Increasing Software Testing Coverage and Portability with Spack

Jon Rood and Shreyas Ananthan

National Renewable Energy Laboratory

What is Spack?

- Spack[1] is a package manager spawned out of Lawrence Livermore National Laboratory
- Designed for use on supercomputers
- Original focus on combinatorial building of software

Spack

- Spack is a knowledgebase for automating building of scientific applications
- Spack is also a domain specific language (DSL) for building and managing software
- Due to its popularity, we can leverage this to assemble testing environments much easier than previously possible
  - Currently contains more than 3000 built-in packages

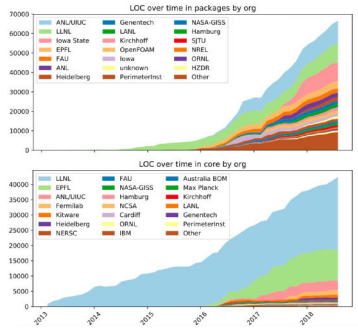

Fig. 1: Spack lines of code over time.[2]

Nalu-Wind Example

- Nalu-Wind[3] is an application developed under the Exascale Computing Project (ECP) for modeling wind farms with blade-resolved wind turbines

Fig. 2: Nalu-Wind simulation for blade-resolved wind turbine.

- Nalu-Wind is tested against multiple machines, operating systems, compilers, code branches, and optional features by exploiting Spack to prepare each testing environment

The Spack Spec Syntax

- Spack provides a powerful command line interface
- Customize installs on the command line

```
spack install mpileaks                  # unconstrained
spack install mpileaks@3.3              # @custom version
spack install mpileaks@3.3 %gcc@7.3.0   # %custom compiler
spack install mpileaks@3.3 %gcc@7.3.0 +threads  # +/~ option AKA variant
spack install mpileaks@3.3 cppflags="-O3 -g3"   # custom compiler flags
spack install mpileaks@3.3 os=CNL10 target=haswell  # cross compile
spack install mpileaks@3.3 ^mpich@3.2   # ^ custom dependencies
spack <command> /a4947f                 # use spec hash
```

Fig. 3: Examples of *specs* defined on the command line.[2]

- Each expression resolves to a unique *spec*
  - Each clause adds a constraint
  - Constraints are *optional*
- Spec syntax is recursive
- *Concretization* fills in implicit details
- Concretized specs reduce to a unique hash

Spack Advantages

- Portable Python framework for fulfilling dependencies
- Easily affect the entire stack by building everything with:
  - Particular compiler, particular flags, different library versions, specified application options

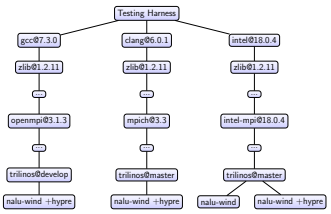

Fig. 4: Example dependency tree of environments generated for testing.

- Automatic rpath means binaries work despite environment
- Easily automate and isolate the environment
- Easily test/track/update dependencies
- Easily query Spack's database for dependency locations
- Spack operations able to execute concurrently
  - Single package building is parallel by default

Nalu-Wind Stack

- Nalu-Wind depends on a *large* software stack
  - Trilinos, YAML-CPP, HDF5, NetCDF, Boost, MPI, SuperLU, OpenFAST, FFTW, TIOGA, HYPRE, Paraview Catalyst, etc...

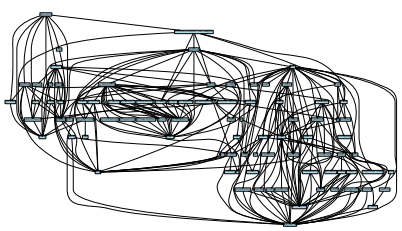

Fig. 5: Nalu-Wind dependency graph with all optional libraries.

- We rely on existing knowledge in Spack for fulfilling our application's dependencies
- Developing the recipe for building our Nalu-Wind application using Spack's DSL can be quite succinct as shown below

The Spack DSL

- Nalu-Wind uses CMake as its low-level build system

```
from spack import *
class NaluWind(CMakePackage):
    """Nalu-Wind: Wind energy focused variant of Nalu."""
    homepage = "https://github.com/exawind/nalu-wind"
    git      = "https://github.com/exawind/nalu-wind.git"

    version('master', branch='master')

    variant('hypre', default=False,
            description='Compile with Hypre support')

    depends_on('mpi')
    depends_on('trilinos@master,develop')
    depends_on('hypre+mpi+int64', when='+hypre')

    def cmake_args(self):
        options = []
        options.extend([
            '-DTrilinos_DIR:PATH=%s' % self.spec['trilinos'].prefix,
            '-DCMAKE_CXX_COMPILER=%s' % self.spec['mpi'].mpicxx,
        ])
        if '+hypre' in self.spec:
            options.extend([
                '-DENABLE_HYPRE:BOOL=ON',
                '-DHYPRE_DIR:PATH=%s' % self.spec['hypre'].prefix
            ])
        return options
```

Listing 1: An abbreviated Spack recipe (package.py) for Nalu-Wind.

- Recipe development is easiest if your application uses CMake, Autotools, or GNU Make build system

Nalu-Wind Testing Harness

- Easy to use nested loop for entire configuration matrix
- Probably don't want to test *every* configuration
- Better to parse a list of configurations

```
declare -a CONFIGURATIONS
# CONFIGURATION: COMPILER_ID, TRILINOS_BRANCH, MPI_ID, NALU_OPTS
CONFIGURATIONS[0]='gcc@7.3.0:develop:openmpi@3.1.3:+hypre'
CONFIGURATIONS[1]='clang@6.0.1:master:mpich@3.3:+hypre'
CONFIGURATIONS[2]='intel@18.0.4:master:intel-mpi@18.0.4:'
```

Listing 2: Example list of test configurations in bash script.

- Spack can create independent software stack environments concurrently

```
# Test Nalu-Wind for the list of configurations asynchronously
for CONFIGURATION in "${CONFIGURATIONS[@]}"; do
  CONFIG=(${CONFIGURATION//:/ })
  COMPILER_ID=${CONFIG[0]}
  TRILINOS_BRANCH=${CONFIG[1]}
  MPI_ID=${CONFIG[2]}
  NALU_OPTS=${CONFIG[3]}
  (test_configuration) &
done
```

Listing 3: Main loop in bash for testing configurations concurrently.

- Spack has many lower level commands we are able to exploit to orchestrate each step of the creation of our testing environment

```
test_configuration() {
  # Uninstall any dependencies we are tracking
  spack uninstall -a -y trilinos@${TRILINOS_BRANCH} %${COMPILER_ID}
  # Stage unique nalu-wind repo for this configuration
  spack stage nalu-wind ${NALU_OPTS} %${COMPILER_ID} ^${MPI_ID}
  # Update git repos of nalu-wind and dependencies we are tracking
  spack cd trilinos@${TRILINOS_BRANCH} %${COMPILER_ID} ^${MPI_ID} && \
    git fetch --all && git reset --hard origin/${TRILINOS_BRANCH} && \
    git clean -df
  spack cd nalu-wind ${NALU_OPTS} %${COMPILER_ID} ^${MPI_ID} && \
    git fetch --all && git reset --hard origin/master && \
    git clean -df
  # Install all dependencies for our application for this configuration
  spack install --dont-restage --keep-stage --only dependencies nalu-wind \
    ${NALU_OPTS} %${COMPILER_ID} ^trilinos@${TRILINOS_BRANCH} ^${MPI_ID}
  # Load the required executable dependencies into our environment
  spack load ${MPI_ID} %{COMPILER_ID}
  # Query locations of libraries to pass to our application
  TRILINOS_DIR=$(spack location -i trilinos@${TRILINOS_BRANCH} \
    %{COMPILER_ID} ^${MPI_ID})
  # Run test script that builds, tests, and reports results
  ctest -S CTestNightlyScript.cmake -DTRILINOS_DIR=${TRILINOS_DIR} \
    -DNALU_OPTS=${NALU_OPTS}
  # Unload executable dependencies from our environment
  spack unload ${MPI_ID} %{COMPILER_ID}
}
```

Listing 4: Function in bash for testing a single configuration.

Nalu-Wind Results

- Using Spack we are able to test our Nalu-Wind application, which has a large software stack, portably and efficiently, with much more coverage across environments than previously possible without Spack

Fig. 6: Test results generated nightly for Nalu-Wind using CDash.

- In the future we would like to:
  - Parallelize building of independent packages in the DAG
  - Parallelize building of packages and testing across nodes
  - Extend test coverage to GPU architectures

References

[1] T. Gamblin, M. P. LeGendre, M. R. Collette, G. L. Lee, A. Moody, B. R. de Supinski, and W. S. Futral, "The spack package manager: Bringing order to hpc software chaos," in *Supercomputing 2015 (SC'15), Austin, Texas*, LLNL-CONF-669890, November 2015.

[2] T. Gamblin, G. Becker, M. Legendre, M. Melara, and P. Scheibel, "Spack tutorial," in *ECP Annual Meeting 2019*, (Houston, TX, USA), January 2019.

[3] Nalu-Wind. https://github.com/exawind/nalu-wind, 2019.

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering, and early testbed platforms, in support of the nation's exascale computing imperative.

This work was authored in part by the National Renewable Energy Laboratory, operated by Alliance for Sustainable Energy, LLC, for the U.S. Department of Energy (DOE) under Contract No. DE-AC36-08GO28308. The views expressed in the article do not necessarily represent the views of the DOE or the U.S. Government. The U.S. Government retains and the publisher, by accepting the article for publication, acknowledges that the U.S. Government retains a nonexclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this work, or allow others to do so, for U.S. Government purposes.

SIAM CSE 2019
Spokane, Washington Feb. 25 - Mar. 1, 2019
NREL/PO-2C00-73335