SAND2019-3102C

Creating Stable Productive CSE Software Development and Integration Processes in Unstable Environments on the Path to Exascale

Roscoe A. Bartlett
Department of Software Engineering & Research
Sandia National Laboratories
https://bartlettroscoe.github.io
rabartl@sandia.gov

Joseph R. Frye
Department of Software Engineering & Research
Sandia National Laboratories
jfrye@sandia.gov

***Abstract*—The Sandia National Laboratories (SNL) Advanced Technology Development and Mitigation (ATDM) project focuses on R&D for exascale computational science and engineering (CSE) software. Exascale application (APP) codes are co-developed and integrated with a large number of 2nd generation Trilinos packages built on top of Kokkos for achieving portable performance. These efforts are challenged by needing to develop and test on many unstable and constantly changing pre-exascale platforms using immature compilers and other system software. Challenges, experiences, and lessons learned are presented for creating stable development and integration workflows for these types of difficult projects. In particular, we describe automated workflows, testing, and integration processes as well as new tools and multi-team collaboration processes for effectively keeping a large number of automated builds and tests working on these unstable platforms.**

***Index Terms*—software engineering, software testing, testing techniques, exascale systems, software libraries, scientific computing, high performance computing**

I. Introduction

The Advanced Technology Development and Mitigation (ATDM) project was initiated in 2015 to investigate algorithms and software challenges and solutions for future simulation and modeling software to run efficiently on the next generation of the exascale supercomputers.

With increases in the clock speed of individual computer processing units (CPUs) leveling off and reaching basic physical limits, further increases in computing power are coming from massive increases in the number of smaller processing units (cores) that are packed onto increasingly fatter CPUs or in Graphics Processing Units (GPUs) from hardware vendors such as Cray, IBM, Intel, Arm, and NVIDIA. On these new CPUs and GPUs that will be used in the first generation of exascale computers in 2023, fine-grained parallelism must be used in order to exploit the full potential for parallel calculations which requires the usage of various threading models, new numerical algorithms, and specialized programming languages and compilers (e.g. Intel KNL, OpenMP, NVIDA CUDA). This is a significant departure from the standard successful single program multiple data (SPMD) approach of running single-thread exectuables on large numbers of CPUs tied together with the Message Passing Interface (MPI) using course-grained parallelism which has been a stable development and deployment environment for two decades.

This shift from simpler SPMD MPI approaches to fine-grained parallelism and accelerators represents a major disruption of the computational science and engineering (CSE) community. The large uncertainty and risk associated with this shift is the motivation for the ATDM projects with a primary goal to devise strategies for refactoring many current codes to preserve the large investment over decades of work that went into their development.

The goal of the ATDM program is to produce some reusable mathematical algorithms software libraries and a few demonstration CSE applications (APPs) that will be able to efficiently run on DOE's leading exascale computers by 2023.

This paper focuses on the Sandia National Laboratories (SNL) ATDM program but many of its challenges are very similar to other exascale software R&D efforts and some of the solutions and lessons learned will be applicable to those projects as well.

A. Overview of Development and Integration Efforts

The primary SNL ATDM Application (APP) codes SPARC [5] and EMPIRE [4] and the libraries they use are shown in Figure-1. These APPs use solvers and other basic algorithms implemented in the Trilinos [7] software collection. Each APP uses a different subset of packages from Trilinos. The APPs frequently drive Trilinos development and they need consistent updates of Trilinos in order to make steady progress on their own development and simulation efforts.

EMPIRE [4] is a massively parallel simulation code for electro-magnetic plasma flows in radiation environments. It

This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government. Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energys National Nuclear Security Administration under contract DE-NA0003525.

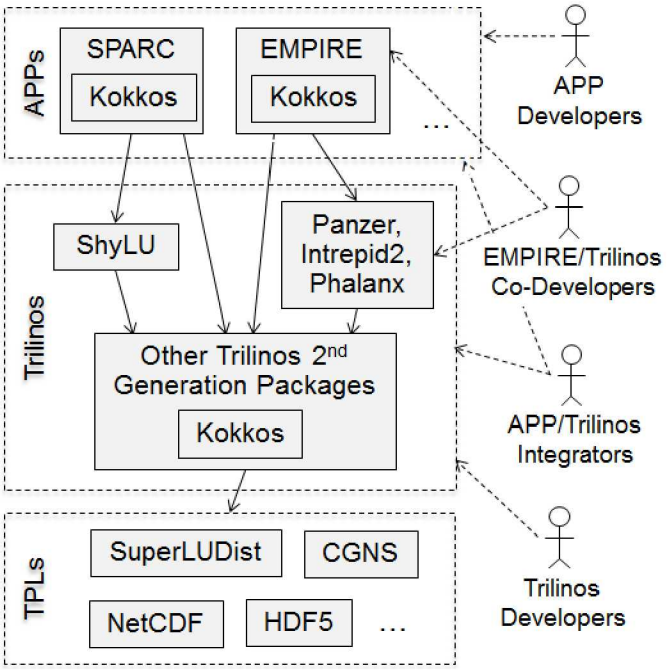

Fig. 1. SNL ATDM APPs, Trilinos packages, and TPLs (Third Party Software)

has deep dependencies on many Trilinos packages. In addition to basic Trilinos solver packages, it uses the discretization packages Panzer, Phalanx, and Intrepid2 to implement the basic residual right-hand-side and matrix computations at a low level. Therefore, many EMPIRE developers actively develop on these Trilinos packages and are essentially EMPIRE/Trilinos co-developers as depicted in Figure-1. In addition, EMPIRE requires Trilinos solvers and preconditioners including the very actively developed 2nd-generation MueLU package of multi-level preconditioners for exascale architectures. Because of this intimate dependency and EMPIRE/Trilinos co-development, EMPIRE requires almost daily updates of Trilinos.

SPARC [5] is a compressible computational fluid dynamics (CFD) code for solving aerodynamics and aerothermodynamics problems. It has strong required dependencies on Kokkos [6] and SEACAS (parallel file IO and mesh handling support) that must be built and installed before building any SPARC software. However, SPARC only has optional dependencies on any other Trilinos packages. While SPARC can use many of the solvers in Trilinos such as KokkosKernels (dense linear algebra kernels based on Kokkos) and ShyLU (sparse on-node linear solvers based on Kokkos), it has its own native solvers as well. While SPARC does feed requirements into some of the Trilinos solver packages, SPARC's dependencies on these packages are weak. Therefore, SPARC usually does not require frequent updates of Trilinos to keep making progress on its own development efforts.

The EMPIRE and SPARC APP projects provide a contrast into different approaches for multi-component development and integration. On one end, there are limited dependencies on software components developed outside of the APP team (SPARC). On the other end, there are significant dependencies on externally developed reusable components (EMPIRE). Therefore, the SNL ATDM project provides an interesting test bed for these two approaches.

Kokkos [6] is a performance portability library that provides an abstraction layer to enable the development of kernels that will run performantly on all architectures of interest. It has multiple back-end implementations that use OpenMP, CUDA, or other programming models and languages to achieve good on-node and on-GPU parallel performance. Kokkos classes pervade almost every line of numerical code in the Trilinos 2nd generation packages and the APPs. This embedded nature of Kokkos is depicted in Figure-1 with the box `Kokkos` embedded in the boxes for `SPARC` and `EMPIRE`.

Trilinos [7] is a large collection of software packages that provide numerical algorithms and supporting software ranging from low-level memory management utilities to massively parallel iterative linear solvers and preconditioners, nonlinear solvers, optimization, uncertainty qualification, and tools and frameworks for various discretization methods. The majority of Trilinos development is funded by the DOE ASC program. However, several other non-ASC projects also fund developers to do basic algorithms research. As a result, several projects and developers that submit changes to Trilinos are not primarily concerned about exascale issues. This creates a challenge to keep Trilinos working on the pre-exascale platforms that drive R&D work.

This stack of software from Kokkos on up is written in modern C++11 using advanced templating techniques that yield portable performance on pre-exascale machines but does so at the cost of expensive and memory intensive compilation and linking. In fact, build times dominate test runtimes on most development and test machines with most compilers. The high cost of building the software makes it challenging to perform sufficient automated testing on many of the these machines where great competition exists for computer time.

B. Unstable Pre-Exascale System Environments

One of the chief drivers of the instability of pre-exascale systems is that the exact hardware details of the exascale computers of 2023 and beyond are not known (because basic R&D on these systems is still under way). And even less is known about the compilers, detailed programming models, run-time libraries and other tools that will be available on these exascale machines. As a result, early efforts to prepare for exascale involve chasing an ever-moving target. The SNL Advanced Architecture Test Beds effort sets up and maintains clones of many of these intermediate pre-exascale systems and the ATDM project uses these to drive a lot of development and testing of the codes. On these intermediate R&D systems compiler bugs, bugs in system software like MPI, and other defects are fairly common and greatly impact the productivity of developers. Many developers who started their work in the stable SPMD MPI environment of the 2000s are not

accustomed to these types of problems. In the words of Michael Heroux (Trilinos lead and current ECP Software Technology Lead) "HPC is becoming a bloody sport again."

While developers have had to accept the nature of these unstable pre-exascale development environments, they still need to be productive and frequent integrations of new versions of Trilinos are needed to effectively drive development and capabilities in the APPs.

C. Early Development and Integration Processes

In the early years of the SNL ATDM program, the APP and Trilinos development teams used simpler development and integration processes that are common in many CSE efforts. Trilinos developers directly pushed to the main development branch with no testing requirement. Each APP had their own custom configuration of Trilinos and there was no automated testing of these custom configurations on any platform and little testing of any Trilinos configuration on challenging platforms like CUDA/GPUs. Most APP developers were pulling updated versions of Trilinos directly from the main Trilinos development branch. As is common for many CSE projects, there were many build and test failures that persisted for long periods of time in the builds submitted to the Trilinos CDash[1] site. Because of these perpetually failing builds and tests, it was difficult to detect when new failures occurred. This made new build and test failures difficult to identify and they often went unnoticed until someone happened to discover a new defect as part of their daily work.

As a result of these simpler development approaches, problems with managing clean Trilinos builds on many platforms, and the challenges of these demanding pre-exascale environments, APP developers that pulled Trilinos from the main development branch often experienced broken builds, broken behavior, and other problems with the software. These problems were most common on CUDA GPU platforms and there were long periods of time where some of the builds were broken. Even when Trilinos built and ran on these platforms, one would expect a large number of native Trilinos tests to fail. This resulted in the APP development teams not running native Trilinos tests as part of their regular development and integration efforts with Trilinos (problems this created are discussed in Section II-B).

Trilinos developers also had difficulty reproducing Trilinos issues reported by APP developers because the APP's custom Trilinos configuration was difficult to reproduce and often required access to machines unavailable to Trilinos developers.

This situation caused significant reductions in Trilinos and APP developer productivity and was straining the project.

II. Recent Improvements in Development and Integration Processes

Improving the stability and productivity of development and integration processes requires construction of better workflows that inject more effective testing to gate different steps in the process (while not imposing more complexity and overhead than is needed to achieve the desired result). Because the APPs require a regular flow of updates of Trilinos and since EMPIRE development often requires co-development with Trilinos, arguably the most attractive development and integration workflow for this situation is ***Almost Continuous Integration*** [3], [8]. This approach was used for integration of Trilinos with SNL customer codes Charon in 2007 [1], [2], Sierra in 2009 [3], and CASL VERA between 2013 and 2014. What differentiates the ATDM project being discussed here from past projects is the added complexity and instability of the pre-exascale development environments.

These workflows are composed of the following parts:

***Basic development and integration git workflows*:** How version control git repositories and branches are set up, how merges occur, what git commands are run, etc.

***Testing gates for integration steps in the workflows*:** What test suites are run and must pass for each integration step in the various git workflows.

***Detecting, triaging, and correcting failures*:** How new failures are detected and triaged, how work is organized to address the failures in a timely way, how one assures that issues are getting resolved, etc.

These workflow parts are described in the following subsections for development and integration.

A. Basic Development and Integration git Workflows

The basic git workflow that has recently been adopted for the SNL ATDM project is shown in Figure-2. In this workflow, each APP maintains its own stable mirror of the Trilinos git repository from which their APP developers directly pull. Trilinos developers no longer directly commit to the main Trilinos development branch `develop` but instead create commits on *topic branches* which are merged into the `develop` branch using GitHub Pull Requests (PR). Each of these Trilinos PRs are tested on a small number of general build configurations using a new PR auto-testing system (see Section II-B). Updated versions of Trilinos `develop` are tested daily against a larger set of Trilinos builds and tests on many platforms (i.e. the "ATDM Trilinos builds & tests"). Also, Trilinos `develop` is tested daily against each APP's own native test suite on a number of platforms (i.e. the "ATDM APP builds & tests"). If a filtered set of Trilinos builds & tests pass and the native APP builds & tests pass for a candidate version of Trilinos `develop`, then the APP can accept the new Trilinos version. The "APP Trilinos Integrator" for each APP is watching over testing and integration processes for their APP and their daily APP build & test suites run against Trilinos `develop`. (Note that the workflow in Figure-2 only covers the development and updates of Trilinos to the APPs. The more complex git workflow elements that support APP + Trilinos co-development are beyond the scope of this paper.)

B. Automated Testing Gates for Integration Workflows

The APP Trilinos development and integration workflow shown in Figure-2 described above and the testing gates in

[1]CDash: Open-source web-based build and test results database and dashboard provided by the company Kitware; `https://cdash.org`.

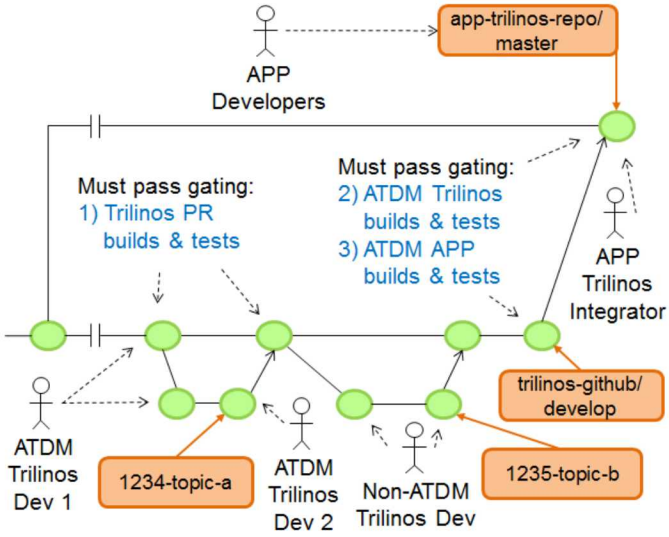

Fig. 2. ATDM Trilinos development and integration workflow

the workflow evolved over the calendar years 2016 through 2018. The evolution and current status of the workflows, as well as the testing gates, are described below.

Early efforts began in mid 2016 focused on the basic stabilization of the Trilinos `develop` branch. Efforts initially focused on encouraging pre-push testing of Trilinos using a single standard build configuration and the `checkin-test.py` tool which resulted in noticeable gains in stability. The next effort was to get a GitHub PR testing system in place and in early 2018 the new Trilinos PR auto-testing system was made mandatory. The PR system was refined throughout 2018 to address a number of issues and by early 2019 the **Trilinos PR builds & tests** system contained four different build configurations of Trilinos using three GCC compilers versions and one Intel compiler version all on x86 Linux RHEL6 test machines where all builds and tests must 100% pass before merging. Even though the PR system does not directly test the ATDM-specific configuration on ATDM platforms, it did further improve the stability of the Trilinos `develop` branch (due to requiring the builds and tests run and pass before merging).

Both SPARC and EMPIRE also have their own native **ATDM APP builds & tests** to demonstrate and protect the development of the APP software itself. APP Trilinos integrators have always run some subset of the APP builds & tests against a candidate version of Trilinos before accepting an update. But, in general, the APP codes did not run the native Trilinos test suite on any platform using their configurations. A major reason for this was that the Trilinos test suite had many failing tests on almost all of these platforms (see Section I-C). Therefore, they could not tell if failures they were seeing were already there or were new and if these failures should cause them concern about updating Trilinos. The APP developers are currently still updating their copy of Trilinos based purely on the status of their APP's native test suite. (The one exception is that EMPIRE does run Panzer tests to gate Trilinos updates but no other native Trilinos package tests).

However, the approach used by SPARC and EMPIRE to only run their APP builds & test suites as the only gate for updating their copy of Trilinos is not ideal. The problem is that when run-time defects are injected into Trilinos, one can only hope to catch these defects in the downstream APP test suite. But tricky defects are harder to detect and triage in downstream APP tests (due to the course-grained system-level tests on which APP codes tend to rely). This puts APP developers in charge of doing first-round triaging of failures that they have to report upstream to Trilinos developers (APP developers don't like having to debug defects in upstream software). Another problem with not running native Trilinos tests is that APPs tend to not to have high coverage of upstream software functionality. As a result, many defects in upstream software don't get caught in the downstream customer APP test suite. When this happens, these defects lie in wait and then often will only get caught by customers trying to use the APP. When this occurs, it damages the trust of the users in the APP codes. Once the APP developers triage the problem and determine it to be caused by an uncaught defect in the upstream software, APP developers lose trust in the upstream software and developers. This cycle creates a culture of mistrust and leads to a pull-back of the usage of external software.

The way to mitigate the problems created by running just the APP builds & tests as the only gate to updating the APP's copy of Trilinos described above is to run the native Trilinos test suite on all of the platforms and build configurations that are used by the downstream APPs. Also, to reduce the computational load for running the automated Trilinos builds and tests, a single configuration of Trilinos is in the final stages of development which merges the SPARC and EMPIRE Trilinos configurations to allow a single configuration of Trilinos to support both APPs. In early 2018, an effort was started to run and clean up the native Trilinos test suite on all of the major pre-exascale platforms required by the APPs. This included many pre-exascale platforms such as Intel KNL machines, IBM Power8 and Power9 machines with GPUs, and several other systems. This new set of **ATDM Trilinos builds & tests** is specifically targeted to the APP codes, runs daily on the target platforms, and submit results to CDash with an example dashboard shown in Figure-3.

All of the gating test suites described above require 100% passing builds and tests on all selected builds. But most failing tests in CSE software don't necessarily indicate a defect in the underlying functional code. During the process to set up the Trilinos builds on the various target platforms, clean up the initial failing tests, and address new failures, it was observed that ***approximately 90-95% of the failing tests were not due to defects in functional Trilinos code that impacted the ATDM APPs*** but instead were defects in functionality that APPs were not using, defects in only test code, problems with the underlying system software, or some other non-code issue (see section-II-C). So why then do we insist on 100% passing

ATDM — 33 builds

Site	Build Name	Update	Configure		Build		Test			Start Time	Labels
		Revision	Error	Warn	Error	Warn	Not Run	Fail	Pass		
cee-rhel6	Trilinos-atdm-cee-rhel6-clang-5.0.1-openmpi-1.10.2-serial-static-opt	5487fd	0	0	0	42	0	0	2064	Jan 04, 2019 - 05:11 UTC	27 labels
cee-rhel6	Trilinos-atdm-cee-rhel6-gnu-4.9.3-openmpi-1.10.2-serial-static-opt	5487fd	0	0	0	5	0	0	2066	Jan 04, 2019 - 06:12 UTC	27 labels
cee-rhel6	Trilinos-atdm-cee-rhel6-gnu-7.2.0-openmpi-1.10.2-serial-static-opt	5487fd	0	0	0	10	0	0	2066	Jan 04, 2019 - 07:05 UTC	27 labels
cee-rhel6	Trilinos-atdm-cee-rhel6-intel-17.0.1-intelmpi-5.1.2-serial-static-opt	5487fd	0	0	0	50	0	0	2063	Jan 04, 2019 - 08:40 UTC	27 labels
chama	Trilinos-atdm-chama-intel-debug-openmp	5487fd	0	19	0	7	0	0	1996	Jan 04, 2019 - 12:32 UTC	25 labels
chama	Trilinos-atdm-chama-intel-opt-openmp	5487fd	0	19	0	6	0	0	1997	Jan 04, 2019 - 12:29 UTC	25 labels
hansen	Trilinos-atdm-hansen-shiller-gnu-debug-openmp	5487fd	0	19	0	5	0	0	1996	Jan 04, 2019 - 06:22 UTC	25 labels
hansen	Trilinos-atdm-hansen-shiller-gnu-debug-serial	5487fd	0	19	0	5	0	0	1886	Jan 04, 2019 - 07:13 UTC	25 labels
hansen	Trilinos-atdm-hansen-shiller-gnu-opt-openmp	5487fd	0	19	0	5	0	0	1997	Jan 04, 2019 - 10:11 UTC	25 labels
hansen	Trilinos-atdm-hansen-shiller-gnu-opt-serial	5487fd	0	19	0	5	0	0	1887	Jan 04, 2019 - 07:14 UTC	25 labels
hansen	Trilinos-atdm-hansen-shiller-intel-debug-openmp	5487fd	0	19	0	7	0	0	1996	Jan 04, 2019 - 08:57 UTC	25 labels
hansen	Trilinos-atdm-hansen-shiller-intel-debug-serial	5487fd	0	19	0	7	0	0	1888	Jan 04, 2019 - 07:20 UTC	25 labels
hansen	Trilinos-atdm-hansen-shiller-intel-opt-openmp	5487fd	0	19	0	6	0	0	1997	Jan 04, 2019 - 09:19 UTC	25 labels
hansen	Trilinos-atdm-hansen-shiller-intel-opt-serial	5487fd	0	19	0	6	0	0	1891	Jan 04, 2019 - 08:24 UTC	25 labels
mutrino	Trilinos-atdm-mutrino-intel-opt-openmp-HSW	5487fd	0	19	0	50	0	2	1982	Jan 04, 2019 - 11:10 UTC	25 labels
mutrino	Trilinos-atdm-mutrino-intel-opt-openmp-KNL-panzer	5487fd	0	1	0	50	0	0	170	Jan 04, 2019 - 09:03 UTC	Panzer
sems-rhel6	Trilinos-atdm-sems-rhel6-gnu-debug-openmp	5487fd	0	19	0	30	0	0	1996	Jan 04, 2019 - 10:21 UTC	25 labels
sems-rhel6	Trilinos-atdm-sems-rhel6-gnu-debug-serial	5487fd	0	19	0	30	0	0	1890	Jan 04, 2019 - 14:19 UTC	25 labels
sems-rhel6	Trilinos-atdm-sems-rhel6-gnu-opt-openmp	5487fd	0	19	0	30	0	1	1996	Jan 04, 2019 - 11:35 UTC	25 labels
sems-rhel6	Trilinos-atdm-sems-rhel6-gnu-opt-serial	5487fd	0	19	0	30	0	0	1891	Jan 04, 2019 - 09:34 UTC	25 labels
sems-rhel6	Trilinos-atdm-sems-rhel6-intel-opt-openmp	5487fd	0	19	0	6	0	1	1996	Jan 04, 2019 - 12:28 UTC	25 labels
serrano	Trilinos-atdm-serrano-intel-debug-openmp	5487fd	0	19	0	7	0	0	1995	Jan 04, 2019 - 11:48 UTC	25 labels
serrano	Trilinos-atdm-serrano-intel-opt-openmp	5487fd	0	19	0	6	0	0	1996	Jan 04, 2019 - 10:32 UTC	25 labels
waterman	Trilinos-atdm-waterman-cuda-9.2-debug	5487fd	0	20	0	50	0	0	1974	Jan 04, 2019 - 09:07 UTC	25 labels
waterman	Trilinos-atdm-waterman-cuda-9.2-opt	5487fd	0	20	0	50	0	0	2005	Jan 04, 2019 - 09:44 UTC	25 labels
waterman	Trilinos-atdm-waterman-cuda-9.2-release-debug	5487fd	0	20	0	50				Jan 04, 2019 - 12:09 UTC	25 labels
waterman	Trilinos-atdm-waterman-gnu-opt-openmp	5487fd	0	19	0	16	0	0	1997	Jan 04, 2019 - 09:07 UTC	25 labels
waterman	Trilinos-atdm-waterman-gnu-release-debug-openmp	5487fd	0	19	0	22	0	0	1996	Jan 04, 2019 - 11:41 UTC	25 labels
white	Trilinos-atdm-white-ride-cuda-9.2-gnu-7.2.0-debug	5487fd	0	20	0	50	0	0	1973	Jan 04, 2019 - 08:54 UTC	25 labels
white	Trilinos-atdm-white-ride-cuda-9.2-gnu-7.2.0-release	5487fd	0	20	0	50	0	0	2001	Jan 04, 2019 - 10:22 UTC	25 labels
white	Trilinos-atdm-white-ride-cuda-9.2-gnu-7.2.0-release-debug	5487fd	0	20	0	50	0	0	2002	Jan 04, 2019 - 12:08 UTC	25 labels
white	Trilinos-atdm-white-ride-gnu-7.2.0-openmp-debug	5487fd	0	19	0	15	0	0	1994	Jan 04, 2019 - 07:30 UTC	25 labels
white	Trilinos-atdm-white-ride-gnu-7.2.0-openmp-release	5487fd	0	19	0	16	0	0	1997	Jan 04, 2019 - 08:15 UTC	25 labels

Fig. 3. CDash dashboard for Promoted **ATDM Trilinos builds & tests** on testing day 2019-01-04

tests for all of these gating tests suites before updating Trilinos to an APP? The reason is that ***while likely 90-95% of failing tests don't indicate a problem with functional code impacting customers, these test failures hide the 5-10% of failing tests that do indicate real defects***. This is exactly analogous to why it is important to turn on strong compiler warnings and then clean up all compiler warnings even though most compiler warnings are benign; it is because the large number of benign warnings hide the small number of warnings that point to real code defects.

An example of what can happen when a project does not require 100% passing tests and has many failing tests over long periods of time was a recent case involving an update of Trilinos to an important non-ATDM customer APP code. An update of Trilinos was accepted by the customer APP that injected a significant defect into its functional code. A few months later, a problem was discovered with the APP code that was eventually traced back to Trilinos. But it was not clear what was broken. An expensive and laborious git bisection study was performed across thousands of Trilinos commits that required many builds of the large APP code and running its native tests. Through that effort the change in Trilinos causing the defective behavior was discovered. But it was also discovered that when Trilinos was updated it had actually triggered a new failing APP test that demonstrated the defect. ***But because the APP already had so many existing failing tests, no one noticed the new test failure that demonstrated the new defect in Trilinos***.

The lesson from the above experiences is that one must ***carefully scrutinize every failing test in order to detect new defects***. Well-designed test suites typically will have just a single test that protects a specific use case for a piece of functionality. Therefore, a significant defect may only trigger the failure of a single test so every test counts. The other lesson learned is that one ***must not allow existing failing tests to hide the injection of new failing tests***. The tools and processes described in Section II-C below outline how this can be accomplished.

Finally, note that even though there is a single set of Trilinos builds that test the union of Trilinos packages used by SPARC and EMPIRE, the gating checks for these two APPs filter the results differently according to their own requirements. That is, when examining the Trilinos builds posted on the CDash site using a CDash query, the queries for EMPIRE and SPARC filter out different sets of build sites and Trilinos packages that are not of interest to each particular APP. For example, for the CDash results shown in Figure-3, the CDash analysis and reporting tool (described in Section II-C) that determines pass/fail for the *ATDM Trilinos builds & tests* for SPARC uses a CDash query that filters out results for the sites 'sems-rhel6', 'hansen', and 'white' and filters out the build and test results for the Trilinos packages 'Phalanx', 'Intrepid2', and 'Panzer' (using `SubProjects` filters). Therefore, failures for builds on 'white' or in the package 'Panzer' (in any build) will not block the update of Trilinos to SPARC.

C. Detection, Triage, and Correction of Failures

The implementation of the integration workflow shown in Figure-2 requires Trilinos `develop` to pass a selected subset of the Trilinos builds & tests and the APP builds & tests in order for an APP to accept an updated version. Producing a passing version of Trilinos `develop` boils down to a race between fixing existing defects verses the injection of new defects. If the "mean-time to fail" is shorter than the "mean-time to fix", then (on average) the test suites will always be broken and the APPs will never get updates of Trilinos. How can one address this problem? One option is to move to a release-branch workflow where a new branch is periodically created off of `develop` and issues are fixed on the branch until the gating tests all pass. However, a release-branch workflow increases the computer time and involves more complex workflows. The other option is to adjust processes to keep a single Trilinos `develop` branch clean.

Stabilizing the Trilinos `develop` branch requires reducing the time that it takes to:

detect new failures in the build or tests,
triage new failures to determine if they represent real blocking or critical defect, and
address new failures to fix defects or deal with benign failures.

The processes and tools for these three tasks are interrelated and therefore need to be discussed together. First, once a new failure on the Trilinos CDash dashboard is detected, it must be triaged and have a Trilinos GitHub issue created to characterize the failure and to investigate it. The act of creating a new Trilinos GitHub issue and keeping track of which test failures are mapped to which Trilinos GitHub issues is critical to being able to detect new failures even while other tests are already failing (see below). The outline for the template for Trilinos GitHub issues is shown in Figure-4.

````
CC: @trilinos/<package-name>, ...

## Next Action Status
<Status and/or next action>

## Description
As shown in [this query](<cdash-link>) the tests:
* `<full-test-name-1>`
* `<full-test-name-2>`
are failing in the builds:
* `<full-build-name-1>`
* `<full-build-name-2>`
<Add more details about what is failing>

## Current Status on CDash
The status of these tests/builds for the current
testing day can be found on CDash [here](<cdash-link>).

## Steps to Reproduce
<Information on what machines can reproduce the failure>
```
$ cd <some_build_dir>/
$ source $TRILINOS_DIR/cmake/std/atdm/load-env.sh \
  <build-name>
$ cmake \
  -GNinja \
  -DTrilinos_CONFIGURE_OPTIONS_FILE:STRING=\
  cmake/std/atdm/ATDMDevEnv.cmake \
  -DTrilinos_ENABLE_TESTS=ON \
  -DTrilinos_ENABLE_<package-name>=ON \
  $TRILINOS_DIR
$ make NP=16  # Or just 'ninja -j16'
$ <command-to-run-on-compute-node> ctest -j16
 ```
````

Fig. 4. Markdown skeleton template for Trilinos GitHub Issues

Other than the Issue title, there are four key sections in each GitHub Issue. ***Description*** lists the full names of the failing tests with their associated builds, and contains information to characterize how the tests are failing. (This allows one to query GitHub to find issues related to failures seen on CDash.). ***Steps to Reproduce*** provides exact commands to reproduce the Trilinos build and/or test failure(s) (where `<build-name>` can be the CDash build name). (So that Trilinos developers can trivially reproduce the failures.) ***Current Status on CDash*** provides links to CDash queries where the current status of impacted builds or tests can be observed. (So Trilinos developers can always see the current status to know if the problems are resolved or not yet.) Finally, ***Next Action Status*** contains a short precise statement of the current status of the issue and the next action needed to make progress.

One of the novel aspects of this effort was creating a system that makes it trivial for APP and Trilinos developers to (re)produce any Trilinos build configuration in any supported target system. This is shown in the "Steps to Reproduce" section in Figure-4 and involves sourcing the shell script '`atdm/load-env.sh <build-name>`', doing the configuration with raw `cmake` passing a single configuration file `ATDMDevEnv.cmake` and setting the Trilinos packages to enable, and then executing `make` and `ctest` to run the build and tests. The correct machine is automatically detected in `atdm/load-env.sh` by examining `hostname` and matching that to a list of known supported machines while the build configuration settings are parsed from `<build-name>` argument. For example, the build name `gnu-7.2.0-openmp-static-opt` specifies the GNU 7.2.0 compiler on the local system, using the Kokkos OpenMP backend implementation, produces static libraries (versus shared libraries), and creates an optimized build of the code (versus a debug build). The command `atdm/load-env.sh` then loads modules and sets up environment variables that describe the requested build configuration. The CMake fragment file `ATDMDevEnv.cmake` passed to `cmake` then reads in this set of environment variables and tells the build system what configuration to produce. This simple set of commands are the same on every supported system and the only thing that changes is the command to run the tests on a compute node with `ctest`.

One of the key tasks after the creation of a new Trilinos GitHub Issue is determining the severity/criticality of the failure. This is done by setting one of the three ***ATDM Trilinos severity/criticality labels*** which are as follows.

'**`ATDM Sev: Critical`**" issues critically damage the ability to even run the Trilinos builds. Examples include a library build failure on an important platform (e.g. CUDA on ATS-2) that takes out many tests; or a runtime defect in an upstream package that takes out hundreds of tests on several platforms.

"**`ATDM Sev: Blocker`**" issues make Trilinos unfit to be adopted by one or more APPs but do not seriously damage the ability to run automated builds. An example is a runtime issue that breaks the functioning of an important Trilinos capability used by an APP but only impacts a small number of Trilinos tests.

"**`ATDM Sev: Nonblocker`**" issues either don't impact the APPs or are minor problems that should not block APPs from getting Trilinos updates. Examples include failures in tests for functionality that is not even used by any of the APPs; a failure in a Trilinos test on a small number of platforms that has been confirmed to be a bug in the test code and not in Trilinos library code itself; or warnings that need to be cleaned up.

One of the important realizations that we came to was that one cannot determine the stability of Trilinos on these platforms and the suitability of Trilinos to be adopted by APP customers by just observing the Trilinos CDash dashboard alone. First, while some of the "red" on the dashboard represents legitimate defects in Trilinos that should block updates, ***other "red" should not block the update of Trilinos by ATDM APP customers*** and fall into two broad categories.

1) Some "red" will be from random system failures such as mpi startup failures, non-code build failures (e.g. Intel compiler can't communicate with its license server), and other occasional unexplained failures (e.g. disk I/O failures), etc.

2) Some "red" will be "`ATDM Sev: Nonblocker`" failing tests that are allowed to continue to fail in order to allow developers a chance to clean them up and still get feedback if they are successful (as opposed to disabling the tests and then getting no feedback).

Alternatively, ***a lack of "red" (i.e. only "green") may be due to some builds not submitting results to CDash*** for

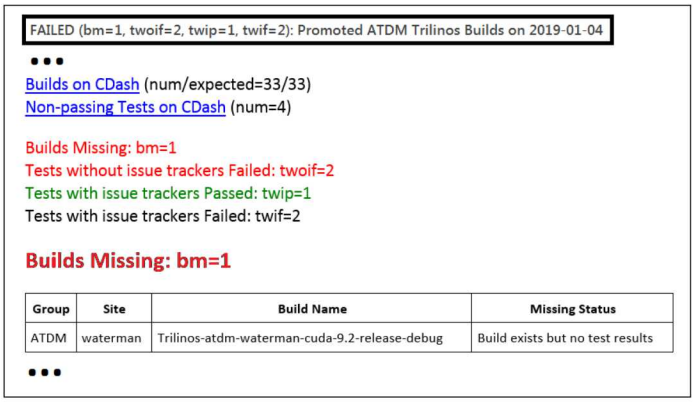

Fig. 5. CDash analysis email part 1 for 2019-01-04: email summary line & body header

a variety of reasons such as system maintenance, upgrades, overloading of shared build nodes crashing the build, crashing of jenkins jobs, overloaded batch queues not running build jobs, and the CDash site not accepting data.

Therefore, since one cannot just look at the lack of "red" on the dashboard, a different approach is required. To address all of these issues, the new ***command-line Python tool*** `cdash_analyze_and_report.py` was developed. The inputs to this tool are the CDash base site, the CDash project name, the filters for the builds page `cdash/index.php`, the corresponding filters for non-passing tests for the page `cdash/queryTests.php`, CSV files listing the expected builds and failing tests with known issues trackers, and the CDash testing date (e.g. "2019-01-04"). Given that information, this tool pulls build and test data using the CDash REST API, analyzes the data, generates an HTML report, emails that report to a given set of email addresses, and returns `0` for "PASSED" and nonzero for "FAILED" to the shell.

An example email generated by this tool (broken into two parts) is provided in Figures 5 and 6. The top part of the email in Figure-5 shows the subject line "`FAILED (bm=1, twoif=2, twip=1, twif=2) Promoted ATDM Trilinos Builds on 2019-01-04`" which gives acronyms for the types and number of failures and other information where the acronyms are spelled out at the top of the email body. In this example, there was one build missing test results (`bm=1`). There were two new test failures that don't yet have issue trackers created for them (`twoif=2`) (therefore they may need to be triaged and have Trilinos GitHub Issues created for them). There was one test with an issue tracker that was previously failing but is now passing (`twip=1`) (therefore it might be fixed and one may be able to close the corresponding GitHub issue). And finally, it shows that there were two failing tests that already have issue trackers (`twif=2`) (therefore, the Trilinos developers already know about the failures and should be working on a fix).

The second part of the email body (Figure-6) tabulates the tests in each of these categories. These tables have hyperlinks to CDash and include the number of days the test has been consecutively passing/non-passing and history of passing/non-passing (last 30 days). This test history is incredibly useful and massively reduces the amount of time that it takes to monitor the builds and tests, detect new failures, and identify possibly randomly failing tests. In the case of the failing test `Belos_BlockGmsPoly_Epetra_File_Ex_MPI_4` (table `twiof=2`), it is clear that this is the first time this test has failed over the last 30 days. But the fact that this one test failed in two different builds on the same day suggests that this test failure might be due to a new bug in Trilinos instead of a random system failure. To determine more, one must click on the "Failed" hyperlinks to examine the detailed test output. Doing this type of top-level triaging given all of this information takes just a few minutes. Having to get this information directly off of CDash by running individual queries can take hours.

This CDash reporting tool is not only used to aid in top-level detection and triaging of new failures like described above, but (as mentioned in Section II-B) is also used to determine PASS/FAIL for the Trilinos builds & tests for each APP. There is one set of CDash filter fields for SPARC and a different set for EMPIRE. This requires changing just a few command-line arguments to the tool.

Once new failures have been detected and triaged (resulting in new Trilinos GitHub issues), the last step is for Trilinos developers to follow up and resolve them in a timely way. That, of course, can turn out the be the most difficult and time-consuming step. There are three options for addressing these failures where:

> ***fixing the failure*** is typically the best option if the issue can be fixed quickly;
> ***(temporarily) disabling the failing code and/or test*** is only an option if the issue has the "`ATDM Sev: Nonblocker`" label; or
> ***reverting the git commit(s) that caused the failures*** is typically only done if the issue has the "`ATDM Sev: Critical`" label and it can't be fixed quickly.

In the case of new failures, finding the commits that need to be reverted is typically easy since the builds and/or tests passed the previous testing day. However, when standing up new builds that need to be cleaned up, one generally cannot revert git commits because one does not know when the change, if any, was made that caused the failures.

D. Organizational processes

The most difficult part of the process to achieve a stable and productive development environment for the developers is the timely resolution of Trilinos issues. The work to stay on top of and resolve these issues is more than any one person can do and must engage the Trilinos Product Area Leaders. In coordination with the Trilinos Product Area Leaders, the SNL ATDM project has set up a number of specific roles and responsibilities which include the:

Tests without issue trackers Failed (limited to 20): twoif=2

Site	Build Name	Test Name	Status	Details	Consecutive Non-pass Days	Non-pass Last 30 Days	Pass Last 30 Days	Issue Tracker
sems-rhel6	Trilinos-atdm-sems-rhel6-intel-opt-openmp	Belos_BlockGmresPoly_Epetra_File_Ex_0_-MPI_4	Failed	Completed (Failed)	1	1	22	
sems-rhel6	Trilinos-atdm-sems-rhel6-gnu-opt-openmp	Belos_BlockGmresPoly_Epetra_File_Ex_1_-MPI_4	Failed	Completed (Failed)	1	1	26	

• • •

Tests with issue trackers Failed: twif=2

Site	Build Name	Test Name	Status	Details	Consecutive Non-pass Days	Non-pass Last 30 Days	Pass Last 30 Days	Issue Tracker
mutrino	Trilinos-atdm-mutrino-intel-opt-openmp-HSW	Anasazi_Epetra_BKS_norestart_test_-MPI_4	Failed	Completed (Failed)	21	21	3	#3499
mutrino	Trilinos-atdm-mutrino-intel-opt-openmp-HSW	Anasazi_MultiVecTraitsTest2_MPI_4	Failed	Completed (Failed)	24	24	0	#3499

Fig. 6. CDash analysis email part 2: new non-triaged failures (twoif) and existing known failures (twif) (leaving out the table for 'twip')

Top-level ATDM Trilinos Triager who gets the daily CDash summary emails, creates new ATDM Trilinos GitHub Issues, and hands them off to the
Trilinos Product Area Leads who examine the Issues in their area, find *Trilinos Developers* and supporting funding, then follow up to make sure the Issues get addressed (according to priority); and the
Trilinos Developers who resolve the issues.

These roles and this process are in their infancy but the APP teams are already starting to see the benefits.

III. Conclusions and Future Work

Pre-exascale platforms, programming environments & models present challenges that the CSE community has not faced in several decades which are exacerbated due to the greater amount of more complex algorithms and software being integrated into single applications. Experience in the SNL ATDM program has shown that ***improving developer productivity in these changing pre-exascale environments requires better designed development and integration processes and better automated testing***. One must ***carefully scrutinize every failing test in order to detect new defects*** and ***must not allow existing failures to hide new failures***. Also, effectively staying on top of a larger number of automated builds and tests on these platforms ***requires an analysis tool that takes a broad view of build and tests results to show trends, commonality, and history***.

While much progress has been made to improve stability and productivity, the APP's updates of Trilinos are not yet gated on 100% passing supporting ATDM Trilinos builds & tests. There are some planned improvements that will facilitate this such as a) allowing "`ATDM Sev: Nonblocking`" tests to continue to fail but not trigger global failure, and b) detecting and automatically filtering out occasional known random system failures. In addition, work is needed to make the processes less labor intensive which includes a) simplifying creation of new GitHub Issues, and b) creating a tool to automatically update GitHub Issues about the status of their associated tests. The latter would automatically notify Trilinos developers when associated failing tests start passing and add regular reminders of still-failing tests.

References

[1] R. Bartlett. Daily integration and testing of the development versions of applications and Trilinos. Technical Report SAND2007-7040, Sandia National Laboratories, 2007.

[2] R. Bartlett and et. al. ASC vertical integration milestone. Technical Report SAND2007-5839, Sandia National Laboratories, 2007.

[3] R.A. Bartlett. Integration strategies for computational science. In *Software Engineering for Computational Science and Engineering, 2009. SECSE '09. ICSE Workshop on*, pages 35 –42, 23-23 2009.

[4] Matthew Tyler Bettencourt, Eric C Cyr, Richard Michael Jack Kramer, Sean Miller, Roger P. Pawlowski, Edward Geoffrey Phillips, Allen C. Robinson, and John N. Shadid. Empire - em/pic/fluid simulation code. 8 2017.

[5] Paul Crozier, Micah Howard, William J. Rider, Brian Andrew Freno, Steven W. Bova, and Brian Carnes. Advanced technology and mitigation (ATDM) SPARC re-entry code fiscal year 2017 progress and accomplishments for ECP. 9 2017.

[6] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202 – 3216, 2014. Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

[7] Michael A. Heroux, Roscoe A. Bartlett, Vicki E. Howle, Robert J. Hoekstra, Jonathan J. Hu, Tamara G. Kolda, Richard B. Lehoucq, Kevin R. Long, Roger P. Pawlowski, Eric T. Phipps, Andrew G. Salinger, Heidi K. Thornquist, Ray S. Tuminaro, James M. Willenbring, Alan Williams, and Kendall S. Stanley. An overview of the trilinos project. *ACM Trans. Math. Softw.*, 31(3):397–423, 2005.

[8] Andrew E. Slaughter, John W. Peterson, Derek R. Gaston, Cody J. Permann, David Andr, and Jason M. Miller. Continuous integration for concurrent moose framework and application development on github. 3(1), 2015. Exported from https://app.dimensions.ai on 2019/01/27.