The Evolution of Alegra’s Devops Ecosystem

Tim Fuller

June 21, 2023
Alegra
ALEGRA

Summary

Alegra is a roughly 25-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 ferro electric effects.
## ALEGRA

### Challenges

**Code base**
- roughly 25 year old “legacy code”
- large code base with C++, Fortran, C, and other language components
- extremely complex physics
ALEGRA

Challenges

**Code base**
- roughly 25 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
ALEGRA

Challenges

Code base
- roughly 25 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
ALEGRA

Challenges

<table>
<thead>
<tr>
<th>Code base</th>
<th>Testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>roughly 25 year old “legacy code”</td>
<td>most testing done on gifted, and aging, hardware</td>
</tr>
<tr>
<td>large code base with C++, Fortran, C, and other language components</td>
<td>thousands of tests with tens of Gb of data</td>
</tr>
<tr>
<td>extremely complex physics</td>
<td>some tests take longer then 24 hours to execute</td>
</tr>
</tbody>
</table>

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 then 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

## Challenges

<table>
<thead>
<tr>
<th>Code base</th>
<th>Testing</th>
<th>Building</th>
</tr>
</thead>
<tbody>
<tr>
<td>roughly 25 year old “legacy code”</td>
<td>most testing done on gifted, and aging, hardware</td>
<td>maintaining builds on all SNL CEE-LAN and HPC machines</td>
</tr>
<tr>
<td>large code base with C++, Fortran, C, and other language components</td>
<td>thousands of tests with tens of Gb of data</td>
<td>maintaining builds on select SNL test beds</td>
</tr>
<tr>
<td>extremely complex physics</td>
<td>some tests take longer then 24 hours to execute</td>
<td>providing builds on customer machines for which there are no SNL counterparts</td>
</tr>
</tbody>
</table>

### 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 then 24 hours to execute
ALEGRA

Challenges

Code base
- roughly 25 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 then 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.
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

The legacy toolset: addressing challenges

**Code base**: single SVN repository for code and data

⇒ code/data kept in consistent states
⇒ difficult to manage access controls
⇒ no (easy to use) pull/merge request mechanism
ALEGRA

The legacy toolset: addressing challenges

Code base: single SVN repository for code and data
  ⇒ code/data kept in consistent states
  ⇒ difficult to manage access controls
  ⇒ no (easy to use) pull/merge request mechanism

Dependencies: “vendor” TPLs in to code repository, patch if necessary, write custom build scripts for each
  ⇒ consistent builds
  ⇒ must do double duty as a package manager and build system
  ⇒ TPLs have drifted from upstream versions and are difficult to update

Data: store all data in centralized location
  ⇒ easy to find/navigate
  ⇒ difficult access controls
  ⇒ must be filtered for releases to remove sensitive data

Testing: scripts to run nightly and weekly tests
  ⇒ code is kept safe from regressions
  ⇒ scripts duplicate existing tools (cron, CDash, etc)
  ⇒ test invocation does not match user invocation
  ⇒ no automated commit testing

Building: use custom build system
  ⇒ consistent builds that we control from end to end
  ⇒ duplicates specialized tools Spack, CMake, etc
  ⇒ requires considerable expertise to maintain compiler files, MPI files, etc

Running: provide scripts for interacting with the code
  ⇒ Alegra (usually) invoked in a consistent way
  ⇒ scripts are not terribly consistent/integrated
  ⇒ Many of the Python scripts wrap older csh scripts
  ⇒ Python scripts written in Python2
ALEGRA

The legacy toolset: addressing challenges

**Code base**: single SVN repository for code and data
  - code/data kept in consistent states
  - difficult to manage access controls
  - no (easy to use) pull/merge request mechanism

**Dependencies**: “vendor” TPLs in to code repository, patch if necessary, write custom build scripts for each
  - consistent builds
  - must do double duty as a package manager and build system
  - TPLs have drifted from upstream versions and are difficult to update

**Data**: store all data in centralized location
  - easy to find/navigate
  - difficult access controls
  - must be filtered for releases to remove sensitive data

**Testing**: scripts to run nightly and weekly tests
  - code is kept safe from regressions
  - scripts duplicate existing tools (cron, CDash, etc)
  - test invocation does not match user invocation
  - no automated commit testing

**Building**: use custom build system
  - consistent builds that we control from end to end
  - duplicates specialized tools Spack, CMake, etc
  - requires considerable expertise to maintain

**Running**: provide scripts for interacting with the code
  - Alegra (usually) invoked in a consistent way
  - scripts are not terribly consistent/integrated
  - Many of the Python scripts wrap older csh scripts
  - Python scripts written in Python2
ALEGRA

The legacy toolset: addressing challenges

**Code base**: single SVN repository for code and data
- ⇒ code/data kept in consistent states
- ⇒ difficult to manage access controls
- ⇒ no (easy to use) pull/merge request mechanism

**Dependencies**: “vendor” TPLs in to code repository, patch if necessary, write custom build scripts for each
- ⇒ consistent builds
- ⇒ must do double duty as a package manager and build system
- ⇒ TPLs have drifted from upstream versions and are difficult to update

**Data**: store all data in centralized location
- ⇒ easy to find/navigate
- ⇒ difficult access controls
- ⇒ must be filtered for releases to remove sensitive data

**Testing**: scripts to run nightly and weekly tests
- ⇒ code is kept safe from regressions
- ⇒ scripts duplicate existing tools (cron, CDash, etc)
- ⇒ test invocation does not match user invocation
- ⇒ no automated commit testing
## ALEGRA

The legacy toolset: addressing challenges

<table>
<thead>
<tr>
<th><strong>Code base:</strong> single SVN repository for code and data</th>
</tr>
</thead>
<tbody>
<tr>
<td>⇒ code/data kept in consistent states</td>
</tr>
<tr>
<td>⇒ difficult to manage access controls</td>
</tr>
<tr>
<td>⇒ no (easy to use) pull/merge request mechanism</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Dependencies:</strong> “vendor” TPLs in to code repository, patch if necessary, write custom build scripts for each</th>
</tr>
</thead>
<tbody>
<tr>
<td>⇒ consistent builds</td>
</tr>
<tr>
<td>⇒ must do double duty as a package manager and build system</td>
</tr>
<tr>
<td>⇒ TPLs have drifted from upstream versions and are difficult to update</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Data:</strong> store all data in centralized location</th>
</tr>
</thead>
<tbody>
<tr>
<td>⇒ easy to find/navigate</td>
</tr>
<tr>
<td>⇒ difficult access controls</td>
</tr>
<tr>
<td>⇒ must be filtered for releases to remove sensitive data</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Testing:</strong> scripts to run nightly and weekly tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>⇒ code is kept safe from regressions</td>
</tr>
<tr>
<td>⇒ scripts duplicate existing tools (cron, CDash, etc)</td>
</tr>
<tr>
<td>⇒ test invocation does not match user invocation</td>
</tr>
<tr>
<td>⇒ no automated commit testing</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Building:</strong> use custom build system</th>
</tr>
</thead>
<tbody>
<tr>
<td>⇒ consistent builds that we control from end to end</td>
</tr>
<tr>
<td>⇒ duplicates specialized tools Spack, CMake, etc.</td>
</tr>
<tr>
<td>⇒ requires considerable expertise to maintain compiler files, MPI files, etc.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Running:</strong> provide scripts for interacting with the code</th>
</tr>
</thead>
<tbody>
<tr>
<td>⇒ Alegra (usually) invoked in a consistent way</td>
</tr>
<tr>
<td>⇒ scripts are not terribly consistent/integrated</td>
</tr>
<tr>
<td>⇒ Many of the Python scripts wrap older csh scripts</td>
</tr>
<tr>
<td>⇒ Python scripts written in Python2</td>
</tr>
</tbody>
</table>
ALEGRA

The legacy toolset: addressing challenges

**Code base**: single SVN repository for code and data
- code/data kept in consistent states
- difficult to manage access controls
- no (easy to use) pull/merge request mechanism

**Dependencies**: “vendor” TPLs in to code repository, patch if necessary, write custom build scripts for each
- consistent builds
- must do double duty as a package manager and build system
- TPLs have drifted from upstream versions and are difficult to update

**Data**: store all data in centralized location
- easy to find/navigate
- difficult access controls
- must be filtered for releases to remove sensitive data

**Testing**: scripts to run nightly and weekly tests
- code is kept safe from regressions
- scripts duplicate existing tools (cron, CDash, etc)
- test invocation does not match user invocation
- no automated commit testing

**Building**: use custom build system
- consistent builds that we control from end to end
- duplicates specialized tools Spack, CMake, etc.
- requires considerable expertise to maintain compiler files, MPI files, etc.

**Running**: provide scripts for interacting with the code
- Alegra (usually) invoked in a consistent way
- scripts are not terribly consistent/integrated
- Many of the Python scripts wrap older csh scripts
- Python scripts written in Python2
Alegra tooling modernizations
ALEGRA TOOLING MODERNIZATIONS

Requirements

- Reduce technical debt: use outside tools to do what they do well
- Protect UCNI, ITAR, and other proprietary data
- Be as future “proof” as possible
- Consistent and integrated design
toolset2 is a Python library that glues together the pieces of our CI/CD workflow:

- Spack
- VVTest
- GitLab
- CDash

Like Spack, toolset2 has a command line utility `nevada` with many subcommands
Workflows
Workflows: Building ALEGRA

```bash
$ git clone --recursive git@cee-gitlab.sandia.gov:alegra/source-code/toolset2
$ source ./toolset2/share/nevada/setup-env.sh
$ nevada config add developer:spack:env:SPACK_ENV
$ spack install alegranevada@master
```
WORKFLOWS

Workflows: Developer

- Developer opens merge request
- Merge request cannot be merged until
  - approved by another developer
  - automated tests pass
- Developers cannot push to master

```
$ git clone --recursive git@cee-gitlab.sandia.gov:alegra/source-code/alegranevada
$ nevada config add developer:spack:env:SPACK_ENV
$ spack develop -p $(pwd)/alegranevada alegranevada@master
$ spack install alegranevada@master
$ # do work...
$ spack install alegranevada@master
$ # open merge request
```
WORKFLOWS

Workflows: Developer integration testing

```bash
$ cp $(spack location -i alegranevada@master)/etc/alegranevada/alegranevada.yaml .
$ vvtest --run-dir=./vvtest [options] <paths>
$ cd ./vvtest
$ nevada post-to-cdash --cdash-track=Experimental ./vvtest/results.json
```
WORKFLOWS

Workflows: Developer integration testing

```
$ cp $(spack location -i alegranevada@master)/etc/alegranevada/alegranevada.yaml .
$ vvtest --run-dir=./vvtest [options] <paths>
$ cd ./vvtest
$ nevada post-to-cdash --cdash-track=Experimental ./vvtest/results.json
```

A simple test file

```python
#VVT: keywords : fast 2D
#VVT: parameterize (autotype) : np = 1 4
#VVT: parameterize (autotype) : temp = 100 500 1000
import vvtest_util as vvt
from nevada.contrib import alegra_job, exo_diff
def test():
    with alegra_job(vvt.NAME, dimension=3) as job:
        job.preprocess_input_file(TEMP=vvt.temp)
        job.run(nproc=vvt.np)
        job.compare_with_baseline()
if __name__ == "__main__":
    test()
```

```
ls ./vvtest
results.json test_name.np=4.temp=100/
test_name.np=1.temp=100/ test_name.np=4.temp=1000/
test_name.np=1.temp=1000/ test_name.np=4.temp=500/
test_name.np=1.temp=500/ vvtestlist
```
WORKFLOWS

Workflows: Developer integration testing

A simple test file

```python
#VVT: keywords : fast 2D
#VVT: parameterize (autotype) : np = 1 4
#VVT: parameterize (autotype) : temp = 100 500 1000
import vvtest_util as vvt
from nevada.contrib import alegra_job, exo_diff
def test():
    with alegra_job(vvt.NAME, dimension=3) as job:
        job.preprocess_input_file(TEMP=vvt.temp)
        job.run(nproc=vvt.np)
        job.compare_with_baseline()
    if __name__ == "__main__":
        test()
```

```bash
$ cp $(spack location -i alegranevada@master)/etc/alegranevada/alegranevada.yaml .
$ vvtest --run-dir=./vvtest [options] <paths>
$ cd ./vvtest
$ nevada post-to-cdash --cdash-track=Experimental ./vvtest/results.json

$ ls ./vvtest
results.json test_name.np=4.temp=100/
test_name.np=1.temp=100/  test_name.np=4.temp=1000/
test_name.np=1.temp=1000/ test_name.np=4.temp=500/
test_name.np=1.temp=500/ vvtestlist
```
WORKFLOWS

Workflows: CI/CD merge request

```
generate_merge_request_config:
  rules:
  - if: '${CI_MERGE_REQUEST_TITLE} =~ /^wip:/i || ${CI_MERGE_REQUEST_TITLE} =~ /^draft:/i'
    when: never
  - if: ${CI_MERGE_REQUEST_ID}
    when: always
  script: .gitlab/pipelines/merge_request/generate_mr_config2.py -o merge_request-ci.yaml
  artifacts:
    paths:
    - merge_request-ci.yaml

merge_request:
  stage: merge_request
  rules:
  - if: '${CI_MERGE_REQUEST_TITLE} =~ /^wip:/i || ${CI_MERGE_REQUEST_TITLE} =~ /^draft:/i'
    when: never
  - if: ${CI_MERGE_REQUEST_ID}
    when: always
  trigger:
    strategy: depend
    include:
    - artifact: merge_request-ci.yaml
  job: generate_merge_request_config
  forward:
    yaml_variables: true
    pipeline_variables: true
```

- In the first job, a GitLab CI configuration file is created with instructions for how to build, test, and post results to CDash.
- In the second, the configuration is executed by the `gitlab-runner`
Workflows: Nightly, Weekly, and Performance Testing

- Testing is launched by scheduled GitLab runners
- GitLab runners interact with toolset2
- Results posted to CDash
WORKFLOWS

Workflows: CI/CD nightly testing

```
# pieces:
  toolset2:
    clone:
      - rm -rf toolset2
      - git clone --recursive git@cee-gitlab.sandia.gov:alegra/source-code/toolset2.git
    setup:
      - source ./toolset2/share/nevada/setup-env.sh
    sandbox:
      create:
        - nvidia sandbox init --E ${BNB_SPACK_ENV} .
      activate:
        - nvidia sandbox activate .
    build:
      - rm -rf $PREFIX
      - spack config add "config:install_tree:root:$PREFIX/opt"
      - spack develop -p 'pwd' ${BNB_SPEC}
      - spack add ${BNB_SPEC}
      - spack concretize -f
      - spack -k install --show-log-on-error --only=dependencies --fail-fast
      - chown -R alegra:wg-alegrausers $PREFIX
      - chmod -R g+rX $PREFIX
      - mkdir -p .bnb/build
      - tail -n +4 spack-build-01-cmake-out.txt > .bnb/build/configure-log.txt
      - sed -n '/==> \[.*cmake/p' spack-build-01-cmake-out.txt | cut -d ' ' -f 3- > .bnb/build/configure-args.txt
      - tail -n +4 spack-build-02-build-out.txt > .bnb/build/build-log.txt
      - sed '3q;d' spack-build-02-build-out.txt | cut -d ' ' -f 3- > .bnb/build/build-args.txt
      - tail -n +4 spack-build-03-install-out.txt > .bnb/build/install-log.txt
      - sed '3q;d' spack-build-03-install-out.txt | cut -d ' ' -f 3- > .bnb/build/install-args.txt
      - cp spack-stage-progress.json .bnb/build
      - cp $(spack location -i ${BNB_SPEC})/etc/alegranevada/alegranevada.yaml .bnb/build
      - cp $(spack location -b ${BNB_SPEC})/vvtest/results.json .bnb/build/vvtest.json
    report:
      single:
        - nvidia bnb create-cdash-reports
          --test=. bnb/cdash
          --dash-track="${BNB_CDASH_TRACK}" 
          --dash-buildname="${BNB_CDASH_BUILDNAME}" 
          --dash-sizes=${SNLCLUSTER} 
          --build-dir=. bnb/build 
          --test-file=. bnb/build/vvtest.json 
          --test-file=. bnb/test/vvtest.json || true
```
WORKFLOWS
Workflows: CI/CD nightly testing

.toolset2:
  clone:
    - rm -rf toolset2
    - git clone --recursive git@cee-gitlab.sandia.gov:alegra/source-code/toolset2.git
  setup:
    - source ./toolset2/share/nevada/setup-env.sh
  sandbox:
    create:
      - nvidia sandbox init -E "$BNB_SPACK_ENV"
    activate:
      - nvidia sandbox activate
  build:
    - rm -rf $PREFIX
    - spack config add "config:install_tree:root:$PREFIX/opt"
    - spack develop -p "pwd" "$BNB_SPEC"
    - spack add "$BNB_SPEC"
    - spack concretize -f
    - spack -k install --show-log-on-error --only=dependencies --fail-fast
    - chown -R alegra:wg-alegrausers $PREFIX
    - chmod -R g+rX $PREFIX
    - mkdir -p .bnb/build
    - tail -n +4 spack-build-01-cmake-out.txt > .bnb/build/configure-log.txt
    - sed -n '/==> \[.*cmake/p' spack-build-01-cmake-out.txt | cut -d ' ' -f 3- > .bnb/build/configure-args.txt
    - tail -n +4 spack-build-02-build-out.txt > .bnb/build/build-log.txt
    - tail -n +4 spack-build-03-install-out.txt > .bnb/build/install-log.txt
    - cp spack-stage-progress.json .bnb/build
    - cp $spack location -i "$BNB_SPEC")/etc/alegranevada/alegranevada.yaml .bnb/build
    - cp $spack location -b "$BNB_SPEC")/vvtest/results.json .bnb/build/vvtest.json
  report:
    single:
      - nvidia bnb create-cdash-reports
        --dest bnb/cdash
        --cdash-track="$BNB_CDASH_TRACK"
        --cdash-buildname="$(BNB_CDASH_BUILDNAME)"
        --cdash-site="SNNCLUSTER"
        --build-dir= bnb/build
        --test-file= bnb/build/vvtest.json
        --test-file= bnb/test/vvtest.json || true

.include: /gitlab/pipelines/templates/pieces.yaml

.stage: [bnb]
  variables:
    GIT_SUBMODULE_STRATEGY: recursive
  nightly:
    timeout: 18h
    stage: bnb
    variables:
      BNB_VERSION: votd
      BNB_CDASH_TRACK: Nightly
      BNB_SPEC: alegranevada@master build_type=RelWithDebInfo
      BNB_VVTEST_ARGS: --batch --timeout-multiplier=4
      before_script:
        - !reference [.pieces, toolset2, clone]
        - !reference [.pieces, toolset2, setup]
        - !reference [.pieces, sandbox, create]
        - !reference [.pieces, sandbox, activate]
  after_script:
    - echo "$CI_JOB_STATUS" > .bnb/status.txt
    - !reference [.pieces, build]
    - !reference [.pieces, test, setup]
    - !reference [.pieces, test, single]
  tags: [tlcc2]
  variables:
    BNB_TOOLCHAIN: gnu
    BNB_BUILD_TYPE: opt
    BNB_SPACK_ENV: "$BNB_SPACK_ENV_TLCC2_GNU"
    BNB_CDASH_BUILDNAME: 'AlegraNevada/RelWithDebInfo Environment=tlcc2-gnu Tests!Long'
Workflows: CDash Integration

toolset2 collects build and test data from Spack and VVtest, writes the CDash XML files, and posts the files to the CDash server.
Containerization
CONTAINERIZATION

The problem

ALEGRA’s dependency graph is complex

- Building ALEGRA is not trivial
- Porting ALEGRA to new systems is time consuming and painful
- Recent adoption of Spack helps, but does not eliminate our problems
- Devops team are engineers/physicists who do devops out of necessity
CONTAINERIZATION

A solution?

Build a stripped down container with ALEGRA

- We have total control of the build environment
- We can build ALEGRA on our desktops and run it on HPC resources
- Integrate containers in to our existing CI processes
- Customers run the same executables that developers and nightly tests run
- We don’t have to port ALEGRA to new systems
CONTAINERIZATION

Containerized ALEGRA build: Overview

- Write a Docker manifest with instructions to build ALEGRA
- Build ALEGRA in a container using podman
- Push container to GitLab container registry
- Convert Docker container image to Singularity Image Format (SIF)
- Run container on HPC resources using Singularity
- Post process simulation results as usual
CONTAINERIZATION

Podman for on-platform container builds

- Sometimes need to build containers directly on HPC platforms or dedicated systems
- Laptop may not look like target supercomputer
- Cannot use Docker due to root-level requirements
- Podman (and Buildah) provide rootless container builds while maintaining user-level permissions
  - CLI equivalent to Docker – easy to use!
  - User namespaces and shadow-utils = trusted
  - Overlay and FUSE for mounting
  - Ongoing collaboration with Red Hat
- Prototyped solution on Astra/Stria
- Rolling out Podman on production build and test systems

```
podman build -t "gitlab.doe.gov/atse/astra:1.2.4" .
podman push gitlab.doe.gov/atse/astra:1.2.4

singularity build atse-astra-1.2.4.sif docker://gitlab.doe.gov/atse/astra:1.2.4

salloc -N 2048 && mpirun -np $NP singularity exec atse-astra-1.2.4.sif /app
```

Pedretti, et. al, Chronicles of Astra: Challenges and Lessons from the First Petascale Arm Supercomputer , in Supercomputing 2020
CONTAINERIZATION

Containerized ALEGRA build: multi-stage build

In this first build stage, a base image containing compilers, MPI, and other developer tools. In it, the entire ALEGRA application dependency graph is built.

The second FROM instruction starts a new build stage. The COPY --from=builder lines copy libraries and executables stage. All of the developer tools and intermediate artifacts are left behind.
CONTAINERIZATION

Containerized ALEGRA build: build and run

Build the container image

```bash
$ podman build --target alegra --tag alegra-VERSION -f Dockerfile
```

Post the imate to a Gitlab registry

```bash
$ podman login URL:ID
$ podman tag alegra-VERSION URL:ID/NAME
$ podman push URL:ID/NAME
```

Convert to Singularity SIF

```bash
$ singularity build --docker-login alegra-VERSION.sif docker://URL:ID/NAME
```

Run ALEGRA

```bash
$ mpiexec -n <N> singularity exec \
   -B "$(pwd)" alegra-VERSION.sif alegra3d -a <RUNID>
```
CONTAINERIZATION

- How big is the container image?
  - The resultant SIF file is smaller than the standard ALEGRA executable.

- How do simulation runtimes compare?
  - Our initial testing has shown that simulations run with the containerized ALEGRA can be faster than with natively built executables.

- Is it portable?
  - Yes, we build the container images on our Apple laptops and run them on Sandia’s HPC resources.

- What’s the catch?
  - See next slide . . .
CONTAINERIZATION

- Portable container images can be moved from one resource deployment to another with ease
- Reproducibility is possible
  - Everything (minus kernel) is self contained
  - Traceability is possible via build manuscripts
  - No image modifications
- Performance can suffer - no optimizations
  - Can’t build for AVX512 on Haswell
  - Unable to leverage latest GPU drivers

- Performant container images can run at near-native performance compared to natively built applications
- Requires targeted builds for custom hardware
  - Specialized interconnect optimizations
  - Vendor-proprietary software
- Host libraries are mounted in containers
  - Load system MPI library (glibc issues!?)
  - Match accelerator libs to host driver
- Not portable across multiple systems

How do we strike the right balance?
Conclusion
CONCLUSION
Alegra abandoned its devops infrastructure of over 20 years in favor of the composite system described

Lessons Learned
- aggressively update dependencies
- don’t wrap, adapt
- switching to CMake was relatively fast, easy, and painless
- adopting Spack not as painless as thought

Positive effects
- consistent interfaces across the board
- modern tools and workflows easier to keep up to date
- easier to protect various flavors of UCI
  ⇒ modern fork/merge request workflows
  ⇒ google and stackexchange are useful to the project
- familiar dev ops tools eases onboarding of new team members
- reduced technical debt