PECOS

Predictive Engineering and Computational Sciences

Tools and Guidelines for Software Development

Christopher S. Simmons

Institute for Computational Engineering and Sciences
The University of Texas at Austin

July 8, 2012

Outline

Prediction

- Predicting the behavior of the physical world is central to both science and engineering
- Advances in computer simulation has led to prediction of more complicated physical phenomena
- The complexity of recent simulations . . .
  - Makes reliability difficult to assess
  - Increases the danger of drawing false conclusions from inaccurate predictions

"Prediction is very difficult, especially if it's about the future." N. Bohr

Predictive Science Must Advance

Douglass Post (*Computational Science Demands a New Paradigm*, Physics Today, Jan. 2005):

" The prediction challenge is now the most serious limiting factor for computational science. . . New methods of verifying and validating complex codes are mandatory if computational science is to fulfill its promise for science and society."

Dimitri Kusnezov (ASC 2007 PI Meeting)

". . . in the area of codes, we are not anywhere near achieving predictive capability yet. We are pioneering computational science at the most remarkable scales, and together we are learning how to define notions of predictivity through verification, validation and uncertainty quantification."

The Multiple Roles of a Computational Scientist

Funding agencies typically fund SCIENCE not SOFTWARE therefore

Computational Scientists are expected to be:

- Physicist – addressed in curriculum
- Numerical Analyst – address in curriculum
- Software Developer? – seriously?

Software Engineering (and creating . . .) is vital to the work we do.

- understandable code
- well-documented code
- verifiable code

Software Engineering is not part of the formal curriculum thus most scientific software developed in academia is unverified. Furthermore, the software developed is unable to ever be verified without significant time investment.

Version Control

Minimum Guidelines – Actually using version control is the first step

Ideal Usage

- Put everything under version control
- Consider putting parts of your home directory under VC
- Use a consistent project structure and naming convention
- Commit often and in logical chunks
- Write meaningful commit messages
- Do all file operations in the VCS
- Set up change notifications if working with multiple people

Common Version Control Systems

- CVS
- Subversion (SVN)
- Git

Build Systems

Minimum Guidelines

- Any build system is a good start but it's more than an alias
- Bash script
- Makefile
- Host specific Makefiles

A proper build system

- Cross platform
- Supports dependencies
- Supports finding include files and libraries
- Supports parallel builds

Common build tools are Make, autoconf/automake, cmake, SCons

Scientific vs. Other Computing

- Scientific Computing
  - has to be right (or at least quantifiably wrong)
  - has to be fast
  - doesn't have to be easy to use
  - doesn't have to be pretty
- Other Types of Computing
  - errors are tolerable
  - speed is negotiable
  - often has to be pretty or easy to use or both

Trust No One . . . not even yourself

Unit Testing

- Tests individual units of source code
- A Unit is the smallest testable part of a code
- Each Unit test should be independent from others
- Best to write the Unit and the Unit test at the same time

Regression Testing

- Examples applications, unit tests, benchmark problems
- Catches unintended consequences of revisions
- Design of a suite of regression tests is an art

- Consider Code and Feature Coverage
- Test early and Test often
- Consider automating tests

Types of Documentation

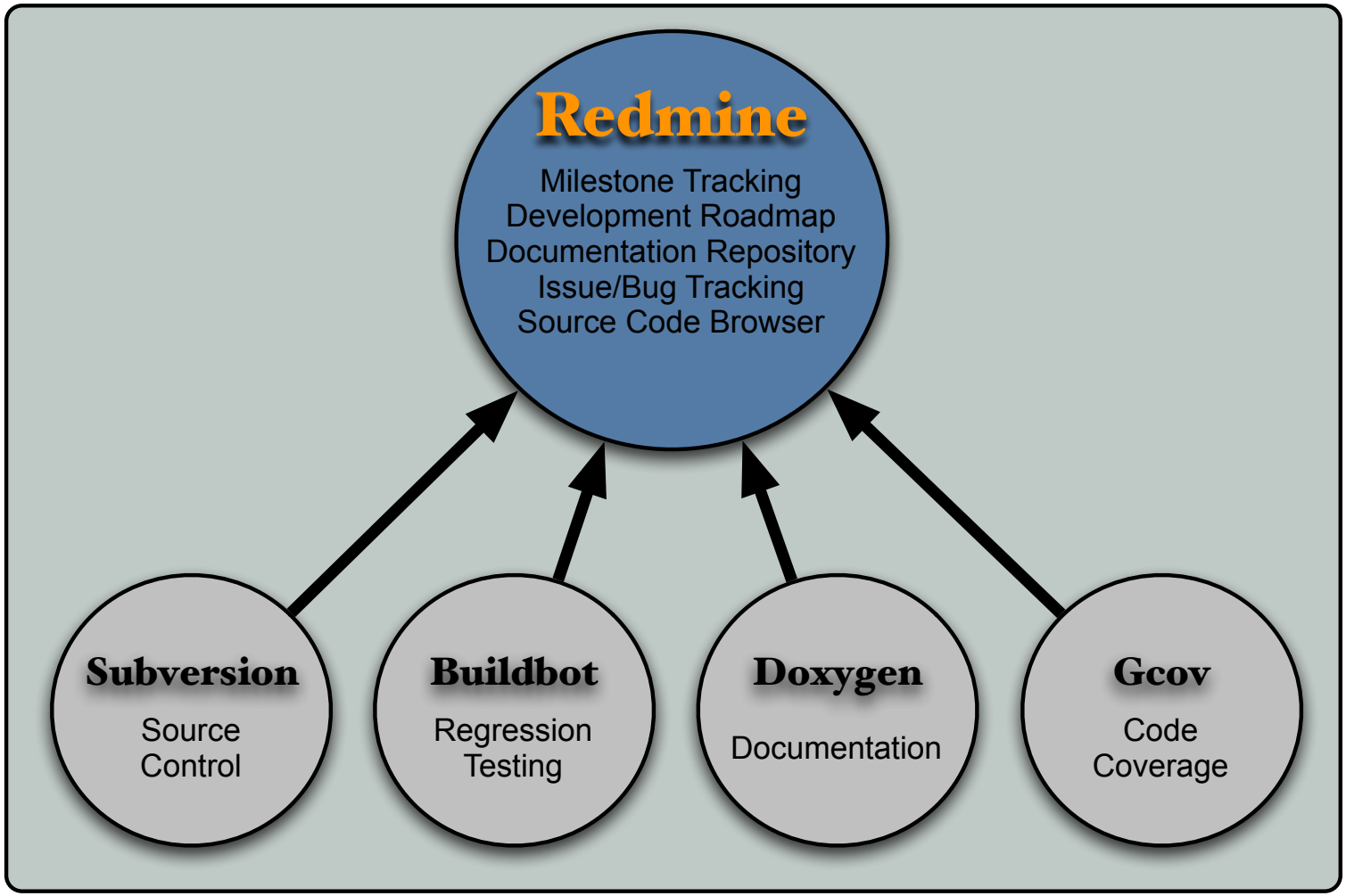
Minimal acceptable documentation

- ReadMe file
- Commented Source
- Bugs / TODO lists
- Changelog

Full Blown Documentation

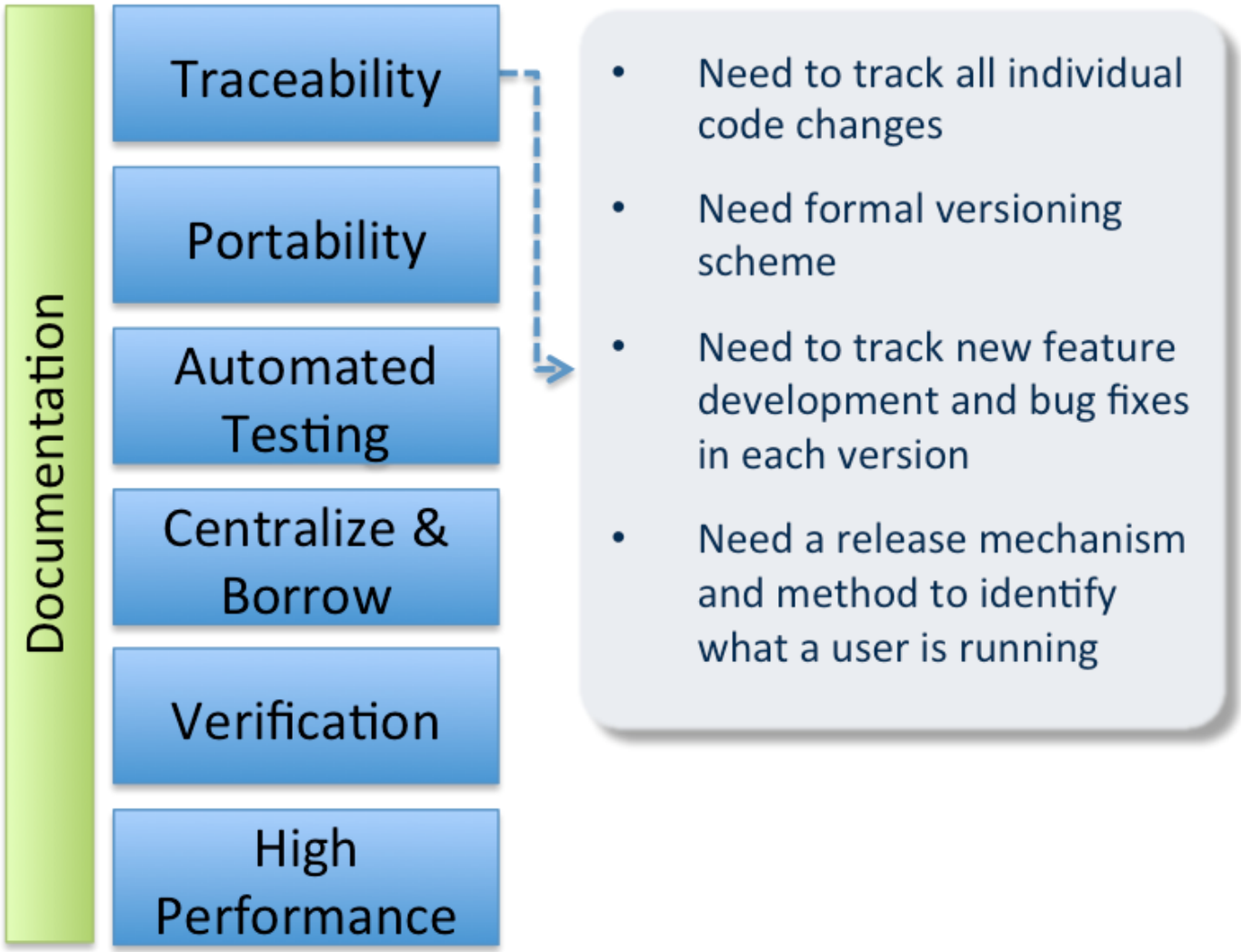
- Model, Discretization and Code Verification Documentation
- Requirements Document
- Autogenerated Technical Documentation and code structure from commented source files
- End-User Document if applicable
- Issue Tracking

Overall Software/Project Management Scheme

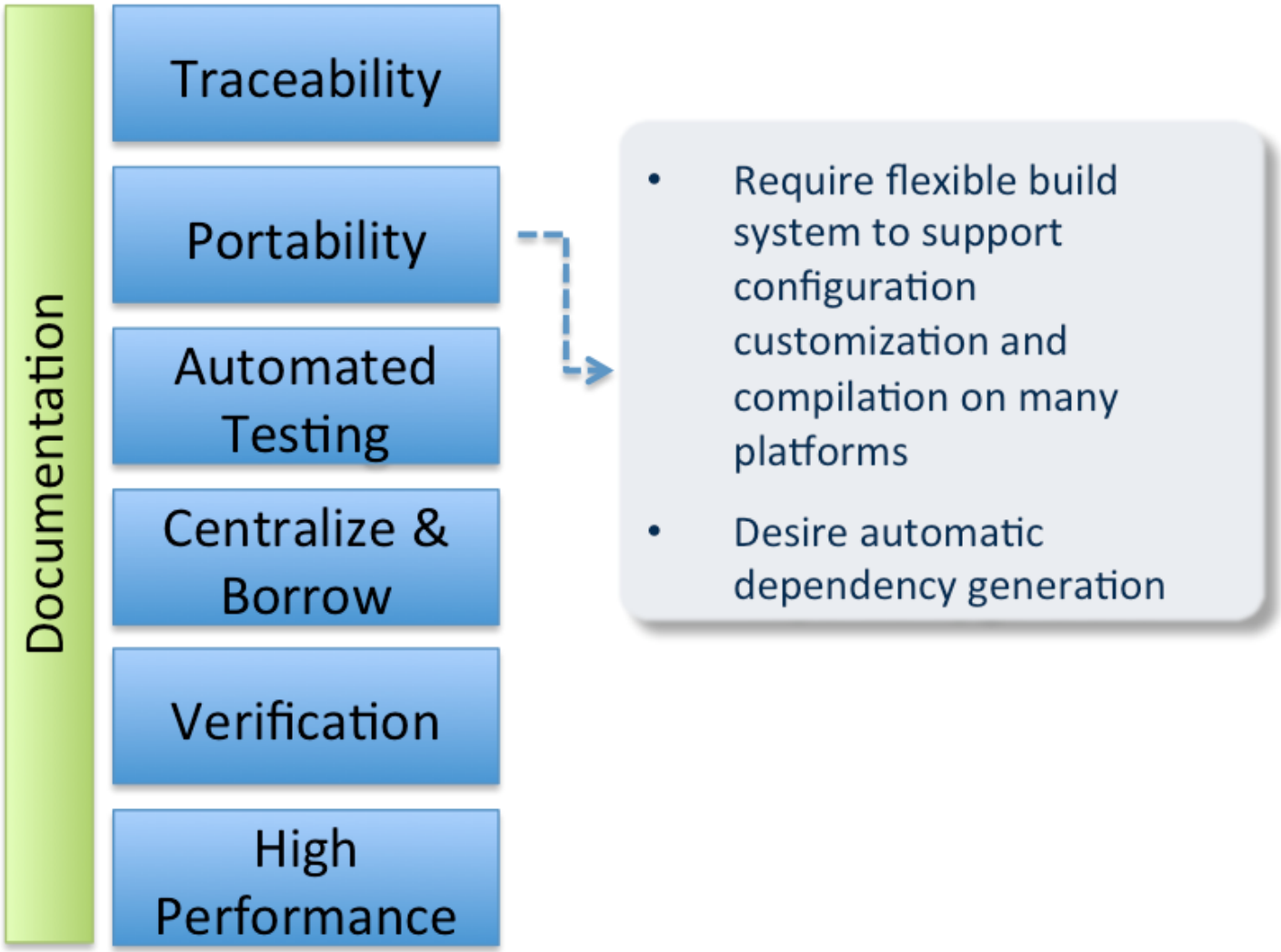


"If you want to know the truth of a city, you need look no further than the state of its infrastructure"

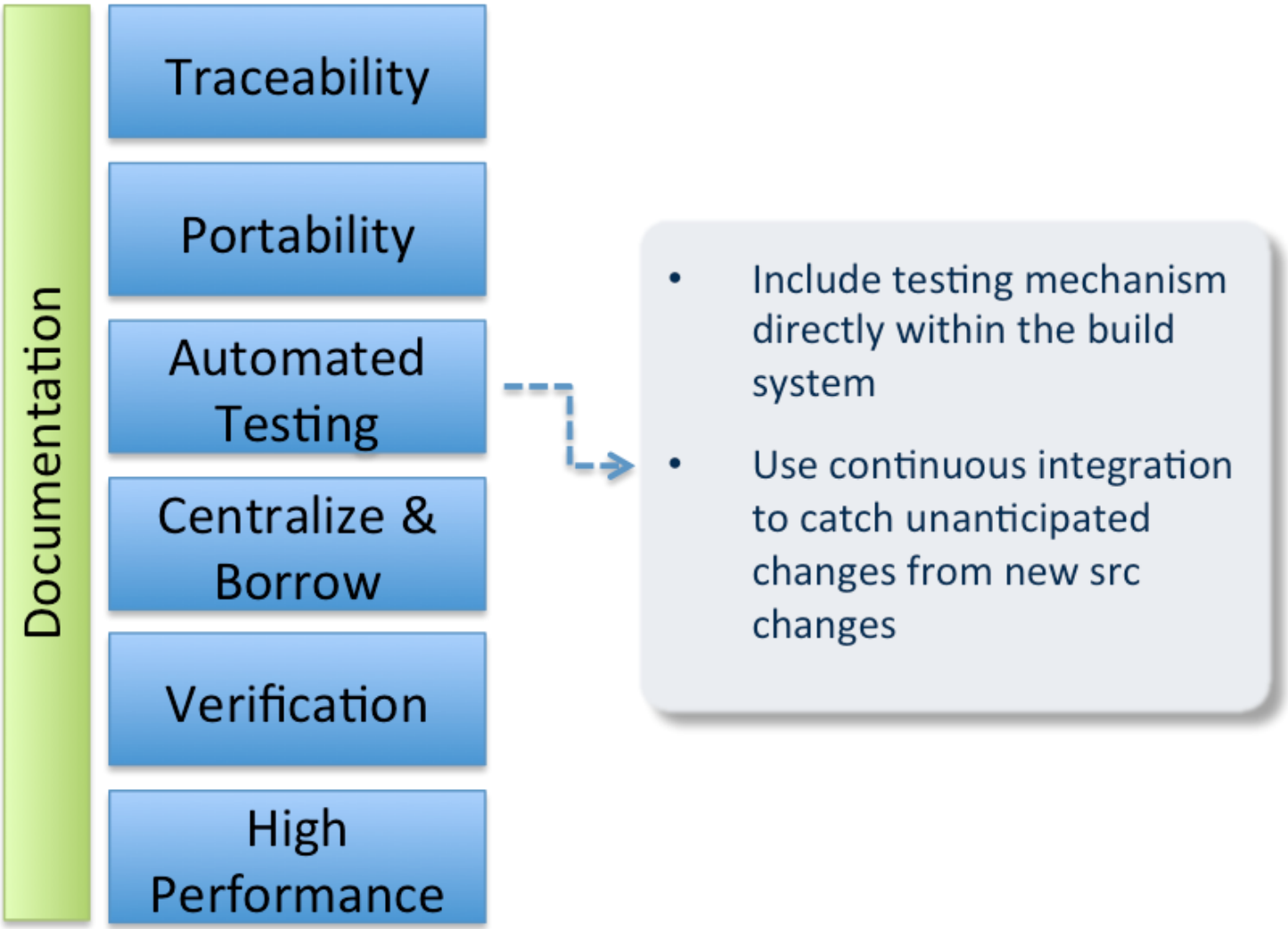
General PECOS Phiolosophy



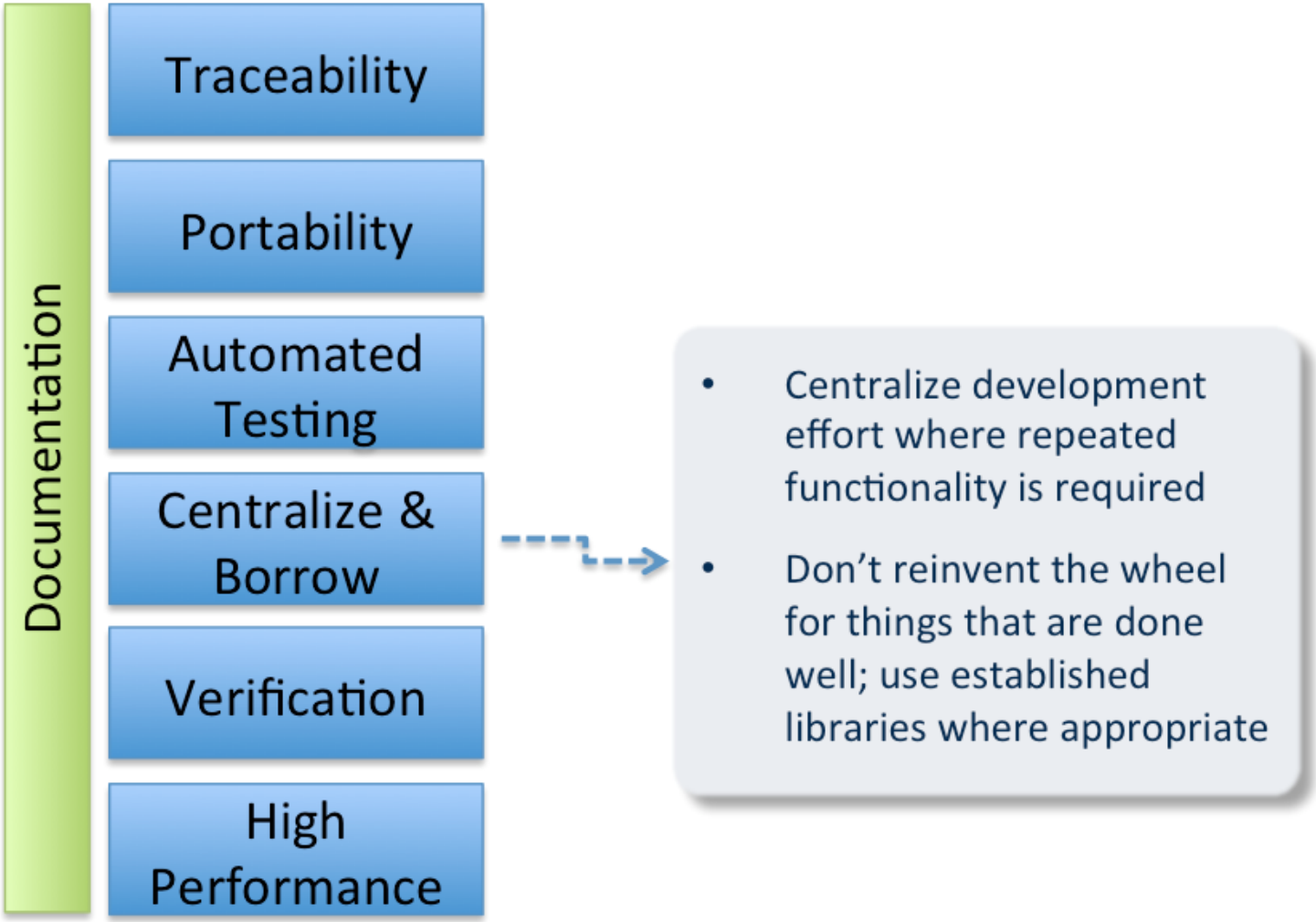
General PECOS Phiolosophy



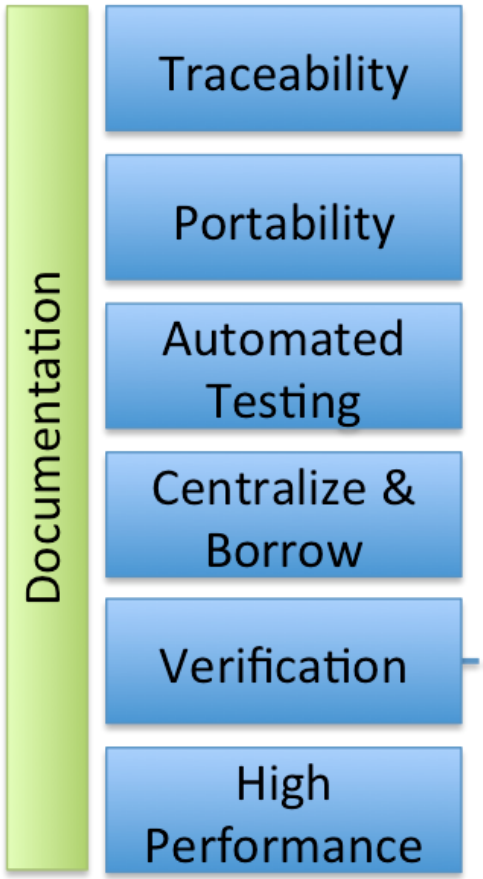
General PECOS Phiolosophy



General PECOS Phiolosophy

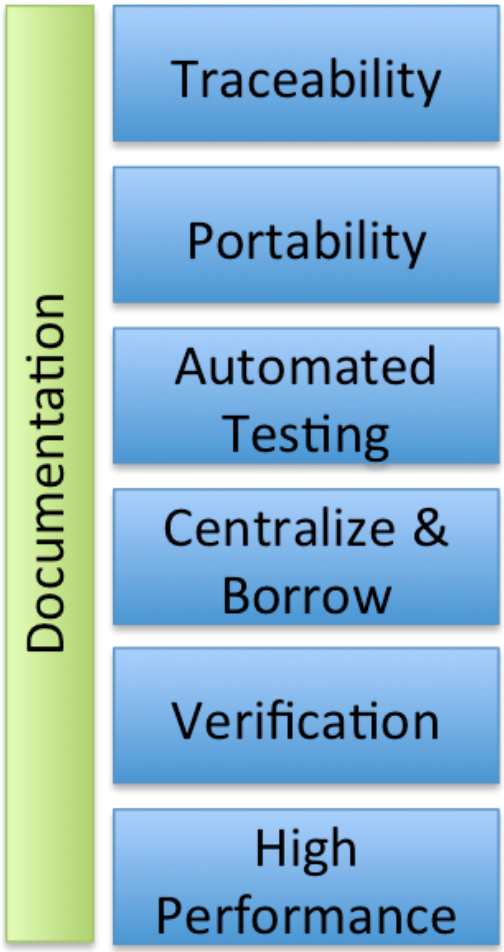


General PECOS Phiolosophy



- Verification is a necessary condition; for support libraries we require individual unit tests for specific features
- For numerical applications:
  - Model documentation: assumptions, governing equations, boundary conditions, numerical implementation
  - Unit testing
  - Method of manufactured solutions
  - Solution verification: convergence analysis, discretization error

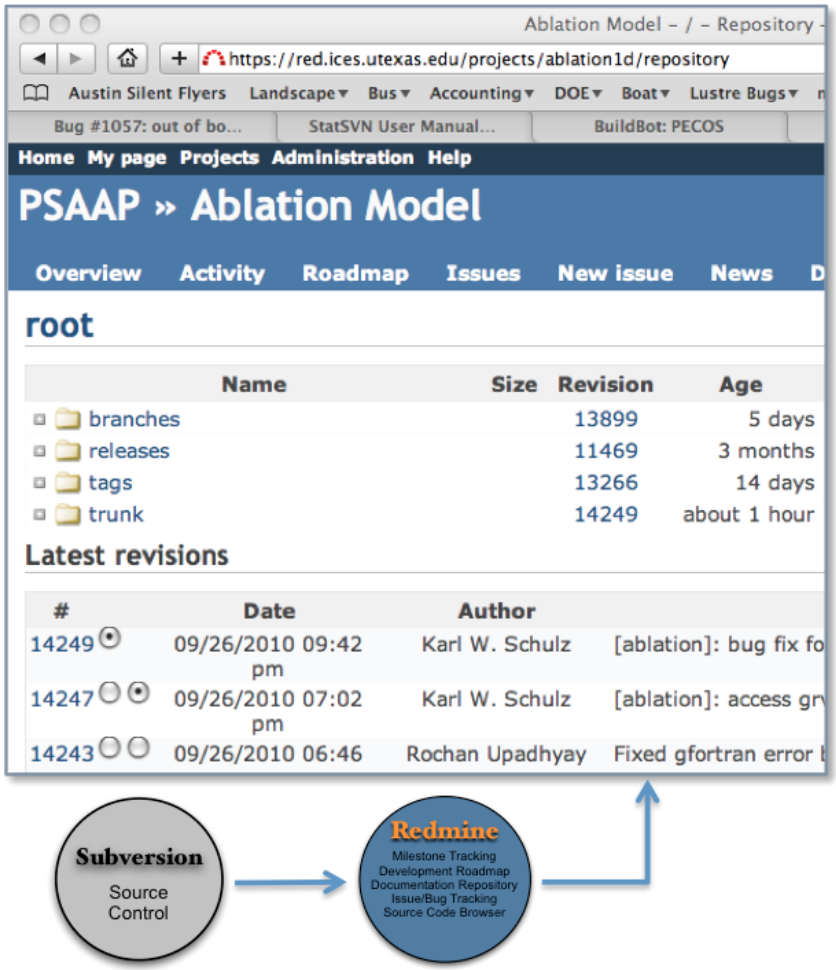
General PECOS Phiolosophy



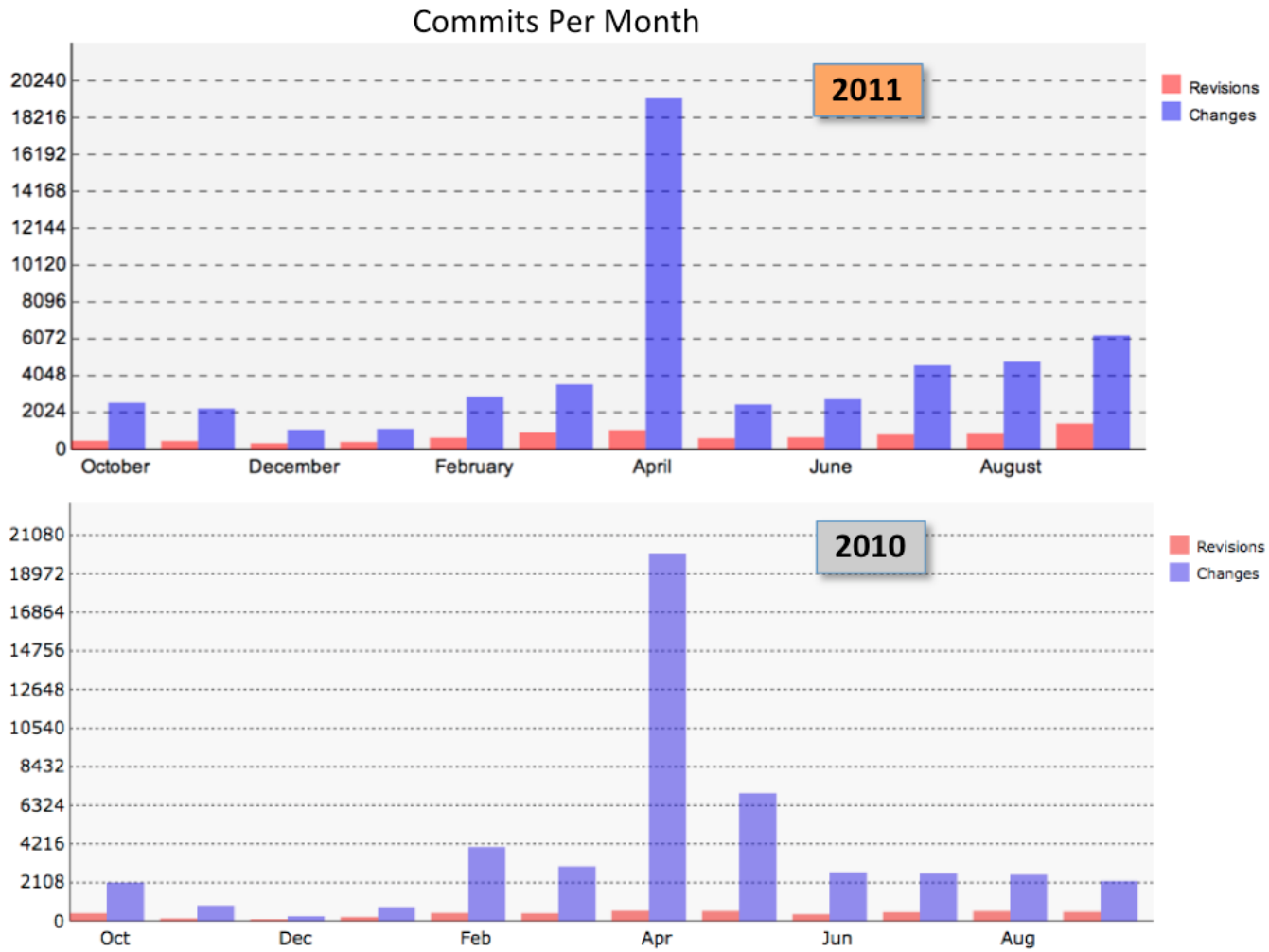
- Efficient, high-performance applications is particularly critical in UQ
- With sufficient regression tests in place, we must repeat performance analysis periodically

Traceability – Source Control

- We continue to use Subversion for underlying source code control (and for many other things related to PECOS):
  - experimental data
  - reference solutions
  - papers/presentations
- Maintain one global repo; each software project managed as subdirectory of top-level repo
  - the repo has grown to a healthy 80GB as of Feb. 2012
  - For each project, we adopt a standard development layout convention:
    - trunk/
    - branches/
    - tags/
    - releases/

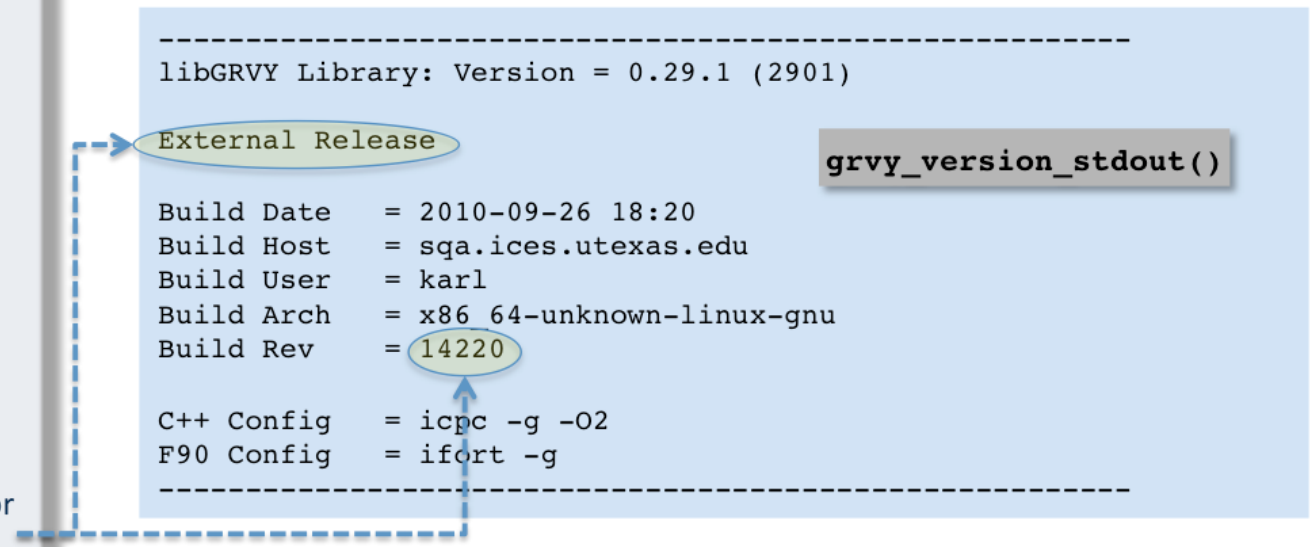


Traceability – Source Control



Traceability – Versioning Info

- Software versioning is important for traceability
- We adopt a **MAJOR.MINOR.MICRO** versioning strategy (e.g. 0.29.1)
- Version info encapsulated multiple ways
  - library headers
  - API calls
- Libraries have standalone "`version`" binary which is included as part of the installation process
- Caches svn revision numbering for traceability (for both development/external usage)
- Important external library linkage also cached



Traceability – Versioning Info

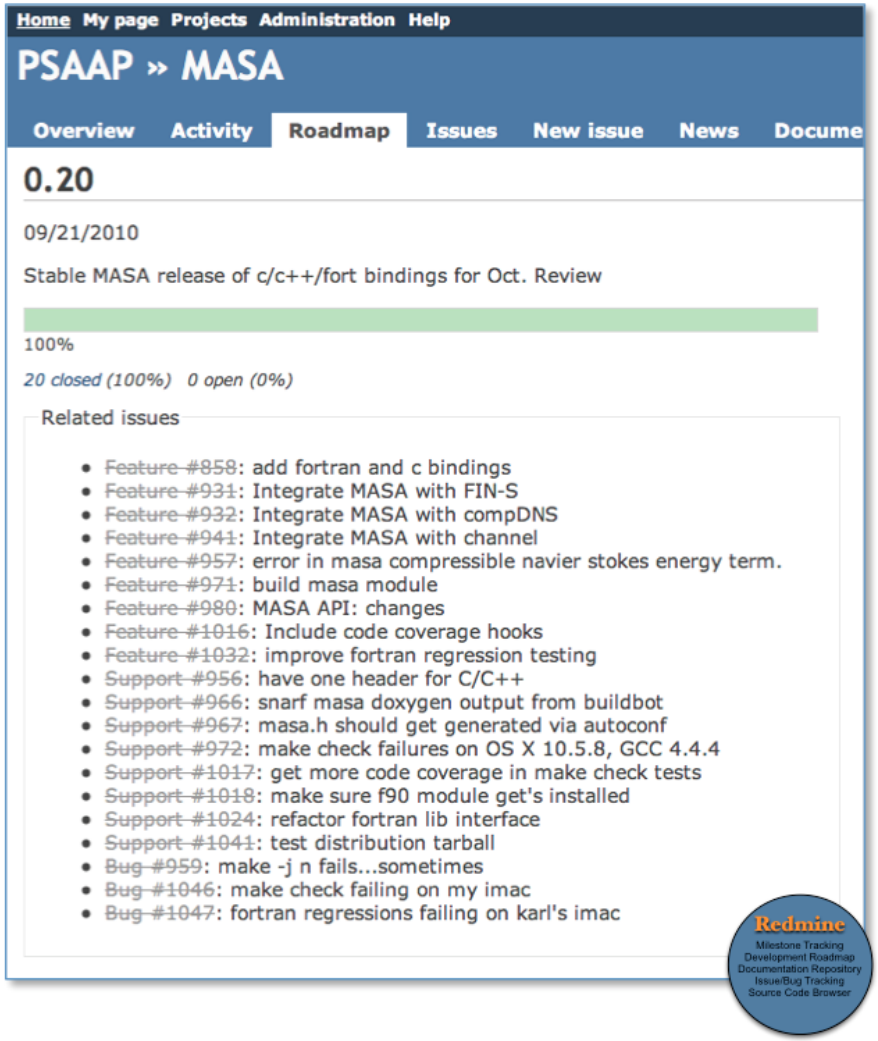
- Software versioning is important for traceability
- We adopt a **MAJOR.MINOR.MICRO** versioning strategy (e.g. 0.29.1)
- Version info encapsulated multiple ways
  - library headers
  - API calls
- Libraries have standalone "`version`" binary which is included as part of the installation process
- Caches svn revision numbering for traceability (for both development/external usage)
- Important external library linkage also cached

```
-------------------------------------------------------------
 PECOS 1D Ablation Model: Version = 0.24

 Development Build

 Config Date  = 2010-09-26 16:12
 Build Host   = sqa.ices.utexas.edu
 Build User   = karl
 Build Arch   = x86_64-unknown-linux-gnu
 Build Rev    = 14230M

 F90 Config   = gfortran  -O3
   C Config   = gcc  -O3

 GRVY Version =  2901
 GRVY DIR     = /org/centers/pecos/LIBRARIES/grvy/gcc-4.3/0.29.1
-------------------------------------------------------------
```

grvy_get_numeric_version()

Traceability – Release Mechanism

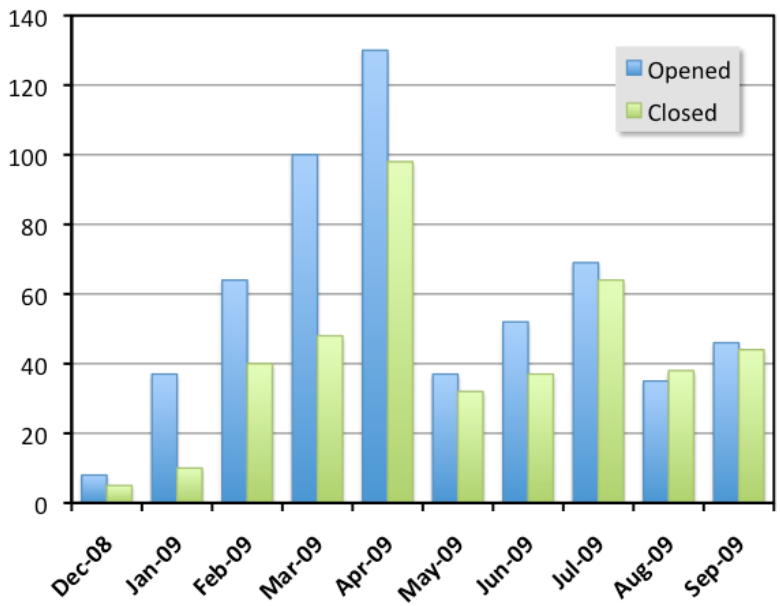
- We use Redmine to track enhancements, features added and fixes for each release
  - formal changelog kept within each projects repository as well
  - support “make dist” releases to generate portable tarballs which do not require local autotools support
- License updates made based on local UT feedback
  - Using LGPL V2.1 for libraries
  - Using GPL V2 for most applications
  - GNU Free Documentation License (GFDL) for documentation
- License changes necessitated updating source headers across many packages
  - we realized we could benefit from a tool to help with this
  - created a simple licensing utility which integrates a unified license header across all desired source files
  - To update, we edit top level LICENSE and run “make”



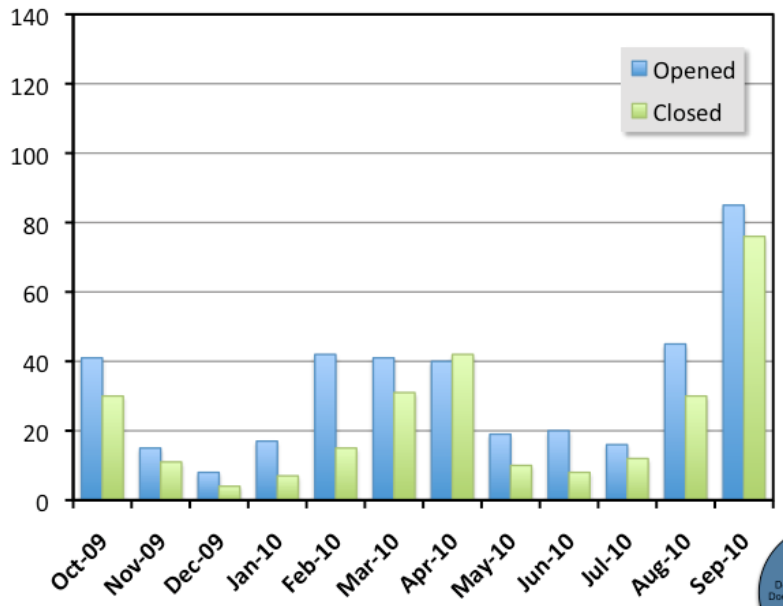
Traceability – Issue Tracking

Issue tracking continues to be part of the daily grind; not just used for bug tracking, also used for center milestones, new feature suggestions, performance improvements, regression failures, etc.

2010

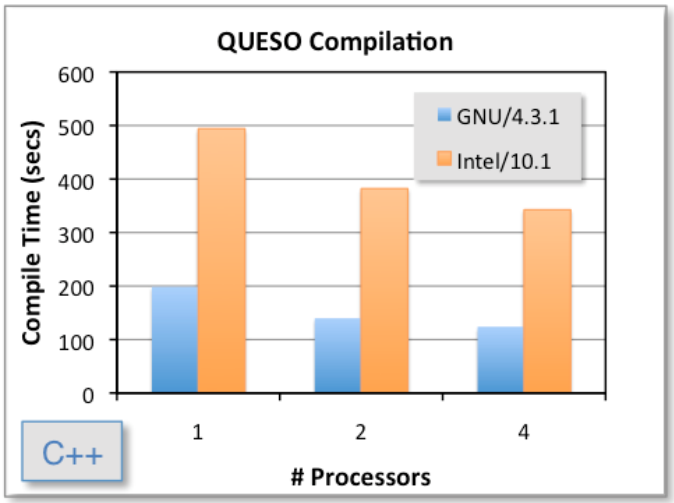


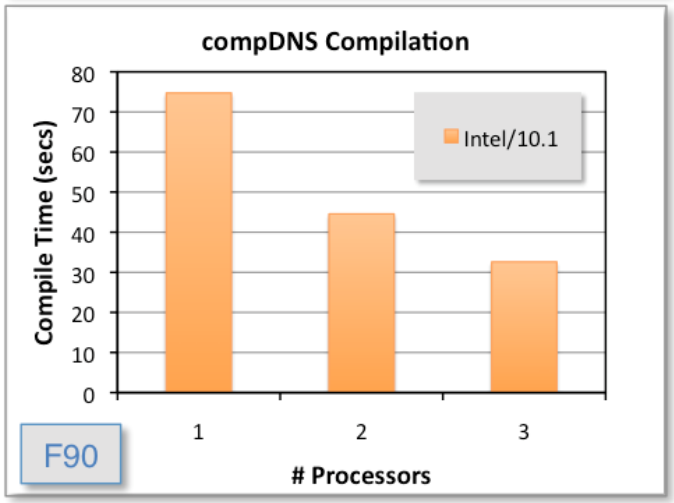
2011



Portability

- Key component to portability is a flexible configuration system
  - need easy way to find FFTW or BLAS routines on multiple systems
  - override default compiler settings as desired
- Using the autotools suite (*autoconf, automake, libtool)*
  - provide familiar "`configure; make check; make install; make docs`" build targets
  - Extra benefit of parallel build support is implicitly provided for C/C++
  - Have devised a mechanism to support auto dependencies in Fortran90 (via a perl-based module dependency tool)





Portability – Mixed Language Improvement

- One new approach taken this year to aid in mixed C/Fortran language portability is the adoption of **iso_c_bindings**
  - allows Fortran/C code to interoperate (without the usual cadre of hacks employed)
  - part of the Fortran 2003 standard
  - provides mechanism to deal with pass by value/reference directly and character strings
  - handles namespace underscoring issues
  - support is available in recent GNU, Intel, and PGI compilers
- To interface with FIN-S (C++), we first adopted use of the bindings within the ABLATION library
  - Entire MASA Fortran interface supported via these bindings
  - Will update GRVY library which was originally done “old-school”

```
interface
  real (c_double) function masa_eval_1d_t_source(value) bind (C,name='cmasa_eval_1d_t_source')
  use iso_c_binding
  implicit none

   real (c_double), value :: value

  end function masa_eval_1d_t_source
end interface
```

Automated Testing

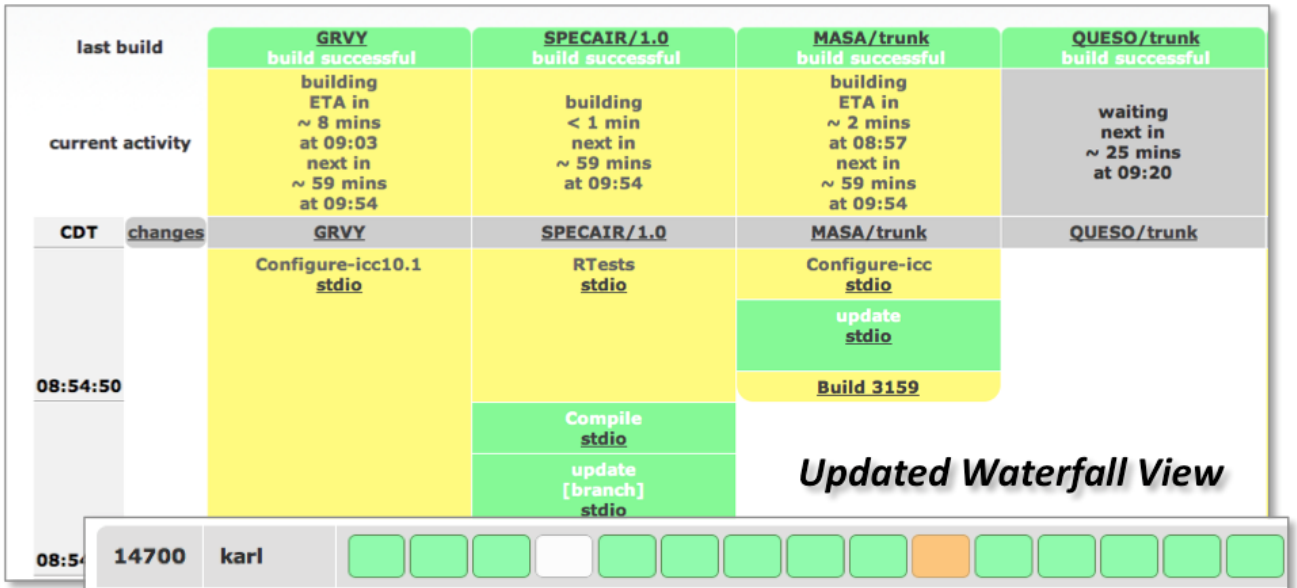
- Continue to rely on Buildbot for automatic regression testing of internal software projects
  - compilation errors
  - runtime errors
  - regression/verification failures
- Regression testing follows standard hierarchical framework
  - grab latest source code (via *svn*)
  - run fresh configuration and compile using the desired build system
  - verify solutions via application dependent regression tests
  - build updated documentation
- Each software package provides autotools testing mechanism – "`make check`"
- Buildbot provides standalone web-page for quick visual inspection (red/green type feedback)

```
> ./configure –prefix=$HOME/bin/grvy
> make
> make check
PASS: init.sh
PASS: C_input_read_variables
PASS: C_input_nofile_available
PASS: C_input_read_missing_variable
PASS: C_input_register_variables
PASS: C_timer_sum
PASS: C_timer_stats
PASS: C_math_isnan
PASS: C_math_isinf
PASS: C_check_file_path
PASS: C_unique_dir
PASS: C_wrapper_scratch_dir.sh
PASS: C_input_fdump
PASS: C_wrapper_log.sh
...
==================
All 34 tests passed
==================
```

Typical Test Sequence

Continuous Integration – *Test Suite Continues to Grow*

- Currently have ~23 active "builders" running on our updated Buildbot server (v. 0.8.1)
- Continuous regressions run at multiple intervals
  - 1 hour
  - 3 hours
  - 18 hours
- Commit changesets exposed to Buildbot
- Developers can add new tests directly which take effect at next build cycle



Centralize and Borrow

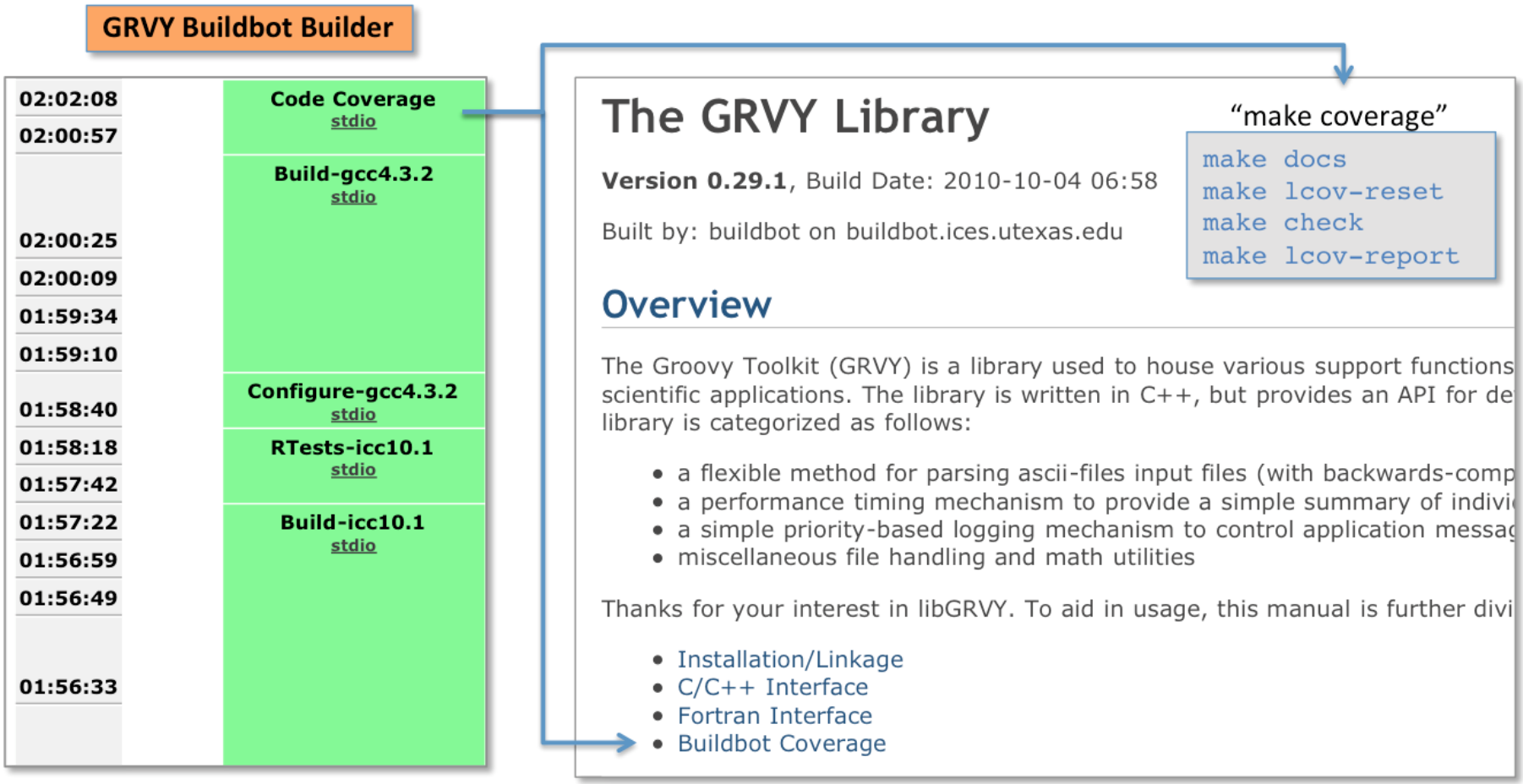
- Centralize development effort where repeated functionality is required
  - GRVY – utility library
  - MASA – verification support
  - ESIO – structured parallel I/O library
- Use of svn:externals to share/ integrate common repositories
  - autoconf macros
  - license utilities
  - Fortran auto-dependency tools

- Leverage existing codes and libraries where appropriate
  - don't reinvent the wheel for things that are done well
  - use established libraries
- Primary external library usage:
  - Boost
  - Dakota
  - Trilinos
  - GSL
  - PETSc
  - HDF5
  - GLPK
  - FFTW3
  - BLAS/LAPACK

Verification

- Identified goals for improvement:
  - Expanded use of unit testing
  - Inclusion of code coverage analysis to assess completeness of our existing tests
  - Expanded use of manufactured solutions
- Results:
  - Completed initial code coverage analysis using GNU gcov tool
  - Have embedded configuration options to easily include coverage within PECOS software packages
    - generates HTML output summarizing coverage (lcov)
    - accessible via our Doxygen/Redmine integration
    - can browse source directly to view areas of the code which are not sufficiently tested
  - Formalized the creation of an MMS library; identified a *Minister of Verification*

Verification - Code Coverage Analysis



Verification - Code Coverage Analysis

GRVY Buildbot Builder

02:02:08	**Code Coverage** stdio
02:00:57	
	Build-gcc4.3.2 stdio
02:00:25	
02:00:09	
01:59:34	
01:59:10	
	Configure-gcc4.3.2 stdio
01:58:40	
01:58:18	**RTests-icc10.1** stdio
01:57:42	
01:57:22	**Build-icc10.1** stdio
01:56:59	
01:56:49	
01:56:33	

LCOV - code coverage report

Current view: top level - build/src
Test: libGRVY
Date: 2010-10-04

	Hit	Total	Coverage
Lines:	778	849	91.6 %
Functions:	179	188	95.2 %

Filename	Line Coverage		Functions	
grvy.h	100.0 %	1 / 1	100.0 %	1 / 1
grvy_classes.cpp	88.9 %	375 / 422	94.4 %	68 / 72
grvy_classes.h	100.0 %	7 / 7	100.0 %	1 / 1
grvy_input.cpp	97.0 %	160 / 165	95.9 %	47 / 49
grvy_int.h	100.0 %	3 / 3	100.0 %	1 / 1
grvy_log.cpp	100.0 %	20 / 20	100.0 %	6 / 6
grvy_math.cpp	100.0 %	14 / 14	100.0 %	7 / 7
grvy_timer.cpp	100.0 %	81 / 81	100.0 %	26 / 26
grvy_utils.cpp	85.5 %	112 / 131	86.4 %	19 / 22
grvy_version.cpp	100.0 %	5 / 5	100.0 %	3 / 3

Generated by: LCOV version 1.9

Verification - Code Coverage Analysis

- Note that 100% line coverage can be difficult to obtain with sufficient defensive checks in place (e.g. ENOMEM errors and file-system calls)
- Performing the code coverage exercise was quite beneficial
  - exposed missing test coverage which we improved
  - caught several bugs (bug #1004, bug 1027, bug 1065)
- Coverage option now available for all formal PECOS software products to enable continued periodic analysis

```
129          28 :     if(stat(dirname,&st) != 0)
130             :       {
131          18 :         if( mkdir(dirname,0700) != 0 )
132             :           {
133           0 :             _GRVY_message(GRVY_ERROR,__func__,"unable to create directory",dirname);
134           0 :             return -1;
135             :           }
136             :       }
```

GRVY Coverage (grvy_utils.cpp)

MASA – 0.20 Code Coverage

LCOV - code coverage report

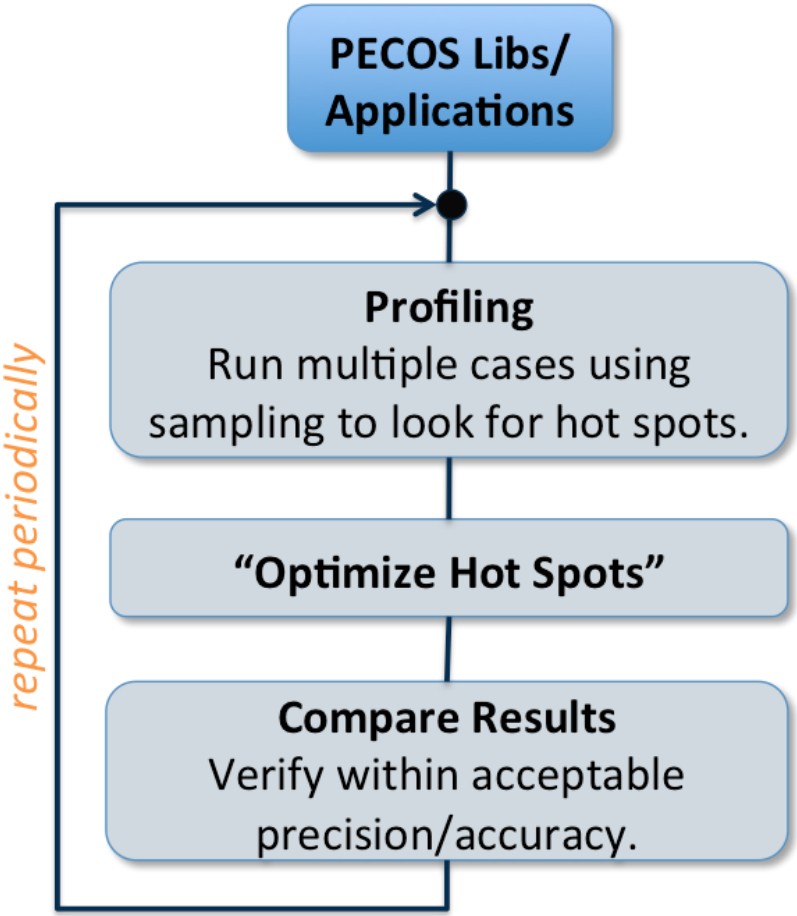
		Hit	Total	Coverage
Current view: top level - build/src				
Test: MASA	**Lines:**	1670	1676	99.6 %
Date: 2010-10-04	**Functions:**	352	401	87.8 %

Filename	Line Coverage		Functions	
axi_cns.cpp	100.0 %	72 / 72	92.3 %	12 / 13
axi_euler.cpp	100.0 %	70 / 70	92.3 %	12 / 13
cmasa.cpp	100.0 %	55 / 55	100.0 %	38 / 38
cns.cpp	100.0 %	199 / 199	92.3 %	24 / 26
euler.cpp	100.0 %	252 / 252	91.4 %	32 / 35
heat.cpp	100.0 %	306 / 306	77.4 %	41 / 53
masa.f90	100.0 %	10 / 10	100.0 %	4 / 4
masa_class.cpp	100.0 %	149 / 149	85.0 %	17 / 20
masa_core.cpp	100.0 %	263 / 263	100.0 %	61 / 61
masa_internal.h	100.0 %	69 / 69	73.7 %	70 / 95
masa_map.cpp	100.0 %	26 / 26	100.0 %	6 / 6
rans_sa.cpp	100.0 %	92 / 92	96.4 %	27 / 28
sod.cpp	94.7 %	107 / 113	88.9 %	8 / 9

Generated by: LCOV version 1.9

High Performance

- Performance always matters, and is particularly important in UQ settings where we run thousands or millions of forward solves
- We treat performance optimization as an ongoing process
- Impossible without sufficient regressions and solution "diff'ers" to assess correctness and impact on floating-point precisions
- Mention that productivity performance counts as well when in development mode
  - FINS compile time reduced dramatically
- New feature coming in our libGRVY utility library
  - mechanisms to track named "timer" performance over time
  - results are cached on a per machine (or per supercomputer) basis
  - stored in HDF5 repository associate with the application



Generally Accepted Recommendations

General Recommendations

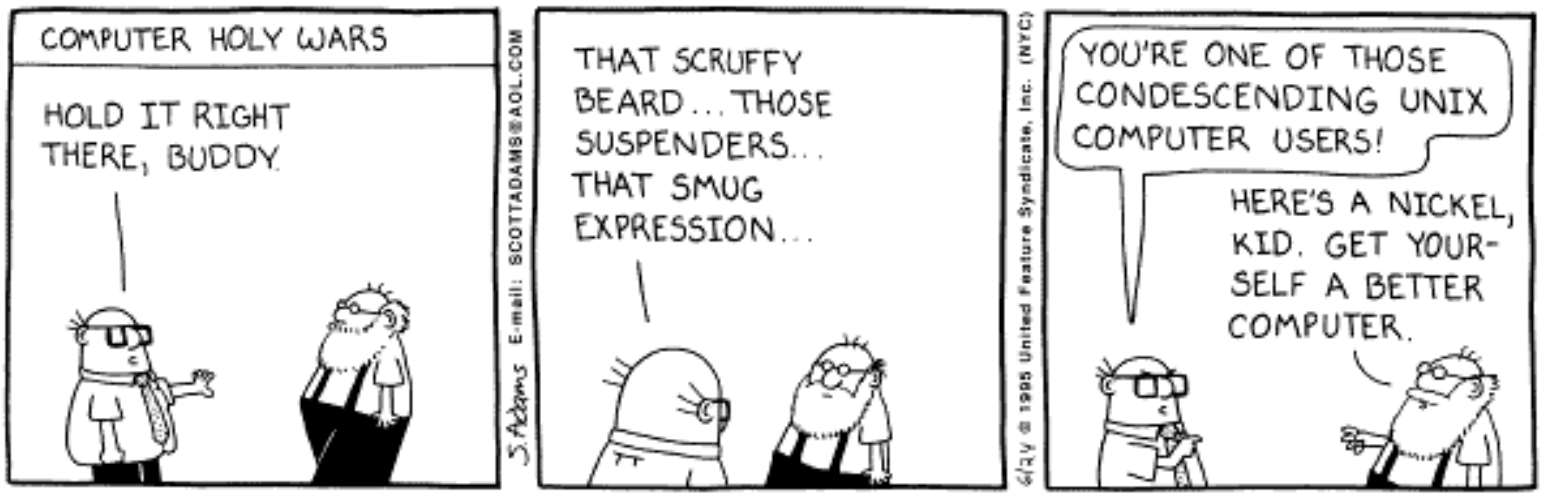
- Show other people your code early and often
- Reuse good software and evaluate multiple options
- Avoid platform dependent files (endianness)
- Always consider performance versus portability
- Prefer OSS solutions if possible
- Backup everything and anything

Don’t reinvent the wheel

- Don’t write your own input parser
- Don’t write your own solver unless necessary
- Don’t write anything until you evaluate options

Learn as much Unix as you can tolerate

Unix: A Culture in Itself



"Two of the most famous products of Berkeley are LSD and Unix. I don't think that this is a coincidence."
(Anonymous quote from The UNIX-HATERS Handbook.)

Highly Opinionated Recommendations

Consistent tools from desktop to supercomputer

- Learn a console-based text edition (vi(m) / emacs)
- Learn a unix shell
- Learn LaTeX
- Learn a scripting language (python / perl)
- man (-k) is your friend

Understand:

- compile / link cycle
- static versus dynamic libraries
- processor optimization
- storage hierarchy (local versus network disk)
- environment variables ($PATH and $LD_LIBRARY_PATH)

More Highly Opinionated Recommendations

More things to understand

- Actually learn how to use the compiler
- Learn about the hardware (memory, interconnect, disk)
- Learn about debugging and profiling
- Always have a restart capability

Soapbox

- Use text-based formats when possible
- screen will change your life
- Goggle is an amazing resource
- Be NICE to SysAdmins

Thank you!

Questions?