

TESTING OF HPC SCIENTIFIC SOFTWARE

SC16

Salt Lake City, Utah

November 14, 2016

Anshu Dubey, Alicia Klinvex, Jeff Johnson, and the
IDEAS team

The IDEAS project

2

- A DOE project aimed at increasing **software productivity for extreme-scale computational science**
- IDEAS resources
 - ▣ On various topics in software engineering and productivity, including testing
 - ▣ <https://ideas-productivity.org>
- More info:
 - ▣ See last slide for info on additional software testing resources
 - ▣ CSE Software Forum:
<https://cse-software.org>

Collaborators in IDEAS project:

ANL

LANL

LBNL

LLNL

ORNL

PNNL

SNL

Colorado School
of Mines

Outline

3

- Introduction
 - ▣ Motivation for verification and testing
 - ▣ Importance of granularity in testing
 - ▣ Definitions of test types and their role in the testing regime
 - ▣ Code coverage
 - ▣ Continuous integration
- Scientific software verification and validation
 - ▣ Definitions
 - ▣ Challenges specific to scientific and high-performance computing
 - ▣ Examples
- How to evaluate needs of a project and devise a testing regime
 - ▣ Examples - Alquimia, Amanzi, Trilinos
- Testing during refactoring
 - ▣ General guidelines
 - ▣ Detailed case study with FLASH
- TravisCI tutorial

Introduction

Why is testing important?

Granularity of tests

Types of tests

Code coverage

Continuous integration

Benefits of testing

5

- Promotes high-quality software that delivers correct results and improves confidence
- Increases quality and speed of development, reducing development and maintenance costs
- Maintains portability to a variety of systems and compilers
- Helps in refactoring
 - ▣ Avoid introducing new errors when adding new features
 - ▣ Avoid reintroducing old errors

How common are bugs?

6

Programs do not acquire bugs as people acquire germs, by hanging around other buggy programs. Programmers must insert them.

- Harlan Mills

- Bugs per 1000 lines of code (KLOC)
- Industry average for delivered software
 - ▣ 1-25 errors
- Microsoft Applications Division
 - ▣ 10-20 defects during in-house testing
 - ▣ 0.5 in released product

Code Complete (Steven McConnell)

Why testing is important: the protein structures of Geoffrey Chang

7

- Some inherited code flipped two columns of data, inverting an electron-density map
- Resulted in an incorrect protein structure
- Retracted 5 publications
 - ▣ One was cited 364 times
- Many papers and grant applications conflicting with his results were rejected

Why testing is important: the 40 second flight of the Ariane 5

8

- ❑ Ariane 5: a European orbital launch vehicle meant to lift 20 tons into low Earth orbit
- ❑ Initial rocket went off course, started to disintegrate, then self-destructed less than a minute after launch
- ❑ Seven variables were at risk of leading to an Operand Error (due to conversion of floating point to integer)
 - ▣ Four were protected
- ❑ Investigation concluded insufficient test coverage as one of the causes for this accident
- ❑ Resulted in a loss of \$370,000,000.

Why testing is important: the Therac-25 accidents

9

- ❑ Therac-25: a computer-controlled radiation therapy machine
- ❑ Minimal software testing
- ❑ Race condition in the code went undetected
- ❑ Unlucky patients were struck with approximately 100 times the intended dose of radiation, $\sim 15,000$ rads
- ❑ Error code indicated that no dose of radiation was given, so operator instructed machine to proceed
- ❑ Recalled after six accidents resulting in death and serious injuries

Granularity of tests

10

- Unit tests
 - ▣ Test individual functions or classes
 - ▣ Build and run fast
 - ▣ Localize errors
 - ▣ Usually written before or during code development
 - Prevent faults from being introduced
 - ▣ Example: Can I correctly compute a dot-product?

If a unit test fails, you should know *exactly* what is broken.

Granularity of tests

11

- Integration tests
 - ▣ Test interaction of larger pieces of software
 - ▣ Do not build or run as fast as unit tests
 - ▣ Example: Does the preconditioner class work with the Krylov solver class?

Granularity of tests

12

- System-level tests
 - ▣ Test the full software system at the user interaction level
 - ▣ Example: Does my CFD code compute the correct solution?

Types of tests

13

□ Verification tests

- Does the code implement the intended algorithm correctly?
- Check for specific mathematical properties
- Example
 - Solving $Ax=b$ where A has 5 distinct eigenvalues
 - Does my Krylov solver converge in 5 iterations?
- Can be any granularity

Types of tests

14

- Acceptance tests
 - ▣ Assert acceptable functioning for a specific customer
 - Different from other types of tests, which don't involve customers
 - ▣ Generally at the system-level
 - ▣ Example: Does my linear solver achieve the correct convergence rate for a particular customer's linear system?

Types of tests

15

- Regression (no-change) tests
 - ▣ Compare current observable output to a gold standard
 - Gold standard frequently comes from previous version of software
 - ▣ Similar to verification tests
 - Must independently verify that the gold standard is correct
 - ▣ Example
 - My Krylov solver took 10 iterations last week; does it still take 10 iterations?
 - Does it achieve the same solution?
 - ▣ Bounded change tests are better for floating point computations

Types of tests

16

- Performance tests
 - ▣ Focus on the runtime and resource utilization
 - ▣ Nothing to do with correctness
 - Orthogonal to other types of tests
 - ▣ Example: It took my code 10s to solve this linear system last week; does it take longer now?

Types of tests

17

- Installation tests
 - ▣ Verify that the configure-make-install is working as expected
 - ▣ Example: Can I build and run a simple driver using my library after the library is installed?

Additional resources

18

- <https://www.udacity.com/course/software-testing--cs258>
- http://www.tutorialspoint.com/software_testing/software_testing_levels.htm

Good testing practices

19

- Test-driven development – acceptance tests are written before the software
 - ▣ Gain clarity on code
 - ▣ Guarantees tests will exist
 - ▣ Useful when testing is viewed as unsustainable tax on resources
- Provide users a regression test suite
- Test software regularly, preferably daily

Policies on testing practices

20

- Must have consistent policy on dealing with failed tests
 - ▣ Issue tracking
 - How quickly does it need to be fixed?
 - Who is responsible for fixing it?
 - ▣ Add regression test afterwards (to avoid reintroducing issue later)
- Someone needs to be in charge of watching the test suite

Policies on testing practices

21

- When refactoring or adding new features, run a regression suite before checkin
 - ▣ Be sure to add new regression tests for the new features
- Make sure at least two people are familiar with every portion of code
- Require a code review before releasing test suite
 - ▣ Another person may spot issues you didn't
 - ▣ Incredibly cost-effective

Policies on testing practices

22

- Avoid regression suites consisting of system-level no-change tests
 - ▣ Tests often need to be re-baselined
 - Often done without verification of new gold-standard
 - ▣ Hard to maintain across multiple platforms
 - ▣ Loose tolerances can allow subtle defects to appear

Use of test harnesses

23

- Essential for large code
 - ▣ Set up and run tests
 - ▣ Evaluate test results
- Easy to execute a logical subset of tests
 - ▣ Pre-push
 - ▣ Nightly
- Automation of test harness is critical for
 - ▣ Long-running test suites
 - ▣ Projects that support many platforms

Automated test harnesses

24

- crontab
 - ▣ Time-based scheduler for Linux
 - ▣ Execute specific command at specific time
- Newer tools...
 - ▣ Allow centralized servers to execute tests on multiple platforms
 - ▣ Assist in load balancing and scheduling on available test resources
 - ▣ Test execution can be triggered by
 - Time
 - An event (such as repository modification)
 - Manual request by developer

Reporting test results

25

- Output results to screen
 - ▣ Appropriate for pre-push testing
- Send email to a mail list
 - ▣ Can be generated by dashboard
- Test results dashboard
 - ▣ Can display results from a range of dates
 - ▣ Can detect changes in pass/fail conditions
 - ▣ Allows results to be sorted and searched
 - ▣ Enhances visibility of failing builds and tests

Motivating people to write tests

26

- Tests protect YOU from other people from breaking your work
 - ▣ If someone else's changes break your code, they are responsible for fixing it
- Testing is cheaper and easier than debugging
- You may already have some tests lying around
 - ▣ Drivers for generating conference or paper results
 - ▣ User submitted bugs
 - ▣ Examples

How do we determine what other tests are needed?

27

□ Code coverage tools

- Expose parts of the code that aren't being tested

- gcov

 - standard utility with the GNU compiler collection suite

 - counts the number of times each statement is executed

- lcov

 - a graphical front-end for gcov

 - available at <http://ltp.sourceforge.net/coverage/lcov.php>

How to use lcov

28

- Compile and link your code with `--coverage` flag
 - ▣ It's a good idea to disable optimization
- Run your test suite
- Collect coverage data using `lcov`
- Generate html output using `genhtml`

A simple example

```
#include<iostream>
#include "isEven.hpp"

int main()
{
    int num = 8;

    if(isEven(num))
        std::cout << num << " is an even number.\nTEST PASSED";
    else
        std::cout << num << " is an odd number.\nTEST FAILED";

    return 0;
}
```

```
bool isEven(int x)
{
    if(x%2 == 0)
        return true;
    return false;
}
```

A simple example

30

- Compile and link with `--coverage` flag
 - ▣ `g++ --coverage evenExample.cpp -o evenExample`
 - ▣ This creates a file called `evenExample.gcno`
- Run the test
 - ▣ `./evenExample`
 - ▣ This creates a file called `evenExample.gcda`
- Collect coverage data using `lcov`
 - ▣ `lcov --capture --directory . --output-file evenExample.info`
 - ▣ This creates `evenExample.info`
- Generate html output using `genhtml`
 - ▣ `genhtml evenExample.info --output-directory evenHTML`
 - ▣ This generates html files in the directory `evenHTML`

A simple example

31

LCOV - code coverage report

Current view: **top level** - `/home/amklinv/IDEAS/testingTalk/examples/simpleExample`

Test: `evenExample.info`

Date: `2016-05-24 14:13:07`

	Hit	Total	Coverage
Lines:	9	11	81.8 %
Functions:	4	4	100.0 %

Filename	Line Coverage ↕	Functions ↕
evenExample.cpp	<div><div></div></div> 85.7 % 6 / 7	100.0 % 3 / 3
isEven.hpp	<div><div></div></div> 75.0 % 3 / 4	100.0 % 1 / 1

Generated by: [LCOV version 1.12-4-g04a3c0e](#)

This is the file
we're testing.

A simple example

32

LCOV - code coverage report

Current view: [top level](#) - [home/amkllnv/IDEAS/testingTalk/examples/simpleExample](#) - [IsEven.hpp](#) (source / functions)

Test: [evenExample.info](#)

Date: 2016-05-24 14:13:07

	Hit	Total	Coverage
Lines:	3	4	75.0 %
Functions:	1	1	100.0 %

Line data

Source code

1	1	:	bool isEven(int x)
2		:	{
3	1	:	if(x%2 == 0)
4	1	:	return true;
5		:	
6	0	:	return false;
7		:	}

We never tested this line of code
(which activates when x is odd)

Let's add another test

```
#include<iostream>
#include "isEven.hpp"

int main()
{
    int num = 7;

    if(isEven(num))
        std::cout << num << " is an even number.\nTEST FAILED";
    else
        std::cout << num << " is an odd number.\nTEST PASSED";

    return 0;
}
```

```
bool isEven(int x)
{
    if(x%2 == 0)
        return true;
    return false;
}
```

A simple example

34

- Compile and link with `--coverage` flag
 - ▣ `g++ --coverage oddExample.cpp -o oddExample`
 - ▣ This creates a file called `oddExample.gcno`
- Run the test
 - ▣ `./oddExample`
 - ▣ This creates a file called `oddExample.gcda`
- Collect coverage data for BOTH TESTS using `lcov`
 - ▣ `lcov --capture --directory . --output-file twoExamples.info`
 - ▣ This creates `twoExamples.info`
- Generate html output using `genhtml`
 - ▣ `genhtml twoExamples.info --output-directory totalHTML`
 - ▣ This generates html files in the directory `totalHTML`

A simple example

35

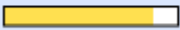


LCOV - code coverage report

Current view: [top level](#) - [/home/amklinv/IDEAS/testingTalk/examples/simpleExample](#)

Test: [twoExamples.info](#)

Date: [2016-05-24 15:17:38](#)

	Hit	Total	Coverage
Lines:	16	18	88.9 %
Functions:	7	7	100.0 %

Filename	Line Coverage ↕			Functions ↕	
evenExample.cpp		85.7 %	6 / 7	100.0 %	3 / 3
isEven.hpp		100.0 %	4 / 4	100.0 %	1 / 1
oddExample.cpp		85.7 %	6 / 7	100.0 %	3 / 3

Generated by: [LCOV version 1.12-4-g04a3c0e](#)

This is the file
we're testing

A simple example

36

LCOV - code coverage report

Current view: [top level](#) - [home/amklInv/IDEAS/testingTalk/examples/simpleExample](#) - [IsEven.hpp](#) (source / functions)

Test: [twoExamples.info](#)

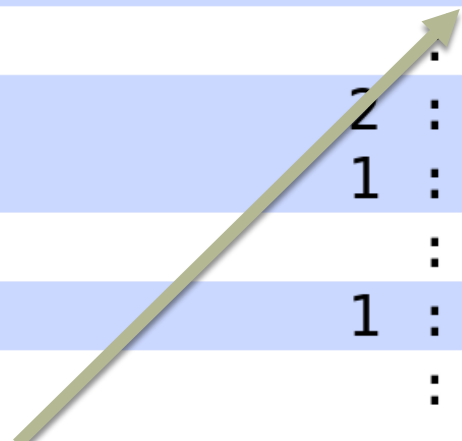
Date: 2016-05-24 15:17:38

	Hit	Total	Coverage
Lines:	4	4	100.0 %
Functions:	1	1	100.0 %

Line data

Source code

1	2	:	bool isEven(int x)
2	:		{
3	2	:	if(x%2 == 0)
4	1	:	return true;
5	:		
6	1	:	return false;
7	:		}



We tested every line
of this function

A real example - xSDKTrilinos

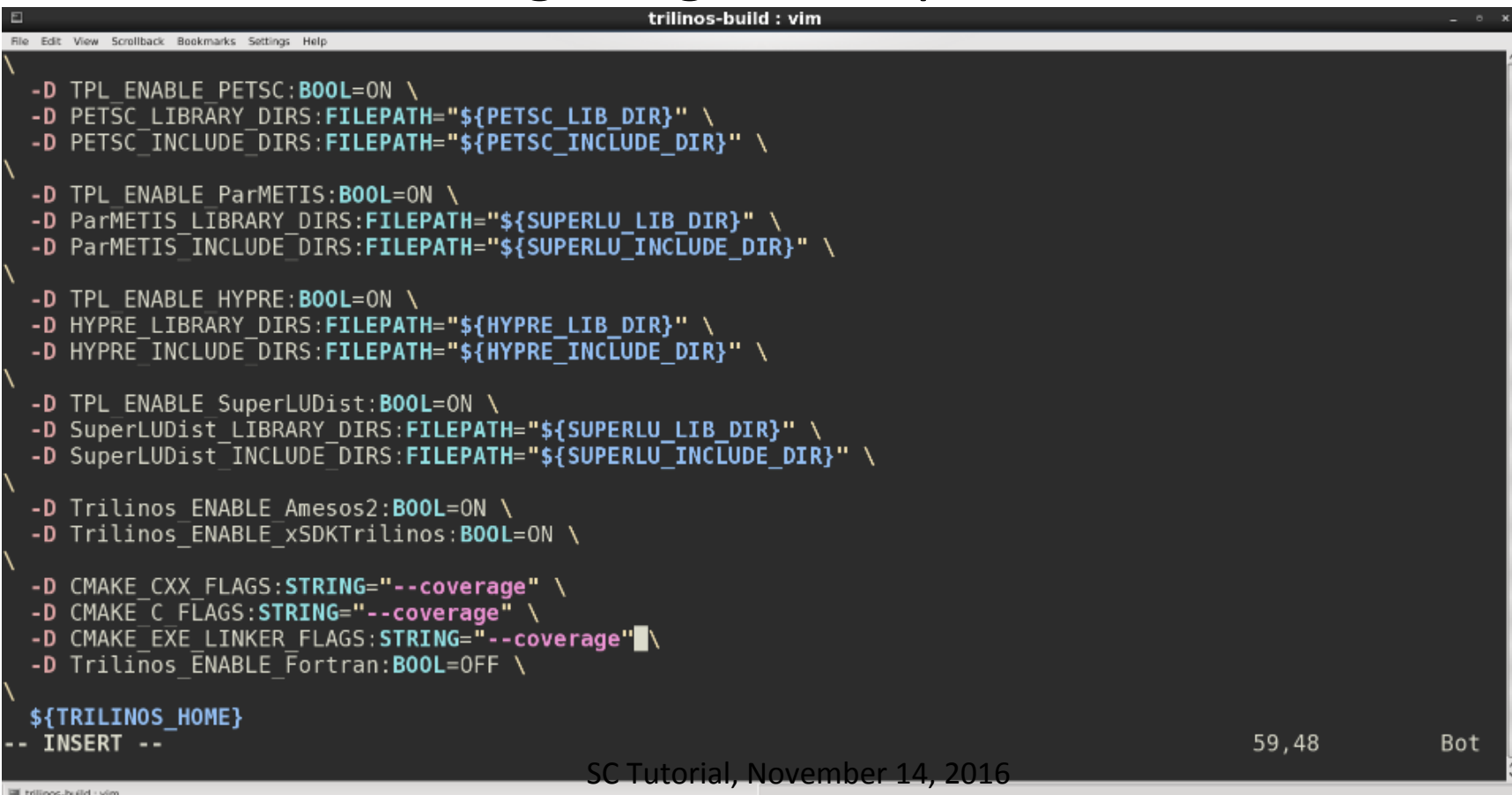
37

- Part of the Trilinos library, developed at SNL as part of the IDEAS project
- Contains the interfaces between Trilinos, PETSc, and hypre (various DOE codes)
- Available at <https://github.com/trilinos/xSDKTrilinos>
- Ten automated tests are run nightly
 - ▣ Six are actually examples that were converted into tests
- Did we leave anything out?

A real example - xSDKTrilinos

38

- Step 1: Modify our CMake configuration file to use the `--coverage` flag to compile and link



```
trilinos-build : vim
File Edit View Scrollback Bookmarks Settings Help

-D TPL_ENABLE_PETSC:BOOL=ON \
-D PETSC_LIBRARY_DIRS:FILEPATH="${PETSC_LIB_DIR}" \
-D PETSC_INCLUDE_DIRS:FILEPATH="${PETSC_INCLUDE_DIR}" \

-D TPL_ENABLE_ParMETIS:BOOL=ON \
-D ParMETIS_LIBRARY_DIRS:FILEPATH="${SUPERLU_LIB_DIR}" \
-D ParMETIS_INCLUDE_DIRS:FILEPATH="${SUPERLU_INCLUDE_DIR}" \

-D TPL_ENABLE_HYPRE:BOOL=ON \
-D HYPRE_LIBRARY_DIRS:FILEPATH="${HYPRE_LIB_DIR}" \
-D HYPRE_INCLUDE_DIRS:FILEPATH="${HYPRE_INCLUDE_DIR}" \

-D TPL_ENABLE_SuperLUDist:BOOL=ON \
-D SuperLUDist_LIBRARY_DIRS:FILEPATH="${SUPERLU_LIB_DIR}" \
-D SuperLUDist_INCLUDE_DIRS:FILEPATH="${SUPERLU_INCLUDE_DIR}" \

-D Trilinos_ENABLE_Amesos2:BOOL=ON \
-D Trilinos_ENABLE_xSDKTrilinos:BOOL=ON \

-D CMAKE_CXX_FLAGS:STRING="--coverage" \
-D CMAKE_C_FLAGS:STRING="--coverage" \
-D CMAKE_EXE_LINKER_FLAGS:STRING="--coverage" \
-D Trilinos_ENABLE_Fortran:BOOL=OFF \

${TRILINOS_HOME}
-- INSERT --

59,48 Bot
```

A real example - xSDKTrilinos

39

- Build Trilinos (including xSDKTrilinos)
 - ▣ `./do-configure`
 - ▣ `make -j`
- This will create a whole bunch of .gcno files
- This will also build the xSDKTrilinos tests because the configure file included
 - ▣ `-D Trilinos_ENABLE_TESTS:BOOL=ON`
 - ▣ `-D Trilinos_ENABLE_EXAMPLES:BOOL=ON`
 - ▣ `-D Trilinos_ENABLE_ALL_OPTIONAL_PACKAGES=ON`

A real example - xSDKTrilinos

40

□ Run the tests using ctest

```
trilinos-build : ctest
File Edit View Scrollback Bookmarks Settings Help
[amklinv@s995692 trilinos-build]$ ctest
Test project /home/amklinv/IDEAS/testingTalk/trilinos-build
  Start 1: Amesos2_KLU2_UnitTests_MPI_4
1/18 Test #1: Amesos2_KLU2_UnitTests_MPI_4 ..... Passed    1.46 sec
  Start 2: Amesos2_SuperLU_DIST_Solver_Test_MPI_4
2/18 Test #2: Amesos2_SuperLU_DIST_Solver_Test_MPI_4 ..... Passed    2.80 sec
  Start 3: Amesos2_SolverFactory_UnitTests_MPI_4
3/18 Test #3: Amesos2_SolverFactory_UnitTests_MPI_4 ..... Passed    1.46 sec
  Start 4: Amesos2_Tpetra_MultiVector_Adapter_UnitTests_MPI_4
4/18 Test #4: Amesos2_Tpetra_MultiVector_Adapter_UnitTests_MPI_4 ... Passed    1.36 sec
  Start 5: Amesos2_Tpetra_CrsMatrix_Adapter_UnitTests_MPI_4
5/18 Test #5: Amesos2_Tpetra_CrsMatrix_Adapter_UnitTests_MPI_4 ..... Passed    1.42 sec
  Start 6: Amesos2_Epetra_MultiVector_Adapter_UnitTests_MPI_4
6/18 Test #6: Amesos2_Epetra_MultiVector_Adapter_UnitTests_MPI_4 ... Passed    1.35 sec
  Start 7: Amesos2_Epetra_RowMatrix_Adapter_UnitTests_MPI_4
7/18 Test #7: Amesos2_Epetra_RowMatrix_Adapter_UnitTests_MPI_4 ..... Passed    1.35 sec
  Start 8: Amesos2_CrsMatrix_Adapter_Consistency_Tests_MPI_4
8/18 Test #8: Amesos2_CrsMatrix_Adapter_Consistency_Tests_MPI_4 .... Passed    1.47 sec
  Start 9: xSDKTrilinos_PETScAIJMatrix_MPI_4
9/18 Test #9: xSDKTrilinos_PETScAIJMatrix_MPI_4 ..... Passed    1.42 sec
  Start 10: xSDKTrilinos_PETSc_Amesos2_example_MPI_4
10/18 Test #10: xSDKTrilinos_PETSc_Amesos2_example_MPI_4 ..... Passed    1.42 sec
  Start 11: xSDKTrilinos_PETSc_Anasazi_example_MPI_4
11/18 Test #11: xSDKTrilinos_PETSc_Anasazi_example_MPI_4 ..... Passed    2.71 sec
  Start 12: xSDKTrilinos_PETSc_Ifpack2_example_MPI_4
12/18 Test #12: xSDKTrilinos_PETSc_Ifpack2_example_MPI_4 ..... Passed    1.47 sec
  Start 13: xSDKTrilinos_PETSc_MueLu_example_MPI_4
```


A real example - xSDKTrilinos

41

□ All tests passed. Yay!

```
trilinos-build : ctest
File Edit View Scrollback Bookmarks Settings Help
Start 1: xSDKTrilinos_PETSc_Amesos2_example_MPI_4
10/18 Test #10: xSDKTrilinos_PETSc_Amesos2_example_MPI_4 ..... Passed      1.42 sec
Start 11: xSDKTrilinos_PETSc_Anasazi_example_MPI_4
11/18 Test #11: xSDKTrilinos_PETSc_Anasazi_example_MPI_4 ..... Passed      2.71 sec
Start 12: xSDKTrilinos_PETSc_Ifpack2_example_MPI_4
12/18 Test #12: xSDKTrilinos_PETSc_Ifpack2_example_MPI_4 ..... Passed      1.47 sec
Start 13: xSDKTrilinos_PETSc_MueLu_example_MPI_4
13/18 Test #13: xSDKTrilinos_PETSc_MueLu_example_MPI_4 ..... Passed      2.34 sec
Start 14: xSDKTrilinos_example_TpetraKSP_MPI_4
14/18 Test #14: xSDKTrilinos_example_TpetraKSP_MPI_4 ..... Passed      1.50 sec
Start 15: xSDKTrilinos_example_EpetraKSP_MPI_4
15/18 Test #15: xSDKTrilinos_example_EpetraKSP_MPI_4 ..... Passed      1.37 sec
Start 16: xSDKTrilinos_HypreTest_MPI_4
16/18 Test #16: xSDKTrilinos_HypreTest_MPI_4 ..... Passed      1.42 sec
Start 17: xSDKTrilinos_Hypre_Belos_example_MPI_4
17/18 Test #17: xSDKTrilinos_Hypre_Belos_example_MPI_4 ..... Passed      1.38 sec
Start 18: xSDKTrilinos_Hypre_Solve_example_MPI_4
18/18 Test #18: xSDKTrilinos_Hypre_Solve_example_MPI_4 ..... Passed      1.36 sec

100% tests passed, 0 tests failed out of 18

Label Time Summary:
Amesos2      = 12.67 sec (8 tests)
xSDKTrilinos = 16.39 sec (10 tests)

Total Test time (real) = 29.11 sec
[amklinv@s995692 trilinos-build]$
```

A real example - xSDKTrilinos

42

- Collect coverage data for the tests using lcov
 - ▣ `lcov --capture --directory . --output-file xSDKTrilinos.info`
 - ▣ This creates xSDKTrilinos.info
 - ▣ lcov processes 634 gcda files in this step, so this does take a few minutes

A real example - xSDKTrilinos

43

- Generate html output using genhtml
 - ▣ `genhtml xSDKTrilinos.info --output-directory xSDKTrilinos`
 - ▣ This generates html files in the directory xSDKTrilinos
 - ▣ This step takes a few minutes too

A real example - xSDKTrilinos

44




LCOV - code coverage report

Current view: [top level](#) - xSDKTrilinos/petsc/src

Test: xSDKTrilinos.info

Date: 2016-06-02 15:36:10

	Hit	Total	Coverage
Lines:	342	420	81.4 %
Functions:	77	117	65.8 %

Filename	Line Coverage ↕		Functions ↕	
BelosPETScSolMgr.hpp		84.7 %	166 / 196	68.2 % 30 / 44
Tpetra_PETScAI_Graph.hpp		75.3 %	67 / 89	62.5 % 20 / 32
Tpetra_PETScAI_Matrix.hpp		80.7 %	109 / 135	65.9 % 27 / 41

Generated by: [LCOV version 1.12-4-g04a3c0e](#)

Let's take a look at
the solver interface.

```


766 : //=====
767 : template<class ScalarType, class MV, class OP>
768 192 : PetscErrorCode PETScSolMgr<ScalarType,MV,OP>::applyPrec(PC M, Vec x, Vec Mx)
769 : {
770 :     using Teuchos::RCP;
771 :     typedef PETScSolMgrHelper<ScalarType,MV,OP> Helper;
772 :
773 :     PetscErrorCode ierr;
774 :     const PetscScalar * xData;
775 :     PetscScalar * MxData;
776 :     void * ptr;
777 :
778 :     // Get the problem out of the context
779 192 :     ierr = PCShellGetContext(M,&ptr); CHKERRQ(ierr);
780 192 :     LinearProblem<ScalarType,MV,OP> * problem = (LinearProblem<ScalarType,MV,OP>*)ptr;
781 :
782 :     // Rip the raw data out of the PETSc vectors
783 192 :     ierr = VecGetArrayRead(x, &xData); CHKERRQ(ierr);
784 192 :     ierr = VecGetArray(Mx, &MxData); CHKERRQ(ierr);
785 :
786 :     // Wrap the PETSc data in a Trilinos Vector
787 192 :     RCP<MV> trilinosX, trilinosMX;
788 192 :     Helper::wrapVector(const_cast<PetscScalar*>(xData), *problem->getLHS(), trilinosX);
789 192 :     Helper::wrapVector(MxData, *problem->getLHS(), trilinosMX);
790 :
791 :     // Perform the multiplication
792 192 :     if(problem->isLeftPrec()) {
793 192 :         problem->applyLeftPrec(*trilinosX, *trilinosMX);
794 :     }
795 :     else {
796 0 :         problem->applyRightPrec(*trilinosX, *trilinosMX);
797 :     }
798 :
799 :     // Unwrap the vectors; this is necessary if we copied data in the wrap step
800 192 :     Helper::unwrapVector(MxData, trilinosMX);
801 :
802 :     // Restore the PETSc vectors
803 192 :     ierr = VecRestoreArrayRead(x,&xData); CHKERRQ(ierr);
804 192 :     ierr = VecRestoreArray(Mx,&MxData); CHKERRQ(ierr);
805 :
806 192 :     return 0;
807 : }
808 :

```

A real example - xSDKTrilinos

46

```
791      :    // Perform the multiplication
792  192 :    if(problem->isLeftPrec()) {
793  192 :        problem->applyLeftPrec(*trilinosX, *trilinosMX);
794      :    }
795      :    else {
796  0 :        problem->applyRightPrec(*trilinosX, *trilinosMX);
797      :    }
```



Oops. I never tested the *right* preconditioning branch...

A hands-on gcov tutorial

47

- <https://amklinv.github.io/morpheus/index.html>

Coverity scan

48

- A free cloud-based static analysis product for open source code
 - ▣ Also available for non open source code, but not free
- Analyzes over 4000 open source projects
- Used to analyze
 - ▣ Sudden unintended acceleration of Toyota vehicles
 - ▣ Large Hadron Collider software
 - ▣ Mars Curiosity rover flight software
 - ▣ Libre Office

Coverity scan

49

- Automatically looks at all the different paths
- Finds
 - ▣ Resource leaks
 - ▣ Dereferences of null pointers
 - ▣ Use of uninitialized data
 - ▣ Memory corruptions
 - ▣ Control flow issues
 - ▣ Use of resources that have been freed
 - ▣ And more!

How to use coverity scan

50

- Create a project
- Tell it about your open source license
- Options for performing the scan:
 - ▣ Upload a tarball
 - ▣ Point it to a URL
 - ▣ Use TravisCI (https://scan.coverity.com/travis_ci)

Coverity analysis of Trilinos

51

- <https://scan.coverity.com/projects/1680>

Continuous integration (CI): a master branch that always works

52

- ❑ Code changes trigger automated builds/tests on target platforms
- ❑ Builds/tests finish *in a reasonable amount of time*, providing useful feedback when it's most needed
- ❑ Immensely helpful!
- ❑ Requires some work, though:
 - ▣ A reasonably automated build system
 - ▣ An automated test system with significant test coverage
 - ▣ A set of systems on which tests will be run, and a controller

Continuous integration (CI): a master branch that always works

53

- Has existed for some time
- Adoption has been slow
 - ▣ Setting up and maintaining CI systems is difficult, labor-intensive (typically requires a dedicated staff member)
 - ▣ *You have to be doing a lot of things right to even consider CI*

Formulating a continuous testing regime

54

- Identify verification needs within software
 - ▣ Defines code-coverage requirements
 - ▣ Pick features of the code necessary for correct behavior
 - Individual units
 - Interaction between units
 - ▣ Know the purpose of testing each feature
 - Reduces wasted effort and computing resources
 - Helps identify the most appropriate type of test
- Identify behaviors of code with detectable response to changes

Cloud-based CI is available as a service on GitHub

55

- Automated builds/tests can be triggered via pull requests
- Builds/tests can be run on cloud systems – no server in your closet. *Great use of the cloud!*
- Test results are reported on the pull request page (with links to detailed logs)
- Already being used successfully by scientific computing projects, with noticeable benefits to productivity
- Not perfect, but *far* better than not doing CI

Travis CI is a great choice for HPC

56

- Integrates easily with GitHub
- *Free* for Open Source projects
- Supports environments with C/C++/Fortran compilers (GNU, Clang, Intel[?])
- Linux, Mac platforms available
- *Relatively* simple, *reasonably* flexible configuration file
 - ▣ Documentation is sparse, but we now have working examples

Travis CI live demo

57

- <https://github.com/amklinux/morpheus>

Verification and Validation

Challenges specific to scientific software

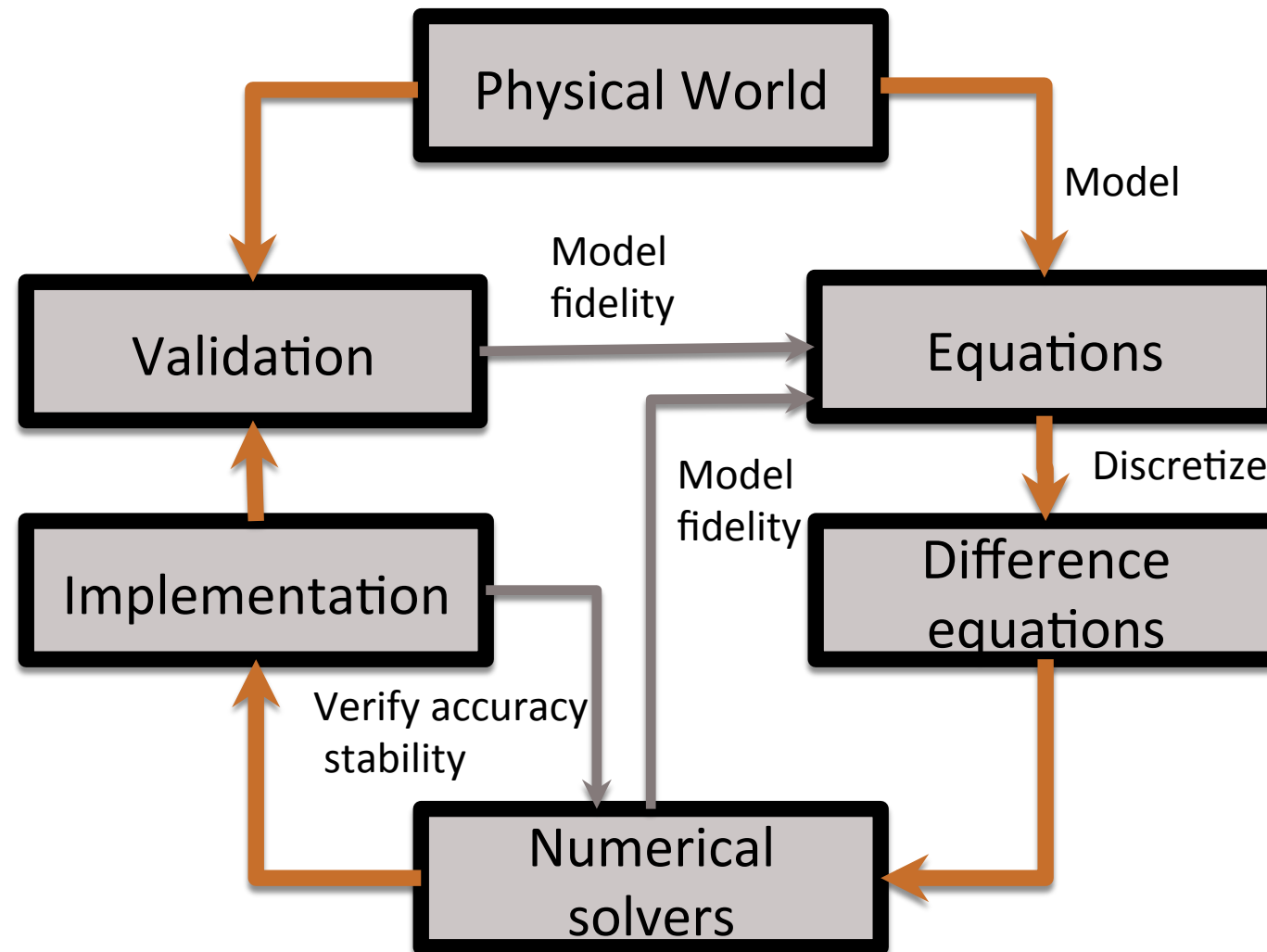
V&V during different stages

Model validation

Verification of methods and implementation

Simplified schematic of science through computation

59



This is for simulations, but the philosophy applies to other computations too.

Many stages in the lifecycle have components that may themselves be under research => need modifications

Definitions

60

- At highest level
 - ▣ Verification – the implementation has expected behavior
 - ▣ Validation – the model reflects the phenomenon it is meant to capture
- Different validation definitions can be applied in other circumstances

Verification

61

- Code verification uses tests
 - ▣ It is much more than a collection of tests
- It is the holistic process through which you ensure that
 - ▣ Your implementation shows expected behavior,
 - ▣ Your implementation is consistent with your model,
 - ▣ Science you are trying to do with the code can be done.

Validation

62

- Model validation
 - ▣ Compare with observations
 - From sensors, telescopes, experiments
 - Others
- Not necessary to capture whole reality
 - ▣ Features of interest
 - Are the approximations appropriate?
- Method validation
 - ▣ Validate method order
 - ▣ Construct code-to-code comparisons

CSE testing challenges

63

- Floating point issues
 - ▣ Different results
 - On different platforms and runs
 - Ill-conditioning can magnify these small differences
 - Final solution may be different
 - Number of iterations may be different
- Unit testing
 - ▣ Sometimes producing meaningful testable behavior too dependent upon other parts of the code
- Definitions don't always fit

CSE verification challenges

64

- Integration testing may have hierarchy too
- Particularly true of codes that allow composability in their configuration
- Codes may incorporate some legacy components
 - ▣ Its own set of challenges
 - No existing tests of any granularities
- Examples – multiphysics application codes that support multiple domains
 - ▣ FLASH case study later

CSE validation challenges

65

- Interdisciplinary
 - ▣ Domain knowledge
 - ▣ Applied mathematics
 - ▣ Software engineering
- Exploring uncharted territories
 - ▣ Existing knowledge is of limited interest
 - ▣ Need to push the boundaries
 - ▣ The behavior of solvers not always predictable in regimes of interest

Stages and types of verification

66

- During initial code development
 - ▣ Accuracy and stability during development of the algorithm
 - ▣ Matching the algorithm to the model
 - ▣ Interoperability of algorithms
- In later stages
 - ▣ While adding new major capabilities or modifying existing capabilities
 - ▣ Ongoing maintenance
 - ▣ Preparing for production
- If refactoring
 - ▣ Ensuring that behavior remains consistent and expected
- All stages have a mix of automation and human-intervention

Note that the stages apply to the whole code as well as its components

Development phase

67

- Development of tests and diagnostics goes hand-in-hand with code development
 - ▣ Non-trivial to devise good tests, but extremely important
 - ▣ Compare against simpler analytical or semi-analytical solutions
 - They can also form a basis for unit testing
- In addition to testing for “correct” behavior, also test for stability, convergence, or other such desirable characteristics
- Many of these tests will be worth preserving for the maintenance phase

Remember that these apply to the whole code as well as its components

Development phase – adding on

68

- Few more steps when adding new components to existing code
 - ▣ Know the existing components it interacts with
 - ▣ Verify its interoperability with those components
 - ▣ Verify that it does not inadvertently break some unconnected part of the code
- May need addition of tests not just for the new component but also for some of the old components
 - ▣ This part is often overlooked to the detriment of the overall verification

Maintenance phase

69

- ❑ Concerns mature, mostly unchanging code
- ❑ Testing mostly automated
- ❑ Verify ongoing correctness
 - ▣ With incremental changes
- ❑ Code and interoperability coverage are critical
- ❑ Software process should include policies about handling failures
 - ▣ Prioritization
 - ▣ Turn-around time

Examples: Tpetra verification

70

- Distributed basic linear algebra subroutines
 - ▣ Sparse matrices
 - ▣ Dense matrices
- Check for correct linear algebra
- Check for correct errors
 - ▣ Does the program throw an exception if I try to multiply things with incompatible dimensions?

Belos verification

71

- Krylov solvers
- Use problems with known solutions
 - ▣ Given A and Y , generate $B=AY$
 - Ensures B is in the range of A
 - ▣ Solve $AX=B$
- Some tests use Belos matrix and vector classes
- Some tests use Epetra/Tpetra classes
- Test with and without preconditioning
 - ▣ Left and right

Anasazi verification

72

- Eigensolvers
- Use problems with known solutions
 - ▣ Generated using Matlab's sprand
 - ▣ Problems with analytic solutions
 - Discretization of the Laplace operator
- Measure the residual of the computed eigenvectors
 - ▣ $R = AX - BX\Lambda$
- Number of iterations are compared to a gold standard

Zoltan(2) verification

73

- Graph partitioning
 - ▣ Some Sandia-developed code
 - ▣ Some TPL wrappers
- Gold standard solutions
 - ▣ Labor intensive
 - ▣ Gold standard changes when algorithms change
 - ▣ Upgrades to a TPL such as ParMETIS require gold standard to be updated
- Uses metrics to determine whether the solution is correct
 - ▣ Edge cuts
 - ▣ Balance criteria

SuperLU verification

74

- SuperLU – sparse Gaussian elimination code
- Test suite
 - ▣ Many unit and integration level tests
 - ▣ Entire suite can be run in a few minutes
 - ▣ Demonstrates validation and acceptance testing, also no-change or bounded-change testing
 - ▣ Demonstrates how to deal with floating point issues

SuperLU test suite

75

- Suite has two main goals
 - ▣ Tests query functions to floating-point parameters
 - Machine epsilon, underflow and overflow thresholds, etc
 - ▣ Provide coverage of all routines
 - Tests all functions of the user-callable routines

SuperLU test suite

76

- Many input matrices are generated
 - ▣ Different numerical and structural properties
- Uses several numerical metrics to assert accuracy of solution
 - ▣ Stable LU factorization
 - ▣ Small forward and backward errors

Example: SuperLU test suite

77

- Performs exhaustive testing of a large number of input parameters

```
For each set of valid values {  
  For each set of valid values {  
    ...  
    For each set of valid values {  
      For each matrix type {  
        Generate the input matrix A and rhs b;  
        Call a user-callable routine with input values {, ,..., };  
        Compute the test metrics;  
        Check whether each metric is smaller than a prescribed  
threshold;  
      }  
    }  
    ...  
  }  
}
```

- Runs over 10,000 tests in a few minutes

FLASH verification and validation

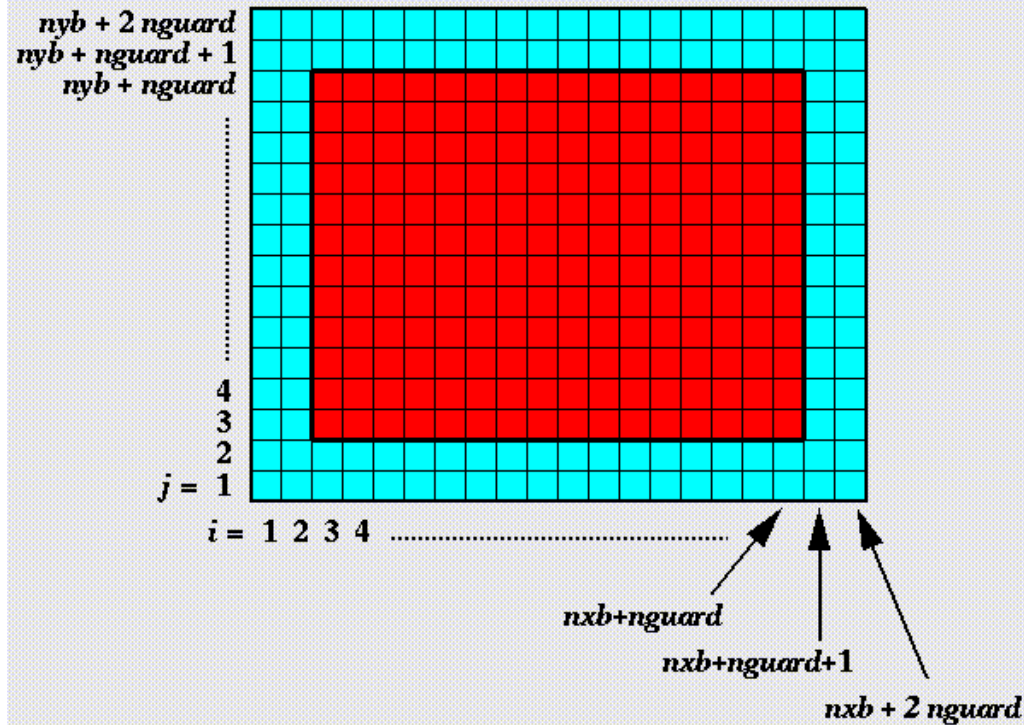
78

- http://flash.uchicago.edu/site/testsuite/viewer/viewBuilds.py?target_dir=/home/tester/flashTest/output/gin-nag/2012-06-06
- Note the combination of unit/composite tests
 - ▣ Terminology is inconsistent with standard definitions
 - ▣ It serves the developers and users well
- Unit tests compare against analytical, semi-analytical or manufactured solutions
- Composite tests are integrated or system level
 - ▣ Compare output against gold standard

Against manufactured solution

79

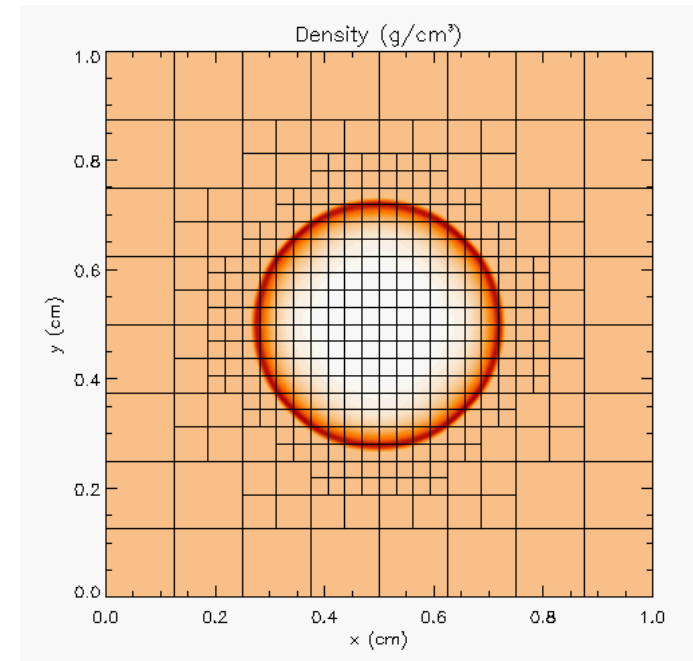
- Verification of guard cell fill
- Use two variables A & B
- Initialize A including guard cells and B excluding them
- Apply guard cell fill to B



Against analytical solution

80

- ❑ Sedov blast wave
- ❑ High pressure at the center
- ❑ Shock moves out spherically
- ❑ FLASH with AMR and hydro
- ❑ Known analytical solution



Though it exercises both mesh, hydro and eos, if mesh and eos are verified first, then this test verifies hydro

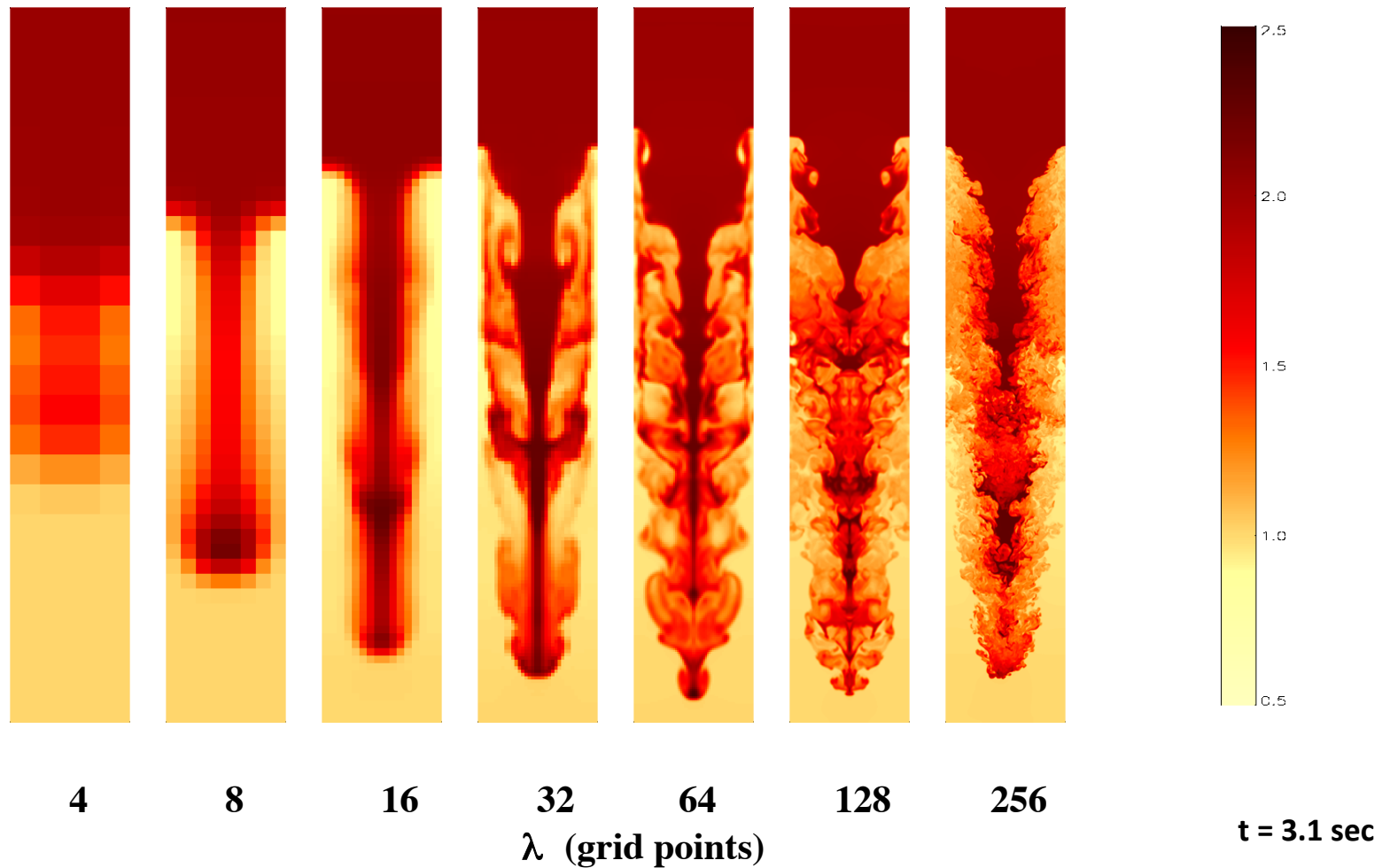
Building confidence

81

- Use ghost cell fill and Sedov tests, add one more
 - ▣ Eos unit
 - Use initial conditions from a known problem
 - Apply eos in two different modes – at the end all variables should be consistent within tolerance
- First two tests are stand-alone
- The third test depends on Grid and Eos
 - ▣ Not all of Grid functionality it uses is tested
 - Flux correction in AMR
- If Grid and Eos tests passed and Hydro failed
 - ▣ If UG version failed then fault is in hydro
 - ▣ If UG passed and AMR failed the fault is likely in flux correction

Validation single-mode Rayleigh-Taylor

82



Density (g/cc)

Methodology

Evaluating project needs

Devising testing regime

Examples from Alquimia, Amanzi and Trilinos

Why not always use the most stringent testing?

84

- Effort spent in devising tests and testing regime are a tax on team resources
- When the tax is too high...
 - ▣ Team cannot meet code-use objectives
- When is the tax is too low...
 - ▣ Necessary oversight not provided
 - ▣ Defects in code sneak through

Evaluating project needs

85

□ Objectives

- ▣ Proof of concept
- ▣ Limited research use
- ▣ Library
- ▣ Production – simulations and analysis

□ Team

- ▣ Number of developers
- ▣ Background of developers
- ▣ Geographical spread

Evaluating project needs

86

- Lifecycle stages
- Lifetime
 - ▣ How long a code is expected to live
 - ▣ New code versus some legacy components
- Complexity
 - ▣ Number of modules, models, data structures, solvers
 - ▣ Degree of coupling and interoperability requirements

Commonalities

87

- Unit testing is always good
 - ▣ It is unlikely to be sufficient
- Verification of expected behavior
- Understanding the range of validity and applicability is always important
 - ▣ Especially for individual solvers

Building a test suite for CSE codes

88

Ideal time is *during development*

- When software is mature, must ensure new code does not break old features
 - ▣ Without regular testing, adding new code is error-prone
 - ▣ Structural changes are tedious without a way to verify ongoing correctness
 - ▣ Regular automated testing can provide a huge savings

Consider the project scope

89

- Proof of concept
 - ▣ Nothing more than the common testing of previous slide
- Limited use
 - ▣ Manually run test-suite before each use may suffice
 - Coverage is still important
- Library
 - ▣ Depends on team and complexity
- Regular simulation and analysis
 - ▣ Depends on team and complexity
 - ▣ Testing coverage needs system level integrated coverage

Customizing for project needs: Team

90

- One to two developers – periodic manual testing and verification
- Mid-size to large team – automated test suite running regularly
- Subgroups within the team – automated test suite with tests of different granularity
 - ▣ May also need multiple suites run on their own schedules

Considering complexity and lifetime

91

- ❑ What runs in the regular test-suite?
- ❑ If there are subgroups, what goes into the separate test-suites?
- ❑ How often should each test-suite run?
- ❑ How do you ensure interoperability coverage?

The question to answer: how do you balance the tax amount for maximum productivity?

Other factors

92

- Frequency of testing depends upon lifecycle stage
 - ▣ Mid-size to large team working on the same code component doing rapid development – ideally continuous integration
 - ▣ Stable mature code - regular automated testing
 - ▣ Refactoring – needs its own strategy
- Complexity and lifetime
 - ▣ Affect the testing regime being devised
 - ▣ Testing needs and strategy differ when code incorporates legacy components

Maintenance of a test suite

93

- Testing regime is only useful if it is
 - ▣ Maintained
 - ▣ Monitored regularly
 - ▣ Has rapid response to failure
- Maintenance includes
 - ▣ Updating tests and benchmarks
 - ▣ Adjustments to software stack
 - ▣ Archiving and retrieval of test suite output
 - Helpful in tracing change in code behavior

Maintenance of a test suite

94

- Monitoring individual tests manually is unreasonable and should be automated
 - ▣ Manual inspection should be limited to failing tests
 - ▣ For repository code, failure can be correlated to check-ins within a particular time-frame
 - Only certain developers need to be involved

Maintenance of a test suite

95

- Tests should pass most of the time
 - ▣ Easy when code changes are infrequent
 - ▣ Harder when code is large and rapidly changing
 - Difficult to determine cause of failure
 - Pre-commit test suites are a good idea

Maintenance of a test suite

96

- Periodically review collection of tests
 - ▣ Look for gaps and redundancies
 - ▣ Pruning is important to conserve testing resources
 - ▣ Deprecated features can be removed
 - ▣ New tests may be necessary when new features are added

Selection of tests

97

- Important to aim for quick diagnosis of error
 - ▣ A mix of different granularities works well
 - Unit tests for isolating component or sub-component level faults
 - Integration tests with simple to complex configuration and system level
- Some rules of thumb
 - ▣ Simple
 - ▣ Enable quick pin-pointing
 - ▣ Coverage

For a large code experience with test selection see [Dubey et al 2015](#)

Selection of tests

98

	Hydro	EOS	Gravity	Burn	Particles
AMR	CL	CL		CL	CL
UG	SV	SV			SV
Multigrid	WD	WD	WD	WD	
FFT			PT		

Tests	Symbol
Sedov	SV
Cellular	CL
Poisson	PT
White Dwarf	WD

- A test on the same row indicates interoperability between corresponding physics
- Similar logic would apply to tests on the same column for infrastructure
- More goes on, but this is the primary methodology

Examples

99

- From Alquimia, amanzi and Trilinos
- Focus on different team sizes and objectives
- Different lifetime spans

How is real DOE code tested?

100

	How many developers?	How much code?	How frequent are changes?
Alquimia	< 1 FTE	O(1,000) lines of code	Every few months
Amanzi	About a dozen	O(100,000) lines of code	A few commits every day
Trilinos	A few dozen	O(1,000,000) lines of code	About 12 per day

What is Alquimia?

101

- ❑ Biogeochemistry API and wrapper library
- ❑ Provides a unified interface to existing geochemistry engines
 - ❑ CrunchFlow
 - ❑ PFLOTRAN
- ❑ Allows subsurface flow and transport simulators to access a range of functionality
- ❑ NOT an implementation of a biogeochemistry reaction library
- ❑ Does NOT perform geochemical calculations

How is Alquimia tested?

102

- Continuous integration testing using Travis CI
- Works for them because
 - ▣ Alquimia builds fast
 - ▣ Test suite runs fast
 - ▣ Commits happen in short bursts

What is Amanzi/ATS?

103

□ Amanzi

- ▣ A parallel flow and reactive transport simulator
- ▣ Used to analyze multiple DOE waste disposal sites
- ▣ Example application: modeling hydrological and biogeochemical cycling in the Colorado River System
 - Carbon cycling is especially important because of its role in regulating atmospheric CO₂

What is Amanzi/ATS?

104

- ATS
 - ▣ Advanced Terrestrial Simulator
 - ▣ Built on Amanzi
 - ▣ Adds physics capability to solve equations for ecosystem hydrology

Amanzi/ATS testing practices

105

- 156 tests that can be run via “ctest”
 - ▣ No continuous integration, but developers are expected to run the test suite before committing
- New physics contributions are required to come with new system-level tests
- Various granularity tests

Amanzi/ATS testing granularity

106

- Unit tests
 - ▣ Code is highly componentized
- Medium-grained component tests
 - ▣ Discretizations
 - ▣ Solvers
- Coarse-grained system-level tests
 - ▣ Test full capability
 - ▣ Serve as example for new users

What is Trilinos?

107

- A collection of libraries intended to be used as building blocks for the development of scientific applications
- Organized into 66 packages
 - ▣ Linear solvers
 - ▣ Nonlinear solvers
 - ▣ Eigensolvers
 - ▣ Preconditioners (including multigrid)
 - ▣ And more!

How is Trilinos tested?

108

- Trilinos has 1500 tests between its 66 packages
- Developers are strongly advised to run a checkin test script when committing
- Automated testing on a variety of different platforms

Checkin test script

109

- ❑ Detects which packages were modified by your commits
- ❑ Determines which packages you potentially broke
- ❑ Configures, builds, and tests those packages
 - ▣ On success, pushes to repo
 - ▣ On failure, reports why it failed
- ❑ Useful for ensuring your changes don't break another package
- ❑ May take a while, but many people run it overnight

Why do we do automated testing if everyone uses the checkin script?

110

- May test a different set of packages
- May test different environments
 - ▣ Do your changes work with Intel compilers as well as GNU?
 - ▣ Do your changes work on a mac?
 - ▣ Do your changes work with CUDA?
- Identifies a small set of commits that could have broken a build or test
 - ▣ Identifies the person who knows how to un-break it
- Bugs are easier to fix if caught early

What if “bad people” don’t use the checkin script?

111

- Their commit doesn’t include the checkin script information

The screenshot shows a GitHub commit page for the `trilinos / Trilinos` repository. The commit title is **Tpetra: Add "compare Maps" utility executable to examples**. The commit message states: `@trilinos/tpetra This is useful for #438 and #558, among others.` Below the message, there is a section for **Build/Test Cases Summary** with the following details:

- Enabled Packages: `TpetraCore`
- Disabled Packages: `FEI,PyTrilinos,Moertel,STK,SEACAS,ThreadPool,OptiPack,Rythmos,Intrepid,ROL`
- 0) `MPI_DEBUG => passed: passed=100,notpassed=0 (5.17 min)`
- 1) `SERIAL_RELEASE => passed: passed=74,notpassed=0 (2.29 min)`
- Other local commits for this build/test group: [b945495](#)

The commit is on the `master` branch, committed by `mhoemmen` 4 days ago. The commit hash is `566b0db7a2dac6455644bcc3ebb01d02f47dac3c`, with parent `b945495`. The commit shows 5 changed files with 226 additions and 0 deletions. The page includes navigation links for Code, Issues (276), Pull requests (28), Wiki, Pulse, and Graphs. Repository statistics show 73 Unwatch, 98 Unstar, and 58 Forks.

Checkin test script examples

112

- Example 1: a harmless change to a comment
- Example 2: breaking the build
- Example 3: breaking some tests

Example 1: a harmless change

113

```
// You should have received a copy of the GNU Lesser General Public
// License along with this library; if not, write to the Free Software
// Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301
// USA
// Questions? Contact Michael A. Heroux (maherou@sandia.gov)
//
// *****
// @HEADER

/*! \file AnasaziTraceMinDavidson.hpp
  \brief Implementation of the TraceMin-Davidson eigensolver
 */

#ifndef ANASAZI_TRACEMIN_DAVIDSON_HPP
#define ANASAZI_TRACEMIN_DAVIDSON_HPP

#include "AnasaziConfigDefs.hpp"
#include "AnasaziEigensolver.hpp"
#include "AnasaziMultiVecTraits.hpp"
-- INSERT --
```



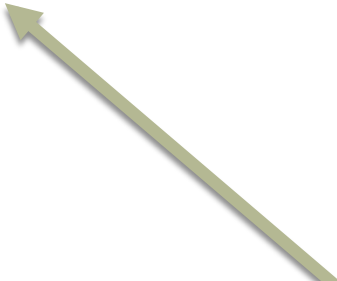
I modified a
comment.

30,61

Example 1: a harmless change

114

 The image cannot be displayed. Your computer may not have enough memory to open the image, or the image may have been corrupted. Restart your computer, and then open the file again. If the red x still appears, you may have to delete the image and then insert it again.



Now I run the checkin test script to make sure I didn't break anything.

Example 1: a harmless change

115

```
Determining the set of packages to enable by examining /home/amklnv/TrilinosDir/github/Trilinos/CHECKIN/modifiedFiles.out
```

```
Modified file: 'packages/anasazi/src/AnasaziTraceMinDavidson.hpp'  
=> Enabling 'Anasazi'!
```

```
Full package enable list: [Anasazi]
```


```
Removing package enables: [FEI,Moertel,STK,Phalanx,PyTrilinos]
```

```
Filtering the set of enabled packages according to allowed package types ...
```

```
Final package enable list: [Anasazi]
```

```
Enabling forward packages on request!
```

```
Adding hard disables for specified packages 'FEI,Moertel,STK,Phalanx,PyTrilinos' ...
```



Note that the checkin script correctly identified which files were modified and which packages they belong to.

Example 1: a harmless change

116

E) Analyze the overall results and send email notification (MPI_DEBUG) ...

E.1) Determine what passed and failed ...

The pull passed!

The configure passed!

The build passed!

testResultsLine = '100% tests passed, 0 tests failed out of 237'

All of the tests ran passed!

E.2) Construct the email message ...



Configure, build, and test
passed for MPI_DEBUG

Example 1: a harmless change

117

```
READY TO PUSH: Trilinos: s995692.srn.sandia.gov
```

```
Thu Apr 21 16:22:57 MDT 2016
```

We are ready to push
because all tests passed

```
Enabled Packages: Anasazi
```

```
Disabled Packages: FEI,Moertel,STK,Phalanx,PyTrilinos
```

```
Enabled all Forward Packages
```

```
Build test results:
```

```
-----  
0) MPI_DEBUG => passed: passed=237,notpassed=0 (8.42 min)  
1) SERIAL_RELEASE => passed: passed=243,notpassed=0 (2.71 min)
```

```
*** Commits for repo :
```

```
982db3b Anasazi: Modified a comment in TraceMin-Davidson
```

Example 2: broken build

118

```
    const Teuchos::RCP<MatOrthoManager<ScalarType,MV,OP> > &ortho,  
    Teuchos::ParameterList &params  
    );
```

private:

```
    //  
    // Convenience typedefs  
    //  
    typedef MultiVecTraits<ScalarType,MV> MVT;  
    typedef OperatorTraits<ScalarType,MV,OP> OPT;  
    typedef Teuchos::ScalarTraits<ScalarType> SCT;  
    typedef typename SCT::magnitudeType MagnitudeType;  
  
    // TraceMin specific methods  
    void addToBasis(const Teuchos::RCP<const MV> Delta);  
  
    void harmonicAddToBasis(const Teuchos::RCP<const MV> Delta);  
}
```

-- INSERT --

99,4

26%

Oops! I accidentally removed the semicolon at the end of the class. This will break the build for sure!

Example 2: broken build

119

C) Do the build (MPI_DEBUG) ...

Running: make -j48

Writing console output to file make.out ...

Runtime for command = 8.235778 minutes

Build failed returning 2!

The checkin script detected
that I broke the build

Traceback (most recent call last):

File "/home/amklinv/TrilinosDir/github/Trilinos/cmake/tribits/ci_support/CheckinTest.py", line 1586, in runBuildTestCase
raise Exception("Build failed!")

Exception: Build failed!

E) Analyze the overall results and send email notification (MPI_DEBUG) ...

E.1) Determine what passed and failed ...

Example 2: broken build

120

```
      from /home/amklnv/temp/Trilinos/packages/anasazi/tpetra/example/TraceMinDavidson/TraceMinDavidsonUserOpEx.cpp:8:
/home/amklnv/temp/Trilinos/packages/anasazi/src/AnasaziTraceMinDavidson.hpp:99:3: error: expected ';' after class definition
In file included from /home/amklnv/temp/Trilinos/packages/anasazi/src/AnasaziTraceMinDavidsonSolMgr.hpp:40:0,
      from /home/amklnv/temp/Trilinos/packages/anasazi/tpetra/example/TraceMinDavidson/TraceMinDavidsonLaplacianEx.cpp:8:
/home/amklnv/temp/Trilinos/packages/anasazi/src/AnasaziTraceMinDavidson.hpp:99:3: error: expected ';' after class definition
make[2]: *** [packages/anasazi/tpetra/test/TraceMinDavidson/CMakeFiles/Anasazi_Tpetra_TraceMinDavidson_largest_standard_test.dir/cxx_main_standard_noprec.cpp.o] Error 1
make[1]: *** [packages/anasazi/tpetra/test/TraceMinDavidson/CMakeFiles/Anasazi_Tpetra_TraceMinDavidson_largest_standard_test.dir/all] Error 2
make[1]: *** Waiting for unfinished jobs....
make[2]: *** [packages/anasazi/tpetra/example/TraceMinDavidson/CMakeFiles/Anasazi_Tpetra_TD_UserOp_example.dir/TraceMinDavidsonUserOpEx.cpp.o] Error 1
make[1]: *** [packages/anasazi/tpetra/example/TraceMinDavidson/CMakeFiles/Anasazi_Tpetra_TD_UserOp_example.dir/all] Error 2
make[2]: *** [packages/anasazi/tpetra/example/TraceMinDavidson/CMakeFiles/Anasazi_Tpetra_TD_UserOp_example.dir/all] Error 2
```

The checkin script also creates a log file with the build error

Example 3: broken tests

121

```
// set the block size and allocate data
int bs = params.get("Block Size", problem_->getNEV());
int nb = params.get("Num Blocks", 1);
// setSize(bs,nb);

NEV_ = problem_->getNEV();

// Create the Ritz shift operator
ritzOp_ = rcp (new tracemin_ritz_op_type (Op_, MOp_, Prec_));

// Set the maximum number of inner iterations
const int innerMaxIts = params.get ("Maximum Krylov Iterations", 200);
ritzOp_->setMaxIts (innerMaxIts);

alpha_ = params.get ("HSS: alpha", ONE);
}
```

Added a logic error to the code

Example 3: broken tests

122

```
FAILED CONFIGURE/BUILD/TEST: Trilinos: s9962.srn.sandia.gov
```

```
Thu Apr 21 17:14:53 MDT 2016
```

```
Enabled Packages: Anasazi
```

```
Disabled Packages: FEI,Moertel,STK,Phalanx,PyTrilinos
```

```
Enabled all Forward Packages
```

```
Build test results:
```

```
-----
```

```
0) MPI_DEBUG => FAILED: passed=233,notpassed=4 => Not ready to push! (8.43 min)
```

```
1) SERIAL_RELEASE => FAILED: passed=239,notpassed=4 => Not ready to push! (2.74 min)
```

```
Failed because one of the build/test cases failed!
```

```
*** Commits for repo :
```

```
6bb949b Anasazi: Broke some TraceMin tests.  Oops!
```

The checkin script detected that I broke several tests

Example 3: broken tests

123

```
98% tests passed, 4 tests failed out of 237
```

```
Label Time Summary:
```

```
Anasazi      = 100.15 sec
```

```
NOX          = 165.35 sec
```

```
Rythmos      = 124.19 sec
```

```
Total test time (real) = 389.89 sec
```

```
The following tests FAILED:
```

```
56 - Anasazi_Tpetra_TraceMin_smallest_proj_test_MPI_4 (Failed)
```

```
57 - Anasazi_Tpetra_TraceMin_smallest_schur_test_MPI_4 (Failed)
```

```
58 - Anasazi_Tpetra_TraceMin_largest_standard_test_MPI_4 (Failed)
```

```
59 - Anasazi_Tpetra_TraceMinDavidson_largest_standard_test_MPI_4 (Failed)
```

```
Errors while running CTest
```

The testing log tells us which tests failed


Trilinos automated testing

124

Login

All Dashboards

Monday, June 06 2016 08:58:08 MDT



Dashboard


Calendar

Previous

Current

Project

Project

Project	Error	Configure Warning	Pass	Error	Build Warning	Pass	Test Not Run	Fail	Pass
Trilinos 	1	531	530	0	272	257	0	14	3976

SubProjects

Project	Error	Configure Warning	Pass	Error	Build Warning	Pass	Test Not Run	Fail	Pass
Teuchos	0	21	21	0	12	9	0	0	227
ThreadPool	0	1	1	0	0	1			
Sacado	0	2	2	0	2	0	0	0	564
RTOp	0	20	20	0	0	20			
Kokkos	0	19	19	0	0	19	0	0	9
Epetra	0	21	21	0	12	9	0	1	244
Zoltan	0	21	21	0	13	8	0	0	135
Shards	0	1	1	0	0	1			
GlobiPack	0	1	1	0	0	1			

Trilinos automated testing

125

Nightly											
Site	Build Name	Update	Configure		Build		Test			Build Time	Labels
		Files	Error ^	Warn	Error	Warn	Not Run	Fail	Pass		
artemis.srn.sandia.gov	Linux-intel-15.0.2-MPI_RELEASE_DEV_DownStream_ETI_SERIAL-OFF_OPENMP-ON_PTHREAD-OFF_CUDA-OFF_COMPLEX-OFF	68	1	140	0	216	0	3	1256	6 hours ago	(44 labels)
lightsaber.srn.sandia.gov	Linux-GCC-4.7.2-RELEASE_DEV_MueLu_Matlab	69	0	111	0	51	0	0	431	10 hours ago	(25 labels)
enigma.sandia.gov	Linux-GCC-4.8.3-OPENMPI_1.6.4_DEBUG_DEV_MueLu_Basker	69	0	227	0	117	0	0	96	9 hours ago	(25 labels)
hansel.sandia.gov	Linux-GCC-4.4.7-MPI_OPT_DEV_XYCE	121	0	70	0	28	0	0	553	9 hours ago	(13 labels)
enigma.sandia.gov	Linux-GCC-4.8.3-OPENMPI_1.6.4_DEBUG_DEV_MueLu_KLU2	69	0	225	0	91	0	0	73	8 hours ago	(25 labels)
enigma.sandia.gov	Linux-GCC-4.8.3-OPENMPI_1.6.4_DEBUG_DEV_MueLu_ExtraTypes_EI	69	0	227	0	117	0	0	97	8 hours ago	(25 labels)
enigma.sandia.gov	Linux-GCC-4.8.3-SERIAL_DEBUG_DEV_MueLu_ExtraTypes	69	0	227	0	117	0	3	94	7 hours ago	(25 labels)
enigma.sandia.gov	Linux-GCC-4.8.3-SERIAL_RELEASE_DEV_MueLu_Experimental	69	0	227	0	113	0	4	107	6 hours ago	(25 labels)

Trilinos automated testing

126

- Several Amesos2 (direct solver) tests are broken

SubProject Dependencies										
Project	Configure			Build			Test			Last submission
	Error	Warning	Pass	Error	Warning	Pass	Not Run	Fail	Pass	
Teuchos	0	22	22	0	13	9	0	0	227	2016-06-06 09:01:20
Epetra	0	22	22	0	13	9	0	1	244	2016-06-06 09:02:05
Triutils	0	22	22	0	0	21	0	0	2	2016-06-06 09:02:16
Tpetra	0	20	20	0	18	2	0	0	285	2016-06-06 08:10:13
EpetraExt	0	21	21	0	3	18	0	0	26	2016-06-06 08:11:16
ThreadPool	0	1	1	0	0	1				2016-06-06 02:51:44
Amesos	0	21	21	0	1	20	0	0	41	2016-06-06 08:16:59

- Are any of its dependencies broken?
 - ▣ Yes, there is a broken Epetra (basic linear algebra) test
 - ▣ Maybe this broke Amesos2?

Trilinos automated testing

127

❑ Which tests were broken in Amesos2?

Testing started on 2016-06-06 07:42:35

Site Name:enigma.sandia.gov

Build Name:Linux-GCC-4.8.3-SERIAL_DEBUG_DEV_MueLu_ExtraTypes

Total time:16s 840ms

OS Name:Linux

OS Platform:x86_64

OS Release:3.10.0-229.4.2.el7.x86_64

OS Version:#1 SMP Fri Apr 24 15:26:38 EDT 2015

Compiler Version:unknown

3 tests failed.

Name	Status	Time	Details	Labels	Summary
Amesos2_Epetra_RowMatrix_Adapter_UnitTests_MPI_4	Failed	1s 860ms	Completed (Failed)	Amesos2	Broken
Amesos2_Epetra_MultiVector_Adapter_UnitTests_MPI_4	Failed	1s 980ms	Completed (Failed)	Amesos2	Broken
Amesos2_Tpetra_CrsMatrix_Adapter_UnitTests_MPI_4	Failed	1s 900ms	Completed (Failed)	Amesos2	Broken

Trilinos automated testing

128

- If you may have broken something, you will get an email about it



CDash <trilinos-regression@sandia.gov>

4:05 AM (5 hours ago) ☆



to anasazi-regres. ▾

A submission to CDash for the project Trilinos has failing tests.
You have been identified as one of the authors who have checked in changes that are part of this submission or you are listed in the default contact list.

Details on the submission can be found at <http://testing.sandia.gov/cdash/buildSummary.php?buildid=2469557>

Project: Trilinos

SubProject: Anasazi

Site: artemis.srn.sandia.gov

Build Name: Linux-intel-15.0.2-MPI_RELEASE_DEV_DownStream_ETI_SERIAL-OFF_OPENMP-ON_PTHREAD-OFF_CUDA-OFF_COMPLEX-OFF

Build Time: 2016-06-06T03:59:42 MDT

Type: Nightly

Tests failing: 1

Tests failing

Anasazi_Epetra_MVOPTester_MPI_4 (<http://testing.sandia.gov/cdash/testDetails.php?test=33891492&build=2469557>)

New master/develop workflow

129

- Want master branch to remain stable
- All developer changes are now pushed to develop branch
- If changes are “okay”, merge develop to master
 - ▣ Currently a manual process for Trilinos framework team
 - ▣ If no new tests are failing on the dashboard, merge
 - ▣ Will eventually be automated

An important consideration: commits are so frequent, and the test suite is so large, it is impractical to run the test suite after each commit.

130

Refactoring

Testing needs during code refactor

Case study with FLASH

Considerations

131

- Know why you are refactoring
 - ▣ Is it necessary?
 - ▣ Where should the code be after refactoring?
- Know the scope of refactoring
 - ▣ How deep a change?
 - ▣ How much code will be affected?
- Know the type of refactor
 - ▣ Is the behavior expected to remain unchanged?
 - To what degree?

Verification methodology

132

- Map from here to there
 - Know bounds on acceptable behavior change
 - Know your error bounds
 - ▣ Bitwise reproduction of results unlikely after transition
 - Check for coverage provided by existing tests
 - Develop new tests where there are gaps
- Incorporate testing overheads into refactor cost estimates

New software vs legacy code

133

- ❑ Legacy code often has insufficient tests
 - ▣ First step in doing new, nontrivial development: add more tests
 - ▣ The issue: legacy code is not organized for unit tests

The key to working with legacy code is getting it to a place where it is possible to know that you are making changes *one at a time*.

- Michael Feathers, **Working Effectively with Legacy Code**

Challenges

134

Checking for coverage

- ❑ Legacy codes can have many gotchas
 - ▣ Dead code
 - ▣ Redundant branches
- ❑ Interactions between sections of the code may be unknown
- ❑ Can be difficult to differentiate between just bad code, or bad code for a good reason
 - ▣ Nested conditionals

Code coverage tools are of limited help

Mitigating challenges

135

- How to differentiate between “to be pruned” code and “to be kept but not covered” code?
 - ▣ If experts are around, they can help
 - ▣ Run the code in all useful configurations, tag unused code
 - Reduces the chance of useful code remaining uncovered

The goal: understand the code

The bad news: may not really be an option

Other options

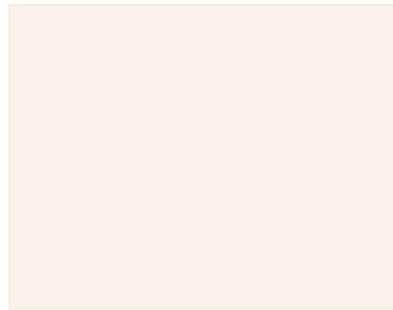
136

- Test coverings
 - ▣ Set of tests used to introduce an invariant
 - ▣ Cover a small area of the system
 - Unit tests might not be possible, given legacy code organization
 - ▣ Correct behavior is defined by what the code did yesterday, not an external standard of correctness
 - If the original legacy code was incorrect, that's a separate issue
 - ▣ Build the invariant, then refactor to make the code clear

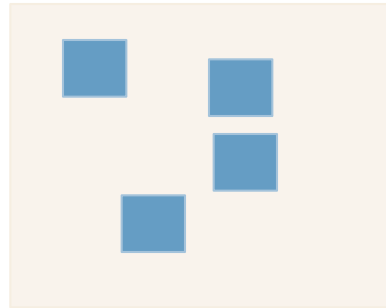
On ramp plan

137

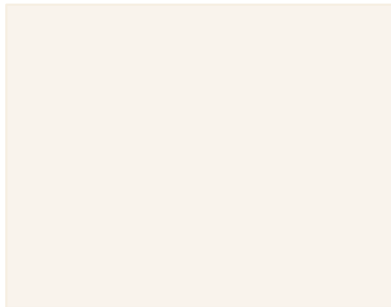
Proportionate to the scope



All at once



**May be
OK**



All at once



**Bad
idea**

On ramp plan 1

138

So how should it be done



Applicable when refactor is shallow
Individual components change
The backbone and global data structures do not

Methodology for plan 1

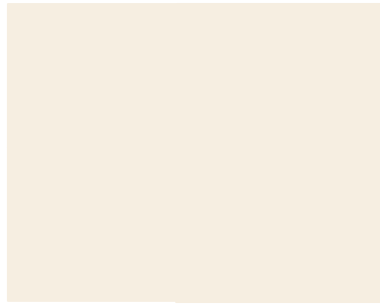
139

- Verify current code version test coverage
- No need to develop complete new testing regime
- Unit tests for isolatable components
- Higher granularity unit-like tests for separately developed code section
- Incorporate new tests into the suite during migration for each new component
 - ▣ May be eliminated later if needed

On ramp plan 2

140

So how should it be done



Methodology for plan 2

141

- ❑ Develop a comparison utility
- ❑ Understand the error-bars
- ❑ Backbone development treated as new development
- ❑ Migrated modules tested in the new infrastructure
- ❑ Unit tests may not need to migrate
- ❑ Applicable tests migrate to the new infrastructure
- ❑ New tests added if new features develop

Takeaway message – devise the methodology for refactoring and then plan a testing regime that meets the combined requirements of code verification within the refactor methodology

Case study : FLASH

142

- FLASH is a multiphysics multicomponent community code for
 - ▣ Astrophysics, cosmology, high energy density physics
- Also used by other communities
 - ▣ Solar physics, computation fluid dynamics, combustion
- Began with an intent to develop a single code usable for multiple applications
- 2+ codes - Prometheus, PARAMESH and other research codes smashed together into one code

Verifying version 1

143

Refactor from version 0-1

- Version created from legacy codes
 - ▣ refactoring challenges applied
- Objective – a more capable code
 - ▣ Moving to a more numerically complex meshing and previously unexplored behavior
 - ▣ Fused code underwent verification of numerical stability and convergence as though a new code
 - ▣ First set of tests used comparison against analytical solutions
- For all practical purposes a new code

Version 1

144

□ The Good

- ▣ Desire to use the same code for many different applications necessitated some thought to infrastructure and architecture
- ▣ Concept of alternative implementations, with a script for plugging different EOS – the setup tool
- ▣ Beginning of inheriting directory structure

□ The Bad

- ▣ F77 style of programming; Common blocks for data sharing
- ▣ Inconsistent data structures, divergent coding practices and no coding standards
- ▣ More capabilities needed but extensibility limited because of code design

Version 1

145

- And the ugly
 - ▣ Two camps
 - Camp 1 – do it right, think about design and then build
 - Camp 2 – do it right, enable science as soon as possible
 - ▣ For a while there were parallel efforts
 - The two camps did not communicate
 - ▣ The resources were not enough for parallel efforts
 - The science centric view won out
- Additional reason for code verification following the methods of a new code testing

Version 2

146

- Objective – make the code manageable and extensive
 - ▣ More physics solvers needed for simulations
 - ▣ Some even required new models and numerics
- Introduce uniformity in coding standards
 - ▣ Interfaces
 - ▣ Data inventory

Transition methodology

147

- Closer to on ramp plan 1
 - ▣ Though objectives were closer to plan 2
- Version 2 features embedded within version 1 code
- Complete backward compatibility, no need for new tests
 - ▣ Code tested by configuring with old version and new version and comparing output
- Developers heavily relied upon nightly testing to catch violations of interoperability

Because methodology did not match the objectives, the refactor had only partial success

Version 2 successes

148

- Addressed the worst of the bad in version 1
 - ▣ Eliminated common blocks
 - ▣ Data inventoried
 - ▣ Variable types classified them
- Enhanced “good”
 - ▣ Setup tool
 - ▣ Config files
 - ▣ Automate testing
- In summary code cleanup, but not extensibility
- Many new tests were added
 - ▣ Code coverage was significantly enhanced

Causes for partial success

149

- Keep the development and production branches synchronized
 - ▣ Enforced backward compatibility in the interfaces
 - ▣ Precluded needed deep changes
 - ▣ Hugely increased developer effort
 - ▣ High barrier to entry for a new developer
- Delayed adoption for production
 - ▣ Development continued in FLASH1.6, and so had to be brought simultaneously into FLASH2 too.

Motivation for another refactor

150

- Version 2 collected data into a central database
 - ▣ Navigating the source tree became more confusing and Config file dependencies became more verbose
 - ▣ No possibility of data scoping; every data item was equally accessible to every routine in the code
- When parsing a function, one could not tell the source of data
- Lateral dependencies were further hidden
- Overhead of database querying slowed the code by about 10-15%
- The queries caused huge amount of code replication and source files became ugly
- Encapsulation became nearly impossible

Version 3

151

- Overarching objective essentially the same as that for version 2
- Other specific objectives from lessons learned
 - ▣ Articulate data ownership in the architecture
 - Arbitrate on modifiability of data
 - ▣ Define component architecture
 - Encapsulation
- The institution of nightly testing with various granularities came in very handy

Version transition 2 to 3

152

- ❑ Controlled by the developers
- ❑ Sufficient time and resources made available to design and prototype
- ❑ No attempt at backward compatibility
- ❑ No attempt to keep development synchronized with production
- ❑ All focus on a forward looking modular, extensible and maintainable code

Version 3 achievements

153

- Kept inheriting directory structure, configuration and customization mechanisms from earlier versions
- Defined naming conventions
 - ▣ Differentiate between namespace and organizational directories
 - ▣ Differentiate between API and non-API functions in a unit
 - ▣ Prefixes indicating the source and scope of data items
- Formalized the unit architecture
 - ▣ Defined API for each unit Resolved data ownership and scope
- Resolved lateral dependencies for encapsulation
- Achieved extensibility

The methodology

154

- On ramp plan 2
 - ▣ Build the framework in isolation from the production code base
 - ▣ Infrastructure units first implemented with a homegrown Uniform Grid.
 - Helped define the API and data ownership
- Unit tests for infrastructure built (new code)
- Infrastructure thoroughly tested before adding physics components
- Test-suite was started on multiple platforms with various configurations (1/2/3D, UG/PARAMESH, HDF5/PnetCDF)
- This took about a year and a half, the framework was very well tested and robust by this time

The methodology

155

- Results could not be bitwise identical
 - ▣ Utility for comparing outputs of the two versions
 - ▣ Tolerances built into the utility to account for error-bars
- New tests needed for physics interaction with infrastructure
 - ▣ Some advancements in solvers created need for new tests
- Unit tests – verifying computed solution against analytical one
 - ▣ Or generating same values in two different ways
- The test-suite advanced simultaneously
 - ▣ Better methodology for verifying coverage
 - ▣ Policies and process

The methodology

156

- In the next stage the mature solvers (ones that were unlikely to have incremental changes) were transitioned to the code
 - ▣ Once a code unit became designated for FLASH3, no users could make a change to that unit in FLASH2 without consulting those doing the refactor.
- The next transition was the simplest production application (with minimal amount of physics)
- Scientists were in the loop for verification and in prioritizing the units to be transitioned at this stage

The outcome

157

- FLASH2 took more than 1.5 years before users transitioned to it
- FLASH3 was in production in the Center long before its official 3.0 release
 - ▣ The ugly had been addressed: the science centric view had given way to a more balanced one; took tremendous effort on the part of the center's leaders
 - ▣ More mutual trust and respect
 - ▣ More reliable code; unit tests provided more confidence, and it was easier to add capabilities

The outcome

158

- Transition was completed in 2006
- Until platform revolution no need for another deep change
- Code is fully extensible
 - ▣ Lagrangian framework imposed on existing framework
 - ▣ Many capabilities added with minimal changes to the backbone
- Code is very well tested
 - ▣ Testing on multiple platforms
 - ▣ By many users in diverse fields
- Bugs still prop up
 - ▣ Many fewer than earlier code versions

Outcomes

159

- A strong culture of verification and validation
- Propagates with alumni
- Provenance of obtained results
- Reproducibility possible within constraint of variations in platforms

Acknowledgments

160

- Argonne, a U.S. Department of Energy Office of Science laboratory, is operated under Contract No. DE-AC02-06CH11357.
- Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000. SAND NO. 2016-8466 C.

Other resources

161

Software testing levels and definitions:

http://www.tutorialspoint.com/software_testing/software_testing_levels.htm

Working Effectively with Legacy Code, Michael Feathers. The legacy software change algorithm described in this book is very straight-forward and powerful for anyone working on a code that has insufficient testing.

Code Complete, Steve McConnell. Excellent testing advice. His description of Structure Basis Testing is good, and it is a simple concept: Write one test for each logic path through your code.

Organization dedicated to software testing: <https://www.associationforsoftwaretesting.org/>

Software Carpentry: <http://katyhuff.github.io/python-testing/>

Tutorial from Udacity: <https://www.udacity.com/course/software-testing--cs258>

Papers on testing:

<http://www.sciencedirect.com/science/article/pii/S0950584914001232>

[https://www.researchgate.net/publication/](https://www.researchgate.net/publication/264697060_Ongoing_verification_of_a_multiphysics_community_code_FLASH)

[264697060_Ongoing_verification_of_a_multiphysics_community_code_FLASH](https://www.researchgate.net/publication/264697060_Ongoing_verification_of_a_multiphysics_community_code_FLASH)

Resources for Trilinos testing:

Trilinos testing policy: <https://github.com/trilinos/Trilinos/wiki/Trilinos-Testing-Policy>

Trilinos test harness: <https://github.com/trilinos/Trilinos/wiki/Policies--%7C-Testing>