SANDIA REPORT

SAND2007-7040
Unlimited Release
Printed October 2007

Daily Integration and Testing of the Development Versions of Applications and Trilinos

A stronger foundation for enhanced collaboration in application and algorithm research and development

Roscoe A. Bartlett

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of Energy's
National Nuclear Security Administration under Contract DE-AC04-94-AL85000.

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2007-7040
Unlimited Release
Printed October 2007

Daily Integration and Testing of the Development Versions of Applications and Trilinos

A stronger foundation for enhanced collaboration in application and algorithm research and development

Roscoe A. Bartlett
Department of Optimization and Uncertainty Estimation

Sandia National Laboratories, Albuquerque NM 87185-1318 USA

Abstract

Daily integration of the development versions of the application code Charon and the numerical solvers collection Trilinos has help us to create a new vision for a deeper level of collaboration between solver developers and application developers which benefits everyone involved. The bridge for this deeper collaboration is based on the foundation of nightly building and testing of the combined development versions of the application code (e.g. Charon) and Trilinos. Here we outline the proposed daily integration and testing process, describe potential advantages and disadvantages, and describe our experiences with Charon + Trilinos Dev during the ASC FY07 Level-2 Vertical Integration Milestone.

Acknowledgments

I would like to thank Scott Collis, Mike Heroux, Russell Hooper, Roger Pawlowski, Andy Salinger, Jim Willenbring and many others for helpful conversations that helped shape this idea. I would also like to thank all of the members ASC FY07 Level-2 Vertical Integration Milestone effort in helping to provide an environment where these ideas could be formed and take hold and in helping to set up and maintain Charon + Trilinos Dev.

Contents

Appendix

1 Introduction

During the course of the ASC Level-2 Vertical Integration Milestone work [3], we found that in order to keep moving forward and avoid backslides in capability (which happened early on), we needed to implement nightly building and testing[1] of the development versions of Charon and Trilinos [4]. Every night, we take what is in the Charon and Trilinos development repositories and build the combined Charon + Trilinos application and run a large set of regression tests. This work had broad implications on the nature of the interaction of applications (APPs) and Trilinos solvers.

In the execution of this daily integration and testing process, we have learned many things about how to do quasi-continuous integration[2] of application and algorithms software and we have realized many important unplanned benefits. This will allow us to reap many more benefits in the future if this process is maintained and extended. There are both production-related benefits and research-related benefits to daily integration that help both the application developers and the algorithm developers to achieve their goals. On the research side, this significantly reduces the overhead required for algorithm developers to try their algorithms out on production quality problems. Developing a numerical solver with a production problem exposes the algorithm developer to a whole host of issues (e.g. poor scaling, ill-conditioning, convergence difficulties, etc.) that are hard to replicate with model problems. On the production side, constant integration insures that the application and Trilinos are always up to date and satisfying the application's requirements. Therefore, when it is time for a release, only a final set of acceptance tests are required and then the codes can be branched and released shortly after. This helps to reduce a whole host of risks such as slipped schedules and broken features[3].

I have seen several different scenarios over the years where this daily integration and testing infrastructure would have facilitated collaboration between application and algorithm developers for the advantage of both. For example, suppose an application developer is running a new problem and discovers some strange behavior from a numerical solver. The algorithm developer may look at the results and speculate what the cause of the behavior might be or if a different variation of the algorithm might help. However, if the algorithm developer is stuck having to use a released version of Trilinos, it will be more difficult to make any major changes in order to investigate the behavior. Also, there may already be improvements made to the algorithm in the development branch of Trilinos that may be able to address the problem. Without the infrastructure of daily integration and testing, it may not be cost effective or practical to bring the development versions of the application and Trilinos up to date in order to try the updated algorithm. The algorithm and application developers may have to wait until the next major release of Trilinos before the new algorithms can be accessed. This time delay works to no one's advantage and can kill the collaboration. With daily integration and testing in place, an easy path for collaboration is maintained where the Trilinos algorithm developer can try their latest and greatest algorithms on these types of challenging production problems and the information learned from trying to solve these production problems can feed back into algorithm development. To some extent, this back and forth development and integration already happens, but without a foundational process in place to streamline it, the bidirectional flow can be greatly restricted.

[1] Charon Dev and Trilinos Dev has been nightly built and tested separately for many years. It was the combined Dev versions of Charon + Trilinos that was not nightly built and tested until the milestone work.

[2] By quasi-continuous integration, we mean that we are suggesting integrating and testing once a day and not every few hours as some have advocated [1].

[3] Also known as regressions

Another example where daily integration and testing of the development versions of the application and Trilinos would ease the path for collaboration is when an application developer wants to try a new capability in Trilinos without a lot of work[4]. When the up-to-date development version of Trilinos is available from within an application, a new algorithm or capability from Trilinos can be accessed much more easily. Typically, even a small increase in the overhead needed to try out a new Trilinos capability in an application can be enough to kill a potentially fruitful collaboration between application and algorithm developers. Daily integration and testing of the development versions of the application and Trilinos removes a unexciting (and therefore demoralizing) but critical obstacle to collaboration and impact.

Daily integration and testing of an application code and Trilinos brings algorithm developers and application developers closer together – exchanging ideas and concerns – and refocuses Trilinos developers on customer efforts while still helping drive publishable numerical algorithm solver research and reduces barriers for new algorithms to have impact through production application codes.

For the reminder of this paper, I will refer to the combined development versions of the application (e.g. Aria [2], Charon[5], Xyce[6] etc.) and Trilinos as APP + Trilinos Dev.

[4] This example really happened and it was the inability to readily access the development version of Trilinos within the application that killed the collaboration.

[5] `http://micro.sandia.gov/charon.html`

[6] `http://www.cs.sandia.gov/Xyce`

2 Outline of proposed APP + Trilinos Dev development and daily integration and testing

The basic idea of this new approach goes like this (which is mostly what we already do with Charon + Trilinos Dev right now):

APP developers mainly work with APP + Trilinos Release: The developers of APP (e.g. Charon, Alegra, SIERRA) would mostly do their production development and run their production test suite against a stable released version of Trilinos.

Trilinos developers mainly work with Trilinos Dev: Most of the developers of Trilinos would do their development work primarily just within Trilinos and run the Trilinos test suite. If radical changes are make, then there should be a way for Trilinos developers to build and run APP + Trilinos Dev and it's test suite to see if anything will break (before a check in). This will require some effort to set up and maintain for each APP (see practices in Section 4).

Hide APP + Trilinos Dev "research" work in APP behind ifdefs: Any "research" development done in APP against Trilinos Dev must be hidden behind #ifdefs so as not to impact the other production-focused development work being done with APP that relies on the more stable release of Trilinos. Differences between Trilinos versions with respect to current APP code would also be handled with #ifdefs.

Perform nightly building and testing of APP + Trilinos Release and APP + Trilinos Dev: In performing this process, it is our goal to avoid unnecessary interactions between APP and Trilinos developers. If a Trilinos developer breaks some software in Trilinos, then we want to avoid bothering an APP developer that has nothing to do with this and no reason to know about this. Likewise, if an APP developer breaks the APP code in some way that does not expose a Trilinos defect, we want to avoid bothering Trilinos developers with this[7]. The following builds are designed to help avoid unnecessary communication between APP and Trilinos developers while still catching APP + Trilinos Dev integration problems as efficiently as possible:

- ***APP + Trilinos Release tested against "production" test suite***: The Dev version of APP would be built and tested against the static/stable current release of Trilinos.
  - ***Send "production" failures only to APP developers***: Test failures for APP + Trilinos Release should only be forwarded to APP developers, not Trilinos developers, since these failures are most likely caused by a defect added by an APP developer. It is possible that latent defects in the current release of Trilinos may be the cause of the failure, but this should be less likely.
- ***APP + Trilinos Dev tested against "research" and "production" test suites***: The combined APP + Trilinos Dev code with enabled #ifdefs and extended "research" test suite would be built and tested (with both "production" and "research" tests).
  - ***Only send "production" failures that did not also fail in APP + Trilinos Release to Trilinos developers (representative)***: Only "production" tests that failed in this version

[7] We did not have unnecessary interactions totally under control for Charon + Trilinos Dev during the milestone work and it resulted in some waisted time. See the practices for how this can be addressed.

that did not fail in the production APP version would be forwarded to Trilinos developers (e.g. to the dedicated APP + Trilinos Representative, see below).

- ***Send "research" failures only to Trilinos developers (representative)***: By default, average APP developers would not see these "research" test failures (unless they wanted to in which case they could be added to an e-mail list).

Release APP + Trilinos together or staged : A release of the APP + Trilinos would go one of two routes:

- ***a) Combined tagging and release of APP + Trilinos***: Right before a release of APP, the APP and Trilinos would be tagged and branched at the same time to make sure both are as current as possible. This could be very challenging to pull off since it would require that Trilinos be releasable at almost any time.
- ***b) Staggered releases of Trilinos and APP***: APP developers make a decision to target a release of APP against a very recent release of Trilinos. At the point of the release (or branch for the release) of Trilinos, the Dev version of APP drops support for older release of Trilinos, moves code from within the protected #ifdefs into main Dev build and then any new work with Trilinos Dev is hidden behind new #ifdefs. As a variation of this, the APP may not be in a position to upgrade to the most recent release of Trilinos right away. In this case, the nightly test harness can be setup to build the APP against three versions of Trilinos: i) the old Trilinos release, ii) the new Trilinos release, and iii) the development version of Trilinos. This requires a little more effort but it guarantees that when the APP developers decide to transition the APP code base to the new Trilinos release, that this will happen smoothly without any problems. At this point, support for the old Trilinos release can be dropped, and the APP developers can do a release of APP at any time that is convenient.

Continue APP + Trilinos Dev nightly building and testing after APP upgrades to new Trilinos release: After APP is upgraded to the next release of Trilinos, using any of the approach approaches described above, the nightly building of APP + Trilinos Dev continues where future incompatibilities between Dev versions of APP and Trilinos are hidden behind new #ifdefs.

3 Advantages and disadvantages to daily integration and testing of APP + Trilinos Dev

There are many research and production advantages to maintaining the daily integration and testing of the development versions of production application (APP) and Trilinos. Some of the more significant advantages are described in the following sections.

3.1 Research advantages to daily integration and testing of APP + Trilinos Dev

Here are some of the research advantages to maintaining the daily integration and testing APP + Trilinos Dev:

Reduces overhead for initial algorithm integration: This massively reduces the overhead needed for a Trilinos algorithms developer to try out a new algorithm on a production quality problem implemented in APP because the codes will build right away. One of the most difficult aspects of doing serious algorithm research is having access to serious (possibly messy) production quality problems.

Improves chances that new algorithms will have impact: Reducing overhead of the initial software integration effort improves the chances that a new algorithm or package will benefit a real application and therefore show real "impact" which is always a criticism for research-driven algorithm developers. This is related to the previous point but is worth mentioning by itself.

Preserves interesting/challenging problems: This preserves really interesting problems that will be constant drivers for future algorithm research. A really challenging numerical test problem that is encountered after an initial algorithm integration and experimentation effort can be preserved so that algorithm researchers can come back to it later again and again to try out new versions of their algorithms. Spending a lot of effort to get an algorithm integrated with some production code and then having that connection and the working examples lost happens again and again with our current environment. Even more important is the ability to show "progress" as our algorithms improve by running them on the same basic production physics problems. Note that preserving interesting test problems is more of an issue for higher-level algorithms like sensitivity solvers and optimization than it is for more basic algorithms like linear solvers (and the associated preconditioners). The ability to solve a linear system is a prerequisite for any (semi)implicit forward simulation solver and therefore challenging linear systems are ubiquitous in scientific computing. Since higher-level analysis problems are not part of low-level modeling and forward solver work, they are often overlooked by APP developers and when the time comes to perform these higher-level analysis, the APP infrastructure to support such methods is gone. Daily integration and testing for these higher-level analysis problems preserves them for future research and future use on critical problems.

3.2 Production advantages to daily integration and testing of APP + Trilinos Dev

Here are some of the production advantages to maintaining the daily integration and testing of the development versions of a production application and Trilinos:

Expands testing for Trilinos: The tests in the APP's test suite represents an extended test suite for Trilinos. In our work with Charon, we have already seen cases where the Charon test suite caught an error that the Trilinos test suite did not. Most APPs also build against an installed version of Trilinos so nightly building and testing of APP + Trilinos Dev helps to test the installation of Trilinos which is very hard to do for Trilinos by itself.

Enables better scalability testing for Trilinos: Production APPs provide ready access to large-scale parallel problems that can serve as a vehicle for testing the parallel scalability of Trilinos algorithms. Teuchos timers can be used which make it easy to isolate timings for specific algorithm features and will show load imbalances if they exist. Automated scripts can query these timings (from the output) and can catch any problems with scalability that are seen.

Reduces time to detection of defects: This is directly related to expanded testing described in the above items but defects introduced in Trilinos code between releases that break customer functionality (but may not be caught by native Trilinos tests) will be caught right away. Even if a test failure is not fixed right away, knowing exactly when a test failed is an extremely useful piece of information in being able to track down and fix the defect later. The cost of fixing defects increases the longer the time between when the defect is introduced and the time it is first detected [5]. Daily testing reduces the risk that a release of Trilinos will regress and/or helps to contain costs and the schedule if a broken feature must be preserved in the next combined release of APP + Trilinos.

Reduces release time and effort: It reduces (or eliminates) the time needed to create a bundled release of APP + Trilinos. This removes uncertainty about how long it will take to put out a release of Trilinos and/or APP.

Allows for more aggressive refactorings and code improvements: It allows for more aggressive refactorings of Trilinos code since the impact of the refactorings on important APP customers can be ascertained and fixed right away. This will allow the architecture to evolve as needed in a safer and less demanding way. The ability to refactor code is the number one issue in making sure a code does not become "legacy code". Without the ability to refactor a code, you automatically insure that the code will be thrown away or relegated to "legacy code" at some point [5].

Better address customer needs: This will bring Trilinos algorithm developers closer to important Trilinos APP customers so that customer needs can be addressed more effectively in a timely manner.

Reduces all kinds of risk: Overall, this simply reduces all kinds of risks, increases predictability of the development and release process, and makes Trilinos more responsive to important customers.

3.3 Potential disadvantages to daily integration and testing of APP + Trilinos Dev

Here are some of the potential disadvantages to maintaining the daily integration and testing of the development versions of a production application and Trilinos:

Will slow down day-to-day development to varying degrees: It will slow down the day-to-day development of Trilinos and the APP to some extent in that problems are dealt with as they are uncovered by daily building and testing. The amount of extra overhead will depend on how

aggressively failing tests are addressed (see the practices in Section 4). Just keeping APP + Trilinos Dev building should not impart much overhead at all is most cases.

Will require better, more coordinated management practices: This will require some more sophisticated management practices and tools to keep all of this running smoothly. This will require some additional effort over what is done now with more up-front effort to set up.

Will impose greater responsibility to meet customer needs: Trilinos developers will have a greater responsibility to meet important APP customer needs. A lot of algorithm researchers don't want to sign up for that kind of responsibility. However, such individuals don't have to write or maintain production algorithmic capabilities. There will always be a place for more pure algorithms research and more theoretical (i.e. less applied) algorithm researchers.

Could increase overall development effort: It may increase the overall development time for Trilinos if not managed well. However, experience by other projects and organizations suggests that the overall development time and effort to create and maintain production capabilities should actually decrease! [5].

4 Suggested practices to support proposed APP + Trilinos Dev development and daily integration and testing

Successfully implementing APP + Trilinos Dev daily integration and testing will require, or will be made much more effective, by the adopting several practices. Some of the more important practices and issues consider are described below (and are summarized in the Appendix).

Separate "production" and "research" tests: Any new test or example added to APP's test suite based on new features in Trilinos Dev must be added as new "research" tests as not to affect existing "production" tests. In this way, we can easily differentiate between "production" regression tests and new "research" or "pre-production" tests. This is not full-proof in differentiating defects in APP code verses Trilinos code, but this will go a long way in helping to suggest where the problem is and avoid unnecessary communication between APP and Trilinos developers.

Refactor APP code to isolate and separate ifdefed code: In order improve the maintainability of the APP code to handle new features and changes in Trilinos Dev verses the current Trilinos release, the APP code should be carefully refactored to segregate and isolate ifdefed code. As much as possible, developers should try to refactor code so that changes that are made in the APP are orthogonal to differences between Trilinos Dev and the current release of Trilinos. This can be done by separating the essence of the differences into well defined functions where one ifdefed version contains code for the current Trilinos release and the other ifdefed version contains code for Trilinos Dev. New features implemented against Trilinos Dev are typically easier to handle since these should be put into new classes and new functions, thereby not disturbing existing code. This type of refactoring should be performed to avoid lots of ifdefs in long functions that can result in defects entering the code and can complicate development. Also, segregating the ifdefed code for APP + Trilinos Dev is needed to remove APP + Trilinos Dev from the daily consciousness of the average APP developer who should not have to be constantly distracted by an unrelated ongoing APP + Trilinos Dev development efforts.

Maintain a dedicated machine for building and testing APP + Trilinos Dev: Having a dedicated, powerful machine for supporting APP + Trilinos Dev would make it easy to maintain an environment to build and test APP + Trilinos Dev. It would also provide Trilinos developers a quick and easy way to access, build and test APP + Trilinos Dev. Issues like keeping third party libraries up to date would only need to be handled on this one machine. Accounts would be granted as needed and would only require SRN access. This would allow any Trilinos or APP developer with an SRN account to quickly log onto the dedicated machine, and then be able to quickly build APP + Trilinos Dev and run the "research" and "production" test suites. Some helper scripts and examples also need to be in place to show how to do this.

Appoint a dedicated APP + Trilinos Representative: One member of the Trilinos or APP development teams should be designated as the point person for the APP + Trilinos Dev effort. This person would be responsible for filtering test failures and forwarding issues to APP or Trilinos developers. This person must be familiar with the APP and Trilinos for this to be effective. Ideally, this person will be a co-developer of APP and Trilinos, so there would be little-to-no learning curve. Having only one person be responsible (with perhaps a backup person in their absences) will make it clear who is accountable for making sure issues are dealt with in a timely manner. This person would take the major responsibility of maintaining the dedicated APP + Trilinos

machine described above. The goal is that this job should not take too much effort if everyone else is doing their job. However, this job could be a nightmare if this effort is not taken seriously by everyone involved (including management). In addition, this responsibility should come with a specific project/task (P/T). Having a specific P/T serves several purposes. First, it lets the APP + Trilinos Representative know that this task is ordained and supported from management. Second, it allows us to track how much time and expense is going into keeping APP + Trilinos Dev working.

Provide easy access for any Trilinos or APP developer to build, test, and develop APP + Trilinos Dev: Trilinos and APP developers need a quick and painless way to build APP + Trilinos Dev in order to diagnose and fix failures. This may include the ability to checkout and change APP code and Trilinos code to fix the problem. Of course, modified APP code would need to go through a code review by APP developers before it was checked into APP's repository. Likewise, an APP developer should be able to change and fix Trilinos code if they are so motivated. Again, a code review by Trilinos developers should be done before an APP developer checks in any code changes to Trilinos. Having a dedicated machine maintained by the APP + Trilinos Representative as described above would make this easy to support. Obviously, another large benefit to providing easy access to APP + Trilinos Dev is that it makes it easier for APP or Trilinos developers to try out new Trilinos algorithms at any time based on the most recent Trilinos code.

Fix failed builds of APP + Trilinos Dev ASAP: It is critical that fixing broken builds of APP + Trilinos Dev be given a high priority and be addressed immediately. Without the software at least building and linking in order to run the tests, we can have no feedback at all about the state of our software and the entire daily integration and testing process falls apart. There can be no exception to this.

Address failing "research" and "production" tests on a schedule appropriate for the APP + Trilinos collaboration: While fixing failed builds of APP + Trilinos Dev must always be given a high priority and fixed immediately, addressing failing "research" and "production" runtime tests can be done on a variety of different schedules, depending on the nature of the APP and/or the APP + Trilinos collaboration at any time. We can imagine two extremes in how and when failing tests are addressed.

One one extreme, every day, each and every failing test is given a high priority and fixed ASAP[8]. While this approach results in the least risk of experiencing a regression, it can significantly harm overall productivity (especially for the APP + Trilinos Representative).

On the other extreme, we might not address any failing tests at all between releases and wait until the next upcoming release before any of the failed tests are addressed. While this other extreme has more risk associated with it as opposed to fixing failing tests instantly, it still offers significant advantages to not performing any daily building or testing at all. First, by keeping APP + Trilinos Dev building, we can address any of the failures at any time we wish. Second, knowing the exact 24 hour period when code changes were made that caused a failing test is a huge piece of information to help find the cause of a test failure. Letting failing tests fail for long periods of time and only requiring that APP + Trilinos Dev keeps building should only impart a very minimal overhead to day-to-day development activities.

In between these two extremes, every morning that one or more failed tests were reported, the APP + Trilinos Representative would spend five to ten minutes looking over the new failing tests and try

[8]This was mostly our policy on the ASC Vertical Integration Milestone but we did not completely keep up with this.

to diagnose them. If the problem can be quickly diagnosed[9], then an e-mail can be sent (or a bug report can be filed) to the parties that can fix the bug. If the problem can not be easily diagnosed in five to ten minutes, then the APP + Trilinos Representative might just make a note of this (e.g. file a general bug report, send a general e-mail, etc.) and then move on with the day's other activities. Then when time becomes available, the root causes of the failing tests can be diagnosed when it will not disturb the flow of other work. Again, knowing the exact 24 hour period when a test first failed is a huge piece of information in finding the root cause of the problem.

In summary, depending on the nature of APP and the relationship between APP and Trilinos, any level of urgency between these two extremes may be acceptable and this will still be much better than not doing any daily building or testing at all. The approach taken to addressing failing tests for APP + Trilinos Dev, for any specific APP, will change as the collaboration goes through periods of greater intensity and lesser intensity. During periods of more intense APP + Trilinos collaboration, we will be more aggressive about addressing failing tests. During periods of less intense (or nonexistent) APP + Trilinos collaboration, we can let tests fail for longer periods of time.

Archive test results for sufficiently long periods of time: Test results from APP + Trilinos Release and APP + Trilinos Dev should be archived for long periods of time. Typically, only smaller output files are needed to diagnose most problems and therefore the largest of output files should typically be excluded from the test results archive. However, all test output files should be archived between 24 hour periods. Having ready access these test results, and being able to compare the outputs from a passing and a failing version of a test (separated by 24 hours) is critical in helping to diagnose failing tests. Older test results should be pruned and thinned as needed to conserve disk space. For example, test results from three months ago could be deleted except for tests on Friday (or some other day) of each week. One exception would be that test results for consecutive days where a test went from passing to failing should be preserved for a long period of time (perhaps a year or more) since this is critical evidence in tracking down failing tests (especially in the extreme where tests are allowed to fail for very long periods of time). Easy access to test results can be provided through a website that anyone with SRN access can access[10] (or limit access to those individuals with Need-to-Know in the case of more sensitive APPs).

Transition "research" to "production" appropriately after each Trilinos release: If a "research" algorithm or feature becomes stable enough and the software implementation is of high enough quality (i.e. a "phase 2" package in Trilinos[11]), then after the next Trilinos release the "research" APP code and tests for that algorithm/feature should be moved to the APP's "production" code and tests. In this way, if the test fails later, the APP developers will be the first to face the bug since it is most likely an APP developer that introduced a new defect.

Perform APP + Trilinos Release and APP + Trilinos Dev nightly testing on the same set of platforms: The nightly building and testing of APP + Trilinos Release and APP + Trilinos Dev should be performed on the same set of platforms. In this way, if a "production" test fails with APP + Trilinos Dev but not with APP + Trilinos Release on the same platform, then we have some assurance that something has been broken by in the "research" work and not the "production" work being done by APP developer. If testing is done on different platforms, then a "production" test failure with APP + Trilinos Dev may just be due to small difference is rounding or other small

[9] Approximately 80% or so of the failing Charon + Trilinos Dev tests were diagnosed in just ten minutes or less.
[10] We have provided web access to the test results archive for the Charon + Trilinos Dev test suite.
[11] See The Trilinos Software Life-cycle Model, `http://www.cs.sandia.gov/~maherou`

issues that result from using different platforms[12].

Enable more communication between APP and Trilinos developers: All of this will require and foster more communication and cooperation between Trilinos and APP developers. This means that Trilinos developers will have to have a closer relationship with their customers.

Provide for instantaneous releasabiliy of Trilinos to important customers: In order to allow for the option of the Dev versions of the APP and Trilinos to be tagged, branched, and released together, the Trilinos release process for such customers needs to be doable in only a few days at most. This will require a change in a great many of the current Trilinos practices. For example, this will require that we port Trilinos Dev to various platforms where the APP runs on a frequent basis (perhaps every few months or less). Also, this will require that we have completely automated tarball testing and installation testing. There are other issues that would also need to be changed and/or improved in how we develop Trilinos. If a staggered release of Trilinos and APP is performed, then instantaneous releasabiliy is not really need (but is useful as a general principle in any case).

[12] In the ASC Vertical Integration Milestone work with Charon + Trilinos Dev, we did not have this in place. As a result, there were several occasions that new "production" tests failed when run with Charon + Trilinos Dev that passed with Charon + Trilinos Release on a different platform. Most of these "production" test failures were just due to minor rounding or other porting issues. The time waisted tracking down these "production" test failures could have been avoided if we had had this policy in place.

5 Experience from the ASC Vertical Integration Milestone with Charon + Trilinos Dev

The purpose of this section is to describe what was done in the FY07 ASC Level-2 Vertical Integration Milestone with Charon + Trilinos Dev. The milestone work served as the inspiration for APP + Trilinos Dev and served as a prototype and case study for APP + Trilinos Dev daily integration and testing in practice. I will describe what we did, what worked well, and what needs to be improved for this and other such efforts. Many of the suggested practicies given in Section 4 came from feedback provided by the Charon + Trilinos Dev relationship conducted during the milestone. As a contrast, I also describe a smaller effort that involved an integration of Aria/SIERRA + Trilinos Dev that was not supported by daily integration and testing.

5.1 Charon + Trilinos Dev daily integration and testing

Here I describe the basic elements of the Charon + Trilinos Dev daily integration and testing process that we had in place for the milestone. First, note that Charon has had its own native nightly test harness in place for many years. However, the Charon test harness was only set up to checkout and build Charon against a static set of third party libraries (TPLs), including Trilinos, and it was not clear how these scripts would be updated to allow for building with Trilinos Dev updated daily. Also, lack of Charon tool developer support made it difficult to see how to add this capability. Therefore, the decision was made to develop a new minimal test harness framework specifically for nightly building and testing of the Dev versions of Charon and Trilinos. Starting in February of 2007, we set up and ran nightly building and testing Charon + Trilinos Dev in debug (dbg) and optimized (opt) mode on my own 64 bit, 4-core, AMD, Linux workstation using GCC 3.4.6 for the compiler.

The Charon + Trilinos Dev test harness itself used the native Nevada/Alegra/Charon test harness to define and run the tests. Therefore, we did not reinvent the core test harness since the existing Nevada/Alegra/Charon test harness is quite good in many ways. The Charon + Trilinos Dev test harness shell scripts just focused on checking-out/updating the sources, running the builds, invoking the Nevada/Alegra/Charon test harness, interpreting the results, archiving the results, and sending out e-mail notifications.

A set of shell-based (i.e. `sh`) scripts were written to perform all tasks of the test harness. A top-level script was written that was directly invoked by a crontab job. Every night (starting at midnight), the test harness script checks out the Dev versions of Charon, the Charon TPLs, and Trilinos (within the Charon TPL directory structure) and builds and tests various sets of build options of Charon + Trilinos Dev. These scripts were not only designed to be used from the automated test harness but were also designed to be used to document how to perform various development and testing tasks. The scripts that are called are designed to show developers how to run the various tools to checkout, build, and test Charon with the Charon TPLs (which have Trilinos Dev embedded in them). Therefore, the nightly test harness not only tests Charon + Trilinos Dev, but it also tests the scripts that document the various tasks and which developers can directly use to perform these tasks.

Two directory trees were established and built from. An `Updated` base directory was used where

CVS updates were done into an existing tree for Charon, Charon_TPL, and Trilinos. From this tree, the optimized (opt) version of Charon + Trilinos Dev was built and tested. This is an important use case for continuing developers that will typically update existing working directories and then rebuild the code. A second `FromScratch` base directory was used where the working directories for code, object files, libraries, and executables were all deleted. Then, all of the source code was checked out from scratch, and built and tested. This is also an important use case since it ensures that new developers can checkout and build all of the code from scratch at any time.

As mentioned above, the Charon + Trilinos Dev test harness not only builds Charon + Trilinos Dev every night, it also builds all of the Charon TPLs (including Trilinos). Early on in the milestone, a change was made to one of the Charon TPLs that resulted in Charon to break. However, the existing Charon nightly test harness maintained by the Charon developers themselves on another machine did not (and still does not at the time of this writing) build the Charon TPLs every day. This delayed integration represents an increased risk (as all delayed integration does) which the Charon + Trilinos Dev test harness has addressed. A similar delayed integration for Charon exists with the Nevada/Alegra framework itself but that integration risk is owned by Charon is beyond the scope of this discussion.

The test harness shell scripts also archive the test results to a web-accessible directory tree. The URLs to these directories for each test are given in passed/failed e-mail notifications that are sent out at the end of build & test invocation for each set of build options (i.e. opt vs. dbg) of Charon + Trilinos Dev. All of the test input and output files are saved in this archive directory tree. However, to avoid massive increases in storage, all files above 1M are deleted. This removes mostly large mesh input files and exodus input and output files but maintains algorithm output traces which are the most useful in diagnosing failed tests. The test harness output clearly records the exact time and date that the source code is checked out which allows one to check out exactly the same versions at a latter time to reproduce the test results. Given that we archive all of the test results and a clear time stamp is given to each build, we can go back in time (months in some cases) to examine the behavior of a test and can then checkout the versions of the code for that build and should be able to more-or-less reproduce the test output. Of course, many issues make it almost impossible to do this if too much time passes (e.g. the OS and other system tools on the test machine may be updated, the Alegra TPLs (which are not rebuilt by the Charon + Trilinos Dev test harness) may have changed, etc.).

The notification e-mails sent out every night give the URL on the test machine running an Apache server to the archive directory that contain the test results. Therefore, a few simple clicks of the mouse are all that is needed to view the details of the test results from the night before (or for any prior nightly test for that matter). Mornings where there were new test failures, I was able to use the archived test results on the web server to diagnose most failed tests in less than 10 minutes. In many cases, I was the cause of the failed Charon “research” test due to a change in Trilinos that I committed that was not verified against Charon before being checked in[13]. In these cases, I was responsible for addressing the failed tests. In other cases, I sent off a short e-mail to the person or persons that I suspected was the cause of the problem and gave them some tips on what I suspected the problem might be given my brief analysis[14].

[13] Note that later in the milestone, I never checked in code into Trilinos that I did not first build and test Charon against. This massively cut down on the number of failed builds and failed tests.

[14] Adding some suggestion on what the problem might be proved to be a very effective catalyst for getting people to address the problems in a timely manner.

All in all, the daily integration and testing process that we set up for Charon + Trilinos Dev was very effective at keeping Charon + Trilinos Dev on track and avoiding backslides in functionality that we were adding during the milestone. However, there were a few aspects of the process that caused some unnecessary effort. Some of these problems are described below.

One problem that we had was that we did not build and test Charon + Trilinos 7.0 (the current Trilinos release at the time) and Charon + Trilinos Dev on the same platforms. As a result, there were several instances where a new "production" test was created and checked into Charon that worked just fine when run as part of Charon + Trilinos 7.0 on the 32 bit platform, where the same test failed or diffed on the 64 bit platform where Charon + Trilinos Dev was being built and tested. As a result, I spent a fairly significant amount of time diagnosing failing "production" tests that were unrelated to any changes in Trilinos and unrelated to the milestone effort. This has since been addressed by adding nightly building and testing of Charon + Trilinos 7.0 on the same 64 bit machine where the rest of the Charon + Trilinos Dev tests are run (see the Section 4 about this practice).

Another problem involved the transition to the post-milestone period. The end of the milestone was marked by the release of Trilinos 8.0 which included new milestone-related capabilities. This would have been an ideal time to upgrade Charon from Trilinos 7.0 to Trilinos 8.0. Since all of the Charon "production" tests were already passing against Trilinos 8.0 which where run every day , performing the upgrade would have been as simple as removing a few `ifdefs` and removing the `tridev` keyword from all of the "research" tests that were ready to become "production" tests. However, Charon has a dependency on an internal circuit simulation code called Xyce which was not being built and tested nightly against Trilinos Dev and typically takes several months or longer to be upgraded to a new release of Trilinos. Therefore, we could not simply upgrade Charon to Trilinos 8.0 since it would have broken the combined Charon + Xyce + Trilinos application. Therefore, we decided to go ahead and put in place nightly testing of Charon against both Trilinos 8.0 and Trilinos Dev. To accomplish this, we had to add a new `define` macro `CHARON_TRI8` and a new test keyword `tri8`, and we had to add new test builds for Charon + Trilinos 8.0 to the nightly test harness. At the time of this writing my personal computer is still running the following five builds of Charon + Trilinos every night:

1. Charon + Trilinos Dev (opt, all tests, updated sources)
2. Charon + Trilinos Dev (dbg, only tridev tests, sources checked out from scratch)
3. Charon + Trilinos 8.0 (opt, all tests, updated sources)
4. Charon + Trilinos 8.0 (dbg, only tri8 tests, updated sources)
5. Charon + Trilinos 7.0 (opt, all tests except tridev and tri8 tests, static Charon TPLs (including Trilinos 7.0))

Above, only the `tridev` and `tri8` tests were run in the `dbg` builds since running all of the tests would have taken too long to complete in a single night. Even with this, the builds and tests took from midnight to after 6:00 AM to fully complete. Note that the existing Charon test harness did not do nightly building or testing for a `dbg` build and therefore the milestone test harness actually improves their testing. By performing all of these nightly builds, we will be guaranteed that when Charon is upgraded to Trilinos 8.0 that this will go smoothly with few, if any, issues.

With all of the new milestone-related "research" tests constantly being built and tested every night, these tests will be preserved such that when we come back to Charon to advance our work, we will be assured that we have a solid foundation to augment functionality. As a result, Charon has become a very attractive foundation for our future algorithmic research. Without the daily integration and testing in place, we could experience significant problems getting our problems running again and the difficulties that we could encounter would likely be enough to cause us to delay or abandon our efforts. This was exactly what happened in our earlier efforts (for example, with MPSalsa).

5.2 Aria/SIERRA + Trilinos Dev

To contrast our experience with Charon + Trilinos Dev, consider the auxiliary effort where Aria/SIERRA was updated to build against Trilinos Dev and MOOCHO[15] was interfaced to Aria to solve a prototype design problem. This effort was meant to demonstrate that the milestone work was more general than just Charon and also served a number of other purposes as well. However, while Charon + Trilinos Dev was supported by nightly testing and Charon + Trilinos Dev was (and still is) instantly available to anyone with SRN access, Aria/SIERRA + Trilinos Dev was only periodically built by a single developer and was not easily accessible to others. The immediate impact of this approach was that some amount of effort was required to get Aria/SIERRA + Trilinos Dev to build again each time development paused for a time and then was continued. Also, the combined Aria/SIERRA + Trilinos Dev application was not easily accessible to the MOOCHO expert on the milestone team so when difficulties solving one of the problems surfaced, they where difficult to addresses.

Of course, the long term implication of not having Aria/SIERRA + Trilinos Dev building and testing in place is that the developed capability could break without anyone ever knowing it. It is very possible that the code may not even build by the time SIERRA upgrades to Trilinos 8.0. Therefore, the developed MOOCHO/Aria capability is fragile and is susceptible to being broken and may be lost if too much time goes by[16].

[15] `http://trilinos.sandia.gov/packages/moocho`

[16] At the time of this writing, planning is under way to establish daily integration and testing of Aria/SIERRA + Trilinos Dev that will protect MOOCHO/Aria and other future algorithmic developments.

6 Conclusions

There are a number of conclusions that we have drawn for the proposed daily integration and testing of the development versions of an application and Trilinos:

- Daily integration and testing of the development versions of the application and Trilinos:
  - results in better production capabilities and better research,
  - brings algorithm developers and application developers closer together allowing for a better exchange of ideas and concerns,
  - refocuses Trilinos developers on customer efforts,
  - helps drive research-quality algorithm development, and
  - reduces barriers for new algorithms to have impact on production applications.
- Other application projects and scientific support software projects should consider adopting the type of continuous integration that is used with Charon + Trilinos that was developed as part of the ASC Vertical Integration Milestone work.

References

[1] Paul M. Duvall. *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley, 2007.

[2] Pat K. Notz et. al. Aria 1.5 user manual. Technical Report SAND2007-2734, Sandia National Laboratories, 2007.

[3] Roscoe A. Bartlett et. al. ASC vertical integration milestone. Technical Report SAND2007-5839, Sandia National Laboratories, 2007.

[4] Mike A. Heroux, Teri Barth, David Day, Rob Hoekstra, Rich Lehoucq, Kevin Long, Roger Pawlowski, Ray Tuminaro, and Alan Williams. Trilinos : object-oriented, high-performance parallel solver libraries for the solution of large-scale complex multi-physics engineering and scientific applications. `http://software.sandia.gov/Trilinos`.

[5] Steve C. McConnell. *Code Complete 2nd Edition: A practical handbook of software construction*. Microsoft Press, 2004.

APP + Trilinos Dev Daily Integration and Testing Checklist

□ Do you have `ifdef`s in place in APP code that are needed to build against Trilinos Dev and against a stable release (or multiple releases) of Trilinos?

□ Have you separated “production” and “research” tests so that you can better differentiate APP defects from Trilinos defects?

□ Have you appointed an official APP + Trilinos Representative to make sure APP + Trilinos Dev is maintained and is responsible for making sure issues are forwarded to the appropriate parties?

□ Have you set up a dedicated machine to do daily integration and testing of APP + Trilinos Dev?

□ Have you provided easy access to APP and Trilinos developers to immediately build a private version of APP + Trilinos Dev to try out new algorithmic capabilities?

□ Do you fix failing builds of APP + Trilinos Dev right away, with no exceptions?

□ Do you address failing “production” and “research” tests with an urgency that is appropriate for the nature of APP and the APP + Trilinos collaboration?

□ Do you archive test results long enough to allow developers to diagnose failing tests?

□ After each major release of Trilinos, do you upgrade APP to the new release in a timely way?

□ During the transitionary period between when Trilinos is branched for a release and when the APP finally gets updated for the new Trilinos release, do you build APP against the old Trilinos release, the current Trilinos release, and Trilinos Dev?

DISTRIBUTION:

2	MS 9018	Central Technical Files, 8944
2	MS 0899	Technical Library, 4536

Sandia National Laboratories