Adoption and Usage of Spack in ALEGRA DevOps and Development

Presented by:
Tim Fuller
Alegra

Summary

Alegra is a roughly 34-year old code that provides approximate solutions to multiphysics problems involving

- large-deformation Lagrangian, Eulerian, or ALE solid dynamics/hydrodynamics;
- electrical conductivity, magnetic induction/diffusion, nonlinear ohmic heating, Lorentz forces;
- finite element discretizations;
- material data and equations of state;
- radiation transport, thermonuclear burn; and
- piezo and ferroelectric effects.
Challenges

**Code base**
- roughly 34 year old “legacy code”
- large code base with C++, Fortran, C, and other language components
- extremely complex physics

**Dependencies**
- complex dependencies: roughly 30 TPLs including Dakota, Trilinos, Xyce
- each having its own build system
- some TPLs have proprietary licenses

**Data**
- relies on material data from a variety of sources
- ITAR, UCNI, LANL proprietary, and LLNL proprietary data
- not all customers are authorized to receive data

**Testing**
- most testing done on gifted, and aging, hardware
- thousands of tests with tens of Gb of data
- some tests take longer than 24 hours to execute

**Building**
- maintaining builds on all SNL CEE-LAN and HPC machines
- maintaining builds on select SNL test beds
- providing builds on customer machines for which there are no SNL counterparts

**Running**
- complex user interface
- interactions with many other tools: MPI, exodus, etc.
Alegra

- manage and build TPLs;
- manage and build alegranevada source code;
- manage source code testing;
- manage source code releases; and
- define compiler interfaces and compiler flags.

The legacy toolset implements functionality from many modern tools.
Alegra tooling modernizations

toolset2 is a Python library that glues together the pieces of our CI/CD workflow:

- Spack
- VVTest
- GitLab
- CDash
It’s...
Spack all the way down
Adoption strategy, part 1

ONE DOES NOT SIMPLY

SPACK

imgflip.com
Adoption strategy, part 1

- Fork and wrap spack with our toolset, hide as many Spack details from developers

```
$ # setup environment
$ spacktivate
$ spack add ...
$ spack concretize...
$ spack install
```

- Provide default environments and reference area (upstreams)
- Provide spackages for every package in our software stack
- Modified Spack to fit our needs
Adoption strategy, part 1

- Fork and wrap spack with our toolset, hide as many Spack details from developers

```
$ # setup environment
$ spacktivate
$ spack add ...
$ spack concretize...
$ spack install
```

- Provide default environments as reference area (upstreams)
- Provide spackages for every package in our software stack
- Modify spack to fit our needs
Adoption strategy, part 2

For every application that uses Spack, there is a wrapper to wrap Spack

(Chris Siefert)
Adoption strategy, part 2

ONE SHOULD SIMPLY SPACK

*For every application that uses Spack, there is a wrapper to wrap Spack*  

(Chris Siefert)
Adoption strategy, part 2

Don’t wrap!
Adoption strategy, part 2

Don’t wrap!

- Don’t wrap, adapt
Adoption strategy, part 2

Don’t wrap!

- Don’t wrap, adapt
- Don’t wrap, extend
Adoption strategy, part 2

Don’t wrap!

- Don’t wrap, adapt
- Don’t wrap, extend
- Don’t wrap, contribute
Don’t wrap, adapt

Expect developers to read Spack documentation and learn basics of Spack

- Spack spec language
Don’t wrap, adapt

Expect developers to read Spack documentation and learn basics of Spack

- Spack spec language
- `spack find`
Don’t wrap, adapt

Expect developers to read Spack documentation and learn basics of Spack

- Spack spec language
- `spack find`
- `spack info`
Don’t wrap, adapt

Expect developers to read Spack documentation and learn basics of Spack

- Spack spec language
- `spack find`
- `spack info`
- `spack develop`
Don’t wrap, adapt

Expect developers to read Spack documentation and learn basics of Spack

- Spack spec language
- `spack find`
- `spack info`
- `spack develop`
- `spack concretize`
Don’t wrap, adapt

Expect developers to read Spack documentation and learn basics of Spack

- Spack spec language
- `spack find`
- `spack info`
- `spack develop`
- `spack concretize`
- `spack install`
Don’t wrap, adapt

Expect developers to read Spack documentation and learn basics of Spack

- Spack spec language
- `spack find`
- `spack info`
- `spack develop`
- `spack concretize`
- `spack install`

provide 90% of the functionality we need for using Spack
Don’t wrap, extend

Spack extensions allow one to extend Spack with custom commands.

- Originally, we provided additional functionality by stitching together different Spack commands with scripts
- These scripts were fragile and often broke when we updated Spack
- Spack provides a better solution in the form of extensions

```
spack:
  config:
    extensions:
      - $toolset2/var/spack/extensions/spack-nevada
```

```
$ tree $toolset2/spack/extensions
$toolset2/spack/extensions/
  __ spack-nevada
  __ nevada
  __ cmd
    __ __init__.py
    __ distribution.py
    __ make.py
    __ multi_develop.py
    __ pull.py
```
Don’t wrap, extend: multi-develop
Don’t wrap, extend: multi-develop

```bash
$ cd $workdir
$ git clone url:app
$ git clone url:dep1
$ git clone url:dep2
$ spack develop -p $(pwd)/app app@version
$ spack develop -p $(pwd)/dep1 dep1@version
$ spack develop -p $(pwd)/dep2 dep2@version
$ spack add app@version ~dep1@version ~dep2@version
$ spack concretize
$ spack install
```
Don’t wrap, extend: multi-develop

$ spack multi-develop -h
usage: spack multi-develop [-h] [-f FILE] ...

add multiple specs to an environment’s dev-build information

options:
- h, --help show this help message and exit

Input format:
- details colon separated list of details
- f FILE File containing develop specs

‘spack multi-develop’ is a wrapper around ‘spack develop’ that allows adding multiple specs to an environment’s dev-build information.
Don’t wrap, extend: multi-develop

```bash
$ spack multi-develop -h
usage: spack multi-develop [-h] [-f FILE] ...

add multiple specs to an environment’s dev-build information

options:
  -h, --help    show this help message and exit

Input format:
  details      colon separated list of details
  -f FILE      File containing develop specs

‘spack multi-develop‘ is a wrapper around ‘spack develop‘ that allows
adding multiple specs to an environment’s dev-build information.
```

```bash
$ cat specs.yaml
develop:
- app@version
  path: $CWD/app
  clone: true
- dep1@version
  path: $CWD/dep1
  clone: true
- dep2@version
  path: $CWD/dep2
  clone: true

$ spack multi-develop -f specs.yaml
$ spack add app@version ^dep1@version ^dep2@version
$ spack concretize
$ spack install
```
Don’t wrap, extend: make

```bash
cd $(spack location -b app@version)

source spack-build-env.txt

cd $(spack location -b app@version)

make install -j40

cd $(spack location -b app@version)

spack build-env app@version --make install -j40

But what we really want to do is

$ spack make app --install -j40
```
Don’t wrap, extend: make

- Packages marked by `spack develop` can be rebuilt as the local source changes with `spack install`
Don’t wrap, extend: make

- Packages marked by `spack develop` can be rebuilt as the local source changes with `spack install`
- `spack install` can be slow
Don’t wrap, extend: make

- Packages marked by `spack develop` can be rebuilt as the local source changes with `spack install`
- `spack install` can be slow
- can we run `make` in the package’s build directory?
Don’t wrap, extend: make

- Packages marked by `spack develop` can be rebuilt as the local source changes with `spack install`
- `spack install` can be slow
- can we run `make` in the package’s build directory?
- yes! but...
Don’t wrap, extend: make

- Packages marked by `spack develop` can be rebuilt as the local source changes with `spack install`
- `spack install` can be slow
- can we run `make` in the package’s build directory?
- yes! but...

```bash
$ cd $(spack location -b app@version)
$ cd..
$ source spack-build-env.txt
$ cd $(spack location -b app@version)
$ make install -j40
$ cd $(spack location -b app@version)
$ spack build-env app@version --make install -j40
```

But what we really want to do is

```bash
$ spack make app --install -j40
```
Don’t wrap, extend: make

- Packages marked by `spack develop` can be rebuilt as the local source changes with `spack install`
- `spack install` can be slow
- can we run `make` in the package’s build directory?
- yes! but...

```
$ cd $(spack location -b app@version)
$ cd ..
$ source spack-build-env.txt
$ cd $(spack location -b app@version)
$ make install -j40
```
Don’t wrap, extend: make

- Packages marked by `spack develop` can be rebuilt as the local source changes with `spack install`
- `spack install` can be slow
- can we run `make` in the package’s build directory?
  - yes! but...

```bash
$ cd $(spack location -b app@version)
$ cd ..
$ source spack-build-env.txt
$ cd $(spack location -b app@version)
$ make install -j40
```

```bash
$ cd $(spack location -b app@version)
$ spack build-env app@version -- make install -j40
```
Don’t wrap, extend: make

- Packages marked by `spack develop` can be rebuilt as the local source changes with `spack install`
- `spack install` can be slow
- can we run `make` in the package’s build directory?
- yes! but...

```
$ cd $(spack location -b app@version)
$ cd..
$ source spack-build-env.txt
$ cd $(spack location -b app@version)
$ make install -j40
```

But what we really want to do is

```
$ cd $(spack location -b app@version)
$ spack build-env app@version -- make install -j40
```

```
$ spack make app -- install -j40
```
$ spack make -h
usage: spack make [-h] ...

make SPEC directly with ‘make’ or ‘ninja’

positional arguments:
  SPEC          Spack package to build (must be a develop spec)

options:
  -h, --help    show this help message and exit

Additional arguments can be sent to the build system directly by separating them from SPEC by ‘--’. Eg, ‘spack make SPEC -- -j16’
import argparse
import os
import llnl.util.tty as tty
import spack.build_environment as build_environment
import spack.builder
import spack.cmd
import spack.paths
from llnl.util.filesystem import working_dir
from spack.util.executable import Executable

def make(parser, args):
    env = spack.cmd.require_active_env(cmd_name="make")
    try:
        sep_index = args.spec.index("--")
        extra_make_args = args.spec[sep_index + 1 :]
        specs = args.spec[:sep_index]
    except ValueError:
        extra_make_args = []
        specs = args.spec
    make_program = "ninja" if os.path.exists("build.ninja") else "make"
    make = Executable(make_program)
    make(*extra_make_args)
Don’t wrap, extend: distribution

The problem:

- Our source code has very tight access controls
- Many customers are external to Sandia
- Most need to install on their own air-gapped systems
Don’t wrap, extend: distribution

The problem:
- Our source code has very tight access controls
- Many customers are external to Sandia
- Most need to install on their own air-gapped systems

The solution:

```bash
$ spack distribution create  # On my system
$ # scp to target
$ spack distribution install  # on target system
```
Don’t wrap, extend: distribution

The problem:
- Our source code has very tight access controls
- Many customers are external to Sandia
- Most need to install on their own air-gapped systems

The solution:

```
$ spack distribution -h
usage: spack distribution [-h] SUBCOMMAND ...

Create and install alegranevada distributions

positional arguments:
  SUBCOMMAND
    create         Create the AlegraNevada distribution
    install        Install the AlegraNevada distribution
    add-compilers  Find compilers and add them to the AlegraNevada distribution

options:
  -h, --help     show this help message and exit
```
Don’t wrap, contribute

- Contribute changes you require back to Spack
- Spack developers are open to collaborations and helpful in getting modifications incorporated upstream
Questions?