Improved productivity through standardized configurations and testing of Trilinos on advanced platforms

Roscoe A. Bartlett
rabartl@sandia.gov

Joseph R. Frye
jfrye@sandia.gov

Evan C. Harvey
eharvey@sandia.gov

Department of Software Engineering & Research
Sandia National Laboratories

I. Background

Trilinos [8] is a large collection of complex software providing implementations of advanced computational science algorithms used to create cutting-edge simulation applications (APP) codes. Many important customers and projects inside of Sandia National Laboratories (SNL) rely on Trilinos and drive its internal development and usage. One particularly important customer at SNL is the Advanced Technology Deployment and Mitigation (ATDM) project. This project is focused on several different physics simulation codes that use Trilinos including SPARC [6], EMPIRE [5], and GEMMA [10]. (The details of these codes are not important to this discussion.) Trilinos and the APP codes must be kept working during active development on a number of advanced platforms on the road to exascale sized super computers. Prior to the work described below, the APP developers and Trilinos developers were suffering some serious productivity problems and frequent pain related to the stability of Trilinos on these platforms and problems with developers being able to reproduce each other's build configurations.

Trilinos software is broken down into about 60 "packages" which then can use upwards of 150 externally pre-installed Third Party Library (TPL) packages. In addition, some Trilinos packages are broken down into "subpackages" resulting in a total of more than 150 packages + subpackages. Required or optional dependencies can be created between any package or subpackage to form a directed acyclic graph (DAG) of dependencies. Individual Trilinos packages and subpackages can be enabled or disabled and will trigger the implicit enable of many upstream packages and TPLs to satisfy dependencies. ATDM currently uses 45 of the 60 Trilinos packages and 117 of the 156 total packages + subpackages. About the only Trilinos packages that ATDM does not use are legacy packages. In other words, ATDM uses almost all of the actively developed and supported software in Trilinos.

The different sets of possible package enables and disables and varying the different sets of global and package-level options likely yields hundreds of thousands or more different possible configurations of Trilinos. Most of these permutations configurations will error out during the configuration stage but many (in theory) should work. Customers only require a very small subset of the possible configuration permutations to work and therefore testing must be focused on these customer focused configurations.

Prior to the work described below, each of these ATDM APP code teams were taking it upon themselves to maintain their own tailored configurations of Trilinos; often fighting similar issues on the same platforms with no coordination across teams. They would often maintain very long CMake configuration bash scripts containing hundreds of lines and more than 100 CMake cache variable settings, which were duplicated across many different such scripts. If any one of these options is wrong, it can result in problems with Trilinos that can take hours, days or longer to debug and correct. In addition, there were no automated builds or testing of Trilinos for these configurations and no regular testing on many of the advanced platforms that drive the ATDM project. These advanced platforms include a variety of bleeding-edge node and accelerator architectures including systems with multiple GPUs per node and various CPU architectures (e.g. Intel Knights Landing, ARM-based systems, etc.). ATDM APP developers and even ATDM Trilinos developers would often encounter broken code when pulling updated versions of Trilinos and trying to build on the various ATDM platforms. When APP developers experienced a broken Trilinos configuration, it was often difficult for Trilinos developers to reproduce the cmake configuration, build, and runtime errors that the APP developers were seeing. This resulted in a serious degradation in the productivity of the APP developers pulling updated versions of Trilinos and also for Trilinos developers who tried to reproduce and address problems with Trilinos.

Over the last two years, many areas of the development, testing, and integration processes related to Trilinos in the ATDM project were selected for development in an effort

SAND2020-6704 C: 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 Energy's National Nuclear Security Administration under contract DE-NA0003525.

to improve productivity over existing approaches described above [3]. Below, we describe the problem of managing configurations of Trilinos for the ATDM APP codes on the targeted advanced platforms in a coordinated way and ensuring the general stability of updated versions of Trilinos for the code teams on these platforms. Teams outside of this project helped improve APP developer productivity by adding basic pull request testing before merging changes to the main Trilinos 'develop' branch as well as APP-specific testing against Trilinos.

II. Standardized configurations of Trilinos for ATDM

As a foundation to address the problems with Trilinos in ATDM, an extensible system for specifying the configurations of Trilinos was created[1]. The best way to introduce this system is to show how a Trilinos or APP developer uses it to configure, build, test, and install Trilinos on any supported system. After cloning the Trilinos git repository (path `<trilinos>`) on any given system, to configure, build, and test any subset of Trilinos packages, one just does:

```
cd <some_build_dir>/

source <trilinos>/cmake/std/atdm/load-env.sh \
  <build-name>

cmake -GNinja \
  -DTrilinos_CONFIGURE_OPTIONS_FILE:STRING=\
    cmake/std/atdm/ATDMDevEnv.cmake \
  -DTrilinos_ENABLE_<Package1>=ON \
  -DTrilinos_ENABLE_<Package2>=ON <...> \
  -DTrilinos_ENABLE_TESTS=ON \
  <trilinos>

ninja -j16

ctest -j4
```

The command `source <...>/atdm/load-env.sh <build-name>` automatically determines the system being used. The standard configuration options are extracted from keywords embedded in the `<build-name>` string (see below). This sets up the shell environment for the requested configuration (including loading the desired compilers, MPI implementation, and the associated TPLs) and sets other environment variables for options specified as keywords in the `<build-name>`. The raw `cmake` command that configures Trilinos includes the CMake fragment file `ATDMDevEnv.cmake`. The CMake code in this file pulls build options out of the shell environment set by the `load-env.sh <build-name>` command and then sets various CMake cache variables that fully specify the configuration of any Trilinos package selected to be enabled. In this way, APP or Trilinos developers can enable any subset of packages they want to develop on or to reproduce failures.

[1] `https://github.com/trilinos/Trilinos/wiki/ATDM-Trilinos-Builds`

The `<build-name>` can contain any number of letters and numbers along with keywords separated by the characters _ or -. The categories of keywords includes `<system_name>`, `<kokkos_arch>`, `<compiler>`, `<kokkos_thread>`, `<rdc>`, `<complex>`, `<shared_static>` and `<release_debug>`. Any missing keywords have defaults (except for `<compiler>` but the keyword `default` will select the default compiler and MPI for the system). An unrecognized substring not matching a defined keyword in `<build-name>` is ignored. Examples of valid build names include `intel-openmp-opt`, `ats1-knl_intel_openmp_static_dbg`, and `ats2-cuda-rdc-complex-static-opt`. The currently supported systems include Advanced Technology System 1 (ATS-1, i.e. Trinity), ATS-2 (i.e. Sierra), Commodity Technology System 1 (CTS-1), Vanguard 1 (i.e. ASTRA) and many other commodity and advanced technology systems at SNL and other laboratories supporting the ATDM program.

The included file `ATDMDevEnv.cmake` also disables all of the Trilinos packages and subpackages that none of the ATDM customers need. Developers and specific customers enable whatever subset of packages they desire setting `Trilinos_ENABLE_<Package>=ON`. One can also enable the superset of all of the Trilinos packages used by the ATDM APP codes by setting the `Trilinos_ENABLE_ALL_PACKAGES=ON`. This is called the "blacklisting" for managing the configurations of Trilinos and is a very successful stategy of which most Trilinos developers and users are not aware. This approach ensures that any Trilinos package that does get enabled has the same configuration independent of any other Trilinos packages that get enabled. For example, if the Tpetra package gets enabled, then it will have the identical configuration regardless what other Trilinos packages also get enabled. This is a critical property that is exploited in automated testing of Trilinos to support ATDM.

New platforms and systems are added in an extensible way by adding a new subdirectory `Trilinos/cmake/std/atdm/<system_name>` which typically includes a single bash shell source file `environment.sh`. In addition, custom configurations can be added by either passing a second argument to `source <...>/atdm/load-env.sh <build-name> <some-base-dir>/<custom-system-name>` or registering a new configuration with `export export ATDM_CONFIG_REGISTER_CUSTOM_CONFIG_DIR= <some-base-dir>/<custom-system-name>` and then including `<custom-system-name>` keyword in the `<build-name>`. In addition to CUDA, compilers, and MPI, the ATDM Trilinos configuration requires up to 17 different external TPLs to be pre-installed on a system before an ATDM Trilinos configuration can be be stood up. The process for installing these TPLs on a new system is non-trivial and remains the most significant challenge in getting ATDM builds of Trilinos up and running on a new platform (but is beyond the scope of this discussion). The file `atdm/`

<system_name>/environment.sh loads modules and sets up environment variables that point these pre-installed TPLs.

There are numerous advantages to this approach for managing Trilinos configurations:

- The set of "knobs" exposed is reduced from hundreds of CMake cache variables to just a handful that are supported in the <build-name> keywords, thus avoiding many of the pitfalls of Trilinos configuration.
- Trilinos developers that need to reproduce Trilinos builds and tests can work with raw cmake, make/ninja and ctest commands instead of having to dig through layers of wrapper scripts.
- APP developers only need to list the Trilinos packages their APP directly use (more on that below).
- The correct system ('ats1', 'ats2', 'cts1', etc.) is determined automatically in most cases.
- It is straightforward to add new systems, compilers, etc., to extend the set of builds supported.
- Developers can provide their own custom system configurations satisfying the open-closed principle [7].

The software supporting this ATDM Trilinos configuration system is written in a combination Bash and CMake. Unit testing is in place for many of the Bash functions that are used in the system (using the shunit2[2] test harness). While bash is not the best programming language, it is well suited to situations where the primary role is to manage environments including loading modules, updating the various PATH variables, etc.

III. ATDM Application usage of Trilinos

ATDM APPs that use this ATDM Trilinos each have a *.cmake fragment file that enables the packages needed for that APP code. For example, to configure, build, and install Trilinos to be used by the EMPIRE APP, one does:

```
cd <some_build_dir>/

export ATDM_DIR=<trilinos>/cmake/std/atdm
export EMPIRE_dir=$ATDM_DIR/apps/empire

source $ATDM_DIR/load-env.sh \
  <build-name>

cmake -GNinja \
  -DTrilinos_CONFIGURE_OPTIONS_FILE=\
    $ATDM_DIR/ATDMDevEnv.cmake, \
    $EMPIRE_DIR/EMPIRETrilinosEnables.cmake \
  -DCMAKE_INSTALL_PREFIX=<trilinos-install> \
  <trilinos>

ninja -j16 install
```

The version-controlled file EMPIRETrilinosEnables.cmake contains set() statements to enable the Trilinos packages used by the EMPIRE APP.

[2]https://github.com/kward/shunit2

Once Trilinos is installed, to build the APP code (EMPIRE in this case) one can load the matching environment for that Trilinos install using:

```
source <trilinos-install>/load_matching_env.sh
```

This loads the same CMake, Ninja, compilers, MPI, and any other tool needed to use Trilinos and also sets the environment variable Trilinos_ROOT=<trilinos-install>. Therefore, the CMake find_package(Trilinos) command inside the APP's CMakeLists.txt file will automatically find this Trilinos install.

This largely eliminates the need for complex wrapper scripts to drive the configuration, building and installing of Trilinos and then the usage of Trilinos by an APP code.

The difference between the SPARC and EMPIRE usage of Trilinos is the set of Trilinos packages that they use. Most of the packages are identical but there are some Trilinos packages that are only used by SPARC and some that are only used by EMPIRE. In the case of GEMMA, it uses a subset of the packages used by both SPARC and EMPIRE but Trilinos must be configured with support for complex floating-point numbers while SPARC and EMPIRE do not require complex support. But other than that, all of the ATDM APP codes can be built against the exact same installation of Trilinos (where one enables the super-set by setting Trilinos_ENABLE_ALL_PACKAGES=ON). This comes in very handy when setting up automated builds as described below.

IV. Stabilization of Trilinos

Just having a lot of different configurations for Trilinos organized in the way described above is useless if Trilinos is broken. Therefore, a critical aspect of providing working configurations is to put strong testing processes in place. Maintaining stable versions of Trilinos requires setting up and running automated builds, running the native Trilinos test suite, monitoring those builds and tests, and then triaging and resolving failures in a timely way once they are discovered. The full set of these processes required [3] is beyond the scope of this discussion. Below, we just focus on the setup and stabilization of automated builds of Trilinos.

The black-listing approach described above allows for us to set up a single set of builds for each platform and build configuration and then enable the entire superset of packages used by all of the ATDM APP codes together. In this way, we don't need to set up and run different Trilinos builds specifically for SPARC, EMPIRE and GEMMA in order to well test Trilinos for usage by these different APP codes (even each APP code only uses a subset of these packages). We can just enable the superset of packages by configuring with Trilinos_ENABLE_ALL_PACKAGES=ON. For example, for the ATS-2 platform, only one ats2-cuda-gnu1-spmpi_static_opt build needs to be tested for both SPARC and EMPIRE. A specially selected set of build configurations on each platform is run nightly on the tip of the Trilinos develop branch. At the time of this writing, this

includes 46 Trilinos builds that submit to the Trilinos CDash site, covering all of the important ATDM platforms. Currently, this includes eight different platforms which include builds for various versions of the Clang, GCC, and Intel compilers, CUDA, and various MPI implementations and versions.

Only the specific build configurations that are used by that APP are set up and run. That is, every permutation of possible `<build-name>` keywords is not tested. These Trilinos builds are run once a day and are posted to the Trilinos CDash site[3]. The configure, build and test results posted to CDash are then monitored automatically using a custom CDash summary tool called `cdash_analyze_and_report.py` which maintains a issue status table that is updated and emailed on a nightly basis.

Failures reported using the `cdash_analyze_and_report.py` tool are then triaged and GitHub issues are created to alert Trilinos developers. Each of these GitHub issues includes the exact set of commands needed to reproduce the failures using the ATDM Trilinos configuration described in Section I. The name of the build that shows up on CDash (e.g. `Trilinos-atdm-ats2-cuda-10.1.243-gnu-7.3.1-spmpi-rolling_static_opt`) can be directly passed as the `<build-name>` in the command `source <...>/atdm/load-env.sh <build-name>`. This has allowed any Trilinos developer to trivially reproduce any build on any machine shown on CDash in a uniform way.

Side Note: The computational load to run these builds and tests is too great to do more than a single set of builds in a 24 hour day. In fact, many days, we don't even get full build and test results because of the other loads on the key advanced architecture machines where these builds have to be run. Therefore, other testing workflows that might require running this set of builds multiple times in the same testing day are just not practical.

V. Application integration testing

Each of the ATDM APP codes maintain their own fork of Trilinos and only update their fork when a sufficient set of APP tests pass. This insulates the APP developers and users from the majority of defects that get injected in the Trilinos `develop` branch. (In the early years of ATDM, APP codes directly pulled from the main Trilinos `develop` branch which often resulted in significant pain and lost productivity.) Testing the tip of the Trilinos `develop` branch against customer codes has been a critical activity for many past projects [1], [2], [4], [9] and the ATDM project is no different. In fact, it is more important given the challenges of maintaining working code on these advanced platforms. In the ATDM project, daily integration testing is performed for the SPARC and EMPIRE APP codes against Trilinos `develop`. Two very different integration testing processes are currently used for SPARC and EMPIRE. Below, we describe the Trilinos integration testing process for the SPARC APP.

[3]`https://testing.sandia.gov/cdash/`

The approach taken for SPARC Trilinos integration testing is to leverage the automated builds of Trilinos already being performed on the various platforms described above. The way this works is that the automated builds of Trilinos that post to CDash also includes the install of Trilinos as a byproduct of the build and test process. Each Trilinos build is installed into a date-based directory with the format `<base-dir>/YYYY-MM-DD/<build-name>` where `<build-name>` is the same as the Trilinos CDash build name (minus the leading `Trilinos-atdm-` prefix). There are then SPARC Trilinos integration builds that run the next day that build against the Trilinos installs from the previous day and post results to the SPARC CDash site under the "Trilinos Integration" CDash group. For example, the build `Trilinos-atdm-ats2-cuda-gnu1-spmpi_static_opt` for the testing day `2020-06-25` is installed under `<base-dir>/2020-06-25/ats2-cuda-gnu1-spmpi_static_opt/` and the matching SPARC Trilinos integration build for the next testing day `2020-06-26` builds against that Trilinos install. As these SPARC Trilinos integration builds show up on the main SPARC CDash site, they are viewed and failures are triaged along with the other automated builds of SPARC (against their older currently accepted installs of Trilinos).

These integration testing processes have been a key part of maintaining the stability of Trilinos for the ATDM APP codes while also allowing for frequent, less risky, and less time consuming updates of Trilinos.

VI. Impact on productivity

The work setting up and maintaining the ATDM Trilinos builds on all of these different advanced and supporting platforms represents the most significant improvement in Trilinos portability testing and stabilization in the 20+ year history of the project. The creation of this ATDM Trilinos configuration system, the automated testing of Trilinos on all these platforms, and the resulting stabilization of Trilinos has improved the productivity of ATDM APP developers significantly. These developers and users of Trilinos spend far less time dealing with broken Trilinos builds, having to triage runtime defects in Trilinos code, and having to report them back to the Trilinos development team. The improved stability of Trilinos has also positively impacted non-ATDM customers as well. Simply put, pulling updated code that does not build or run correctly is extremely disruptive and kills productivity (paraphrase from an ATDM APP developer a couple of years ago). Beyond that, broken software degrades trust and has other larger impacts that are difficult to directly measure. These efforts have significantly reduced the pains experienced by the APP development teams.

The impact on the productivity of Trilinos developers themselves due to these efforts is less clear. On one hand, configuring and building Trilinos on these various platforms to reproduce Trilinos bugs has been made much easier for Trilinos developers (and less work for APP developers to explain this to Trilinos developers). On the other hand, running

and cleaning up the native Trilinos test suite on all of platforms and build configurations has been a lot of work which has fallen on the Trilinos developers. (It is difficult to detect new failures if you already have large numbers of perpetually failing tests in a given build.) Now, when most build or runtime defects get injected into Trilinos, the native Trilinos test suite will often catch them and then Trilinos developers will need to triage the failures and fix them. Before, it was often the APP developers who experienced broken builds and runtime behavior in Trilinos through running their APP's test suite. And then it was the APP developers who had to do the first round of triaging of Trilinos defects and then report them. That saved the Trilinos developers some upfront work. So the net impact on the productivity of these efforts on Trilinos developers themselves is unclear.

Our goal is to improve ATDM APP developer productivity by reducing their bug reporting, triaging, and removal effort across the entire ATDM program; conversations indicate that a good deal of the Trilinos debugging burden was shifted from ATDM APP developers to the Trilinos developers.

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. Bartlett and J. R. Frye. Creating stable productive cse software development and integration processes in unstable environments on the path to exascale. pages 1–8, 2019.

[4] 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.

[5] 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.

[6] 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.

[7] R. Martin. *Agile Software Development (Principles, Patterns, and Practices)*. Prentice Hall, 2003.

[8] The Trilinos Project Team. *The Trilinos Project Website*.

[9] John A. Turner, Kevin Clarno, Matt Sieger, Roscoe Bartlett, Benjamin Collins, Roger Pawlowski, Rodney Schmidt, and Randall Summers. The virtual environment for reactor applications (vera): Design and architecture. *Journal of Computational Physics*, 326:544 – 568, 2016.

[10] Langston William and et. al. Massively parallel frequency domain electromagnetics simulation codes. Technical Report SAND2018-0700C, Sandia National Laboratories, 2018.