Testing Fortran Software with pFunit

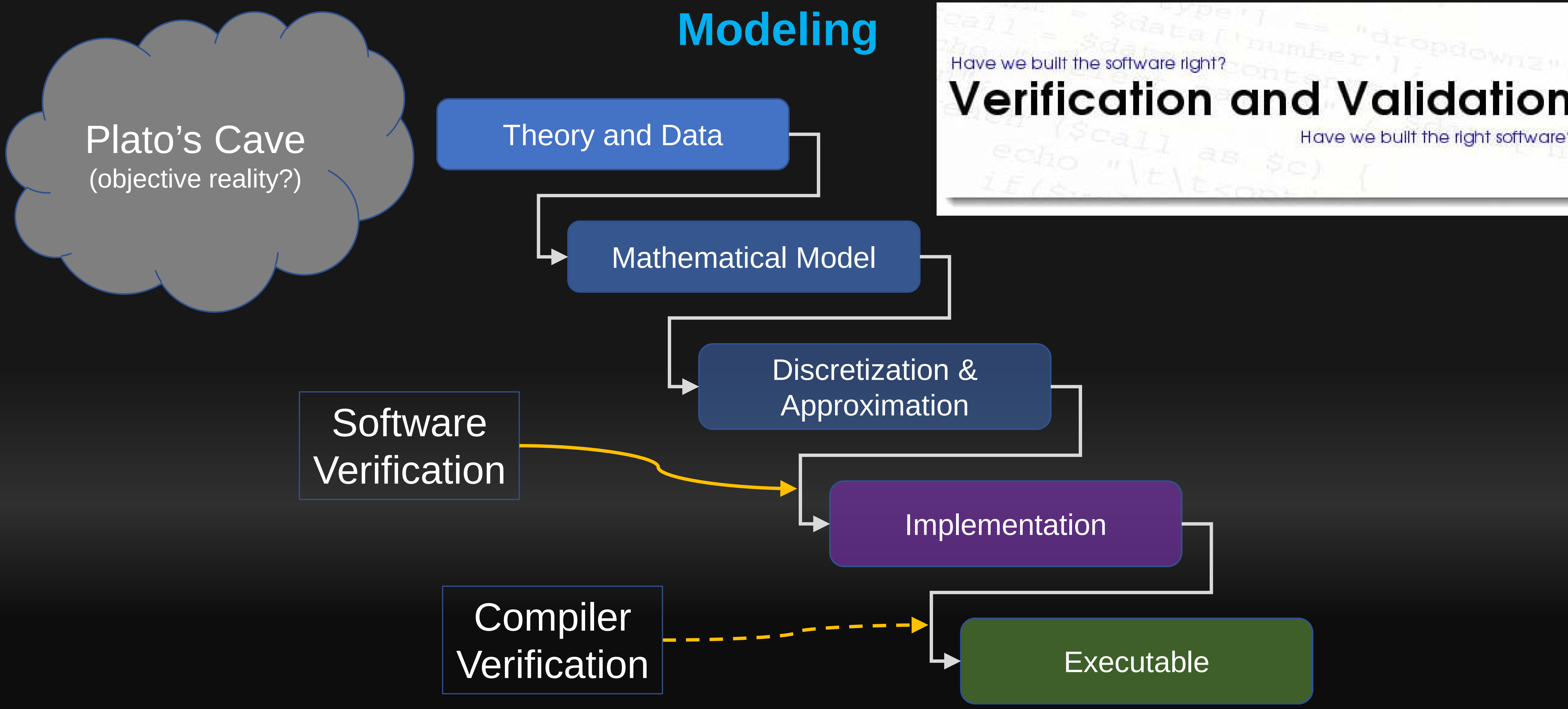
Tom Clune

April 10, 2019

Outline

- Unit Testing and Testing Frameworks
- Break for questions and discussion
- pFUnit – capabilities and examples
- Break for questions and discussion
- Obstacles to testing technical software and suggested remedies

Modeling


Have we built the software right?
Verification and Validation
Have we built the right software
Plato's Cave
(objective reality?)
Theory and Data
Mathematical Model
Discretization & Approximation
Software Verification
Implementation
Compiler Verification
Executable

Not all tests are created equal

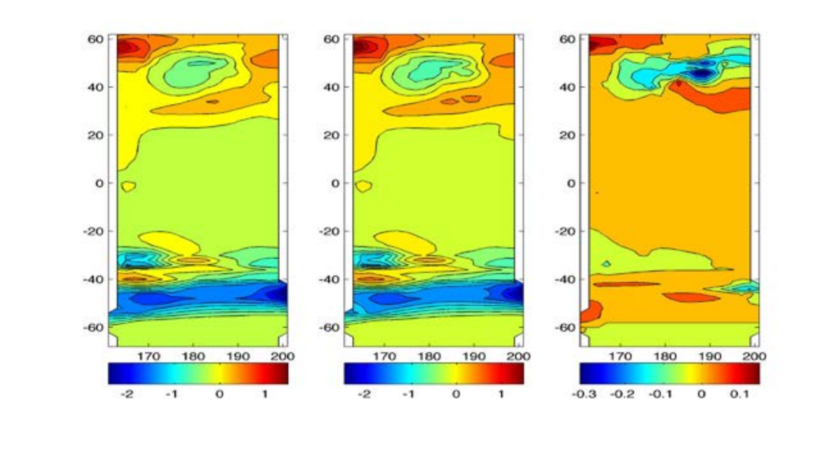
- Abort?:

```
if (x < 0.0) ERROR STOP "ILLEGAL VALUE FOR X"
```

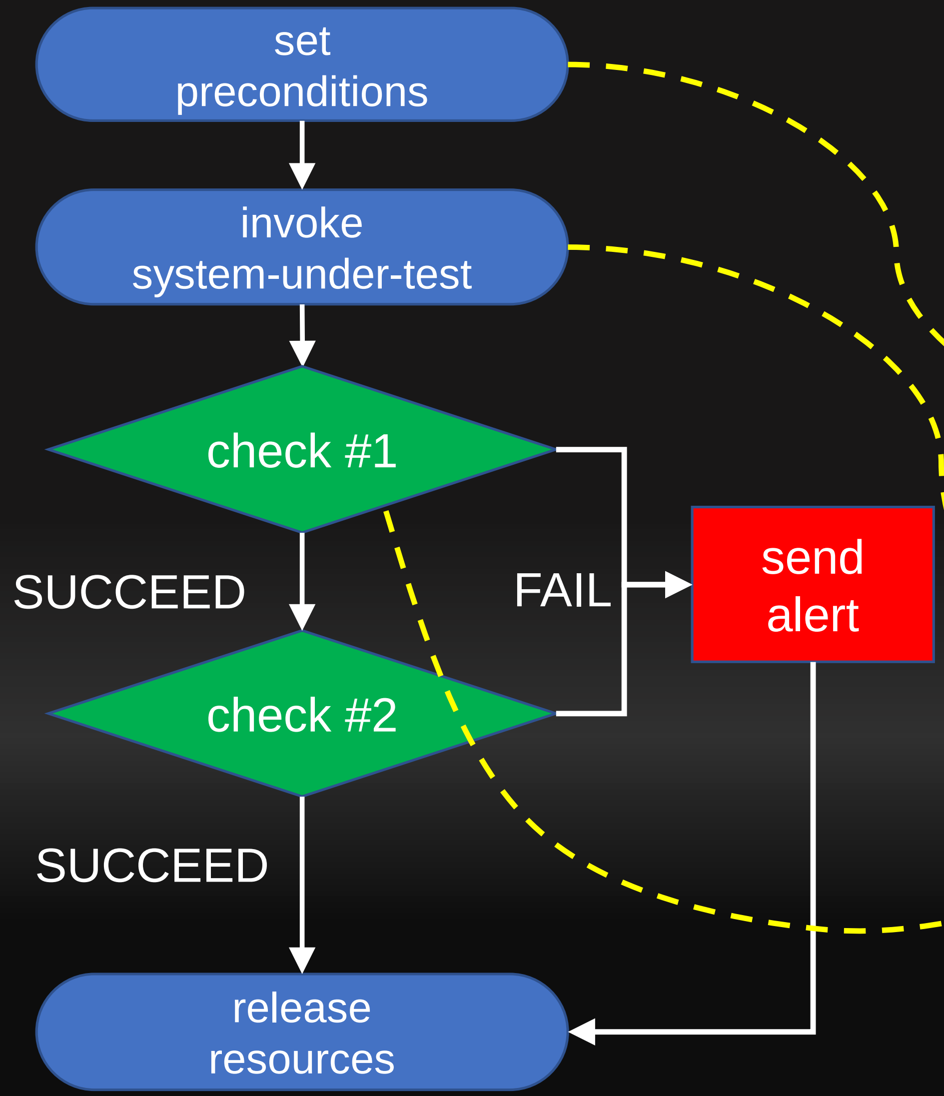
- Diagnostic print statement:

```
print*,  "loss of mass = " , deltaMass
```

- Visual inspection / acceptance threshold for regression:



Anatomy of a Unit Test



$$x(t) = x_0 + v_0(t-t_0) + \tfrac{1}{2}\, a\, (t-t_0)^2$$

```
x0 = 1.0
v0 = 2.0
 a = 4.0
t0 = 1.0
t1 = 2.0
```

```
x_t = compute_distance(x0, v0, a, t0, t1)
```

```
call assert_equal(5.0, x_t)
```

Attributes of Good Unit Tests

- ✓ Silent on success
- ✓ Automated and repeatable
- ✓ Independent (no side effects)
- ✓ Transparent (obvious, but not tautological)
- ✓ Narrow/precise
- ✓ Orthogonal (1 bug ==> 1 failing test)
- ✓ Small / frugal

And in aggregate we want the tests to cover our entire application.

Test Fixtures & Parameterized Tests

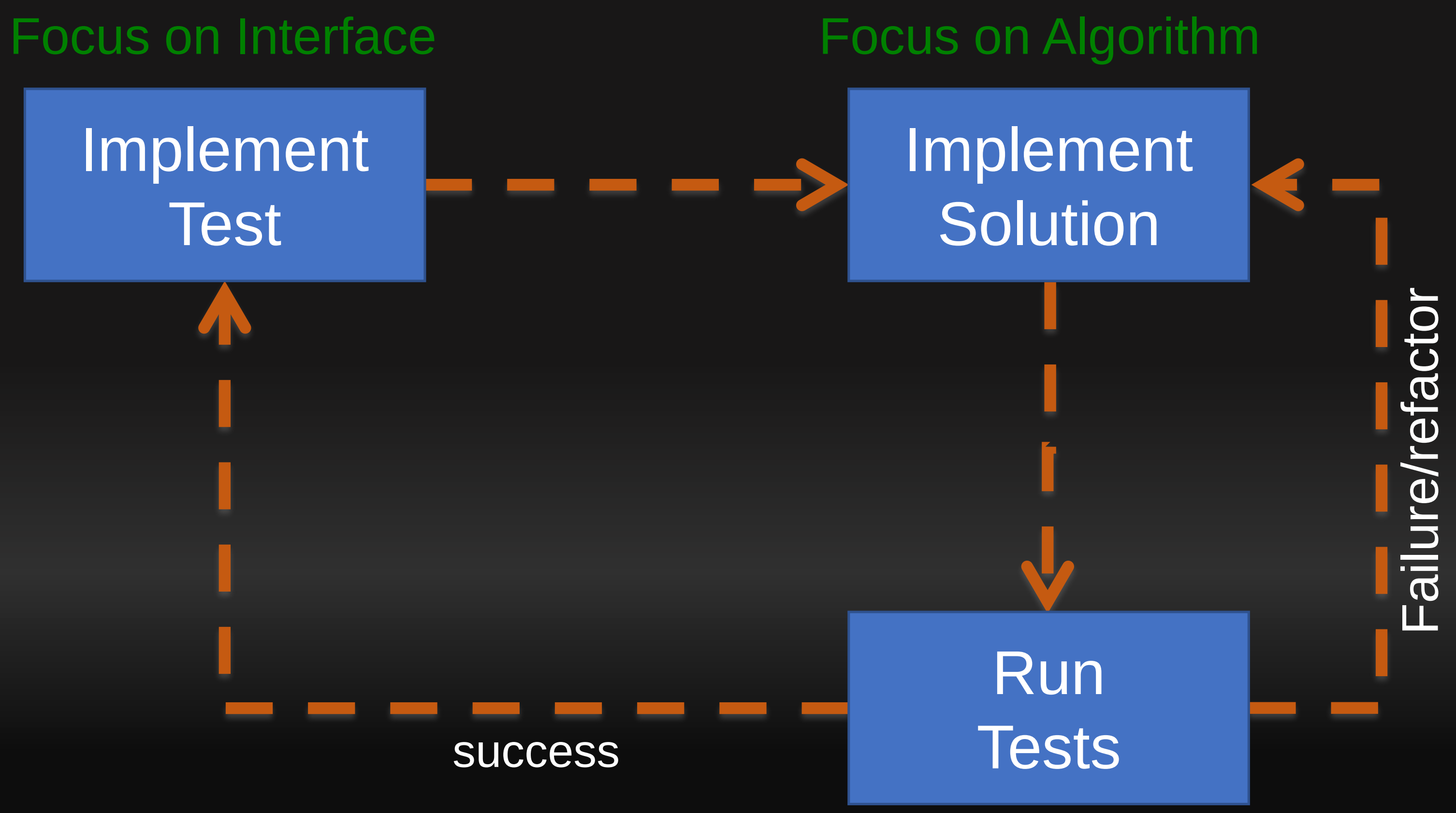
- Test fixture
  - Extracts complex initialization into separate setup procedure run before test itself
  - Ensures release of resources in teardown procedure
    - Even if test fails!
  - Esp. useful if many tests share similar data structures

- Parameterized test: run multiple times but with varying inputs and expected outputs.
  - Generally used in combination with a test fixture
  - Failure messages must identify which case(s) failed

Testing Frameworks

- Greatly simplify testing
  - Test creation
    - post conditions (asserts)
    - Fixtures: set up, tear down, repeat test with different parameters
    - aggregation (test suites)
  - Test execution
    - Summary
    - Failure locations (ftest/suite name, file, line number)
    - *Informative* failure messages
- Have driven major paradigm shifts in testing methodology
  - *Developers* write tests
  - Test driven development (TDD)

The TDD Cycle



- ❖ Very small incremental changes
- ❖ What is a minimal test that moves the design forward?
- ❖ What is the smallest change to make test pass?
- ❖ Rapid cycle << 10 minutes

TDD

- Perceived benefits
  - High test coverage
  - Software always “ready-to-ship”
  - Improved productivity (and lower stress)
  - Tests form a robust ***maintained*** form of documentation
  - Up front focus on interfaces leads to *better design.*
- Downside?
  - 2X-3X total lines of code (tough sell to management)
  - Refactoring is more difficult (but …)
- Challenges
  - Legacy code
  - Esp. procedural legacy code

“To me, legacy code is simply code without tests.”
— **Michael C. Feathers,**
Working Effectively with Legacy Code

Break for Questions

pFUnit

parallel Fortran Unit testing framework

pFUnit: Summary of Features

- Aimed at scientific software written in Fortran (and optionally MPI)
  - A bit of OpenMP as well (locking)
- Leverages Fortran 2003 object-oriented features
  - Very extensible
  - But … requires *very* recent compilers (ifort 18.03, gcc 8.2, NAG 6.2)
  - Developed with TDD
- Python base preprocessor used to simplify things that are hard/tedious in Fortran
  - Provides for expressive @ annotations (@assertEqual, @test …)
- Various command line options: (`--debug`, `–filter`, `--help`, …)

pFUnit: Assertions and Exceptions

- Vast library of numerical assertions
  - `@assertEqual`
    - real, complex (and integer, logical, character)
    - Kinds: default, double, REAL32, REAL64, REAL128
    - Absolute and relative tolerances (default tolerance of 0)
  - `@assertLessThan, @assertGreaterThan` (real)
  - Arbitrary ranks default build is max rank of 5)
    - L_1, L_2, L_∞ norms for arrays (real, complex)
  - `@assertIsNaN, @assertIsFinite`, …
- Exceptions implemented as a global stack (no true exceptions in Fortran)
  - Includes test name, source location, and description of failure
- Simple example: `@assertEqual(3.14159, 22./7, tolerance=1.e-5)`

pFUnit: Tests and Test Runners

Test declarations

- Simple `@test` annotation to indicate a subroutine is a test
- Fixture annotations:
  - `@before, @after,`
- Parameterized tests – advanced
  - Use by extending ParameterizedTestCase
  - Extension annotations: `@testCase, @testParameter`
- RobustRunner will attempt to run tests in a separate process
  - Can (theoretically*) handle hanging and crashing tests
  - Invoke on command line with "`-r robust`"
  - `Alternatively run with debugging "-d"`

pFUnit: MPI support

- MPI test (implemented as subclass of ParameterizedTestCase)
  - Runs a test on varying number of processes
    - Simple annotation extension – e.g., `test(npes=[1,3,7])` runs test 3 times.
    - Each instance gets new communicator with requested num. of pe's.
  - Provides simple type-bound functions to access
    - MPI Communicator (MPI_COMM_WORLD is a no-no)
    - # processes
    - MPI rank
- Exceptions and Asssertions
  - Exceptions on any process gathered and reported on root process
    - Failure description decorated with process and NPES
  - Be careful: failed assertions return immediately
    - Can lead to illegal MPI calls later in test if some processes continue
    - `@mpiAssert` – Blocking; ensures all processes exit if any process fails an assertion

Examples: Installation

1. Build and install pFUnit 4.0 (develop branch)

```
% git clone git://github.com/Goddard-Fortran-Ecosystem/pFUnit.git
% cd pFUnit
% git checkout develop
% mkdir build
% export PFUNIT_DIR=<prefix>
% cmake .. –DCMAKE_INSTALL_PREFIX=$PFUNIT
% make –j tests
% make install
```

2. Clone demos repository (source)

```
% git clone git://github.com/Goddard-Fortran-Ecosystem/pFUnit_demos.git
% cd pFUnit_demos
% …
```

Example: ./Trivial

- Just the minimal amount of code, test, build/run scripts
- Elements
  - square.F90 – the system under test
  - test_square.pf – a single unit test
  - CMakeLists.txt & Makefile
  - Driver scripts:
    - build_with_cmake_and_run.x
    - build_with_make_and_run.x

Trivial: ./Trivial (cont'd)

```
module Square_mod
contains

   pure real function square(x)
      real, intent(in) :: x
      square = x**2
   end function square
end module Square_mod

```

Square.F90

```
@test
subroutine test_square()
   use Square_mod
   use funit

   @assertEqual(9., square(3.), 'square(3)')

end subroutine test_square
```

test_square.pf

Example: ./Trivial (cont'd)

```
cmake_minimum_required(VERSION 3.12)

project (PFUNIT_DEMO_TRIVIAL
  VERSION 1.0.0
  LANGUAGES Fortran)

find_package(PFUNIT REQUIRED)
enable_testing()

# system under test
add_library (sut
  square.F90
  )
target_include_directories(sut PUBLIC ${CMAKE_CURRENT_BINARY_DIR})

# tests
set (test_srcs test_square.pf)
add_pfunit_ctest (my_tests
  TEST_SOURCES ${test_srcs}
  LINK_LIBRARIES sut
  )
```

CMakeLists.txt

Include pFUnit

Macro to build test

Example: ./Trivial (output)

One "." per test - to monitor progress

```
.
Time:              0.000 seconds

OK
(1 test)
```

Success/status

Example: ./Basic

Demonstrates a variety of basic pFUnit features and capabilities

- Source directory has 2 implementations: working and broken
  - Implement elemental square() function and integer factorial function
- Basic assertions: test_simple.pf
- See what failure messages look like: test_failing.pf
- Various mechanisms to skip tests: test_disable.pf
  1. `@disable` annotation – test not run, but tallied in summary
  2. `!@test` not mentioned at all
  3. `@test(#ifdef=foo) – test is run if -Dfoo`
  4. `@test(#ifndef=foo) – test is run if not -Dfoo`
- Very simple example using setup and teardown methods: test_simple_fixture.pf
- Testing source code error handling: test_error_handling.pf

pFUnit: output from failing tests (1 of 3)

```
bash-3.2$ ./broken_tests
.F.F.F
Time:         0.001 seconds

Failure
 in:
test_failing_suite.test_assert_true_and_false_fail
  Location:
[test_failing.pf:14]
intentionally failing test
  …
```

progress bar – 3 tests; 3 failed

Extra message supplied by developer

pFUnit: output from failing tests (2 of 3)

```
…
Failure
 in:
test_failing_suite.test_assert_equal_fail
  Location:
[test_failing.pf:26]
intentionally failing test
AssertEqual failure:
      Expected: <9.000000>
        Actual: <9.000988>
    Difference: <0.9880066E-03> (greater than tolerance of 0.1000000E-03)

Failure
 in:
test_failing_suite.test_fail_array
  Location:
[test_failing.pf:36]
intentionally failing test
ArrayAssertEqual failure:
      Expected: <4.000000>
        Actual: <4.410326>
    Difference: <0.4103265> (greater than tolerance of 0.1000000)
      at index: [2]
```

<suite_name>.<test_name>

[<file>:<line_number>]

Failure description

Index of first incorrect element

pFUnit: failing test output (3 of 3)

```
…
 FAILURES!!!
Tests run: 3, Failures: 3, Errors: 0
, Disabled: 0
ERROR STOP: *** Encountered 1 or more failures/errors during testing. ***
```

summary

pFUnit: disabled test output

```
bash-3.2$ ./disabled_test
.I..
Time:         0.000 seconds

 OK
 (3 tests, 1 disabled)
```

pFUnit: disabled test output

```
bash-3.2$ ./disabled_test  -d


 Start: <test_disable_suite.test_1_active>
.   end: <test_disable_suite.test_1_active>


 Disable: <test_disable_suite.test_2_disabled>
I

 Start: <test_disable_suite.test_3_active>
.   end: <test_disable_suite.test_3_active>


 Start: <test_disable_suite.test_if_not_foo_defined>
.   end: <test_disable_suite.test_if_not_foo_defined>

Time:         0.001 seconds

 OK
 (3 tests, 1 disabled)
```

debug option

Start/end progress

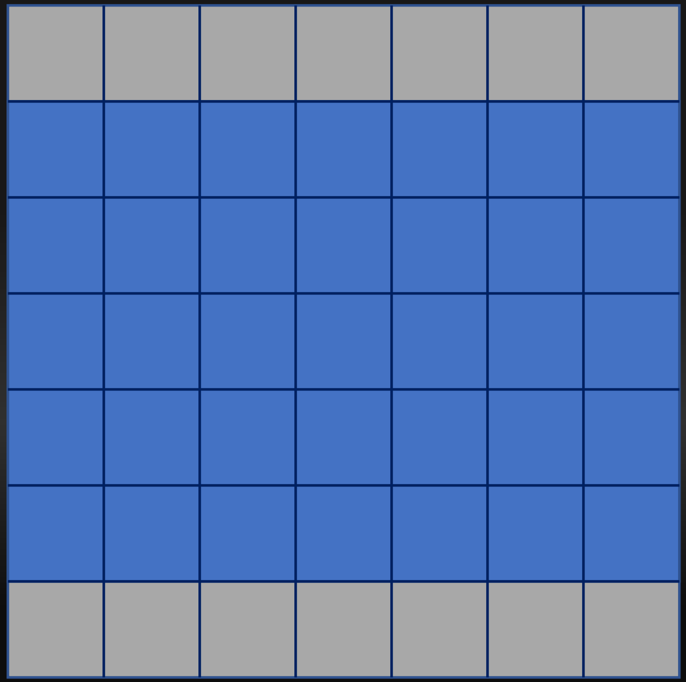
disabled in summary

Example: ./MPI

Demonstrates tests for MPI-based software:

- Tests: test_halo.pf
- Build: CMakeLists.txt
- Things we want to test
  1. Rank of neighbors
  2. Interior not changed
  3. Halo filled from neighbor values

Halo/guard cells on North



Halo/guard cells on South

2D arrays with
1D domain decomposition
Not periodic

Example: ./MPI (cont'd)

```fortran
   @test(npes=[1,2,3,4])
   subroutine test_fill_halo_interior(this)
      type (MpiTestMethod), intent(inout) :: this
      real :: array(NX_LOC,0:NY_LOC+1) ! local domain with halo region
      real :: interior_value

      ! Preconditions: Initialize interior and halos
      interior_value = this%getProcessRank()
      array(1:NX_LOC,1:NY_LOC) = interior_value
      array(1:NX_LOC,0) = HALO_UNDEF
      array(1:NX_LOC,NY_LOC + 1) = HALO_UNDEF

      ! Invoke SUT
      call fill_halo(array, this%getMpiCommunicator())

      ! check that interior values are unchanged
      @MPIassertEqual(interior_value, array(1:NX_LOC,1:NY_LOC))
   end subroutine test_fill_halo_interior
```

Example: ./MPI (cont’d)

```
   @test(npes=[1,2,3])
   subroutine test_fill_halo_south_pole(this)
      type (MpiTestMethod) :: this

      integer :: rank
      real :: array(NX_LOC,0:NY_LOC+1)
      real, parameter :: INTERIOR_VALUE = 1.

      ! Preconditions
      array(1:NX_LOC,0) = HALO_UNDEF
      array(1:NX_LOC,NY_LOC + 1) = HALO_UNDEF
      array(1:NX_LOC,1:NY_LOC) = INTERIOR_VALUE

      call fill_halo(array, this%getMpiCommunicator())

      rank = this%getProcessRank()
      if (rank == 0) then ! southern halo
         @assertEqual(HALO_UNDEF, array(1:NX_LOC,0))
      end if
   end subroutine test_fill_halo_south_pole
```

Example: ./MPI (cont’d)

```
   @test(npes=[1,2,3])
   subroutine test_fill_halo_south_other(this)
      type (MpiTestMethod) :: this

      integer :: rank
      real :: array(NX_LOC,0:NY_LOC+1)

      ! Preconditions
      array(1:NX_LOC,0) = HALO_UNDEF
      array(1:NX_LOC,NY_LOC + 1) = HALO_UNDEF
      array(1:NX_LOC,1:NY_LOC) = rank

      call fill_halo(array, this%getMpiCommunicator())

      rank = this%getProcessRank()
      if (rank > 0) then ! southern halo
         @assertEqual(rank - 1, array(1:NX_LOC,0))
      end if

   end subroutine test_fill_halo_south_other
```

Examples: ./MPI output

```
test 1
    Start 1: mpi_tests

1: Test command: /Users/tclune/installed/Compiler/nag-6.2_clang-
9.1/openmpi/3.1.2/bin/mpirun "--oversubscribe" "-np" "4" "mpi_tests"
1: Test timeout computed to be: 10000000
1: ..................
1: Time:         0.001 seconds
1:
1:  OK
1:  (18 tests)
1:  1/1 Test #1: mpi_tests .......................   Passed    0.13 sec

100% tests passed, 0 tests failed out of 1

Total Test time (real) =   0.14 sec
```

Each #pes is different test

What is new in pFUnit 4.0 (beta)

- Major cleanup of source code and build system
  - Single build for serial and MPI (and ESMF tests)
  - Very few compiler warnings, compiler #ifdef's …
- (Possibly) improved RobustRunner – for crashes and hangs
- Extensible annotations: @disable, @timeout(0.5), …
  - Users can add their own (funitproc needs some tweaks)
- Miscellaneous
  - Improved build macros (cmake and make) for creating executable tests
  - Support for Test Anything Protocol (TAP)
  - Support for testing Earth System Modeling Framework (ESMF) gridded components

New in 4.0 (cont'd)

- fHamcrest (Fortran version of hamcrest)
  - ***Composable*** system of "matchers" – leads to significantly improved extensibility
  - ***Self-describing*** – better error messages
  - Assertions *read almost like sentences*
  - Simple examples:

```
@assert_that(x, is(equal_to(5))
@assert_that([i,j,k], is_not(permutation_of([1,2,3]))
@assert_that(x, is(all_of([greater_than(0),less_than(5)]))
```

  - What about MPI?
    - Not in 4.0 due to a technical issue that needs to be resolved
    - But expect it to look something like:

```
@assert_that(x, on_process(5, comm, is(relatively_near(10.,0.1))))
@assert_that(x, on_all_processes(comm, is(equal_to(5)))
```

Summary

- pFUnit 4.0 (beta) has been released as 4/7/2019
  - Please try it out!

- Expected in 4.1
  - Coarray based tests with CAF_TestCase
    - Requires F2018 teams to be useful
  - Extending fHamcrest
    - Esp. pfHamcrest

References

- Junit: https://github.com/junit-team
- pFUnit: https://github.com/Goddard-Fortran-Ecosystem/pFUnit
- *Test-Driven Development: By Example,* Kent Beck
- *Working Effectively with Legacy Code*, Michael Feathers
- T. Clune, H. Finkel, and M. Rilee “Testing and Debugging Exascale Applications by Mocking MPI”, SE-HPCCSE, 2015.
- T. Clune and R. Rood, “Software Testing and Verification in Climate Model Development”, IEEE Software Volume 28 Issue 6, November 2011.

Questions?

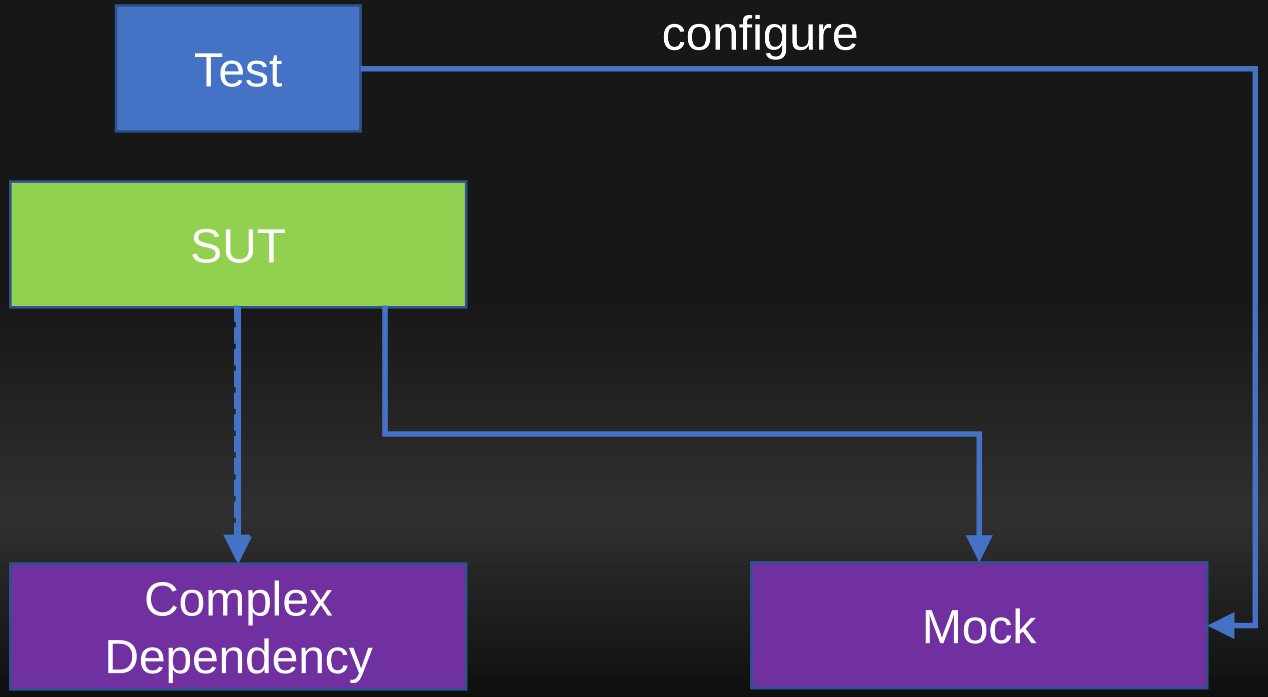
(Stick around for discussion about testing obstacles and mitigations.)

Testing challenges, misconceptions, and methodologies

- Many issues can complicate and even appear to prevent useful unit testing
  - Complexity
  - Floating-point (inexact) arithmetic
  - Distributed parallelism
  - Scalability – testing at petascale, exascale, and beyond

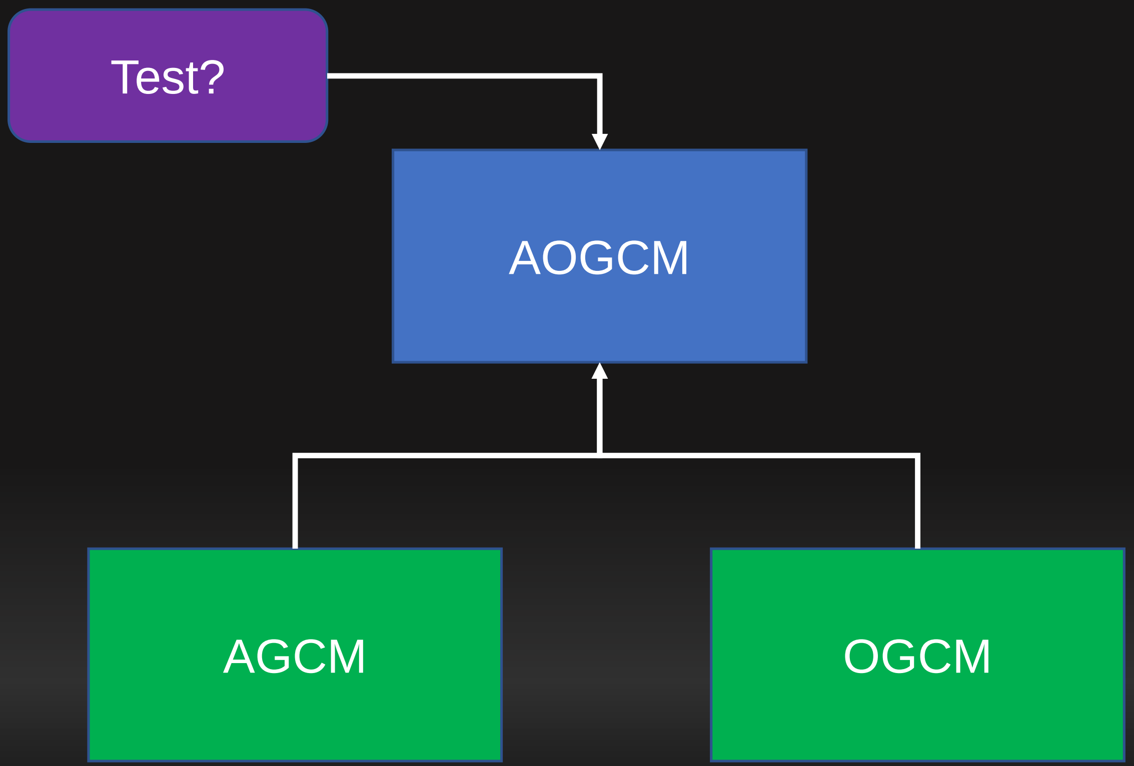
- Many/most of these can be addressed or mitigate by 2 complementary techniques:
  - Use very fine-grained units (subroutines, functions)
  - Use software “mocks” to sidestep complex dependencies.
  - What are mocks? Since you asked …

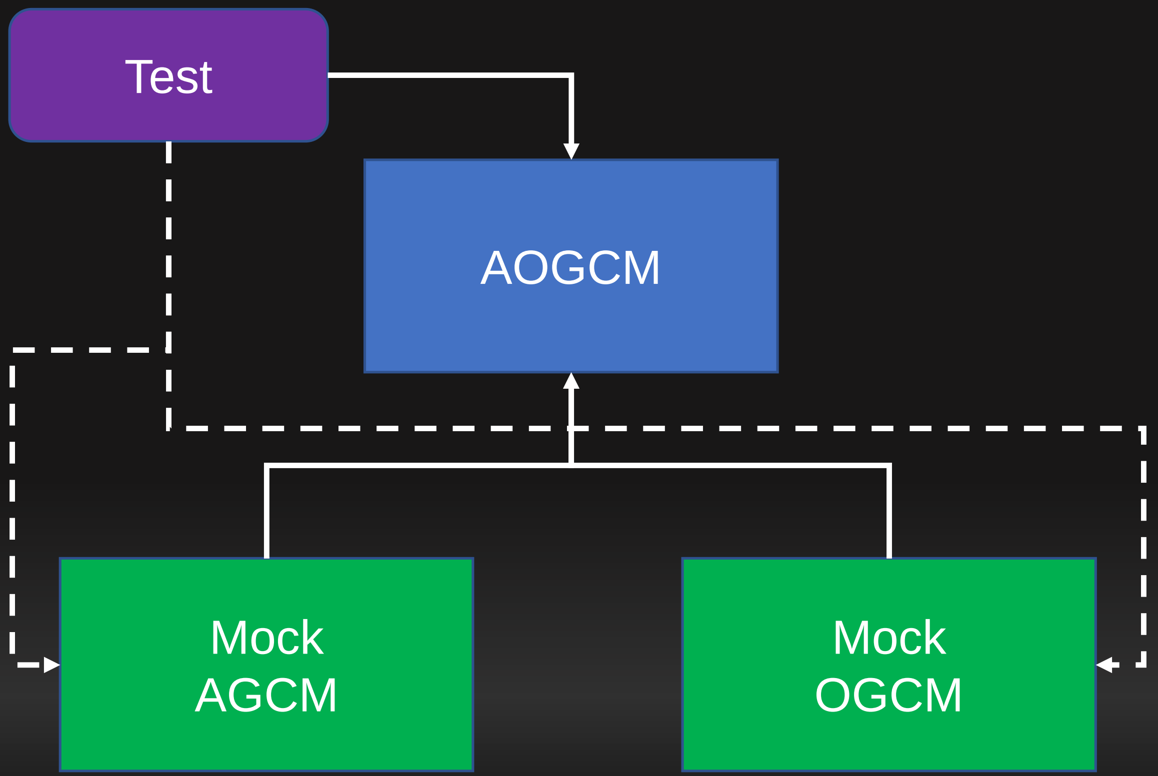
Software Mocks



Mock provides same interface but can be configured to verify inputs and produce preprogrammed outputs

Mock Example: Coupled Climate





Impossible to specify initial conditions with simple obvious outputs

Mock AGCM provides wind stress and expect surface Temp

Mock OGCM provides surface temp and expects wind stress

Challenge: Algorithmic Complexity

- Irreducible complexity?
  - E.g., test of climate model is as complex as climate model?
  - No - each software component is tested in isolation. Complexity is O(N).
  - Essential approach: software “mocks” for nontrivial dependencies

- Lack of analytic solutions?
  - Partial confusion of verification and validation
  - Problem is actually that the SUT is too large.
  - Mitigation
    - Split calculation into small units
    - Lowest levels are easily tested in isolation
    - Higher levels are tested with mocks (still coming back to that)
  - Mitigation of the mitigation – 2 implementations: fused and fine-grained

Challenge: Inexact arithmetic

- Assertions for FP results must generally specify a tolerance
- Estimating a reasonable tolerance is *problematic*
  - Too tight – correct implementation fails
  - Too loose – incorrect implementation succeeds
  - Even when good bounds estimate is available it is impractical
    - E.g. RK4 has error that is $O(h^5)$, but what is the leading coefficient?
    - And who has spare applied mathematicians lying around?
  - Temptation: increase tolerance until test passes (assumes SUT is already correct)

Challenge: Inexact arithmetic (cont'd)

- What gives rise to (nontrivial) roundoff?
  - Subtraction of nearly equal values
  - Iterated operations
  - …
- Mitigation 1: Use smart input values such that arithmetic is nearly exact
  - **You don't need to use physically realistic values to test an expression.**
  - Trivial example on next slide.
- Mitigation 2: Split complex expressions into nested pieces.
  - Test pieces separately with near-exact arithmetic
- Mitigation 3: Split test of iterated calculation
  1. Test individual iteration with smart input values
  2. Test that iteration iterates

Example: The Indiana Pi Bill (this really happened)

- Consider a test for a procedure that calculates the area of a circle:

```
@assertEqual(3.14159265, area(r=1.))
@assertEqual(12.56637060, area(r=2.))  ! Is this output obvious?
```

- Instead we create a helper function that takes pi as a parameter.

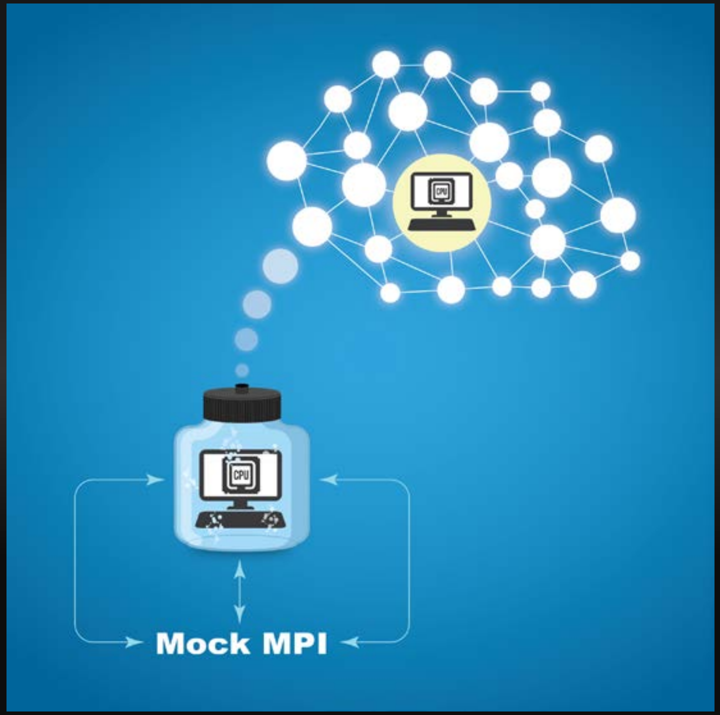
```
real function area_internal(pi, r)
   area_internal = pi*r**2
end function

real function area(r)
  use math_constants, only: pi
  area = area_internal(pi, r)
end real function
```

- Now we can test in a sensible manner:

```
@assertEqual(3, area_internal(pi=3., r=1.))
@assertEqua (12, area_internal(pi=3., r=2.)
@assertEqual(area_internal(pi=pi,r=2.), area(r=2.))
```

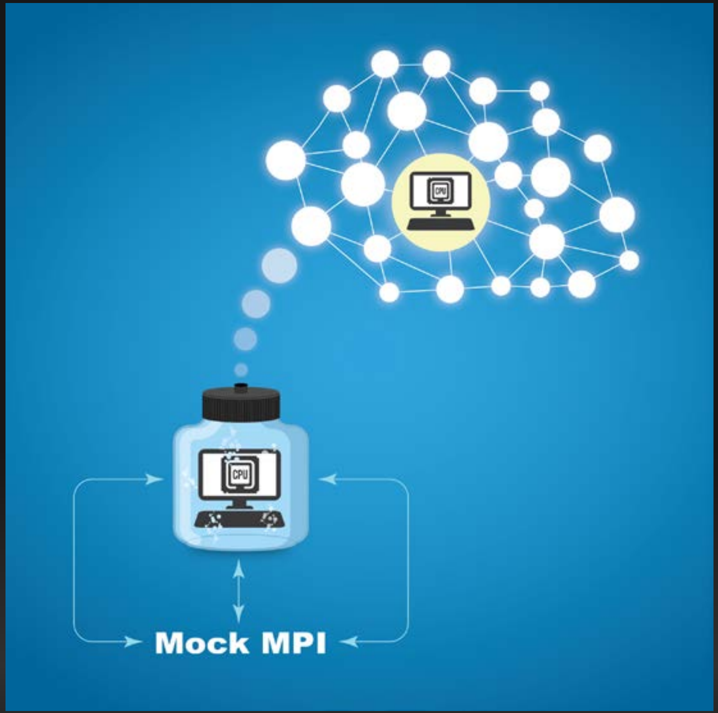
Challenge: Distributed parallelism

- Trivial issues: exercising on multiple processes, collecting exceptions, …
  - pFUnit – been there, done that.
- Real challenges: tests of functionality that may rely on timing
  - Race condition, deadlock, livelock, …
- Solution: Mock MPI (analog of "brain in a vat")
  - Serial software layer with same interfaces as MPI
  - Externally configurable to control MPI outputs
  - Single process of application "sees" a parallel env.
- Example: Testing mutex
  - Cases:
    - I request mutex, and no one else has it
    - I request mutex, but someone else has it
    - I release mutex, but must notify other waiter
    - I release mutex, and there is no other waiter



Challenge: Exascale

- Some defects are only apparent at extreme scale
  - Large number of processes
  - Large memory
- Debugging at extreme scale is expensive
  - Consumes expensive computing resources
  - Developer idle – waiting for queue
  - Delivery is delayed
- Once fixed, how do we ensure fix is preserved?
  - Routine testing too expensive



- Approach: use Mock MPI
  - Use Mock MPI to simulate the exascale environment experienced by a process or node.
  - Replicate issues on a workstation
  - Run ”exascale” regression tests on demand.

Thank you!

(Questions)