Improvements Towards the Release of Pavilion 2.0 Test Harness

Kody J. Everson
Dakota State University
Madison, South Dakota
Kody.Everson@trojans.dsu.edu

Maria Francine Lapid
Los Alamos National Laboratory
Los Alamos, New Mexico
Lapid@lanl.gov

ABSTRACT

High performance computing production support entails thorough testing in order to evaluate the efficacy of a system for production-grade workloads. There are various phases of a system's life-cycle to assess, requiring different methods to accomplish effective evaluation of performance and correctness. Due to the unique and distributed nature of an HPC system, the necessity for sophisticated tools to automatically harness and assess test results, all while interacting with schedulers and programming environment software, requires a customizable, extensible, and lightweight system to manage concurrent testing.

Beginning with the recently refactored codebase of Pavilion 1.0, we assisted with the finishing touches on readying this software for open-source release and production usage. Pavilion 2.0 is a Python 3-based testing framework for HPC clusters that facilitates the building, running, and analysis of tests through an easy-to-use, flexible, YAML-based configuration system. This enables users to configure their own tests by simply wrapping everything in Pavilion's well-defined format.

We took advantage of this system by writing three new commands as well as adding more functionality to two existing commands. Our contributions to the test harness also included improving and adding three result parsers to facilitate automatic parsing of test output files to interactively display and convert results to json format for logging. We improved Pavilion to support advanced configuration capabilities including better variable handling and allowing users to add environment commands to their kickoff scripts. In addition, we developed better table displays and integrated four parallel benchmarks tests to demonstrate these features.

As Pavilion approaches production and community readiness, the improvements this project accomplished aided in the meeting of production-level testing requirements.

KEYWORDS

benchmarking, python, object-oriented, testing, framework

1 INTRODUCTION

The supercomputers in Los Alamos National Laboratory (LANL) each undergo frequent testing to ensure that the systems are healthy. Within this testing we face four major problems: Builds and Environments, Scheduling, Running, and Results and Tracking.

The building of tests on production clusters can be tedious, since the many different machines can have different hardware and environments that may require changes in test configurations to interact with them. In addition, parsing results can be extremely cumbersome. Almost every benchmark displays different output formats, and it would be beneficial to pull relevant information rather than just pass or fail. Also, many of the tests often don't have check points so a user may never know where a test is in its testing process.

With these problems in mind, Los Alamos National Laboratory reworked existing code to produce a robust, extensible framework. Pavilion 2.0 aims to simplify the main issues with running benchmarks and tests in a highly parallel, production environment. The nuances are all simplified using a simple, easy-to-read, predefined YAML configuration format. This allows users to wrap tests in these configuration files and letting Pavilion handle the rest. Pavilion accomplishes this by generating scripts based on system's configuration and eliminates the need to frequently change configurations based on different machine specifics.

2 PAVILION USAGE

2.1 How to run a test using pavilion

2.1.1 YAML config file for tests

In order to run a test, a user must generate a config file (or several). As an example, we compile and run a program called hello.c which is a simple HelloWorld program for MPI (Message-Passing Interface).

The config file, hello.yaml, would look like:

```
hello:
        summary: Hello World MPI program

        build:
                source_location: hello.c
                cmds: "gcc -o hello hello.c"

        run:
                cmds: "srun -N2 ./hello"
```

The test config file can also contain other sections (besides build and run) that specify result parser(s), the scheduler and its settings, variables, and possible permutations.

2.1.2 Commands

After the test config file is written, the user needs to tell Pavilion to build and run the test. The command that reads the test config file and generates the necessary build, kickoff, and run scripts is called pav run. In this example, the user would run `pav run hello`.

The user also has the option to check the current status of the test, get the results, look at the logs, etc.

LA-UR-19-30222

2.2 Underlying process

The build process for Pavilion is rather complex, but most of this is abstracted from the user.

2.1.1 Resolving Test Configurations

In this section of the run command, Pavilion evaluates the current system settings as well as configuration settings to create test objects suitable for the given machine it is running on.

2.2.2 Working directory

The working directory refers to the directory where Pavilion places all of the test, series, build, and download directories necessary for running tests.

Whenever a new test is created, a directory with the corresponding Pavilion test id gets generated. In this directory, you will find the various scripts, logs, and results for a given test. It also creates a symlink to the build directory (if the test requires building) which holds the information necessary for building that specific test. Furthermore, every pav run command generates a test series with its very own test series id. The series directory in the working directory will then hold symlinks to every test directory ran in that series.

3 PAVILION DEVELOPMENT

The features discussed in this section are additions to Pavilion that we implemented this summer.

3.1 Wrapping Tests

We wrapped four different benchmarks and imported them to Pavilion.

Test Name	Details	Comments
IOR	MPI-coordinated test of parallel I/O	
IMB	Intel MPI Benchmark	
Stream	Tests memory bandwidth	Use Table Result Parser
slow test	Determines which nodes are running slower than others	
mpi-slowness	Determines which nodes are running slower than others	Wrote custom result parser
mounts-test	Tests for proper mounting of filesystem	

3.2 Plugins

Pavilion is extensible in that users themselves can write plugins. Plugins are how Pavilion knows how to interact with the specific systems. There are five types of plugins: module wrappers, system variables, schedulers, commands, and result parsers.

3.2.1 New commands and flags

- `pav log`. This command simply prints a log (build, kickoff, or run) from a given test run. This is a good debugging tool as the user can check one of these logs to see where in the Pavilion run process his or her test might have had an error.
- `pav cancel`. The new cancel command allows the user to gracefully cancel any combination of tests or series without having to go through the scheduler.
- `pav clean`. This command deletes directories inside the Pavilion working directory. It is important to note that clean does not delete files and directories of tests that are currently running or scheduled nor the files and directories of builds that those tests depend on.
- `pav status --all`. The flag all is a new addition to the command status. With the all flag, Pavilion can print the statuses of the latest tests (given a limit) found in the working directory, regardless of who ran the test.

3.2.2 Result Parsers

Result parsers look at the benchmark's output. Result parsers can override the default result gathering behavior, which is to simply assign the result as pass if the return value of the run command is zero (and fail otherwise)

- Constant Result Parser. This result parser simply inserts a constant that is specified in the test config into the results. This is useful if a user needs to see the variable used in the config in the results.
- Command Result Parser. This result parser simply inserts a constant that is specified in the test config and looks at either the output or the return value of the command.
- Table Result Parser. This result parser parses a table and stores the values of the table inside a nested dictionary.

3.3 Core Pavilion Features

3.1.1 Variables in Variables

Pavilion can interpret multilevel referencing in the test config file.

3.3.2 Kickoff command scripts

Pavilion has support for additional YAML configuration options to allow users to inject their own commands into the build, kickoff, and run scripts.

3.3.3 Text wrapping

Most of Pavilion's output is done in tables that did not support text wrapping. So, if your window size was too small or your font too large, you would get unpleasant dumps of the Pavilion data. Thus, we developed an algorithm to not only work well, but find the most optimal way to wrap text within a table while also being able to support color output.

4 CONCLUSION

We joined a small team of developers and made important contributions toward making this product usable in production environments not only at LANL but at any HPC center.

ACKNOWLEDGEMENTS

We would like to thank our mentors (Paul Ferrell, Nicholas Sly,and Jennifer Green) and the rest of the Programming and Run-time Environments team (Dan Magee, Jordan Orgas, David Shrader, Calvin Seamons, and Trent Steen) for helping us with our project.