An Automated Approach to Continuous Acceptance Testing of HPC Systems at NERSC

Shahzeb Siddiqui[1], Erik Palmer[1], Sijie Xiang[2], Prathmesh Sambrekar[3] , Sameer Shende[4], Wyatt Spear[4]

[1]NERSC, Lawrence Berkeley National Laboratory, Berkeley, CA, USA
[2]School of Information Systems, Carnegie Mellon University, Pittsburgh, PA, USA
[3]School of Computing and Augmented Intelligence, Arizona State University, Tempe, AZ, USA
[4]Performance Research Laboratory, University of Oregon, Eugene, OR, USA

ABSTRACT

In this paper, we demonstrate a continuous acceptance testing strategy used at NERSC that can be implemented in the broader HPC community. To accomplish this task, we designed a new framework that can handle the complex parts of HPC systems, allowing us to verify a system is working optimally. buildtest [1] is an acceptance testing framework that can automate the testing of HPC systems and enable HPC support teams to painlessly create and run tests. Testing is initiated by changes to the system/software stack at scheduled system outage that demands for NERSC staff to build, run and monitor test results using GitLab's Continuous Integration (CI) [2]. Test results are clearly communicated to developers and users via the CDash [3] web interface and test failures are documented as GitHub issues. Together this framework forms a robust method for verifying cutting edge software stacks' function in challenging HPC environments.

KEYWORDS

HPC Testing, Acceptance Testing, Automation, Continuous Integration

Motivation

The National Energy Research Scientific Computing Center (NERSC) [4] is responsible for supporting highly complex HPC systems where the hardware and software are periodically updated throughout the system's lifetime. NERSC houses two production systems, Cori, a Cray XC49 with Intel Xeon Haswell and Intel Xeon Phi nodes, and Perlmutter a HPE Cray EX with AMD EPYC CPUs and NVIDIA A100 GPUs and supports over 8,000 users with hundreds of software packages. Due to the scale and complexity of software deployments, and modern HPC systems alike, automation is required to achieve reasonable test coverage with a minimal human effort. Moreover, executing tests alone is not sufficient, test results must be presented in a way that can be quickly comprehended and acted on by system administrators.

The user experience is regularly impacted by system level modifications as HPC centers strive for maximum performance by tightly coupling their system and software stacks. For example, the Cray Programming Environment (CPE) is common on HPC systems. The CPE includes compilers, MPI, and scientific software used to build the user-facing software stack. Because of this, system maintenance inevitably leads to changes in the system configuration, software stack, and/or programming environment leading to a significant impact on the users. To effectively address this issue HPC centers can take a proactive approach with a combination of testing and continuous integration to discover bugs and increase user confidence in the system.

The Extreme-Scale Scientific Software Stack (E4S) [5] stack is a collection of open-source software packages (100+) for developing and running scientific software on HPC systems. At NERSC, we deploy E4S via Spack [6], a package manager that enables users to build combinatorial versions and variants of the software package that can coexist together. E4S codes often represent development done in conjunction with hardware, thereby representing the bleeding edge of HPC software capabilities and for this reason, are sensitive to system changes. Maintaining the E4S software stack at NERSC serves as a model for how automated acceptance testing can aid the challenges faced by system administrators.

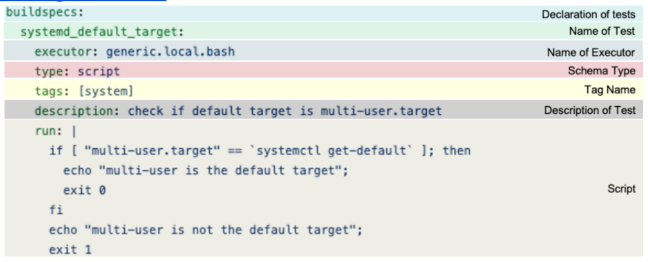
At NERSC we adopted a testing framework to regularly test several versions, and full source rebuilds of the E4S stack over their one year deployment on our systems. The framework involves using buildtest to encode a series of validation tests provided by developers and E4S teams. This is used in conjunction with GitLab with a custom GitLab runner Jacamar CI, to automatically run tests at regular intervals. The results of these tests are then uploaded to a web-interface with CDash.

In the following sections we describe the key elements of the testing framework in further detail. Section 1 outlines an overview of the buildtest framework and how one writes tests. Section 2 discusses our GitLab CI/CD integration. Section 3 describes how tests are pushed to CDash and how we monitor tests. Section 4 provides several example tests run via buildtest at NERSC Although we are specific in our descriptions and examples, the approach is flexible enough to employ at a wide range of HPC centers. For instance, our testing methodology will work with several CI tools, such as Jenkins, CircleCI, TravisCI, Gitlab, GitHub, etc. Here we present GitLab because it was the best solution for NERSC due to its ability to host GitLab runners and provide version control.

1 buildtest

buildtest is an open source framework for automating the testing of HPC systems and their software. The project is available on GitHub at https://github.com/buildtesters/buildtest which can be run on any Linux/MacOS system. Tests are written in YAML and are called **buildspecs (**build specification). buildtest will process the buildspec and create a shell script that executes the test and reports the results.

Shown below is an example buildspec that tests the default systemd target, for more details on how to write buildspecs see the buildspec tutorial.

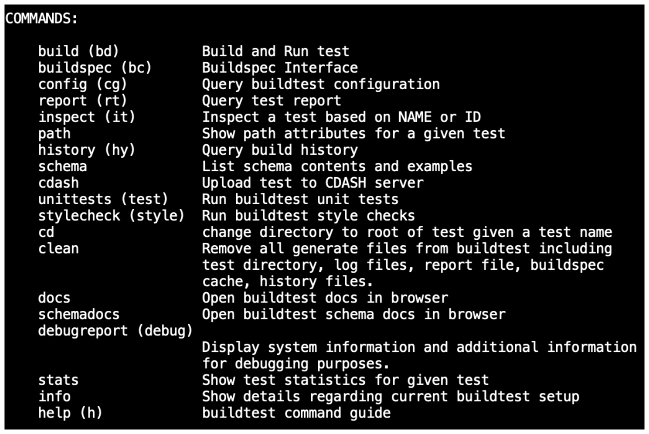


```
buildspecs:
  systemd_default_target:
    executor: generic.local.bash
    type: script
    tags: [system]
    description: check if default target is multi-user.target
    run: |
      if [ "multi-user.target" == `systemctl get-default` ]; then
        echo "multi-user is the default target";
        exit 0
      fi
      echo "multi-user is not the default target";
      exit 1
```

Figure 1: Example buildspec

buildtest has a comprehensive command line interface that allows users to interact with buildtest to build and run tests, inspect test results, view and edit buildspecs along and automatically push test results to CDash. The command line interface is organized into subcommands with several optional and positional arguments similar to how one uses git commands like **git checkout** or **git branch.**

In Figure 2, we show a preview of the buildtest commands that one will see when running **buildtest –help.**



```
COMMANDS:

    build (bd)          Build and Run test
    buildspec (bc)      Buildspec Interface
    config (cg)         Query buildtest configuration
    report (rt)         Query test report
    inspect (it)        Inspect a test based on NAME or ID
    path                Show path attributes for a given test
    history (hy)        Query build history
    schema              List schema contents and examples
    cdash               Upload test to CDASH server
    unittests (test)    Run buildtest unit tests
    stylecheck (style)  Run buildtest style checks
    cd                  change directory to root of test given a test name
    clean               Remove all generate files from buildtest including
                        test directory, log files, report file, buildspec
                        cache, history files.
    docs                Open buildtest docs in browser
    schemadocs          Open buildtest schema docs in browser
    debugreport (debug)
                        Display system information and additional information
                        for debugging purposes.
    stats               Show test statistics for given test
    info                Show details regarding current buildtest setup
    help (h)            buildtest command guide
```

Figure 2: Preview of buildtest commands

buildtest is a standalone test framework which comes with a set of sample tests, however these tests are not practical for testing the HPC system. It is expected that one creates a test repository when using buildtest. At NERSC we have a public facing repository on GitHub https://github.com/buildtesters/buildtest-nersc that contains all of our tests using buildtest. The main project is hosted on GitLab at https://software.nersc.gov/NERSC/buildtest-nersc which is a GitLab server hosted at NERSC.

The GitHub project is a mirror of the main project and kept up to date. We share our tests on GitHub since it's publicly-accessible and encourage the HPC community to reuse our test where applicable and share their tests.

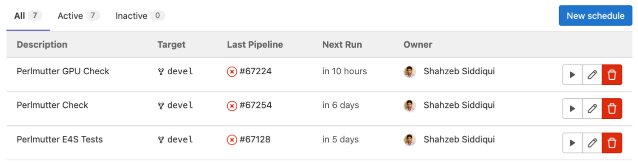
2. GitLab CI/CD Integration

GitLab CI/CD enables software development teams to help catch bugs and errors in the development cycle. GitLab CI configuration is performed via .gitlab-ci.yml that can be placed in the root of your repository.

GitLab Runner is an application that accepts GitLab jobs triggered in the CI pipeline and runs those jobs on the target host. A GitLab runner can be of various types called executors, such as SSH, Shell, Docker, Kubernetes or Custom executor.

We have configured our project to use a custom GitLab runner called Jacamar CI [6] which supports some key features such as setuid, integration with batch schedulers, and downscoping permission of CI jobs. Jacamar CI enables users to run workflow on an HPC system and avoid root escalation (sudo) and impose any security concerns. We have GitLab runners tied to our testing and production systems.

We have configured scheduled pipelines in our GitLab project to run a subset of tests at different intervals. We can't run all tests in a single shot due to queue policies, and resource availability. Furthermore, users are advised not to run compute-intensive workloads on login nodes at NERSC because they can bring down a login node and cause disruption for many users. In Figure 3, we show a list of scheduled pipelines that will trigger a GitLab job.


All 7 | Active 7 | Inactive 0 | New schedule

Description	Target	Last Pipeline	Next Run	Owner
Perlmutter GPU Check	devel	#67224	in 10 hours	Shahzeb Siddiqui
Perlmutter Check	devel	#67254	in 6 days	Shahzeb Siddiqui
Perlmutter E4S Tests	devel	#67128	in 5 days	Shahzeb Siddiqui

Figure 3: Scheduled Pipelines in GitLab

3. Monitoring Test Results

To facilitate communication of test results to system engineers and users we use the web-based software testing server CDash to display results. Displaying the results in this way accomplishes two goals. First, system administrations can quickly identify which tests are passing and get an analysis of test results over time. Second, displaying results in a publicly-accessible format builds confidence among users that software is working properly. The NERSC CDash project is available at https://my.cdash.org/index.php?project=buildtest-nersc and contains test results from Cori and Perlmutter. The process of uploading results to CDash to generate the reports is handled by buildtest via **buildtest cdash upload**. In the example below, buildtest will upload test results from its report file, send results to CDash and provide a URL to view the test results.

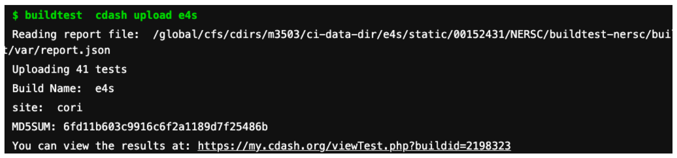
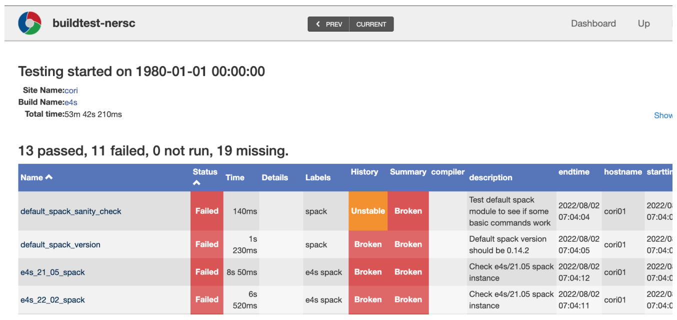

```
$ buildtest  cdash upload e4s
Reading report file:  /global/cfs/cdirs/m3503/ci-data-dir/e4s/static/00152431/NERSC/buildtest-nersc/bui
t/var/report.json
Uploading 41 tests
Build Name:  e4s
site:  cori
MD5SUM: 6fd11b603c9916c6f2a1189d7f25486b
You can view the results at: https://my.cdash.org/viewTest.php?buildid=2198323
```

Figure 4: Uploading test results to CDash via buildtest

We can see the test results in CDash, each entry corresponds to a test name, including the associated metadata with each test. CDash will show failed tests in red which can help pinpoint failures that will require attention from the HPC support team.



Testing started on 1980-01-01 00:00:00
Site Name:cori
Build Name:e4s
Total time:53m 42s 210ms

13 passed, 11 failed, 0 not run, 19 missing.

Name	Status	Time	Details	Labels	History	Summary	compiler	description	endtime	hostname	startti
default_spack_sanity_check	Failed	140ms		spack	Unstable	Broken		Test default spack module to see if some basic commands work	2022/08/02 07:04:04	cori01	2022/0 07:04:0
default_spack_version	Failed	1s 230ms		spack	Broken	Broken		Default spack version should be 0.14.2	2022/08/02 07:04:05	cori01	2022/0 07:04:0
e4s_21_05_spack	Failed	8s 50ms		e4s spack	Broken	Broken		Check e4s/21.05 spack instance	2022/08/02 07:04:12	cori01	2022/0 07:04:0
e4s_22_02_spack	Failed	6s 520ms		e4s spack	Broken	Broken		Check e4s/21.05 spack instance	2022/08/02 07:04:11	cori01	2022/0 07:04:0

Figure 5: Test Results in CDash

Test automation, CI capability and public test dashboard is not enough, we need a human to actively monitor and analyze test results to determine a proper course of action. Therefore, we recently started a bi-weekly meeting to analyze test results in CDash, report test failures into the GitHub issue tracker at https://github.com/buildtesters/buildtest-nersc/issues and get status updates on any incoming user contribution. Oftentimes, we observe infrastructure issues that may prevent tests from running successfully such as a scheduled system outage, a bug in buildtest, or a GitLab runner going offline.

4 HPC Test Cases

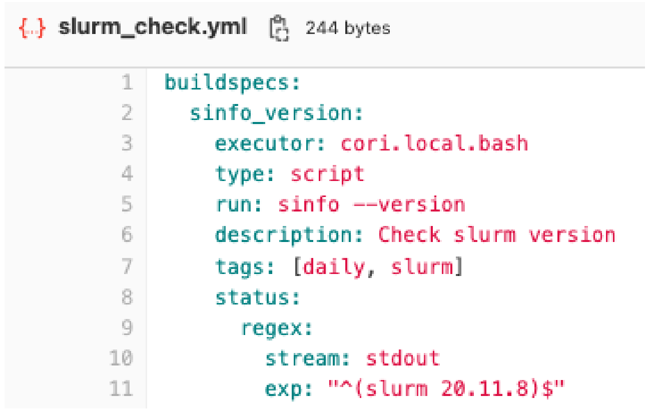
There are a number of use cases that we want to cover when testing an HPC system. A few examples of these are:

1. Job submission to all queues
2. Test your batch scheduler
3. Test your user environment (module, shell startup)
4. Example codes in your user documentation
5. Application-specific test
6. Test your Programming Environment (MPI, OpenACC, OpenMP, CUDA)
7. System Benchmark (HPL, HPCG, IO-500)
8. Filesystem test
9. System-specific utilities
10. GPU test
11. Vendor-provided test
12. Negative Test such as testing a known bug
13. Software Stack testing such as **spack test**

Next, we will show some real example tests at NERSC that can be used at other HPC centers.

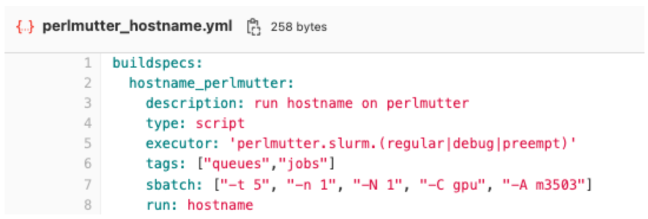
4.1 Example 1. Slurm version

At NERSC, we use the Slurm scheduler. Our first test includes a Slurm version check using **sinfo –version** command. NERSC will periodically upgrade Slurm on Perlmutter which may add or deprecate some Slurm features, therefore we have a simple test to ensure that the Slurm version is the expected version.


slurm_check.yml 244 bytes

```yaml
buildspecs:
  sinfo_version:
    executor: cori.local.bash
    type: script
    run: sinfo --version
    description: Check slurm version
    tags: [daily, slurm]
    status:
      regex:
        stream: stdout
        exp: "^(slurm 20.11.8)$"
```

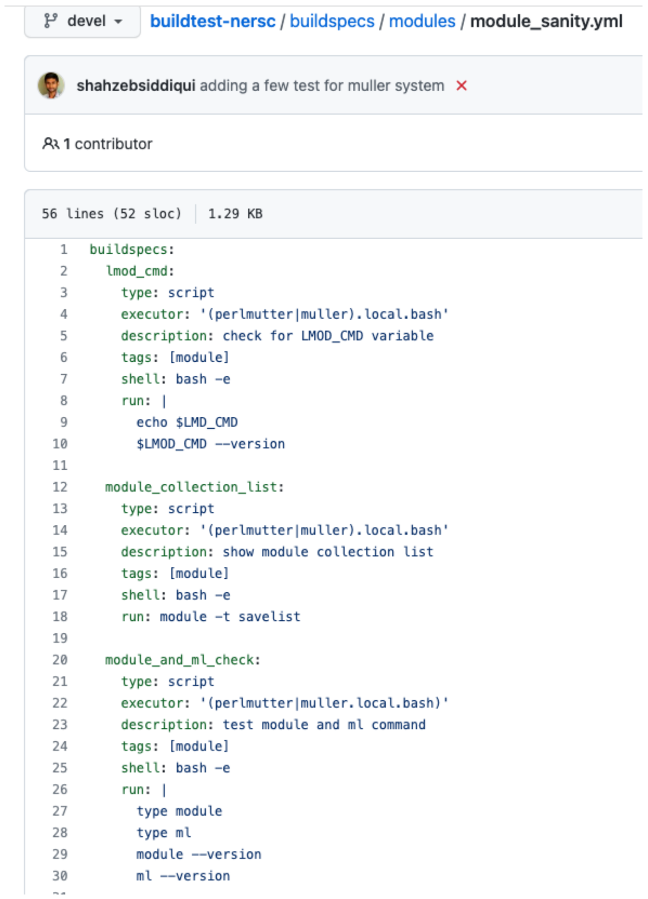
It's a good idea to test job submission to all queues to validate job submission to each queue works as expected. At NERSC, we document example job scripts on how to submit jobs to each of the Slurm queues, therefore we test our documentation examples to ensure our Slurm policy and user documentation matches our expectation with our test. In this next example we run a simple **hostname** across several queues on Perlmutter.


perlmutter_hostname.yml 258 bytes

```yaml
buildspecs:
  hostname_perlmutter:
    description: run hostname on perlmutter
    type: script
    executor: 'perlmutter.slurm.(regular|debug|preempt)'
    tags: ["queues","jobs"]
    sbatch: ["-t 5", "-n 1", "-N 1", "-C gpu", "-A m3503"]
    run: hostname
```

4.2 Example 2. Testing Lmod

On our newest machine, Perlmutter, we are running Lmod [8] as our module system, so we take steps to run basic sanity checks of the module system. For instance, we check if **$LMOD_CMD** is defined and check the version of Lmod. We run some module commands such as **module –version** and **ml –version** to ensure basic functionality of Lmod is working as expected.


devel buildtest-nersc / buildspecs / modules / module_sanity.yml

shahzebsiddiqui adding a few test for muller system

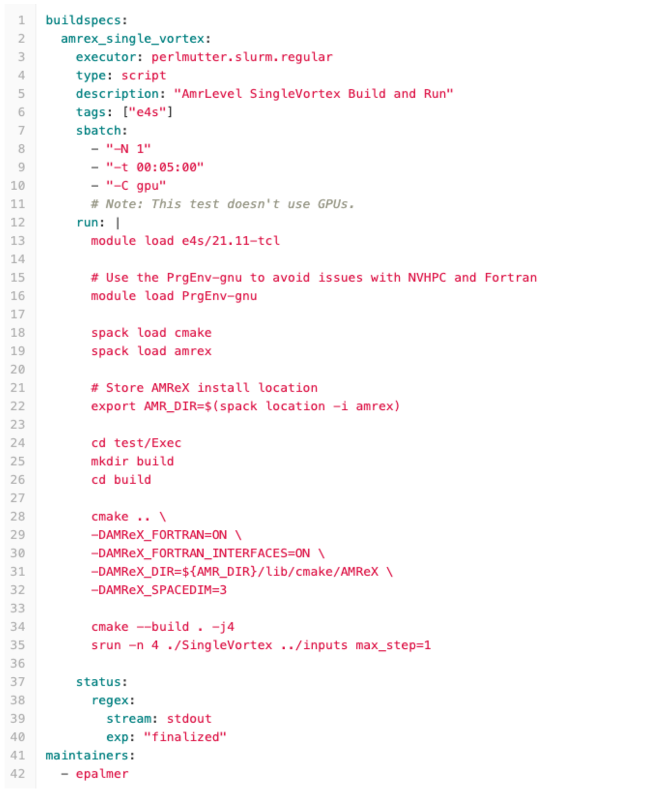
1 contributor

56 lines (52 sloc) 1.29 KB

```yaml
buildspecs:
  lmod_cmd:
    type: script
    executor: '(perlmutter|muller).local.bash'
    description: check for LMOD_CMD variable
    tags: [module]
    shell: bash -e
    run: |
      echo $LMD_CMD
      $LMOD_CMD --version

  module_collection_list:
    type: script
    executor: '(perlmutter|muller).local.bash'
    description: show module collection list
    tags: [module]
    shell: bash -e
    run: module -t savelist

  module_and_ml_check:
    type: script
    executor: '(perlmutter|muller.local.bash)'
    description: test module and ml command
    tags: [module]
    shell: bash -e
    run: |
      type module
      type ml
      module --version
      ml --version
```

4.3 Example 3. Testing AMReX

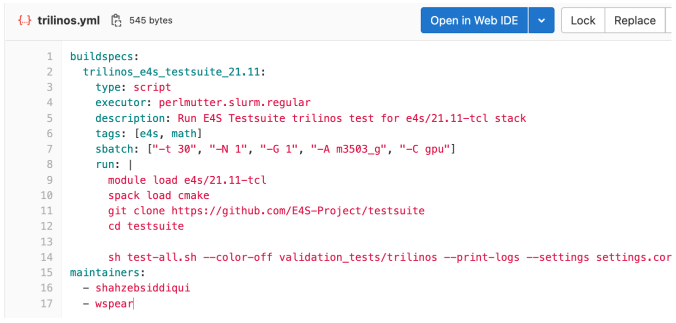
In this example we test an instance of AMReX [9] installed with Spack. AMReX is a software framework for writing massively parallel block-structured adaptive mesh refinement (AMR) applications. First, we specify the sbatch parameters used to submit a job to the scheduler. Then, after the **run:** heading, we load the module **e4s/21.11-tcl** exposing the E4S software stack installed with Spack. AMReX is then loaded into the environment with the command **spack load amrex**. Several additional dependencies for the build are loaded in the same fashion. Then AMReX is built with CMake, and executed via **srun**. In the last step, to determine whether or not the test succeeded, buildtest searches the output stream (stdout) for the regular expression "*finalized*" that AMReX outputs upon error-free execution of its code.



```yaml
buildspecs:
  amrex_single_vortex:
    executor: perlmutter.slurm.regular
    type: script
    description: "AmrLevel SingleVortex Build and Run"
    tags: ["e4s"]
    sbatch:
      - "-N 1"
      - "-t 00:05:00"
      - "-C gpu"
      # Note: This test doesn't use GPUs.
    run: |
      module load e4s/21.11-tcl

      # Use the PrgEnv-gnu to avoid issues with NVHPC and Fortran
      module load PrgEnv-gnu

      spack load cmake
      spack load amrex

      # Store AMReX install location
      export AMR_DIR=$(spack location -i amrex)

      cd test/Exec
      mkdir build
      cd build

      cmake .. \
      -DAMReX_FORTRAN=ON \
      -DAMReX_FORTRAN_INTERFACES=ON \
      -DAMReX_DIR=${AMR_DIR}/lib/cmake/AMReX \
      -DAMReX_SPACEDIM=3

      cmake --build . -j4
      srun -n 4 ./SingleVortex ../inputs max_step=1

    status:
      regex:
        stream: stdout
        exp: "finalized"
maintainers:
  - epalmer
```

4.4 Example 4. Running the E4S Testsuite

We make use of the E4S project's testsuite (https://github.com/E4S-Project/testsuite), a repository of test cases used to validate Spack-deployed software provided by E4S. Shown below is an example trilinos test that will validate our installation from E4S by invoking the testsuite.



```yaml
buildspecs:
  trilinos_e4s_testsuite_21.11:
    type: script
    executor: perlmutter.slurm.regular
    description: Run E4S Testsuite trilinos test for e4s/21.11-tcl stack
    tags: [e4s, math]
    sbatch: ["-t 30", "-N 1", "-G 1", "-A m3503_g", "-C gpu"]
    run: |
      module load e4s/21.11-tcl
      spack load cmake
      git clone https://github.com/E4S-Project/testsuite
      cd testsuite

      sh test-all.sh --color-off validation_tests/trilinos --print-logs --settings settings.cor
maintainers:
  - shahzebsiddiqui
  - wspear
```

Related Work

Among HPC centers, there is a community effort focused on testing. For instance, the HPC System Testing Working Group has hosted several BoFs at SC conferences. This working group focuses on discussing acceptance and regression test procedures from other HPC centers. Along with buildtest there are several open source frameworks such as ReFrame [10] and Pavilion2 [11] that address HPC testing needs. ReFrame is a regression framework for writing tests that support the Cray Programming Environment, several backend job schedulers, module integration, test dependency and parameterized tests. ReFrame2 is led by the Swiss National Supercomputing Center. Pavilion is a framework for running and analyzing tests for HPC systems. Similar to buildtest, the tests are written in YAML and use a Python plugin to modify Pavilion behavior. Pavilion2 is developed by the Los Alamos National Laboratory (LANL).

NERSC recently published a technical document titled Software Deployment Process at NERSC outlining our software deployment process along with testing efforts pertaining to validation of the E4S software stack.

In July 2022, we presented a buildtest tutorial at PEARC22 to help garner interest in HPC testing. The audience was composed of system engineers, HPC user support teams, research software engineers (RSE), and application teams interested in employing HPC testing methodologies at their institutions. In total we had 30 attendees registered for the event.

CONCLUSION

NERSC employs an automated testing framework well suited to the challenges of HPC such as, changing system configurations, software stacks and programming environments. The framework consists of several software packages: buildtest to build and execute tests, GitLab and Jacamar CI to enable continuous testing on the HPC system itself, and CDash to convey the results to system engineers and users. This approach automates several steps of this process and increases user confidence and reliability of the HPC system.

ACKNOWLEDGMENTS

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.

REFERENCES

[1] builtest, https://buildtest.rtfd.io/
[2] GitLab, https://about.gitlab.com/
[3] CDash, https://www.cdash.org/
[4] NERSC, https://www.nersc.gov/
[5] Extreme-Scale Scientific Software Stack, https://e4s-project.github.io/
[6] Spack, https://spack.readthedocs.io/
[7] Jacamar CI, https://gitlab.com/ecp-ci/jacamar-ci)
[8] Lmod, https://lmod.readthedocs.io/en/latest/
[9] AMReX, https://amrex-codes.github.io/amrex/
[10] ReFrame, https://reframe-hpc.readthedocs.io/en/stable/
[11] Pavilion2, https://pavilion2.readthedocs.io/en/latest/