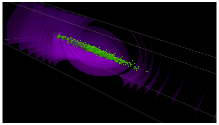
# Unit Testing
# Experience and Advice from the Real World

James Amundson

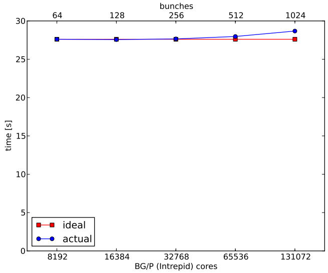
Fermilab

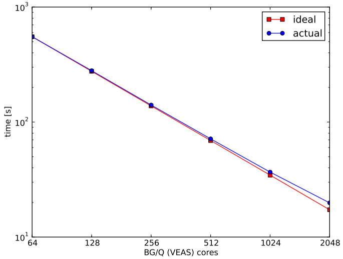
June 17, 2014

# Context: Synergia2.1



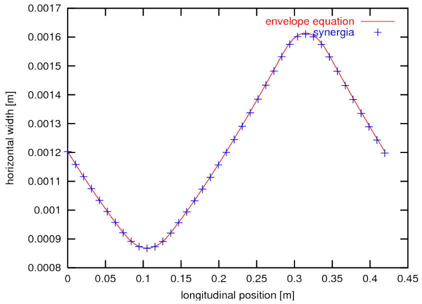
- Particle-in-cell (PIC) accelerator simulation framework
- Designed for supercomputers, usable on laptops
- Weak scaling (fixed problem size/core)
  - 99% efficiency on 65,536 cores
  - 96% efficiency on 131,072 cores
- Strong scaling (fixed problem size)
  - three orders of magnitude
- Roughly 40% of source is test



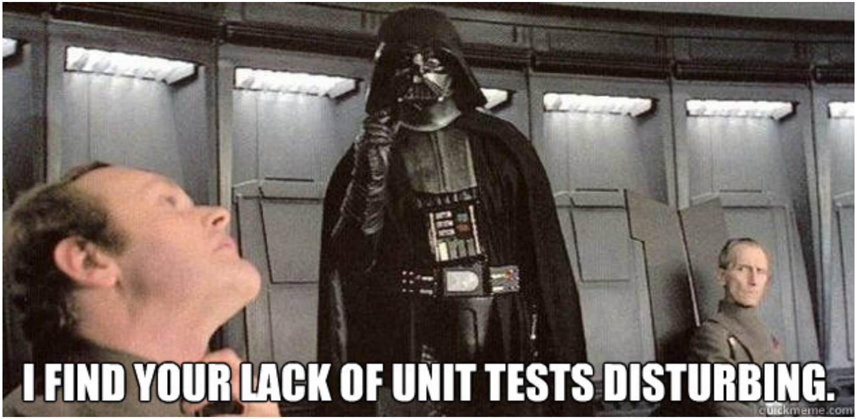


# A test

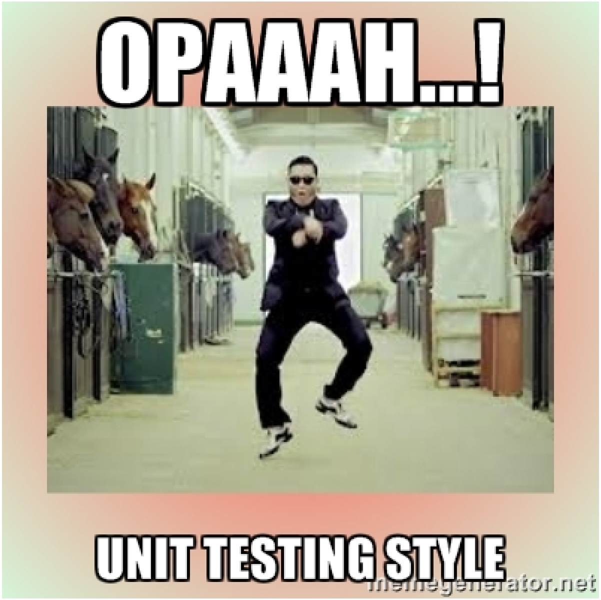
- Analytically calculable case
- Integration test
  - BAD integration test
    - requires human intervention
    - qualitative
    - limited coverage
    - unknown coverage
    - etc., etc.
- No hint as to where to look if test fails



## Darth Vader on Unit Testing



## Psy on Unit Testing



## Definition

A *unit test* is an automated piece of code that invokes a unit of work in the system and then checks a single assumption about the behavior of that unit of work. A unit of work is a single logical functional use case in the system that can be invoked by some public interface (in most cases).

*Definition of a Unit Test - The Art Of Unit Testing*
https://artofunittesting.com/definition-of-a-unit-test/

## Why Unit Test

Unit tests improve the efficiency of your development process in a number of ways:

- Find problems early
  - Write tests as you go along
  - You will never understand your code better than at the time you wrote it
- Get good test coverage
- Avoid the waste of disposable tests
- Provides a set of working examples
  - e.g., I have read the tests in order to understand some Boost interfaces
- Change code with confidence
- Increase the probability that your code produces correct answers

## Unit Testing Tools

- Source code level: testing frameworks
  - Support constructing tests, comparing output, collating results
  - Many packages exist
    - C++: I will focus on *Boost Test*
    - Python: *nose* has many similarities to *Boost Test*
- Build system level
  - CMake provides CTest
  - ART extends CMake testing support within its build system
    - cetbuildtools, will not describe here

## Synergia2.1: Four-vector Class

```
class Four_momentum {
public:
  /// Construct a Four_momentum in the rest frame
  /// @param mass in GeV/c^2
  Four_momentum(double mass);

  /// Set the total energy
  /// @param total_energy in GeV
  void set_total_energy(double total_energy);

  /// Set the kinetic energy
  /// @param kinetic_energy in GeV
  void set_kinetic_energy(double kinetic_energy);

  /// Get the relativistic gamma factor
  double get_gamma() const;

  /// Get the relativistic beta factor
  double get_beta() const;

  /// Check equality to the given tolerance
  /// @param four_momentum another Four_momentum
  /// @param tolerance fractional accuracy for beta and gamma
  bool equal(Four_momentum const &four_momentum, double tolerance) const;

  /// much removed...
};
```

## Basic Unit Tests

```
#define BOOST_TEST_MAIN
#include <boost/test/unit_test.hpp>

// <snip>

const double tolerance = 1.0e-15;
const double mass = 100.0;
const double my_gamma = 1.25; // 5/4
const double beta = 0.6;      // = sqrt(1-1/my_gamma^2) = 3/5
const double total_energy = 125.0;

BOOST_AUTO_TEST_CASE(construct) {
  Four_momentum four_momentum(mass);
}

BOOST_AUTO_TEST_CASE(get_mass) {
  Four_momentum four_momentum(mass);
  BOOST_CHECK_CLOSE(four_momentum.get_mass(), mass, tolerance);
}

BOOST_AUTO_TEST_CASE(get_total_energy) {
  Four_momentum four_momentum(mass);
  BOOST_CHECK_CLOSE(four_momentum.get_total_energy(), mass, tolerance);
}

BOOST_AUTO_TEST_CASE(set_and_get_total_energy) {
  Four_momentum four_momentum(mass);
  four_momentum.set_total_energy(total_energy);
  BOOST_CHECK_CLOSE(four_momentum.get_total_energy(), total_energy, tolerance);
}
```

## Testing Exceptional Conditions

```
BOOST_AUTO_TEST_CASE(set_beta_invalid) {
  Four_momentum four_momentum(mass);
  const double too_large = 4.0;
  bool caught_error = false;
  try {
    four_momentum.set_beta(too_large);
  }
  catch (std::range_error) {
    caught_error = true;
  }
  BOOST_CHECK(caught_error);
}

BOOST_AUTO_TEST_CASE(set_beta_invalid2) {
  Four_momentum four_momentum(mass);
  const double too_small = -0.5;
  bool caught_error = false;
  try {
    four_momentum.set_beta(too_small);
  }
  catch (std::range_error) {
    caught_error = true;
  }
  BOOST_CHECK(caught_error);
}
```

## Test Fixtures

```
struct Fodo_fixture {
  Fodo_fixture()
      : four_momentum(mass, total_energy),
        reference_particle(charge, four_momentum),
        lattice_sptr(new Lattice(name)) {
    BOOST_TEST_MESSAGE("setup fixture");
    // (code to populate lattice_sptr here...)
  }
  ~Fodo_fixture() { BOOST_TEST_MESSAGE("teardown fixture"); }

  Four_momentum four_momentum;
  Reference_particle reference_particle;
  Lattice_sptr lattice_sptr;
};

BOOST_FIXTURE_TEST_CASE(construct, Fodo_fixture) {
  Chef_lattice chef_lattice(lattice_sptr);
}

BOOST_FIXTURE_TEST_CASE(get_brho, Fodo_fixture) {
  Chef_lattice chef_lattice(lattice_sptr);

  double p = sqrt(total_energy * total_energy - mass * mass);
  double brho = p / PH_CNV_brho_to_p;
  BOOST_CHECK_CLOSE(chef_lattice.get_brho(), brho, tolerance);
}
```

## Global Test Fixtures

Every MPI program must have a main loop like this:

```
int main(int argc, char **argv)
{
    MPI_Init(&argc, &argv);
    do_something(argc, argv);
    MPI_Finalize();
    return 0;
}
```

Since Boost Test generates the main loop, something special must be done. Global fixtures provide the solution:

```
#include "synergia/utils/boost_test_mpi_fixture.h"
BOOST_GLOBAL_FIXTURE(MPI_fixture)
```

is equivalent. Of course, custom global fixtures can be created.

# Running a Boost Test with Details

```
|abacus5>export BOOST_TEST_LOG_LEVEL=all
|abacus5>./test_four_momentum
Running 22 test cases...
Entering test suite "Master_Test_Suite"
Entering test case "construct"
Test case construct did not check any assertions
Leaving test case "construct"
Entering test case "get_mass"
test_four_momentum.cc(33):
info: difference{%} between four_momentum.get_mass(){100}
and mass{100} doesn't exceed 1.0000000000000001e-15%
Leaving test case "get_mass"
Entering test case "get_total_energy"
test_four_momentum.cc(40):
info: difference{%} between four_momentum.get_total_energy(){100}
and mass{100} doesn't exceed 1.0000000000000001e-15%
Leaving test case "get_total_energy"
Entering test case "set_and_get_total_energy"
test_four_momentum.cc(48):
info: difference{%} between four_momentum.get_total_energy(){125}
and total_energy{125} doesn't exceed 1.0000000000000001e-15%
Leaving test case "set_and_get_total_energy"
```

## Python Tests using Nose

Nose provides a Python testing framework with capabilities very similar to Boost Test.

```
#!/usr/bin/env python
from foundation import Four_momentum
from nose.tools import *

mass = 3.0
total_energy = 27.4
value = 3.1415
beta_value = 0.678

def test_construct():
    f = Four_momentum(mass)

def test_construct2():
    f = Four_momentum(mass,total_energy)

def test_get_mass():
    f = Four_momentum(mass)
    assert_almost_equal(f.get_mass(),mass)

def test_set_get_total_energy():
    f = Four_momentum(mass)
    f.set_total_energy(value)
    assert_almost_equal(f.get_total_energy(),value)
```

# Running the tests

```
|abacus5>make test
Running tests...
Test project /home/amundson/work/synergia2-checkpoint-rebase4/build/synergia2/src/synergia
    Start 1: test_four_momentum
1/8 Test #1: test_four_momentum ...............   Passed    0.06 sec
    Start 2: test_four_momentum_py
2/8 Test #2: test_four_momentum_py ............   Passed    0.27 sec
    Start 3: test_physical_constants
3/8 Test #3: test_physical_constants ..........   Passed    0.03 sec
    Start 4: test_reference_particle
4/8 Test #4: test_reference_particle ..........   Passed    0.04 sec
    Start 5: test_reference_particle_py
5/8 Test #5: test_reference_particle_py .......   Passed    0.28 sec
    Start 6: test_distribution
6/8 Test #6: test_distribution ................   Passed    0.05 sec
    Start 7: test_distribution_py
7/8 Test #7: test_distribution_py .............   Passed    0.28 sec
    Start 8: test_diagnostics_write_helper
8/8 Test #8: test_diagnostics_write_helper ....   Passed    0.04 sec

100% tests passed, 0 tests failed out of 8

Total Test time (real) =   1.07 sec
|abacus5>
```

## Pitfalls

- The perfect is the enemy of the good
  - Some tests are better than no tests
  - Implemented imperfect tests are better than unimplemented perfect tests
- Writing working tests can feel like a waste of time.
  - You should never feel bad about writing working code, no matter how simple.
- “My code can’t be unit tested.”
  - Bad design
  - Unavoidable reasons
  - You have to decide for yourself
- “But... I wrote all these tests and I still have a bug!”
  - Life is like that
  - Your bugs will be easier to fix in well unit-tested code.