Trilinos Software Engineering Status and Future Issues

Roscoe A. Bartlett
Department of Optimization & Uncertainty Estimation
Trilinos Software Engineering Technologies and Integration Lead

Sandia National Laboratories

Trilinos User Group Meeting, November 5, 2009
Trilinos Software Engineering Overview: Current Status

• Separation of “Stable” Code vs. “Experimental” Code
  • “Primary Stable” code vs. “Secondary Stable” Code
  • “Experimental “code
• Maintaining stability of “Stable” development code and tests
  • “Primary Stable” Code: pre-checkin testing (on “primary platform”)
  • “Secondary Stable” Code: nightly tested
• Maintaining portability
  • Nightly testing on a variety of “secondary platforms”
• Testing infrastructure
  • CTest: Local and pre-checkin testing, drives CI and nightly testing
  • CDash: Displays test results
• Automated testing:
  • Trilinos framework nightly testing (Linux, Mac, Windows) => CDash
  • APP Trilinos Integration testing (Xyce, Charon, Alegra, SIERRA)
• Customer application Integration:
  • Daily integration testing with upgrades to Trilinos releases: Charon, Xyce, Alegra
  • Almost Continuous Integration: SIERRA
Trilinos “Stable” vs “Experimental” Code: Defined

• “Stable” Code and Tests:
  – “Meets one or more of the following criteria:
    • Represents an important capability being used by an existing, or
    • Represents a new capability that the authors are willing to stand behind
    • Does not mean it is being targeted for the next release
  – Expected to be kept working at all times on the primary development platform
  – Developed and maintained to be highly portable
  – Maintained at the high quality as defined by modern SE principles

• “Experimental” Code and Tests:
  – By definition, all remaining code that is not “Stable” code.
  – Represents fundamental research and may be developed with informal low-quality software practices.
  – Any code that has a direct and mandatory dependency on any “Experimental” code must also be considered to be “Experimental” code.
  – Developers should try to avoid depending on other “Experimental” code because it is likely to be unstable and break frequently.
  – “Experimental” code should be protected behind ifdefs with macros that must be defined in order to be built.
Trilinos “Primary Stable” vs “Secondary Stable” Code

• Sub-categorizations of “stable” code:
  – “Primary Stable” code is “Stable” code that only depends on:
    • C, and C++ compilers
    • Fortran 77 compiler (optional)
    • BLAS and LAPACK
    • MPI
  – “Secondary Stable” code
    • Has additional dependencies such as:
      – SWIG/Python (i.e. PyTrilinos)
      – Fortran 2003+ (i.e. ForTrilinos)
      – External direct sparse solvers like UMFPACK, SuperLU, etc. (i.e. Amesos adapters)
    • Or, could be considered “Primary Stable” Code but is excluded from pre-checkin testing
      – Didasko
      – NewPackage
      – ...

• “Stable” code in one package can only depend on “Stable” code in other packages.
• “Stable” code should by default only build “Primary Stable” code.
• Enabling “Secondary Stable” code should require extra configure-time options.
Stable (Primary and Secondary) and Experimental Code

- **Primary Stable Code and Tests:**
  - All affected code should be built and tested *before* a checkin
  - CATEGORY in `cmake/Trilinos[Packages,TPLs].cmake` set to “PS”
  - Required TPL dependencies on BLAS, LAPACK, and MPI (or less)
  - Configured with:
    - `D Trilinos_ENABLE_ALL_PACKAGES:BOOL=ON` \ 
    - `D Trilinos_ENABLE_TESTS:BOOL=ON`

- **Secondary Stable Code and Tests:**
  - Represents an important (released) capability but has extra TPL dependencies
  - *Note* be enabled for pre-checkin testing
  - Tested by central framework resources (nightly integration testing)
  - CATEGORY in `cmake/Trilinos[Packages,TPLs].cmake` set to “SS”
  - Requires explicitly enabling “Stable” optional TPL dependencies
  - Configured with:
    - `D Trilinos_ENABLE_ALL_PACKAGES:BOOL=ON` \ 
    - `D Trilinos_ENABLE_SECONDARY_STABLE_CODE=ON` \ 
    - `D Trilinos_ENABLE_TESTS:BOOL=ON`

- **Tertiary Stable Code and Tests?** (Right now just TPLs)

- **Experimental Code:**
  - CATEGORY in `cmake/Trilinos[Packages,TPLs].cmake` set to “EX”
  - Requires explicit enabling
  - Tested by individual package teams (but posts results to main CDash dashboard)
Trilinos Software Engineering: Issues

- Partitioning of the test suite and testing efforts
- Improving stability of “Stable” code => checkin-test.py script
- CMake sub-package architecture
- Official Trilinos developers toolset
- Automated Installation testing
- Regulated backward comparability
- Streamlined and robustify release process
- Other areas of needed improvement and progress
Partitioning of the Trilinos Test Suite

- "Unit" tests (i.e. TDD tests)
  - Make Trilinos packages better independently tested
  - Goal: Minimize need to enable and test down-stream packages
- "Basic integration" tests (i.e. pre-checkin tests)
  - "Unit tests" + some basic integration tests with all optional packages
  - Required on pre-checkin testing of Primary Stable Code
  - Protects basic functionality and other developers
- "Regression" tests (i.e. basic "nightly" tests)
  - "Basic integration" tests + some heavier tests
  - Runs on all available nightly platforms
  - Protects key correctness functionality
- "Performance" tests (See Teuchos performance tests)
  - Specifically designed to protect serial performance
  - Strong tests with hard time limits (adapted to specific platforms)
  - Runs on specific platforms without any other machine loads
- "Scalability" tests
  - Specifically designed to protect parallel scalability performance
  - Utilize targeted timers around problematic computations
- "User-like" tests (i.e. installation and backward compatibility tests)
  - Subset of "Basic Integration" tests

Need to add a CATEGORIES argument to the PACKAGE_ADD_TEST(…) function and an input cache variable Trilinos_TEST_CATEGORIES
Improving Stability of “Stable” code: Motivation

• Support deep stacks of vertically integrated Trilinos packages with production APPs

• Support tighter coupling and co-development with production APPs
  – SIERRA toolkit packages (STK_Mesh, STK_IO, ...)
  – Replace SIERRA framework code with Trilinos code (Teuchos::ParameterList, ...)
  – Many many others …

• Support more frequent, safer, higher quality, lower risk releases of Trilinos

• Improve overall development productivity and software quality

See:
  Trilinos/doc/DevGuide/TrilinosSoftwareEngineeringImprovements/*\.tex
“Stable” Code: 100% Passing Test Policy

• All “Stable” code should have 100% passing tests 100% of the time on the primary development platforms as the norm instead of the exception.

• Achieving 100% passing tests on auxiliary development platforms is also a priority but is done in a secondary development loop.

• A failing test on any testing platform should be addressed and be made to pass or be disabled using the following algorithm:
  – Fix the test in the strongest way possible
  – Or, loosen the “strength” of test to get it pass on that specific platform (i.e. by loosing a platform-specific tolerance)
  – Or, disable the test and submit a new item to the sprint or product backlog (e.g. Bugzilla bug report) so that it can be prioritized and fixed later
  – Or, remove the test and all of the associated code related to it
Motivations for a 100% Passing Test Policy for “Stable” Code

Why is 100% passing tests important?

• Package Y (reference package):
  – “Broken Window” Phenomenon
    => One broken test begets others
  – Zero (0) is singularly different that 1 or X failing tests
    => People take notice of “all passed” vs “failed”
  – ‘M’ failing tests is not much different that ‘N’ failing tests
  – 100% passing tests is a clear measure of the code health
  – 100% passing test suite is unbiased criteria for code checkins
  – 100% passing test suite is an unbiased measure for if any code has been broken after a checkin
  – Code coverage less meaningful when there are failing tests

• Package X (up-stream package being used by Package Y)
  – 100% passing test suite for Package Z provides a clear means to determine if changes in Package X break anything.

• Package Z (down-stream package that uses Package Y)
  – 100% passing test suite for Package Y gives Package Z developers confidence that they can depend on and trust the code in Package Y.

• Bottom Line:
  – 100% passing test suites help to build trust between developers
  – 100% passing test suites help to avoid unnecessary communication
  – 100% passing test suites help to avoid synchronization points
Waste Created By Lack of Sufficient Pre-Checkin Testing

1) Checkin that breaks Package Z
2) Checks out, builds, tests, and detects problems with Package Z
3) Sends failure email to Package Z developers
4.a) Checks out, builds, & tests
4.b) Wastes time trying to figure out why Package Z is failing (looks at VC logs, looks at dashboard results, etc.)
5) Sends email to Package Y developers to please fix the problem
6) Fixes problem and checks in

- 90% of these problems can be avoided with sufficient pre-checkin testing!
- Catching the problem before checking in saves everyone wasted time!
Automatic Dependency Handling for Pre-Checkin Testing

$ ./do-configure \
-D Trilinos_ENABLE_ALL_PACKAGES:BOOL=OFF \
-D Trilinos_ENABLE_Epetra:BOOL=ON \
-D Trilinos_ENABLE_ALL_FORWARD_DEPENDENCIES:BOOL=ON \
-D Trilinos_ENABLE_TESTS:BOOL=ON
Pre-Checkin Testing: The checkin-test.py script

Python script that performs safe pre-checkin testing:

$ cd SOME_BASE_DIR
$ mkdir CHECKIN; cd CHECKIN
$ $TRILINOS_HOME/cmake/python/checkin-test.py --do-all

• Automatically figures out what Trilinos packages have been changes
• Automatically enables all downstream packages
• Configures, builds and runs tests
  • Built-in Configurations:
    • MPI_DEBUG (Optimized compiler options, checked STL, etc.) (Do at least this build!)
    • SERIAL_RELEASE (varies other configure options)
  • Only enables Primary Stable Code!
  • Strong warning options (warnings as errors is a problem)
• Sends emails after each build case is finished
• Sends final email if it is okay to commit or not
• Can automatically do the commit at the end (Recommended)
• Fully customizable (enabled packages, build cases, etc.)
• Documentation: checkin-test.py --help
checkin-test.py: Example Driver Script

Script I used on my machine (checkin-test-<mymachine>.sh):

```bash
#!/bin/bash
EXTRA_ARGS=＠
echo "-DBUILD_SHARED_LIBS:BOOL=ON" > COMMON.config
echo "-DTrilinos_ENABLE_Sundance:BOOL=OFF" > SERIAL_RELEASE.config

/home/rabartl/PROJECTS/Trilinos.base/Trilinos/cmake/python/checkin-test.py \ 
  --make-options="-j4" \ 
  --ctest-options="-j4" \ 
  --ctest-time-out=180 \ 
  --commit-msg-header-file=checkin_message \ 
  $EXTRA_ARGS
```

Run as (after symbolically linking into CHECKIN directory):

```bash
$ ./checkin-test-<mymachine>.sh --do-all --commit
```

Example driver scripts (I symbolically link these):

- sampleScripts/checkin-test-cygwin-rabartl.sh
- sampleScripts/checkin-test-<mymachine>.sh
- sampleScripts/checkin-test-scicolan-rabartl.sh
- ...
**checkin-test.py: Recommended Workflow**

A) Fill out the checkin checklist message in a temporary text file ‘checkin_message’

B) Do local git commits (once we switch to git)

C) Run the checkin-test.py script:

   $ ./checkin-test-mymachine.sh –do-all –commit

D) Go do something useful (e.g. go home, check email, review a paper, work on a paper, talk with someone, ..)

D) Check your email later to see what happens

**Consequences:**
- Documents a bullet-proof process for configuring, building, and testing Trilinos
- Does the VC commands to do a safe global checkin (ease git transition)
- Enjoy fewer bad checkins
- Spend less time driving the checkin process
Directory Structure for auto-generated log files

CHECKIN/
  checkin-test.out
  update.out
MPI_DEBUG/
  do-configure.out
  make.out
  ctest.out
SERIAL_RELEASE/
  ...

See log files while configure, build, or test is being run:

$ tail -f MPI_DEBUG/make.out
checkin-test.py: Cost of Pre-Checkin Testing (Average Case)

A) Enabling just ML and tests/examples in downstream packages

Enabled packages (libraries) (29/52): Teuchos, RTOp, Kokkos, Epetra, Zoltan, Shards, Triutils, Tpetra, EpetraExt, Thyra, Isorropia, AztecOO, Galeri, Amesos, Pamgen, Ifpack, ML, Belos, Stratimikos, Meros, FEI, Anasazi, , Sacado, Intrepid, NOX, Moertel, Rythmos, MOOCHO, Sundance

Enabled packages (tests/examples) (10/52): ML, Belos, Stratimikos, Meros, FEI, NOX, Moertel, Rythmos, MOOCHO, Sundance

<table>
<thead>
<tr>
<th>Build Type</th>
<th>Build (min)</th>
<th>Test (min)</th>
<th>#tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_DEBUG</td>
<td>24.2</td>
<td>3.9</td>
<td>438</td>
</tr>
<tr>
<td>SERIAL_RELEASE</td>
<td>18.1</td>
<td>1.1</td>
<td>426</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Build Type</th>
<th>Build (min)</th>
<th>Test (min)</th>
<th>#tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_DEBUG</td>
<td>0.7</td>
<td>4.0</td>
<td>438</td>
</tr>
<tr>
<td>SERIAL_RELEASE</td>
<td>0.4</td>
<td>1.2</td>
<td>426</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Build Type</th>
<th>Build (min)</th>
<th>Test (min)</th>
<th>#tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_DEBUG</td>
<td>59.0</td>
<td>6.5</td>
<td>434</td>
</tr>
<tr>
<td>SERIAL_RELEASE*</td>
<td>30.4</td>
<td>1.3</td>
<td>350</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Build Type</th>
<th>Build (min)</th>
<th>Test (min)</th>
<th>#tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_DEBUG</td>
<td>1.4</td>
<td>6.6</td>
<td>434</td>
</tr>
<tr>
<td>SERIAL_RELEASE*</td>
<td>0.7</td>
<td>1.3</td>
<td>350</td>
</tr>
</tbody>
</table>

• With shared libraries, rebuilds can be very fast!
• Use a fast machine to checkin from!

* Sundance disabled on <average-machine> for serial build (see bug ???)
**checkin-test.py: Cost of Pre-Checkin Testing (Worst Case)**

B) Enabling Teuchos and tests/examples in downstream packages


Enabled packages (tests/examples) (22/52): Teuchos, OptiPack, Isorropia, AztecOO, Galeri, Amesos, Ifpack, Komplex, ML, Belos, Stratimikos, Meros, FEI, Anasazi, RBGen, Sacado, Intrepid, NOX, Moertel, Rythmos, MOOCHO, Sundance

<table>
<thead>
<tr>
<th>&lt;fast-machine&gt;, shared libs, from scratch</th>
<th>&lt;fast-machine&gt;, shared libs, rebuilid</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Build Type</strong></td>
<td><strong>Build Type</strong></td>
</tr>
<tr>
<td><strong>Build (min)</strong></td>
<td><strong>Build (min)</strong></td>
</tr>
<tr>
<td><strong>Test (min)</strong></td>
<td><strong>Test (min)</strong></td>
</tr>
<tr>
<td>#tests</td>
<td>#tests</td>
</tr>
<tr>
<td>------------------------------------------</td>
<td>--------------------------------------</td>
</tr>
<tr>
<td>MPI_DEBUG</td>
<td>MPI_DEBUG</td>
</tr>
<tr>
<td>48.0</td>
<td>1.1</td>
</tr>
<tr>
<td>8.34</td>
<td>8.1</td>
</tr>
<tr>
<td>1140</td>
<td>1140</td>
</tr>
<tr>
<td>SERIAL_RELEASE</td>
<td>SERIAL_RELEASE</td>
</tr>
<tr>
<td>37.3</td>
<td>1.2</td>
</tr>
<tr>
<td>1.9</td>
<td>2.1</td>
</tr>
<tr>
<td>1147</td>
<td>1147</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>&lt;average-machine&gt;, shared libs, from scratch</th>
<th>&lt;average-machine&gt;, shared libs, rebuilid</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Build Type</strong></td>
<td><strong>Build Type</strong></td>
</tr>
<tr>
<td><strong>Build (min)</strong></td>
<td><strong>Build (min)</strong></td>
</tr>
<tr>
<td><strong>Test (min)</strong></td>
<td><strong>Test (min)</strong></td>
</tr>
<tr>
<td>#tests</td>
<td>#tests</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>------------------------------------------</td>
</tr>
<tr>
<td>MPI_DEBUG*</td>
<td>MPI_DEBUG*</td>
</tr>
<tr>
<td>103.0</td>
<td>2.3</td>
</tr>
<tr>
<td>12.0</td>
<td>12.0</td>
</tr>
<tr>
<td>1136</td>
<td>1136</td>
</tr>
<tr>
<td>SERIAL_RELEASE*</td>
<td>SERIAL_RELEASE*</td>
</tr>
<tr>
<td>63.5</td>
<td>1.49</td>
</tr>
<tr>
<td>2.5</td>
<td>2.5</td>
</tr>
<tr>
<td>1071</td>
<td>1071</td>
</tr>
</tbody>
</table>

* Sundance disabled on <average-machine> (see bug ???)
B) Enabling Teuchos and tests/examples in downstream packages


Enabled packages (tests/examples) (22/52): Teuchos, OptiPack, Isorropia, AztecOO, Galeri, Amesos, Ifpack, Komplex, ML, Belos, Stratimikos, Meros, FEI, Anasazi, RBGen, Sacado, Intrepid, NOX, Moertel, Rythmos, MOOCHO, Sundance

* Sundance disabled on <average-machine> for serial build (see bug ???)

<table>
<thead>
<tr>
<th>Build Type</th>
<th>Build (min)</th>
<th>Test (min)</th>
<th>#tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_DEBUG</td>
<td>115.3</td>
<td>10.7</td>
<td>1136</td>
</tr>
<tr>
<td>SERIAL_RELEASE*</td>
<td>72.4</td>
<td>2.7</td>
<td>1071</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Build Type</th>
<th>Build (min)</th>
<th>Test (min)</th>
<th>#tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_DEBUG</td>
<td>18.9</td>
<td>10.4</td>
<td>1136</td>
</tr>
<tr>
<td>SERIAL_RELEASE*</td>
<td>6.6</td>
<td>2.4</td>
<td>1071</td>
</tr>
</tbody>
</table>
Speeding up Pre-Checkin Testing: Current Approaches

• 100% safe approaches:
  • Checkin from a fast workstation no matter where you develop (easy with git)
  • Keep private development and checkin builds separate
  • Enabled shared libraries (-DBUILD_SHARED_LIBS:BOOL=ON)
  • Keep the CHECKIN builds up to date (could use crontab or just manually)

• Less than 100% safe approaches (from better to worst):
  • Do only MPI_DEBUG build (--without-serial-release)
  • Disallow enabling all packages (--enable-all-packages=off)
    • Example: Disables enabling all packages when cmake/TrilinosPackages.cmake changes
  • Disable forward packages (--no-enable-fwd-packages)
    • Example: Only tests in the package have changed
    • Example: Good unit tests and minimal changes
  • Disabling specific downstream packages (--disable-packages=P1,…)
    • Example: Disabling Sundance when testing Tpetra
  • Enabling only specific packages (--enable-packages=P1,…)
    • Example: Only test a few packages
      --enable-all-packages=off --enable-packages=Tpetra,Belos,Anasazi
Improving Pre-Checkin Testing: Future Approaches

- Speeding up pre-checkin testing:
  - Move to explicit template instantiation
  - Forward declarations
  - Use pImpl idiom (faster rebuilds)
  - Remove standard C++ headers out of Package_ConfigDefs.hpp
  - Trim down number of “Basic Integration” test executables
  - More unit tests, faster more minimal basic integration tests
  - Move to a sub-package architecture in the CMake build system

- Improving consistency of pre-checkin testing:
  - Standardize versions of GCC, MPI, BLAS, LAPACK etc. …
  => Official Trilinos Developers Toolset

- Improving the portability testing of pre-checkin testing:
  - Strong warnings and warnings as errors
    - Requires standard versions of GCC and MPI!
  => Official Trilinos Developers Toolset
Possible improvements to the checkin-test.py script

• Convert from CVS to git (to be done very soon)

• Allow for extra user-defined build cases:
  -- extra-builds=BUILD1,BUILD2,…,BUILDN
  • Motivation: Allows enabling Secondary Stable and Experimental Code, enabling extra TPLs, etc.
  • Example: Test Secondary Stable Code and TPLs
    $ echo "-DTPL_ENABLE_SCOTCH:BOOL=ON" >> WITH_SCOTCH.config
    $ ./checkin-test-mymachine.sh --extra-builds=WITH_SCOTCH --do-all

• Add more unit testing
Pre-Checkin Testing: Summary

• Using this script will improve the stability of Trilinos for everyone involved!
• Bad reasons to do a sloppy checkin:
  • “I want to integrate my code frequently”
    => Good motivation but not as important good testing
    => Checking in once a day is usually sufficient
  • “I need to get this revision to a collaborator ASAP”
    => Just have them pull directly from your local git repository
  • “I am doing porting work and can’t afford a complete test on the machine”
    => Pull local commits back to your git local working directory your workstation and commit from there
  • “I am in a good point to checkpoint my changes”
    => Do a local git commit
  • “I want to backup my work with history”
    => Use git to publish to a “backup” repository on a different machine
  • “I want to checkin to feel a sense of completion”
    => Mental problem, seek help
• Please read “checkin-test.py –help” and give this a try!
• Please ask questions, give feedback!
CMake Sub-Package Architecture: Motivation

Existing package dependency logic can enable many more packages than is needed for pre-checkin testing

Example: Enable Tpetra

$ checkin-test.py --enable-packages=Tpetra --configure


- Enabled packages (tests/examples) (10/52): Tpetra, Belos, Stratimikos, Meros, Anasazi, RBGen, NOX, Rythmos, MOOCHO, Sundance

=> Problem: Stratimikos, Meros, Rythmos, MOOCHO, and Sundance don’t execute one line of Tpetra code!

- General Problem: Current CMake build system does not respect the existing package partitioning
Package Cohesion OO Principles:
• REP (Release-Reuse Equivalency Principle): The granule of reuse is the granule of release.
• CCP (Common Closure Principle): The classes in a package should be closed together against the same kinds of changes. A change that affects a closed package affects all the classes in that package and no other packages.
• CRP (Common Reuse Principle): The classes in a package are used together. If you reuse one of the classes in a package, you reuse them all.

Package Coupling OO Principles:
• ADP (Acyclic Dependencies Principle): Allow no cycles in the package dependency graph.
• SDP (Stable Dependencies Principle): Depend in the direction of stability.
• SAP (Stable Abstractions Principle): A package should be as abstract as it is stable.

Problem: Many Trilinos packages violate the SE packaging principles most importantly the CRP

CMake Sub-Package Architecture: The Idea

- Partitioning of Trilinos Code:
  - Trilinos packages: More natural feature/social/user packages
  - Trilinos sub-packages: Rigours SE packages (hidden from user)
- Speeds up pre-checkin rebuilds and testing
- Provided greater control over feature selection
- Helps to minimize superficial entangling dependencies
- Minimizes the number of top-level packages
- Hides complexity form the user
- However, some software engineering packages will still be needed due to dependency issues
- Once we have git we can reorganize for this!
Official Trilinos Developers Toolset: Idea and Motivation

• Idea: Define a suite of standard build and other tools along with simple global install script

• Candidate list of software:
  • GCC 4.X.Y (Fortran or no Fortran?)
  • Gold ??? (fast linking)
  • Open MPI ???
  • CMake 2.8.X
  • Git ???, eg ???
  • CLAPACK ???
  • Boost ???
  • Doxygen ???
  • Dot ???

• Motivation:
  • Reduce variability in development and testing for different developers
    • Turn on strong warnings and warnings as errors
  • Simplify setup of new Trilinos development machines
  • Allow more code to be elevated to Primary Stable Code (e.g. boost)
Official Trilinos Developers Toolset: Install scripts

Provide global install script:

$ Install-trilinos-toolset.py --do-all --install-dir=/home/trilinos/install

• Checks out tarballs from Trilinos3PL CVS repository
• Installs all software in single bin, lib, and include directories
• Uses separate install scripts like install-cmake.py, install-git.py etc.
• Would only support basic Linux (perhaps Unix) and Mac computers (not Windows)

ToDo:

• Decide what software should be included
• Decide on versions of all the software packages
• Refactor existing install-git.py and install-cmake.py to enable faster development of simple install steps
• Get software and write basic install scripts and global install script
• Beta users to work out bugs
• Deploy across all Trilinos developers
• Turn on warnings as errors!
• Enjoy more a stable development environment!
Automated Installation Testing

• Idea:
  • BUILD_DIR_1: Build and install Trilinos headers and libraries
    $ do-configure –D CMAKE_INSTALL_PREFIX:PATH=<INSTALLPATH> …
  • BUILD_DIR_2: Configure tests/examples against installed headers/libs
    $ do-configure –D TRILINOS_ENABLE_TESTS:BOOL=ON \
        -D TRILINOS_USE_INSTALLED_LIBS_BASE:PATH=<INSTALLPATH> …

Details:
• Would be handled automatically by the Trilinos CMake macro wappers
• Would not require any changes in Trilinos packages
• Would read from installed export makefiles to get compiler options, list of link libraries, etc.
• Select subset of tests (only user-like tests not all unit tests)

Consequences:
• Automatic testing of installation process!
• Foundation for backward compatibility
Backward Compatibility Considerations

• Backward compatibility is critical for:
  • Safe upgrades of Trilinos releases
  • Composability and compatibility of different software collections
Example of the Need for Backward Compatibility

Multiple releases of Trilinos presents a possible problem with complex applications

Solution:
=> Provide perfect backward compatibility of Trilinos X through Trilinos SIERRA Y+1
Backward Compatibility Considerations

• Backward compatibility is critical for:
  • Safe upgrades of Trilinos releases
  • Composability and compatibility of different software collections

• Maintaining backward compatibility for all time has downsides:
  • Testing/proving backward compatibility is expensive and costly
  • Encourages not changing (refactoring) existing interfaces etc.
    • => Leads to software “entropy” which kills a software product

• A compromise: Regulated backward compatibility (Tentative)
  • Maintain a window of perfect backward compatibility over major version numbers (e.g. 1-2 years)
  • Provide “Deprecated” compiler warnings
    • Example: GCC’s __deprecated__ attribute enabled with
      –DTrilinos_SHOW_DEPRECATED_WARNINGS:BOOL=ON
  • Provide strong automated testing of Trilinos backward compatibility
  • Drop backward compatibility between major version numbers
Regulated Backward Compatibility and Version Numbering?

- Proposal: Trilinos Version Numbering X.Y.Z:
  - X: Defines backward compatibility
  - Y: Major release number in backward compatible set
  - Idea: Even numbers = release, odd numbers = dev (CMake, SIERRA)
    - Makes logic with Trilinos_version.h easier
  - Z: Minor releases off the release branch X.Y

- Backward comparability between releases X.Y and X.Z where Z > Y
  - Example: Trilinos10.6 is backward compatible with 10.0 through 10.4
  - Example: Trilinos 11.X is **not** compatible with Trilinos 10.Y

Maintain backward compatibility of 11.0 with only 10.3 but drop all other deprecated code!

Example: Major Trilinos versions change every 2 years with 2 releases per year
Stay tuned for later discussion
Other Areas of Needed Improvement and Progress

• Code coverage (see TrilinosCMakeQuickstart.txt)
  $ ./do-configure -DTrilinos_ENABLE_COVERAGE_TESTING:BOOL=ON
  $ make dashboard

• Memory checking (see TrilinosCMakeQuickstart.txt)
  $ env CTEST_DO_MEMORY_TESTING=TRUE make dashboard
  • Need a trimmer test suite to allow valgrind to run

• Namespace safety
  • Don’t pollute the global namespace, no ‘using namespace ANTHYING’

• Strong warnings and warnings as errors
  • Need a standard version of GCC and MPI first (Official Trilinos Toolset)

• Code reviews (arguments and evidence seems clear)

• Unit testing (see Todd’s talk)
  • Reduces need to test downstream packages

• Doxygen documentation (Need automated testing of some type)

• Improving exception safety (basic guarantee, strong guarantee, and no-fail guarantee and memory leaks)

• Globing source and header files (SIERRA packages only)
Trilinos Software Engineering: Issues

- Partitioning of the test suite and testing efforts
- Improving stability of “Stable” code => checkin-test.py script
- CMake sub-package architecture
- Official Trilinos developers toolset
- Automated Installation testing
- Regulated backward comparability
- Streamlined and robustify release process
- Other areas of needed improvement and progress