Continuous Regression Testing of the Sustained Petascale Blue Waters Infrastructure

Galen W. Arnold Gregory H. Bauer Brett Bode Tomothy A. Bouvet Celso L. Mendes
National Center for Supercomputing Applications
University of Illinois at Urbana-Champaign
Urbana, USA
{gwarnold,gbauer,brett,tbouvet,cmendes}@illinois.edu

***Abstract*—Regression testing is an important activity that must be carefully conducted to ensure the health, stability and effectiveness of any computational system during its operation. Regression testing is traditionally performed when major system updates or reconfigurations occur, either in hardware or in software. However, to be more effective, testing must be done on a continuous basis, such that any deviation from expected behavior can be detected as early as possible and properly corrected. Here, we describe our approach to continuously test the behavior of the entire Blue Waters infrastructure, including the computational system and various other sub-systems. Our regression testing scheme is based on the Jenkins framework, and includes a broad range of tests, from low-level system aspects to full application performance. By relying on Jenkins, our test executions can be fully automated, and test results remain stored and readily available for human visualization or historical analyses.**

Keywords—regression testing, testing, monitoring, failures

I. Introduction

Regression testing is considered an important activity that must be carefully conducted by system administrators to ensure the health, stability and effectiveness of any computational system during its operation. In particular for large systems, which require large investments for acquisition and deployment, regression testing is an essential procedure to confirm that the system behaves according to specifications and fulfills its intended mission, thus providing the expected return on those investments. Regression testing is traditionally performed when major system updates or reconfigurations occur, either in hardware or in software. However, to be more effective, testing must be done on a continuous basis, such that any deviation from expected behavior can be detected as early as possible. This enables administrators to be alerted about any problems and immediately fix them, minimizing their effect on the system.

Here, we describe our approach to continuously test the behavior of the entire Blue Waters infrastructure, including the computational system (compute nodes and high-speed interconnection network), and various other sub-systems, such as the storage facility, the test-and-development system, and others. Our regression testing scheme is based on the Jenkins framework, and comprises a broad range of tests, from low-level system aspects to full application performance. By relying on Jenkins, our test executions can be fully automated, and test results remain stored and readily available for human visualization or historical analyses.

II. Deployed Testing Framework

After investigating some of the available options for our regression testing infrastructure, we decided to adopt the Jenkins framework [1], as it had many of the features that we needed. We implemented the main Jenkins server inside an NCSA virtual machine. There, the server can be properly secured and at the same time it can connect to all the sub-systems that we need to test. Full details about the design and deployment of our Jenkins infrastructure have been described elsewhere [2].

The main administration interface to our testing structure is a Web-based GUI, shown in Figure-1. This site can be accessed only from the internal NCSA network. The main portion of that GUI, at the right of the figure, is a list containing all the existing tests. For each test, the list indicates the dates for the last successful and failing executions of that test, with corresponding hyperlinks to details about the execution, and also the duration of the test's latest execution. There is also a link that allows manually scheduling an execution of that test.

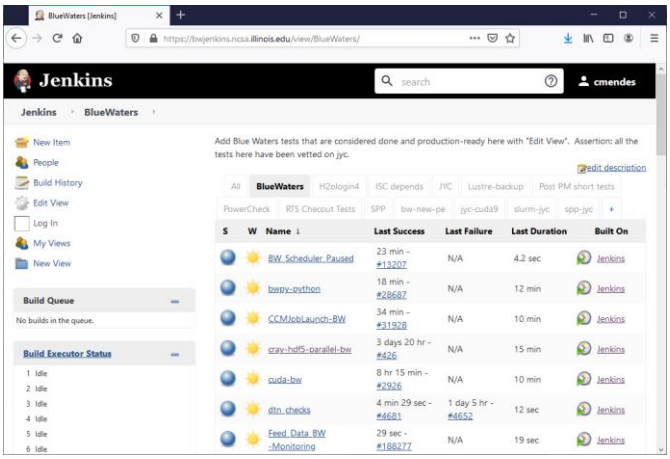

Figure 1- Main GUI for Blue Waters' testing framework.

III. Classes of Tests

In addition to the tests related to the main computational part of Blue Waters, the GUI in Figure-1 allows inspection of tests conducted in other parts of the Blue Waters infrastructure, or for specific collections of features. This can be selected through the tabs above the list of existing tests in the GUI. Major classes of tests that can be selected include the following:

- **Blue Waters**: tests of utilities and full applications that are executed in compute nodes of the system.
- **Login nodes**: tests of serial utilities and application builds that are typically conducted by users in the login nodes.
- **ISC-related**: tests that mostly assess Quality-of-Service on the three filesystems of Blue Waters.
- **JYC**: tests executed in our Test-and-Development System (TDS), a one-cabinet Cray machine with the same hardware and software as in Blue Waters.
- **Lustre backup**: tests related to backup, or to purging old files, in some of the filesystems.
- **Post-maintenance**: short tests that are executed after the system undergoes maintenance, mostly targeting a safe return to regular operations.
- **Power check**: tests for critical components of the system that must be assessed after a power event.
- **SPP applications**: tests with sustained petascale benchmarks, which are full scientific applications.
- **Job scheduler**: tests to evaluate behavior of current version of the job scheduler, done on TDS.

IV. Test Structure and Results

Adding a new test to the existing Jenkins infrastructure is a relatively simple task: one needs to create the sequence of steps required for execution of the test and for analysis of results. For applications, this may optionally include compiling the code and producing an executable. If the test must run on Blue Waters, the required steps also include creation of a proper job script, its submission, and collection of job results. A final step must analyze those results and parse relevant information to reveal success or failure, in addition to any possible metrics of interest.

All those steps described above must be scripted, which is typically done with creation of Shell/Python scripts containing the actions needed. To make this process easier, the GUI enables inspection of scripts from all existing tests. The GUI also allows viewing the console output produced in each test execution. In addition, one can configure, in the GUI, how many execution instances of a test must have results stored for future viewing.

As an example of historical test results, Figure-2 shows the GUI display for executions of IOR, an I/O benchmark that is routinely run on Blue Waters. Each pair of bars corresponds to one execution, with red bars representing *writes* and blue bars representing *reads*. The height of the bars corresponds to the measured rate, in MB/s. From such plots, it may be easy to pinpoint occasions where a filesystem is experiencing problems (e.g. a bar is below a predetermined threshold). Events like those may trigger an alert, which will cause an email to be sent to a system administrator automatically, for further inspections.

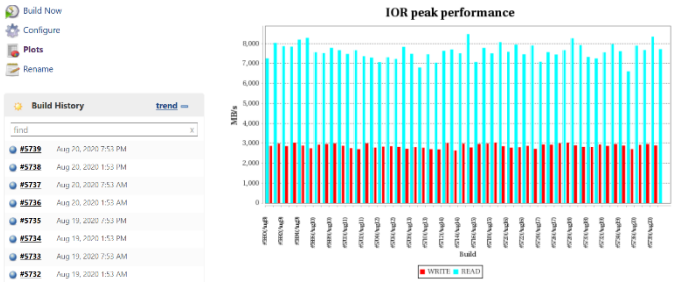

Figure 2 – GUI view showing historical results of IOR tests.

Another recent example of the utility of our testing scheme is one of our strategic applications (SETSM, a geophysical model), where periodic tests enabled catching a subtle issue in the code.

V. Test Execution Frequency

The Jenkins framework allows great flexibility in the selection of execution frequency for the tests. The GUI contains a configurable "build trigger" for each test, which enables periodical executions, or executions after a certain other test is executed. We have tests that run as frequently as every few minutes, while other tests may run once per week or even less. Any test can also be run "manually", with the click of a button in the GUI. We have been running Jenkins tests for three years.

Selecting an appropriate execution frequency sometimes will involve a tradeoff between testing efficiency and system perturbation, since some tests that require a job execution may compete with user jobs for acquisition of Blue Waters nodes. To minimize that contention, we try to avoid excessively frequent executions of heavy tests. In the past, we used a "low priority" job queue on Blue Waters for the tests. Since the deactivation of that queue, many of our tests have been running in the "debug" queue, where the limits in node count and walltime per job are relatively lower than in the regular queues.

VI. Conclusion

We firmly believe that our regression testing infrastructure, implemented with the Jenkins framework, has fully fulfilled the goal of keeping major pieces in Blue Waters working as expected. Whenever a component presents unacceptable behavior, either due to a failure or to an improper configuration, the framework generates alerts that drive immediate attention of our staff. In addition, the historical data stored by the framework is a rich collection of information that can be used for detailed investigations, when a single event is not sufficient to trigger an alert. We plan to keep expanding the set of Blue Waters components that are regularly tested.

Acknowledgments

This research is part of the Blue Waters sustained-petascale computing project, which is supported by the National Science Foundation (awards OCI-0725070 and ACI-1238993), the State of Illinois, and as of December, 2019, the National Geospatial-Intelligence Agency. Blue Waters is a joint effort of the University of Illinois at Urbana-Champaign and its National Center for Supercomputing Applications. We also thank ORNL's Reuben Budiardja for help in deploying Jenkins.

References

[1] CD Foundation, "Jenkins - Build great things at any scale," [Online]. Available: https://www.jenkins.io/. [Accessed 20 August 2020].

[2] T. A. Bouvet, R. D. Budiardja and G. W. Arnold, "Application-Level Regression Testing Framework Using Jenkins," in *Proceedings of the Annual Meeting of the Cray Users Group (CUG)*, Redmond-WA, 2017.