 modulefile was also removed, so now we are planning to use gcc/11.2.0. Shown below is our compiler definition on Perlmutter, with the left image showing our first iteration and image on right showing our updated compilers which we plan to use for our upcoming redeployment.

```bash
siddig@001login31> ml -t av cudatoolkit
/opt/cray/pe/1mod/modulefiles/core:
cudatoolkit/21.3_10.2
cudatoolkit/21.3_11.0
cudatoolkit/21.3_11.2
cudatoolkit/21.9_10.2
cudatoolkit/21.9_11.0
cudatoolkit/21.9_11.4
/opt/modulefiles:
cudatoolkit/21.3_10.2
cudatoolkit/21.3_11.0
cudatoolkit/21.3_11.2
cudatoolkit/21.9_10.2
cudatoolkit/21.9_11.0
cudatoolkit/21.9_11.4
```

**FIGURE 19. Compiler definition of gcc and nvhpc compiler on Perlmutter**

On Jan 22, 2022 we released E4S/21.11 on Perlmutter which was our first deployment of E4S that was based on CPE 21.12. The initial release consisted of 94 packages, we provided TCL and Lua based modules generated by spack. We created a user-facing modulefile `e4s/21.11-tcl` and `e4s/21.11-lmod` that
can be used to access the same software stack but the main difference being the way modulefiles were presented. For more details on this stack see https://docs.nersc.gov/applications/e4s/perlmutter/21.11/.

In parallel to our standard deployment, we provide a containerized deployment of E4S/21.11 on Perlmutter which is a container image provided by the E4S team as part of their release process. The base image is an Ubuntu container using gcc@9.3.0 compiler, we provide this as an alternative to our software stack.

Lessons Learned: The first time on any system can be challenging - have patience

Perlmutter has gone through several changes in the past several months including upgrades to new CPE, we anticipate a few rebuilds will be required until Perlmutter is stable which is expected when bringing a new system into production.

The E4S/21.11 is based on spack version 0.17 which had some significant changes including the clingo concretizer being the default going forward. This affected spack since it now required additional dependency to be installed during the bootstrapping process. We first encountered that spack was unable to bootstrap clingo on Perlmutter so we reported the issue https://github.com/spack/spack/issues/28315 to spack project to get this resolved. Our current workaround was to install clingo via pip in order to satisfy the dependency.

There was a bug in spack in how system detection worked https://github.com/spack/spack/issues/25914 which impacted how we do builds, we were unable to use spack command on Perlmutter which was a serious issue and this issue was addressed promptly by spack team.

Automation

Recently we started a project to centralize our spack configuration and automate our spack deployments. We leverage Gitlab to automate our deployments using scheduled pipelines to perform full source builds of all of our spack stacks. This project is called spack-infrastructure and located at https://software.nersc.gov/NERSC/spack-infrastructure.

We have set up a public facing repo on Github at https://github.com/NERSC/spack-infrastructure which is a mirror of the original repo. In addition, we have user documentation available at https://nerse-spack-infrastructure.readthedocs.io/en/latest/
We have configured a few scheduled pipelines that perform full source builds of our E4S for various systems, these scheduled pipelines will mimic our production deployment but run in a unique directory per CI job.

We have configured gitlab runners to run CI jobs on Cori, Perlmutter and our test systems so we can perform builds on all of our systems where E4S will be deployed. Our spack builds are performed using a single user account which avoids issues with differences between user environments.

Recently, we started building E4S using the spack `develop` branch which contains the bleeding edge of the spack codebase where incoming PRs get merged. We build this stack on a weekly basis which will build
the latest for each software product as new versions are added in spack codebase. These stacks are accessible via modules named **e4s/spack-develop** where we expose users with a spack instance.

```
$ e4s:login28> ml e4s/spack-develop
Welcome to E4S stack built with spack develop

This stack is rebuilt regularly with tip of spack 'develop' branch which means packages will change over time. You can access the log files via $SPACK_INSTALL_LOG, $SPACK_CONCRETIZE_LOG, $SPACK_GITLAB_LOG. You can view the spack configuration (spack.yaml, spack.lock) via environment $SPACK_YAML and $SPACK_LOCK.

For more information regarding this stack see https://software.lbl.gov/NERSC/spack-infrastructure.
```

We plan to leverage this stack as feedback into our E4S deployments and gain insight into what packages can build successfully in future deployments. Take for instance our spack develop stack for Perlmutter has deployed most recent versions of *kokkos*@3.5.00 whereas our most recent deployment (e4s/21.11) contains *kokkos*@3.4.01. We have a high degree of confidence that packages installed via spack develop pipeline will most likely build in our future E4S deployments and this will ease our deployment process since this work is done in advance.

```
$ e4s:login28> spack find --format "{name}@{version} %{compiler.name}@{compiler.version}" kokkos
kokkos@3.5.00 %gcc@11.3.0
kokkos@3.5.00 %gcc@13.0
 kokkos@3.5.00 %gcc@11.3.0

$ e4s:login28> ml e4s/21.11-tcl

The following have been reloaded with a version change:
1) e4s/spack-develop => e4s/21.11-tcl

```

We recently deployed e4s/22.02 on Cori which followed a major system OS upgrade. The E4S deployment contained 385 installed specs, the most we have built so far, and the entire deployment was complete within 2 weeks. This stack was built with gcc@11.2.0 and intel@19.1.2.254, shown below is a breakdown of specs by each compiler.

```

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Root Specs</th>
<th>Implicit Specs</th>
<th>Total Specs</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc@11.2.0</td>
<td>70</td>
<td>166</td>
<td>236</td>
</tr>
<tr>
<td>intel@19.1.2.254</td>
<td>57</td>
<td>92</td>
<td>149</td>
</tr>
<tr>
<td>Total</td>
<td>127</td>
<td>258</td>
<td>385</td>
</tr>
</tbody>
</table>

```

We plan on supporting this release till the end of Cori lifetime (2023) and be our last deployment of E4S on Cori.

**FIGURE 22. Breakdown of installed specs by compilers for e4s/22.02**
MPI Support

We are working with the MVAPICH2 team from Ohio State University to experiment with mvapich2 as an MPI provider for building the E4S stack on Perlmutter. The mvapich2-gdr is an optimized version of mvapich that takes advantage of GPU Direct RDMA technology to improve inter-node data movement on NVIDIA GPUs which is relevant for Perlmutter since we support NVIDIA A100 GPUs. Currently, we are using cray-mpich as our MPI provider which is available on our system but we have run into build errors with certain packages which expect mpi wrapper mpicc instead of cc. We plan on using cray-mpich as the MPI provider for building the stack and introduce mvapich2 for building a subset of packages for future e4s release. The collaboration between the E4S and the NERSC teams has helped install MVAPICH2 and 87 packages with 575 total installed specs from E4S 22.02 as shown in the figures below. These packages use mvapich2-gdr configured with SLURM and CUDA 11.5 on Perlmutter. The total time for installation of these packages was less than one day! The E4S packages may be accessed using the module or spack commands as shown below.

FIGURE 23. How to access e4s/22.02 stack built with mvapich2-gdr
Besides bare-metal installation of E4S, NERSC also supports Shifter, a mature HPC container runtime and both base and full-featured E4S images are installed on Perlmutter. These images contain 100+ HPC and AI/ML packages (such as TensorFlow/PyTorch) with total 621 specs and support the A100 GPUs. Shifter provides a viable deployment option for E4S where only one image needs to be downloaded and is immediately available to all the users. They may continue to use module or spack commands to access the...
packages, as shown in the figure below. Using E4S base or full-featured containers, users may build their own compact, custom configured containers to deploy on any HPC system.

FIGURE 25. Using E4S 22.02 container with Shifter on Perlmutter

The use of conda and the cuda environment in Shifter to use TensorFlow and PyTorch packages with support for A100 GPU on Perlmutter is shown in the figure below.
FIGURE 26. Running TensorFlow from Shifter container

Testing E4S Post Deployment

We test our E4S stack post deployment, we utilize **buildtest** a testing framework to build and run tests on HPC systems. We utilize gitlab to run a subset of E4S tests on Cori and Perlmutter for each of our E4S stacks along with the shifter based container. Our test can be found on our NERSC gitlab server at [https://software.nersc.gov/NERSC/buildtest-nersc](https://software.nersc.gov/NERSC/buildtest-nersc). Shown below is a listing of scheduled pipelines that will run a subset of tests at different schedules. Since we have over 200+ tests we can’t run all of them at once, but instead we run at different intervals.
FIGURE 27. Scheduled pipeline for testing E4S stack

buildtest will publish results to CDASH upon completion of all tests, shown below is an output from the gitlab job that runs E4S tests on the Perlmutter system. Buildtest will show a link to CDASH report file which can be viewed in your browser.

The CDASH report will contain metadata for each test such as name of test, test description, hostname, start and endtime, test duration. CDASH will report the test failures in **RED**.
FIGURE 28. CDASH output for E4S runs on Perlmutter

Shown below is an output from one of our test which will validate trilinos package by testing Zoltan from our shifter container on Perlmutter using two nodes. This test will calculate Preconditioned Conjugate Gradient for problem Epetra::VbrMatrix which will run for 20 iterations.
FIGURE 29. Output for Trilinos Zoltan test in Shifter container

Shown below is the generated build script and test script by buildtest. The test will utilize image `ecpe4s/ubuntu20.04-gpu-x86_64:21.11` which is E4S 21.11 stack built with GPU support. The Zoltan test is available in E4S Testsuite ([https://github.com/E4S-Project/testsuite](https://github.com/E4S-Project/testsuite)), test will allocate 2 nodes with 4 GPUs and run the test from shifter container via `srun`. In order to run the Zoltan trilinos test, we need to load trilinos via `spack load trilinos`.  

```
- For setup - 0.540212 (s)
- For hierarchy setup = 0.538312 (s)
- For smoothers setup = 0.000122315 (s)
- For coarse setup = 0.00076673 (s)
- For final setup = 3.6178e-05 (s)
Total for this setup = 0.519751 (s)
```
Conclusion

Managing an HPC software environment can be a challenging and time-consuming process for any HPC center. Deploying a software stack requires intimate knowledge of the HPC system with in-depth knowledge of the software packages to ensure each package is built optimally for the system. E4S accelerates the development, deployment and use of HPC software, lowering the barriers for HPC users. The E4S software stack community effort creates policies and necessary infrastructure to more easily and quickly deploy software at extreme-scale.

The sheer size of the E4S deployment and the constant upgrades in cutting-edge HPC system technology requires tight integration with HPC facility staff and across the community. The continued success and development of E4S and similar efforts will need to additionally emphasize building the community of support to maintain longevity and impact of the software. In particular, workforce development and community building:

**Workforce Development:** The software deployment team is an integral part of HPC centers, and more focused efforts are needed towards training our existing staff and/or increasing the workforce to support initiatives like E4S at the facilities. An HPC center may have multiple HPC systems and if one wants to deploy E4S for every system we should work towards a sustainable solution where we can deploy E4S relatively quickly while having additional resources so work can be done in parallel. Across the three DOE labs (NERSC, OLCF, ALCF) we noticed that the software deployment group is led by 1-2 individuals who are responsible for building the entire software stack for multiple HPC systems. Since E4S leverages spack as the driver for building E4S stack, this means staff also need spack expertise and a strong sense of how to design software stacks and interface through modules.

**Community Building:** HPC centers can benefit from each other by sharing best practices in the software deployment process, especially for centers that don’t have a well-established process or are trying to deploy an E4S stack for the first time. This report is an effort to share more detailed deployment process information with the community - a behind the scenes look. We encourage others to do similarly.
How to Get Involved

There are several ways to get involved to better support E4S at the facilities.

**Are you an application developer for an E4S product that we install? We need your assistance in troubleshooting the build errors.** Each facility wants to accelerate the software deployment process and provide as many software products as possible to satisfy the user's needs. However, we need your help to debug build failures during our deployment. You will need access to NERSC resources along with our gitlab project [https://software.nersc.gov/NERSC/spack-infrastructure/](https://software.nersc.gov/NERSC/spack-infrastructure/) which contains our spack configuration along with build logs and current issues that need to be addressed. We will try to post these issues in [spack issue tracker](https://software.nersc.gov/NERSC/spack-infrastructure/) to get more visibility. We can get you set up!

**Are you an application developer or user of an E4S software package? We seek guidance on package variants when building a package.** Take for instance, trilinos which comes with several dozen variants

```
trilinos +amesos +amesos2 +anasazi +aztec +belos +boost +epetra +epetraext +ifpack
 +ifpack2 +intrepid +intrepid2 +isorropia +kokkos +ml +minitensor +muelu +nox
 +piro +phalanx +rol +rythmos +sacado +stk +shards +shylu +stokhos +stratinikos
 +teko +tempus +tpetra +trilinoscouplings +zoltan +zoltan2 +superlu-dist gotype=long_long
```

We are unsure if all of these variants are appropriate for our system. Some of these selections were provided by E4S which we incorporated in our spack configuration. We leverage multiple compilers for building E4S stack including gcc, cce, nvhpc, and intel and we would suggest application teams to provide feedback into our decision process since we may choose incompatible compilers or compiler versions.

**Are you an application developer for an E4S software package that we install? We seek your guidance in testing the software on our system.** We are trying to increase test coverage for our e4s deployment by having at least 1-2 sanity tests that can test the software product to increase confidence. You will need access to NERSC resources and our gitlab server [https://software.nersc.gov/NERSC/buildtest-nersc](https://software.nersc.gov/NERSC/buildtest-nersc) with all of our E4S tests. We encourage user contribution to help sustain this effort. Our focus is to test the software provided by E4S stack which will be accessible via **module load e4s**. Currently we are trying to add tests for the latest E4S release with an emphasis on developing tests for Perlmutter.
Acknowledgement

This work was supported by the Office of Science, Office of Advanced Scientific Computing Research of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

We gratefully acknowledge Hai Ah Nam (LBNL, IDEAS-ECP Better Scientific Software Fellowship Coordinator) for her many edits and guidance on this report and the feedback from the ECP E4S collaboration and NERSC staff.
Bio

Shahzeb Siddiqui

Shahzeb Siddiqui is a HPC Consultant/Software Integration Specialist at Lawrence Berkeley National Laboratory at NERSC. He is part of the User Engagement Team that is responsible for engaging with NERSC user community through user support tickets, user outreach, training, documentation. Shahzeb is part of the Exascale Computing Project (ECP) in Software Deployment (SD) group where he is responsible for building Spack Extreme-Scale Scientific Software Stack (E4S) at the DOE facilities. He is the creator of few open source projects including buildtest, lmodule and jobstats. Shahzeb has experience installing and managing large software stack, managing HPC clusters including cluster managers (Bright Cluster Manager, Cobbler) and configuration management tools such as Ansible.

Shahzeb Siddiqui started out his career in High Performance Computing (HPC) in 2012 at King Abdullah University of Science and Technology (KAUST) while pursuing his Masters. His focus in HPC includes Parallel Programming, Performance Tuning, Containers (Singularity, Docker), Linux system administration, Scientific Software Installation and testing, Scheduler Optimization, and Job Metrics. Shahzeb has held multiple roles in his HPC career in the following companies: Dassault-Systemes, Pfizer, Penn State, and IBM. Prior to 2012, he was a software engineer holding multiple roles at Global Science & Technology, Northrop Grumman, and Penn State.

Sameer Shende

Dr. Sameer Shende has helped develop the TAU Performance System, the Program Database Toolkit (PDT), the Extreme-scale Scientific Software Stack (E4S) and the HPCLinux distro. His research interests include tools and techniques for performance instrumentation, measurement, analysis, runtime systems, HPC container runtimes, and compiler optimizations. He serves as a Research Associate Professor and the Director of the Performance Research Laboratory at the University of Oregon, and as the President and Director of ParaTools, Inc., ParaTools, SAS, and ParaTools, Ltd. He leads the SDK project for the Exascale Computing Project (ECP), in the Programming Models and Runtime (PMR). He received his B.Tech. in Electrical Engineering from IIT Bombay, and his M.S. and Ph.D. in Computer and Information Science from the University of Oregon.