LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-TR-738530

Who watches the watchers?: preventing fault in a fault tolerance library

C. D. Stanavige

September 15, 2017

Disclaimer

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

Who watches the watchers?: preventing faults in a fault tolerance library

Cameron Stanavige

15 September 2017

Western Oregon University, Monmouth, OR

LLNL-TR-738530

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

Who watches the watchers?: preventing faults in a fault tolerance library

Cameron Stanavige

Western Oregon University, Monmouth, OR

Abstract—The Scalable Checkpoint/Restart library (SCR) was developed and is used by researchers at Lawrence Livermore National Laboratory to provide a fast and efficient method of saving and recovering large applications during runtime on high-performance computing (HPC) systems. Though SCR protects other programs, up until June 2017, nothing was actively protecting SCR. The goal of this project was to automate the building and testing of this library on the varying HPC architectures on which it is used. Our methods centered around the use of a continuous integration tool called Bamboo that allowed for automation agents to be installed on the HPC systems themselves. These agents provided a way for us to establish a new and unique way to automate and customize the allocation of resources and running of tests with CMake's unit testing framework, CTest, as well as integration testing scripts though an HPC package manager called Spack. These methods provided a parallel environment in which to test the more complex features of SCR. As a result, SCR is now automatically built and tested on several HPC architectures any time changes are made by developers to the library's source code. The results of these tests are then communicated back to the developers for immediate feedback, allowing them to fix functionality of SCR that may have broken. Hours of developers' time are now being saved from the tedious process of manually testing and debugging, which saves money and allows the SCR project team to focus their efforts towards development. Thus, HPC system users can use SCR in conjunction with their own applications to efficiently and effectively checkpoint and restart as needed with the assurance that SCR itself is functioning properly.

I. INTRODUCTION

As software written for high-performance computers has become more and more complex, so has the ability to thoroughly test them to ensure they are working properly. This report will reflect our experience with implementing automated testing and Bamboo, a continuous integration tool, for one particular fault-tolerance library on high-performance computers—the Scalable Checkpoint/Restart library (SCR). The purpose of this report is to inform current and potential software developers for high-performance computers of how we were able to implement automated testing for real-world scenarios on our fault-tolerance library.

SCR was initially created in 2007 at Lawrence Livermore National Laboratory, but has never had any form of proper testing until now. Our approach towards solving this problem using Bamboo, and changes in our build process, will be the primary focus of this report. A short background on what SCR is and how it works will be required in order to gain a complete understanding of our solution. A detailed description of the overall problem and why our previous solutions for continuous integration and automated testing were unfeasible will also be discussed.

II. BACKGROUND

Applications running on large-scale supercomputing systems have a tendency to fail after a given amount of time, for various reasons including faults in code, compute nodes failures, and power loss. These failures can result in a loss of hours, or even days, of valuable computing time and money. One way applications prepare for these failures is by saving their state to checkpoint files. These files are typically written to reliable storage, such as a parallel file system. In the event of a failure, applications can then be restarted from a previous state that was saved on these files. Today, high-performance computing systems have grown vastly in scale and thus writing these checkpoint files to parallel file systems has become more essential—and more expensive.[1]

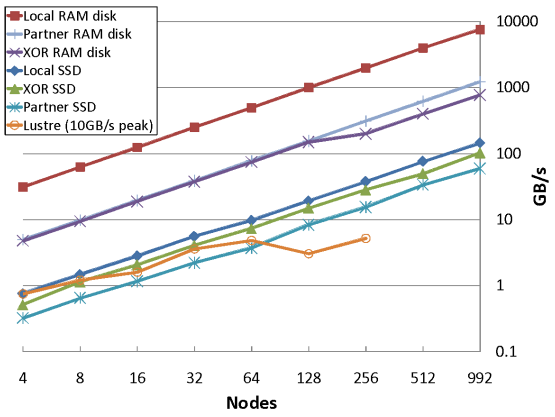

Figure 1. Node count versus checkpoint bandwidth

The Scalable Checkpoint/Restart library (SCR) was created as a potential solution to this problem. Through the use of multi-level checkpointing, SCR uses various redundancy schemes (Fig. 1) to reduce the overhead of writing checkpoint files and for quick and efficient recovery in the event of a failure. SCR uses storage local to the compute nodes on which the application is

running to checkpoint and restart, potentially without the use of the parallel file system at all. The more compute nodes that are used, the better SCR outperforms the parallel file system (Fig. 1).[1] Essentially, SCR is a library that protects applications running on supercomputing systems by allowing them to fail safely and recover efficiently. For a full understanding of SCR, *Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System,* written by scientists at Lawrence Livermore National Laboratory, is a useful and comprehensive resource.[1] In order to thoroughly test if SCR is functioning properly, it needs to be used in conjunction with another application running in parallel across multiple compute nodes.

III. PROBLEM

As previously mentioned, SCR is a library that protects applications in the event of a failure—but before now, nothing has been effectively protecting SCR. The few testing scripts that did already exist were not thorough and were not runnable in a variety of system environments. Thus, they required a user who was already somewhat familiar with the SCR library to manually allocate resources on the desired supercomputing system and then cut, paste, and run the desired testing commands and read through the outputs in order to discover the results. This process was tedious and would need to be repeated on every system that SCR needed to be tested on.

Upon doing this testing process at the beginning of the summer, we immediately discovered a critical bug in the SCR software. This bug was at least a month old, as that was the last time the library had been changed. As discussed above, the manual testing process is tedious and costly; consequently, there was nothing enforcing regular testing or informing the developers when things were broken. Solutions to these types of problems already existed; however, none were entirely helpful when it came to SCR's requirement of the simultaneous use of multiple compute nodes.

IV. SOLUTION

A. Proposed approach

Continuous integration centers around the practice in which developers of a project all use the same shared collection of source code, also known as a repository. Changes made by a developer are uploaded to the repository where the other developers can access those changes. All the while, at desired points in time, or after each change is made, a server watching the source code conducts an automated build to verify nothing was broken by the recent changes (Fig 2).[2]

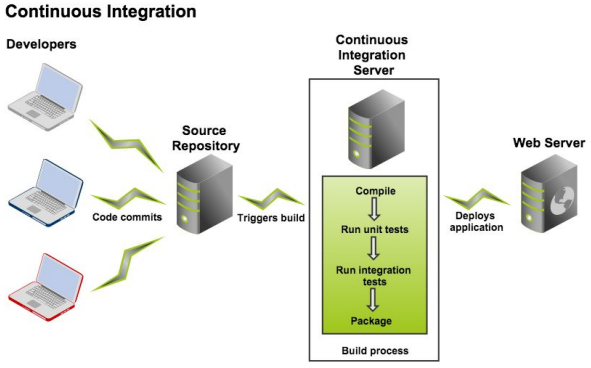

Figure 2. Bamboo continuous integration server

Automated testing works well with continuous integration. After the automated build succeeds, previously written tests are automatically run to ensure no other functionality in the source code has been broken by recent changes. If problems are detected, integration of the changes are prevented and developers are immediately notified to allow them to resolve the issues. These practices were the desired approach towards solving our problem, but how to use them with the parallel requirements of SCR specifically was the real question at hand.

Using this approach was nothing new to our development team. We have several other projects currently using continuous integration through a simple tool called Travis CI. Travis CI is a server that continually watches the desired source code repository for any changes. When a change is detected, Travis CI will copy and build the project and then run any commands or tests as designated by a specific file that exists in the repository.[3] This tool has been successful and works well for projects running serial tests on non-specific hardware. However, Travis CI doesn't have the resources to run parallel programs as it only has its own dedicated server that runs one thing at a time. This is precisely why it could not be used for SCR, as our library required testing in a parallel environment on various hardware-specific systems. In order to fully test SCR, we needed to run a sample program that was using the SCR library to ensure SCR was functioning correctly. We needed a new tool to achieve the results we desired.

B. Our approach

Without knowing if a new tool would work for what we needed, we decided to pursue another continuous integration application called Bamboo, which allowed us to do most of what we set out to accomplish this summer. The primary feature that makes Bamboo different than Travis CI is Bamboo's ability to have remote agents. These agents are similar to the Travis CI server in that they provide the same service, but can be installed remotely on the desired system of choice. Essentially, this allows Bamboo to build and run tests in the same parallel environment and on the same system as though it were a developer doing so manually. Our solution involved installing such agents—one on each of the different system architectures that we needed to thoroughly test SCR.[2]

An essential side project was the task of switching the build process and tools of SCR from Autotools to CMake. This switch allowed for easier creation and addition of unit tests for SCR through CMake's unit testing framework, CTest, which seamlessly integrates with Bamboo. In the end, CTest allowed for fairly easy implementation of automated testing.[4]

C. The solution

As Bamboo's workflow (Fig. 3) is designed to work for a variety of different projects, we had to experiment to find the right structure for SCR .[2]

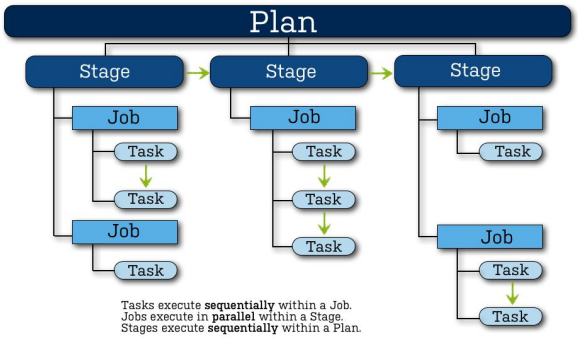

Figure 3. Bamboo workflow

Our solution involved having two distinct plans, each with a different approach to building and testing SCR. Both plans had a single stage with multiple jobs and tasks. As the jobs inside a stage are disconnected and can run in parallel, we created a new job for each system architecture on which SCR was to be tested. A remote agent was then created on each of these systems and assigned to their specific jobs inside each plan. Tasks were created under each job to allow for the actual building and testing of SCR on each system, as well as any system specific customization that may have been required.

1. Plan 1: CMake builds

Ensuring our CMake build works and testing simpler functionality with unit tests was the primary focus of our first plan. Each of this plan's jobs, running in parallel, first retrieves its own copy of the SCR source code. It then proceeds to build and install SCR using CMake. After a successful build, the Bamboo agent calls the CTest framework to run serial and parallel unit tests on the compute nodes of the job's assigned system. The results of these tests are then parsed and displayed by Bamboo (Fig. 4). If all the jobs are successful, the individual build of this plan by the Bamboo agent will be marked as a success.

Figure 4. CTest results displayed in Bamboo

Should any of the individual tasks fail, such as the CMake build, Bamboo will stop the process and notify the developers of the issue. Bamboo will discover if any of the unit tests failed during the parsing process. In the event that one did fail, the overall build of this plan will be displayed as failed and the tests causing the failure will be displayed for the developers to investigate. If an overall job fails on one system but succeeds on all others, the overall plan will still be marked as failed.

2. Plan 2: Spack builds

Our second plan primarily focused on an alternate build process and more complex testing, such as integration tests. Although this plan could be run independently, we made it dependent on the success of our first plan, as a failed first plan would mean SCR itself had issues and there would be no point continuing with more complex testing. This plan uses a common package management tool designed for supercomputing called Spack. Spack handles the build and installation of SCR and any other dependencies automatically, but required a bit more customization for each supercomputing system on which SCR was to be tested.[5]

Each job in this plan would, in parallel, retrieves the source code for Spack and subsequently uses Spack to install SCR. System specific compilers then are identified to allow Spack to correctly build the SCR tests. Additional system environment variables are set up to allow for the more complex integration testing scripts to be run. The Bamboo agent then submits jobs to the system's compute nodes to run these testing scripts. Upon their completion, the agent parses the output from the tests, marks the build of the overall plan as a success or failure depending on the results of the tests, and displays the results for the developers.

V. OUTCOMES

Our efforts led to many favorable outcomes. The switch to CMake and automated testing through Bamboo using CTest has allowed for easy addition of more tests as they are written. Additional integration tests can easily be added as well through the automation of using Spack to build and run them. Hours of developers manually testing, and thus company dollars, are now being saved.

Due to the use of Bamboo, developers and administrators now receive fast and valuable feedback on the status of the SCR library and how it is functioning on a variety of high-performance computing systems. Bamboo automatically detects when changes are made to the SCR library. Our plans then automatically build SCR and run our tests. If recent changes break anything in SCR, Bamboo can prevent the changes from being implemented until they are fixed. Should any of the builds or tests fail, the developers are notified immediately through email, or a variety of other communication methods, allowing them to find and fix the problems right away—a large improvement from waiting a month or more before discovering issues.

Setting up this testing environment has allowed us to automate a number of processes, including building SCR, running unit and integration tests, checking if recent changes caused problems with the rest of the code, and notifying developers of the results of those checks. We have automated the testing of SCR on multiple supercomputing architectures in parallel. With developers no longer having to worry about doing of these tasks manually, we are saving a lot of time and money.

VI. CONCLUSION

In the beginning, SCR was protecting other programs by allowing them to fail safely and successfully recover. However, nothing was protecting SCR. Solutions and tools the development team were using were insufficient, and so we needed a new approach. Bamboo's remote agents and a switch to CMake enabled us to continually ensure that SCR is always functioning properly with a real application on the actual supercomputing systems and environments SCR is being used on. Now, through this use of continuous integration and

automated testing, the watcher is being watched, and SCR is being protected at the same level as the programs it protects.

ACKNOWLEDGEMENTS

Kathryn Mohror, Ph.D., Center for Applied Scientific Computing, Lawrence Livermore Nat. Lab., Livermore, CA, USA

Elsa Gonsiorowski, Ph.D., Livermore Computing, Lawrence Livermore Nat. Lab., Livermore, CA, USA

Adam Moody, M.S., Livermore Computing, Lawrence Livermore Nat. Lab., Livermore, CA, USA

Gregory Becker, Livermore Computing, Lawrence Livermore Nat. Lab., Livermore, CA, USA

David Beckingsale, Ph.D., Center for Applied Scientific Computing, Lawrence Livermore Nat. Lab., Livermore, CA, USA

REFERENCES

[1] Adam Moody, Greg Bronevetsky, Kathryn Mohror, Bronis R. de Supinski, ”Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System,” LLNL-CONF-427742, Supercomputing 2010, New Orleans, LA, November 2010.

[2] Bamboo documentation. Web page. Accessed 7 July 2017. https://confluence.atlassian.com/bamboo/bamboo-documentation-289276551.html

[3] Travis CI user documentation. Web page. Accessed 26 June 2017. https://docs.travis-ci.com

[4] CMake reference documentation. Web page. Accessed 12 July 2017. https://cmake.org/cmake/help/v3.9/

[5] Spack documentation. Web page. Accessed 15 August 2017. http://spack.readthedocs.io/en/latest/