better scientific software

Software Testing and Verification

Presented by

COLABS: Collaboration for Better Software for Science

In collaboration with

FASTMATH RAPIDS STEP SOFTWARE TOOLS ECOSYSTEM PROJECT

With prior support from

ECP EXASCALE COMPUTING PROJECT IDEAS productivity

Anshu Dubey (she/her)
Argonne National Laboratory

Better Scientific Software tutorial @ISC24

Contributors: Anshu Dubey (ANL), David E. Bernholdt (ORNL), Patricia Grubel (LANL), Rinku Gupta (ANL), Alicia Klinvex (SNL), Mark C. Miller (LLNL), Jared O'Neal (ANL), David M. Rogers (ORNL), Gregory R. Watson (ORNL)

CC BY

See slide 2 for license details

License, Citation and Acknowledgements

License and Citation

- This work is licensed under a Creative Commons Attribution 4.0 International License (CC BY 4.0).

- **The requested citation the overall tutorial is:** Anshu Dubey, Better Scientific Software tutorial, in ISC High Performance (ISC24), Hamburg, Germany, and online, 2024. DOI: 10.6084/m9.figshare.25686426.
- Individual modules may be cited as *Speaker, Module Title*, in *Tutorial Title*, …

Acknowledgements

- This work was supported by the U.S. Department of Energy Office of Science, Office of Advanced Scientific Computing Research (ASCR), and by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.
- This work was supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Next-Generation Scientific Software Technologies (NGSST) program.
- This work was performed in part at the Argonne National Laboratory, which is managed by UChicago Argonne, LLC for the U.S. Department of Energy under Contract No. DE-AC02-06CH11357.
- This work was performed in part at the Lawrence Livermore National Laboratory, which is managed by Lawrence Livermore National Security, LLC for the U.S. Department of Energy under Contract No. DE-AC52-07NA27344.
- This work was performed in part at the Los Alamos National Laboratory, which is managed by Triad National Security, LLC for the U.S. Department of Energy under Contract No.89233218CNA000001
- This work was performed in part at the Oak Ridge National Laboratory, which is managed by UT-Battelle, LLC for the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.
- This work was performed in part at Sandia National Laboratories. Sandia National Laboratories is a multi-mission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

What is Testing

Whenever you write a code you are doing it

- When you compile it, you are testing for defects in syntax
- When you run it for the first time you are testing for correctness
- When you add any code and run it again, you are testing it again
- When you break down your development into smaller chunks you test each chunk, then you combine the chunks, and you test again.

What is Testing

Whenever you write a code you are doing it

- When you compile it, you are testing for defects in syntax
- When you run it for the first time you are testing for correctness
- When you add any code and run it again, you are testing it again
- When you break down your development into smaller chunks you test each chunk, then you combine the chunks, and you test again.

Testing is an integral part of code development

What is Testing

Whenever you write a code you are doing it

- When you compile it, you are testing for defects in syntax
- When you run it for the first time you are testing for correctness
- When you add any code and run it again, you are testing it again
- When you break down your development into smaller chunks you test each chunk, then you combine the chunks, and you test again.

Testing is an integral part of code development

So, what is the whole fuss about testing?

What is Testing

Whenever you write a code you are doing it

- When you compile it, you are testing for defects in syntax
- When you run it for the first time you are testing for correctness
- When you add any code and run it again, you are testing it again
- When you break down your development into smaller chunks you test each chunk, then you combine the chunks, and you test again.

Testing is an integral part of code development

So, what is the whole fuss about testing?

Formalization of the process intimidates people because they think of writing tests as an overhead

How to Think About Building Tests

You start by thinking about what is the correct behavior

Next you think about how you are going to be able to tell whether the code is exhibiting correct behavior

How to Think About Building Tests

You start by thinking about what is the correct behavior

Next you think about how you are going to be able to tell whether the code is exhibiting correct behavior

You also think about what would be wrong behavior

Next you think about how you are going to be able to tell whether the code is exhibiting correct behavior

How to Think About Building Tests

You start by thinking about what is the correct behavior

Next you think about how you are going to be able to tell whether the code is exhibiting correct behavior

You also think about what would be wrong behavior

Next you think about how you are going to be able to tell whether the code is exhibiting correct behavior

Let us work through an example …

- You want a large prime number for encryption
- As a part of the development, you first write a function that checks if a given number is prime

Correct behavior: input 13 returns true, input 15 returns false
Incorrect behavior: input 15 returns true

How to Think About Building Tests

You start by thinking about what is the correct behavior

Next you think about how you are going to be able to tell whether the code is exhibiting correct behavior

You also think about what would be wrong behavior

Next you think about how you are going to be able to tell whether the code is exhibiting correct behavior

Let us work through an example …

- You want a large prime number for encryption
- As a part of the development, you first write a function that checks if a given number is prime

Correct behavior: input 13 returns true, input 15 returns false
Incorrect behavior: input 15 returns true

Here are all the ingredients for building a test !!

How to Think About Building Tests

You start by thinking about what is the correct behavior

Next you think about how you are going to be able to tell whether the code is exhibiting correct behavior

You also think about what would be wrong behavior

Next you think about how you are going to be able to tell whether the code is exhibiting correct behavior

Let us work through an example …

- You want a large prime number for encryption
- As a part of the development, you first write a function that checks if a given number is prime

Correct behavior: input 13 returns true, input 15 returns false
Incorrect behavior: input 15 returns true

Here are all the ingredients for building a test !!

- You write a “main” that reads in a number, calls the functions and prints true or false
- You can automate it by including a series of known primes and non-primes and their corresponding true or false values
- This is your “unit test” for the function

How to Think About Building Tests

Next you write a function to get to a large prime for encryption

Then you wish to confirm that it is a large enough prime

So, you write another unit test that counts the number of digits in the prime

How to Think About Building Tests

Next you write a function to get to a large prime for encryption

Then you wish to confirm that it is a large enough prime

So, you write another unit test that counts the number of digits in the prime

Finally, you want to verify that it meets your encryption needs

You integrate your new function with your encryption software

The encryption software is likely to have a way to verify that the cipher can only be translated with the right key

How to Think About Building Tests

Next you write a function to get to a large prime for encryption

Then you wish to confirm that it is a large enough prime

So, you write another unit test that counts the number of digits in the prime

Finally, you want to verify that it meets your encryption needs

You integrate your new function with your encryption software

The encryption software is likely to have a way to verify that the cipher can only be translated with the right key

- Now you have a more complex test that involves several correctly working components
- This is your “integration test”

Types of Tests

Well known tests for enterprise software

- Unit tests – verify a single function, extremely quick to run
- Integration tests – verify functions working together
- System tests – verify functionality of the entire software
- Acceptance tests – verify that the client needs are met
- Regression tests – verify that there is no degradation in code capabilities

Types of Tests

Additional types of tests needed for research software

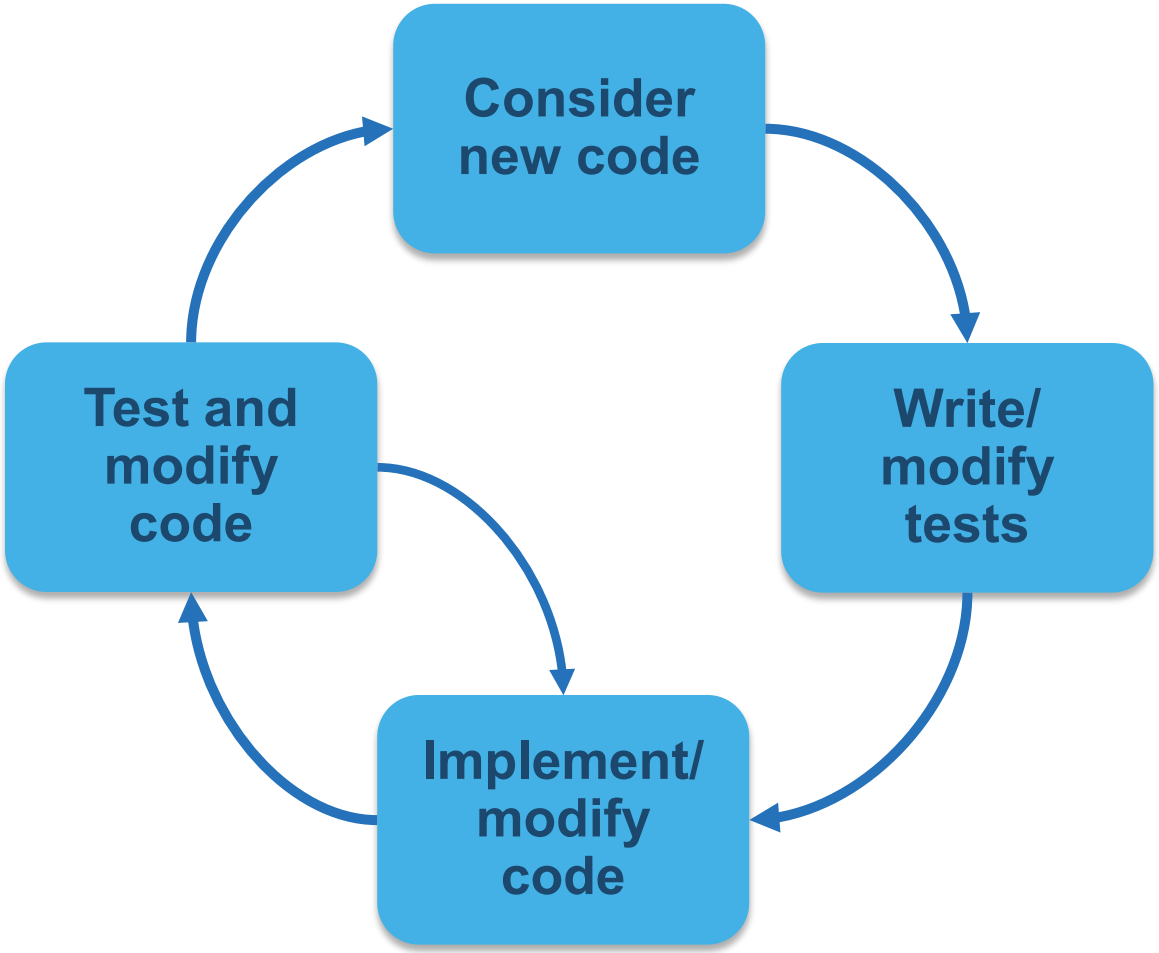
- Composite unit tests – are tests for specific functionalities and/or capabilities
- Granular tests – are integration tests at various granularities verifying correct behavior of interoperating functional units
- Restart tests – verify that a run can restart transparently from a checkpointed state
- Performance tests – apply to high-performance computing codes, verify that there is no performance loss

Classes of Tests

- White box testing – when you know the internals and can modify the code you are testing
  - Likely to be the code you and your collaborators are developing
  - You can insert assertions
  - You can insert code snippets that make testing easier

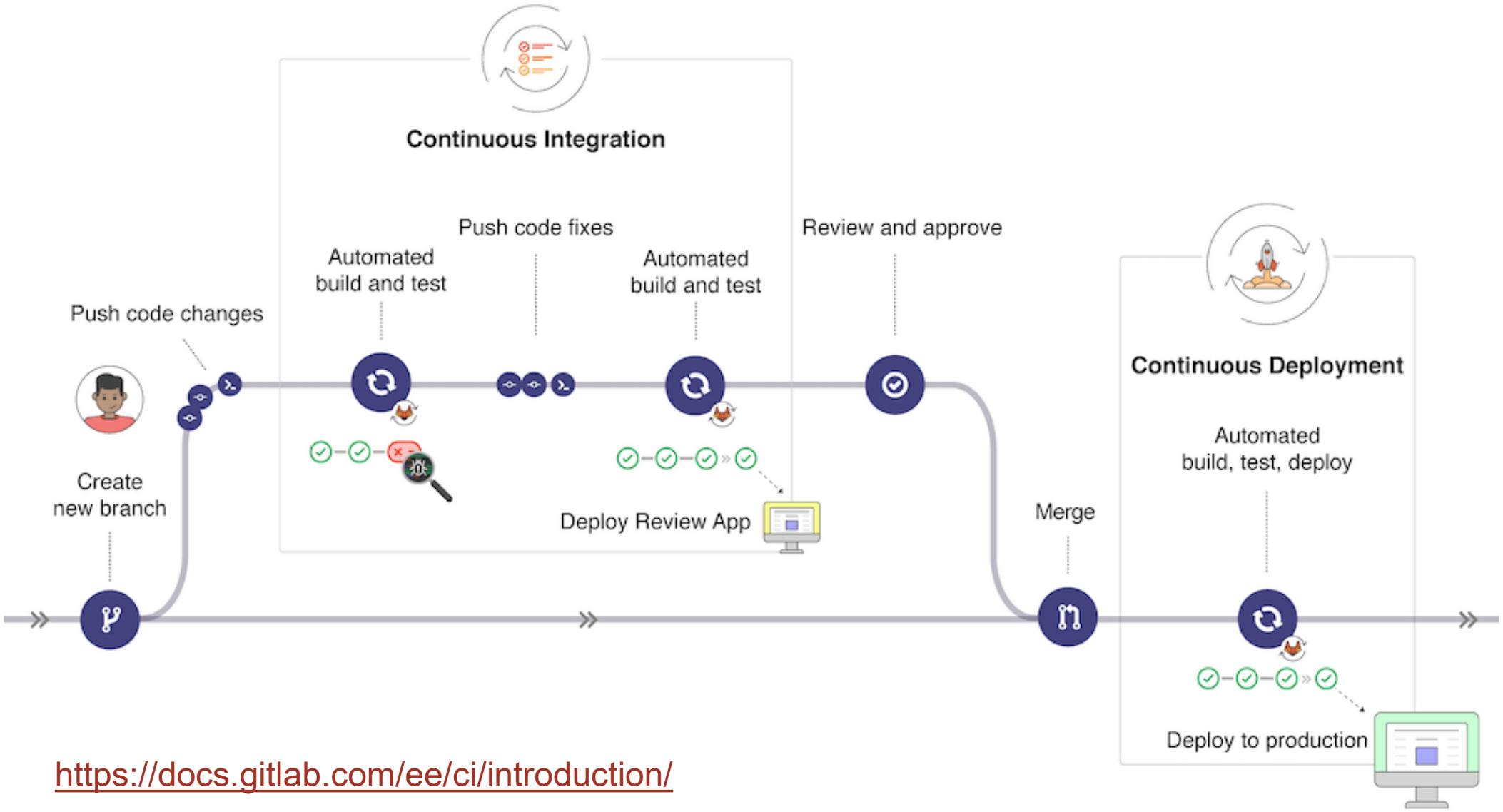
- Black box testing – when you do not know the internals of the code being tested, and cannot modify the code
  - Third party software or legacy code
  - The only means of verification available is reasoning about output to be obtained from supplied input

Test Driven Development



- Documented specifications and requirements of the code
- Ensures that thought is given to what it means for the program to be correct, rather than just what the program should do
- More efficient development cycle
- Much less debugging
- Requires:
  - Care in writing tests
  - Frequent running of tests
  - Wide adoption by development team

Continuous Integration (CI)



https://docs.gitlab.com/ee/ci/introduction/

CI Components

- Testing
  - Focused, critical functionality (infrastructure), fast, independent, orthogonal, complete, …
  - Likely to use a subset of full test-suite, or even develop new simpler tests
- Integration
  - Changes across key branches merged & tested to ensure the “whole” still works
    - Integration can take place at multiple levels
  - Develop, develop, develop, merge, merge, merge, test, test, test…NO!
  - Develop, test, merge, develop, test, merge, develop, test, merge…YES!
- Continuous
  - Changes tested every commit and/or pull-request (like auto-correct)
- CI generally implies a lot of automation

What is CI Good For

- The purpose of CI is to identify problems early
  - Prevent code that would “break the build” or adversely impact other developers being introduced
  - Need to provide sufficient confidence, but run quickly – balance varies by project

- CI should complement (not replace) more extensive automated testing
  - Use scheduled testing for more and more detailed tests, more configurations and platforms, performance testing, etc.

- CI for TDD is a natural fit
  - Writing tests before the code works well with CI

- Many options for where to execute CI tests
  - Free services are a good (easy) place to start
  - But may not be sufficient in the long run (especially large HPC/scientific codes)

- Start simple to get automation working, then build out what you need
  - Focus initially on key software configurations and aspects of the code to be tested
  - Make sure your testing expands to cover new code, use TDD

Building a Test-suite

Elements of test development

- For some tests assertions will suffice
- For others you will need to compare the output against baselines
  - Building a comparison utility is extremely useful
- Also useful to develop diagnostics – indirect ways of verifying behavior
  - Conservation of physical quantities
  - No non-physical values

Building a Test-suite

Elements of test development

- For some tests assertions will suffice
- For others you will need to compare the output against baselines
  - Building a comparison utility is extremely useful
- Also useful to develop diagnostics – indirect ways of verifying behavior
  - Conservation of physical quantities
  - No non-physical values

Building baselines for comparison

- From a known analytical solution
- Manufacture a solution
- Visualize and inspect output and anoint as baseline
- Run a test case upto point A and drop a checkpoint. Run another test case up to a later point B.
  - Use point A to restart and B as the anointed baseline

Building a Test-suite

Elements of test development

- For some tests assertions will suffice
- For others you will need to compare the output against baselines
  - Building a comparison utility is extremely useful
- Also useful to develop diagnostics – indirect ways of verifying behavior
  - Conservation of physical quantities
  - No non-physical values

Building baselines for comparison

- From a known analytical solution
- Manufacture a solution
- Visualize and inspect output and anoint as baseline
- Run a test case up to point A and drop a checkpoint. Run another test case up to a later point B.
  - Use point A to restart and B as the anointed baseline

Apply scaffolding for selection of tests … explained next

Example – Shock Hydrodynamics with Adaptive Mesh Refinement

Components needed

- Mesh
- Hydrodynamics solver
- Equation of state
- Parallelization

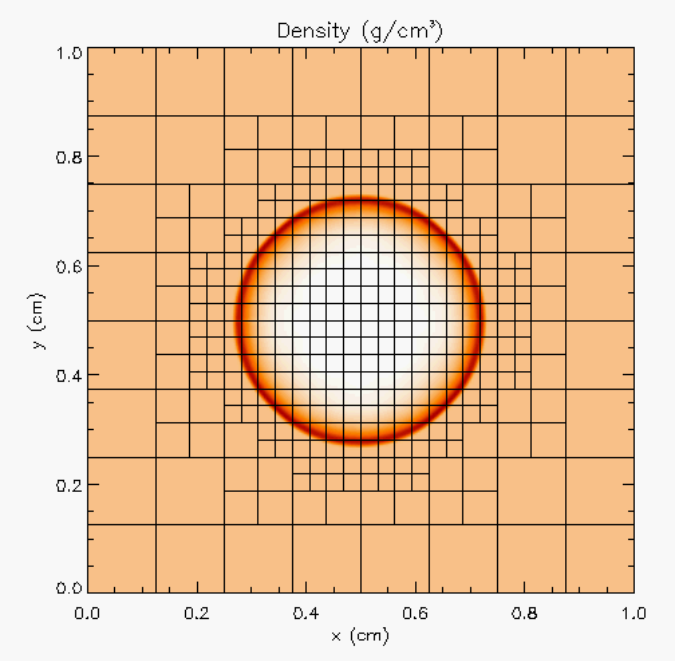
Strategy for development
Think of an application with analytical solution

Example – Shock Hydrodynamics with Adaptive Mesh Refinement

Components needed

- Mesh
- Hydrodynamics solver
- Equation of state
- Parallelization

Strategy for development
Think of an application with analytical solution



- Sedov blast wave
- High pressure at the center
- Shock moves out in a circle
- Analytical solution for how far the shock has travelled

Step 1 – Equation of State

- Initialize density and internal energy with known values
- Compute pressure and temperature using EOS
- Next use density and computed pressure as input and compute internal energy and temperature using EOS
- Compare computed values against initialized values

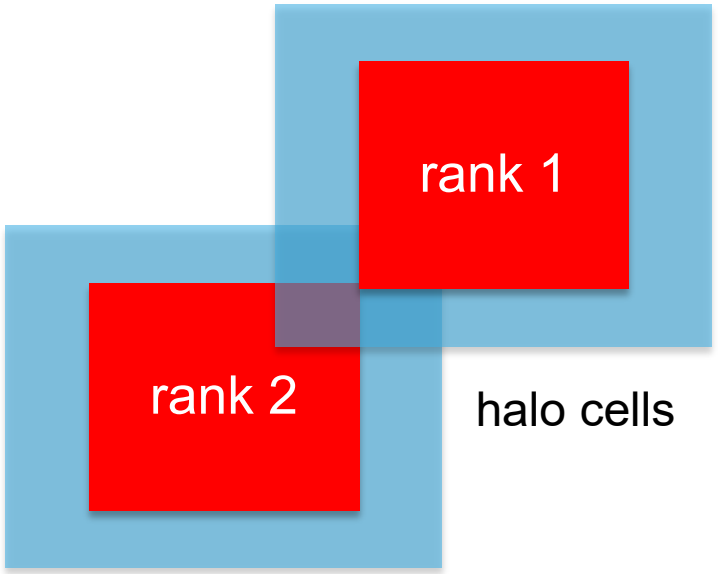
Step 1 – Equation of State

- Initialize density and internal energy with known values
- Compute pressure and temperature using EOS
- Next use density and computed pressure as input and compute internal energy and temperature using EOS
- Compare computed values against initialized values

We have a unit test

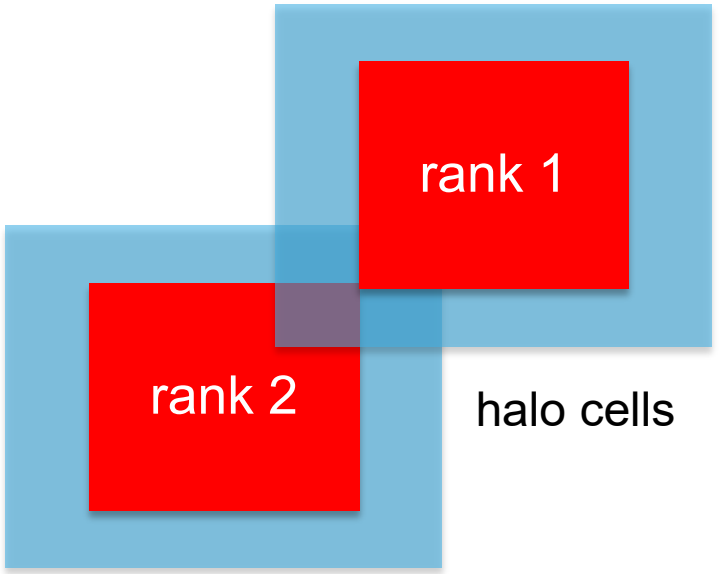
Step 2 – Mesh

- Start with uniform grid
- Domain decomposition for parallelization
  - Halo fill operation
- Initialize the interior (red) with a known function
- Apply halo fill
- Compute values for the halo using the known function
- Compare against filled values



Step 2 – Mesh

- Start with uniform grid
- Domain decomposition for parallelization
  - Halo fill operation
- Initialize the interior (red) with a known function
- Apply halo fill
- Compute values for the halo using the known function
- Compare against filled values



We have another unit test with manufactured solution

Step 3 – Hydrodynamics

- Apply initial conditions to the mesh
  - zeroes everywhere except at the center
- Write code for the analytical expression of the distance traveled by the shock
- Do time integration
- At time **T** compare evolved solution against analytical solution

If both mesh and EOS unit test pass, then any failure is in Hydrodynamics
This is a composite unit test

This is also the idea behind scaffolding

Step 4: AMR

- The same halo fill unit test for mesh also works for AMR
- Additional functionalities to test are:
  - Fine-coarse boundary resolution
  - Regridding
- Steps in testing
  - Run Sedov with UG
  - Run Sedov with AMR, but no dynamic refinement
    - If failed fault is in flux correction
  - Run Sedov with AMR and dynamic refinement
    - If failed fault is in regridding

Step 4: AMR

- The same halo fill unit test for mesh also works for AMR
- Additional functionalities to test are:
  - Fine-coarse boundary resolution
  - Regridding
- Steps in testing
  - Run Sedov with UG
  - Run Sedov with AMR, but no dynamic refinement
    - If failed fault is in flux correction
  - Run Sedov with AMR and dynamic refinement
    - If failed fault is in regridding

We have continued to build scaffolding and are using granular testing to pinpoint the cause of error

Step 4: AMR

- The same halo fill unit test for mesh also works for AMR
- Additional functionalities to test are:
  - Fine-coarse boundary resolution
  - Regridding
- Steps in testing
  - Run Sedov with UG
  - Run Sedov with AMR, but no dynamic refinement
    - If failed fault is in flux correction
  - Run Sedov with AMR and dynamic refinement
    - If failed fault is in regridding

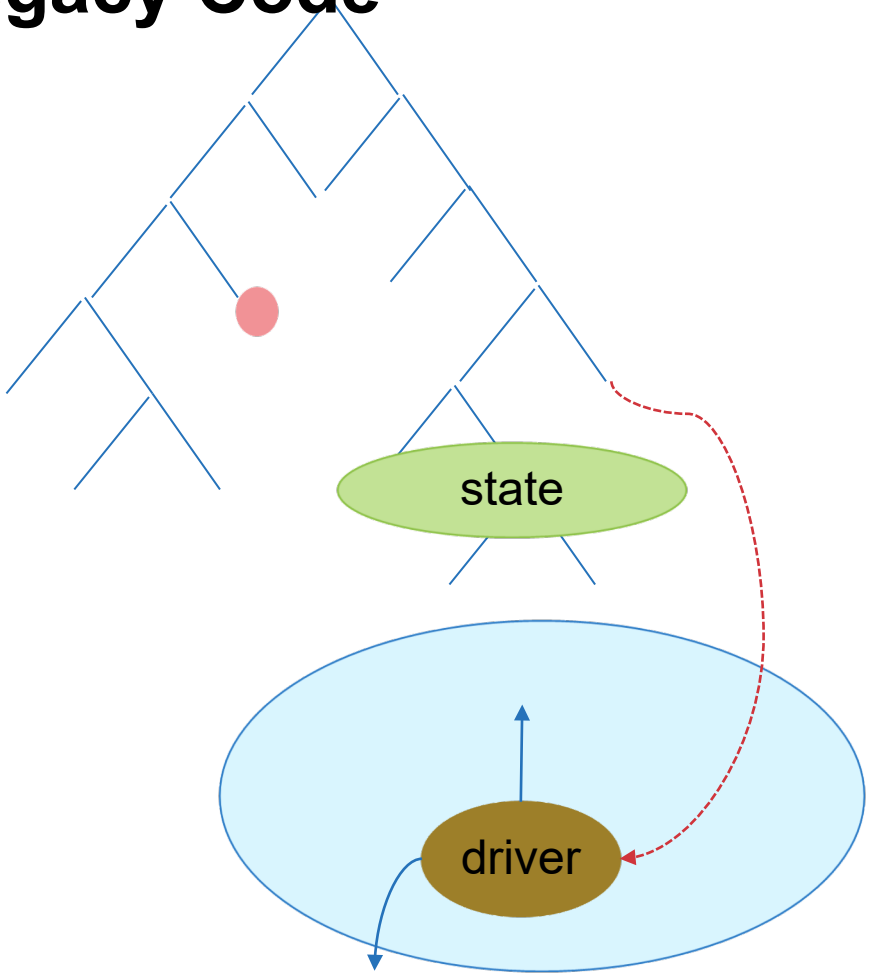
We have continued to build scaffolding and are using granular testing to pinpoint the cause of error

All of these are examples of white box testing

Mixed White/Black Box Testing For a Legacy Code

There may not be existing tests

- Isolate a small area of the code
- Dump a useful state snapshot
- Build a test driver
  - Start with only the files in the area
  - Link in dependencies
    - Copy if any customizations needed
- Read in the state snapshot
- Restart from the saved state
- Verify correctness
  - Always inject errors to verify that the test is working



How to build your test suite?

- 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
  - Restart tests
- Rules of thumb
  - Simple
  - Enable quick pin-pointing

Useful resources https://bssw.io/items?topic=testing

How do we determine what tests are needed?

Code coverage tools

- Expose parts of the code that aren't being tested
  - gcov - standard utility with the GNU compiler collection suite (we will use it in the next few slides)
  - Compile/link with –coverage & turn off optimization
  - Counts the number of times each statement is executed
  - Necessary for testing, but not sufficient
- gcov also works for C and Fortran
  - Other tools exist for other languages
  - JCov for Java
  - Coverage.py for python
  - Devel::Cover for perl
  - profile for MATLAB
- Lcov
  - a graphical front-end for gcov
  - available at https://github.com/linux-test-project/lcov
  - Codecov.io in CI module
- Hosted servers (e.g., coveralls, codecov)
- graphical visualization of results
- push results to server through continuous integration server

Building Test-suite

First line of defense – code coverage tools

- Code coverage tools necessary but not sufficient
- Do not give any information about interoperability

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

- Map your tests and examples – what do they do?
- Follow the order
  - All unit tests – including full module tests (e.g., CL)
  - Tests sensitive to perturbations (e.g., SV)
  - Most stringent tests for solvers (e.g., WD, PT)
  - Least complex test to cover remaining spots (**Aha!**)

Good Rules of Thumb

- Test your tests!
  - Make sure tests fail when they're supposed to!
- Add "regression tests"
  - Ensure that bugs aren't creeping in
- Test regularly
  - Critical when teams are adding code regularly
  - To identify and document where changes to the underlying platform change code behavior/results
- Automate regular testing
  - Inculcate the discipline of monitoring the outcome of regular testing
- Exercise third-party dependencies
- Physics/math-based strategies
  - Conserved quantities, symmetries, synthetic operators
  - Eliminate complete dependence on bitwise reproducibility

Summary

- A testing strategy is essential for producing reliable trustworthy software
  - Invest the time needed to thoroughly test your software at all levels
  - Use automation whenever possible
- Different challenges are associated with exploratory, legacy, and composable codes
  - Adapt your strategy to fit your situation.
  - Eventually you will want to be able to verify all components in a code release.
- Don’t get distracted by all the technologies out there – focus on exercising your code.
  - Scaffolding projects can help with mechanics.

Resources

- Oberkampf, W., & Roy, C. (2010). Verification and Validation in Scientific Computing. Cambridge: Cambridge University Press. doi:10.1017/CBO9780511760396
- Michael Feathers. 2004. Working Effectively with Legacy Code. Prentice Hall PTR, USA. ISBN: 9780131177055
- A Dubey, K Weide, D Lee, J Bachan, C Daley, S Olofin… - Ongoing Verification of a Multiphysics Community Code. Software: Practice and Experience, 2015 **https://doi.org/10.1002/spe.2220**