Overview of Best Practices in HPC Software Development

Better Scientific Software Tutorial

David E. Bernholdt
Computer Science and Mathematics Division and National Center for Computational Sciences
Oak Ridge National Laboratory

Supercomputing 2018
Dallas, TX
November 12, 2018

License, citation, and acknowledgements

License and Citation

- Requested citation: David E. Bernholdt, Anshu Dubey, and Katherine Riley, Overview of Best Practices in HPC Software Development, Better Scientific Software tutorial, in SC ‘18: International Conference for High Performance Computing, Networking, Storage and Analysis, Dallas, Texas, 2018. DOI: 10.6084/m9.figshare.7304408.

Acknowledgements

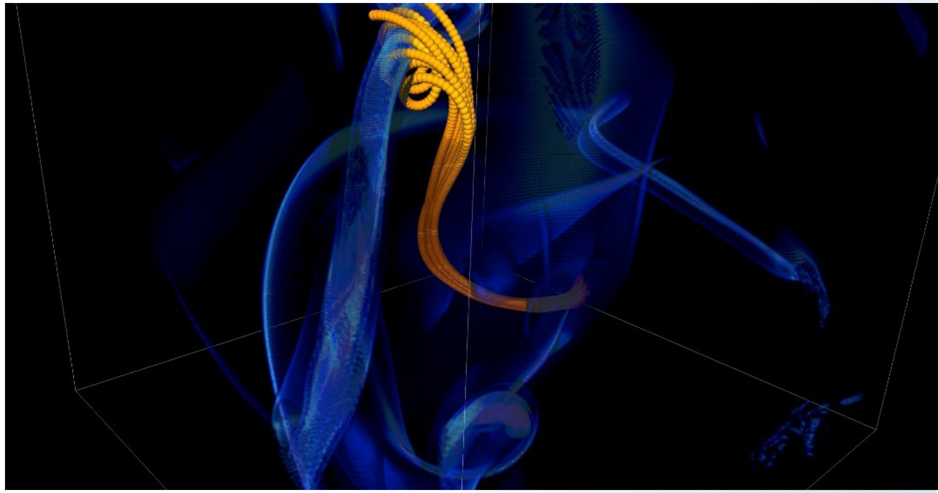
- This work was supported by the U.S. Department of Energy Office of Science, Office of Advanced Scientific Computing Research (ASCR), and by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.
- This work was performed in part at the Argonne National Laboratory, which is managed 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 Oak Ridge National Laboratory, which is managed by UT-Battelle, LLC for the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

Good scientific process
requires
good software practices

Good software practices
increase
scientific productivity

IDEAS productivity EXASCALE COMPUTING PROJECT

You Can Mitigate Risk But It Is Never Zero



- Short notice availability of one of the biggest machines of it’s time
  - **< 1month to get ready, run was 1.5 weeks**
- Quick and dirty development of particle capability in code
- Error in tracking particles resulted in duplicated tags from round-off
- Had to develop post-processing tools to correctly identify trajectories
  - **6 months to process results**

FLASH had a software process in place. It was tested regularly. This was one instance when the full process could not be applied because of time constraints.

Heroic Programming

Usually a pejorative term, is used to describe the expenditure of huge amounts of (coding) effort by talented people to overcome shortcomings in process, project management, scheduling, architecture or any other shortfalls in the execution of a software development project in order to complete it. Heroic Programming is often the only course of action left when poor planning, insufficient funds, and impractical schedules leave a project stranded and unlikely to complete successfully.

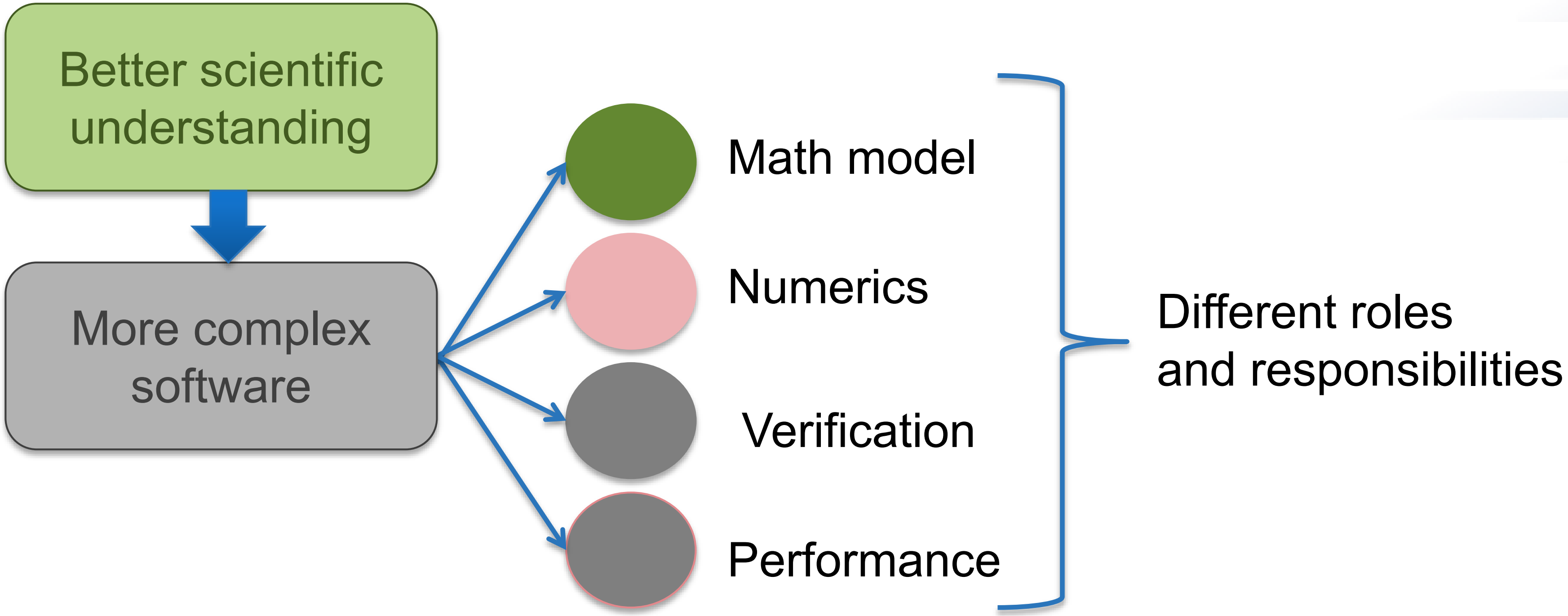
From http://c2.com/cgi/wiki?HeroicProgramming

Science teams often resemble heroic programming

Many do not see anything wrong with that approach

What is wrong with heroic programming

Scientific results that could be obtained with heroic programming have run their course, because:


Better scientific understanding
More complex software
Math model
Numerics
Verification
Performance
Different roles and responsibilities

It is not possible for a single person to take on all these roles

In Extreme-Scale science

- Codes aiming for higher fidelity modeling
  - More complex codes, simulations and analysis
  - More moving parts that need to interoperate
  - Variety of expertise needed – the only tractable development model is through **separation of concerns**
  - **It is more difficult to work on the same software in different roles without a software engineering process**
- Onset of higher platform heterogeneity
  - Requirements are unfolding, not known *a priori*
  - **The only safeguard is investing in flexible design and robust software engineering process**

In Extreme-Scale science

- Codes aiming for higher fidelity modeling
  - More complex codes, simulations and analysis
  - More moving parts that need to interoperate
  - Variety of expertise needed – the only tractable development model is through **separation of concerns**
  - **It is more difficult to work on the same software in different roles without a software engineering process**
- Onset of higher platform heterogeneity
  - Requirements are unfolding, not known *a priori*
  - **The only safeguard is investing in flexible design and robust software engineering process**

Supercomputers change fast
Especially Now

Technical Debt

Consequence of Choices

Quick and dirty collects interest which means more effort required to add features.

Accretion leads to unmanageable software

- Increases cost of maintenance
- Parts of software may become unusable over time
- Inadequately verified software produces questionable results
- Increases ramp-on time for new developers
- Reduces software and science productivity due to technical debt

- "... it seems likely that significant software contributions to existing scientific software projects are not likely to be rewarded through the traditional reputation economy of science. Together these factors provide a reason to expect the over-production of independent scientific software packages, and the underproduction of collaborative projects in which later academics build on the work of earlier ones."

    - Howison & Herbsleb (2011)

Challenges Developing a Scientific Application

Technical

- All parts of the cycle can be under research
- Requirements change throughout the lifecycle as knowledge grows
- Verification complicated by floating point representation
- Real world is messy, so is the software

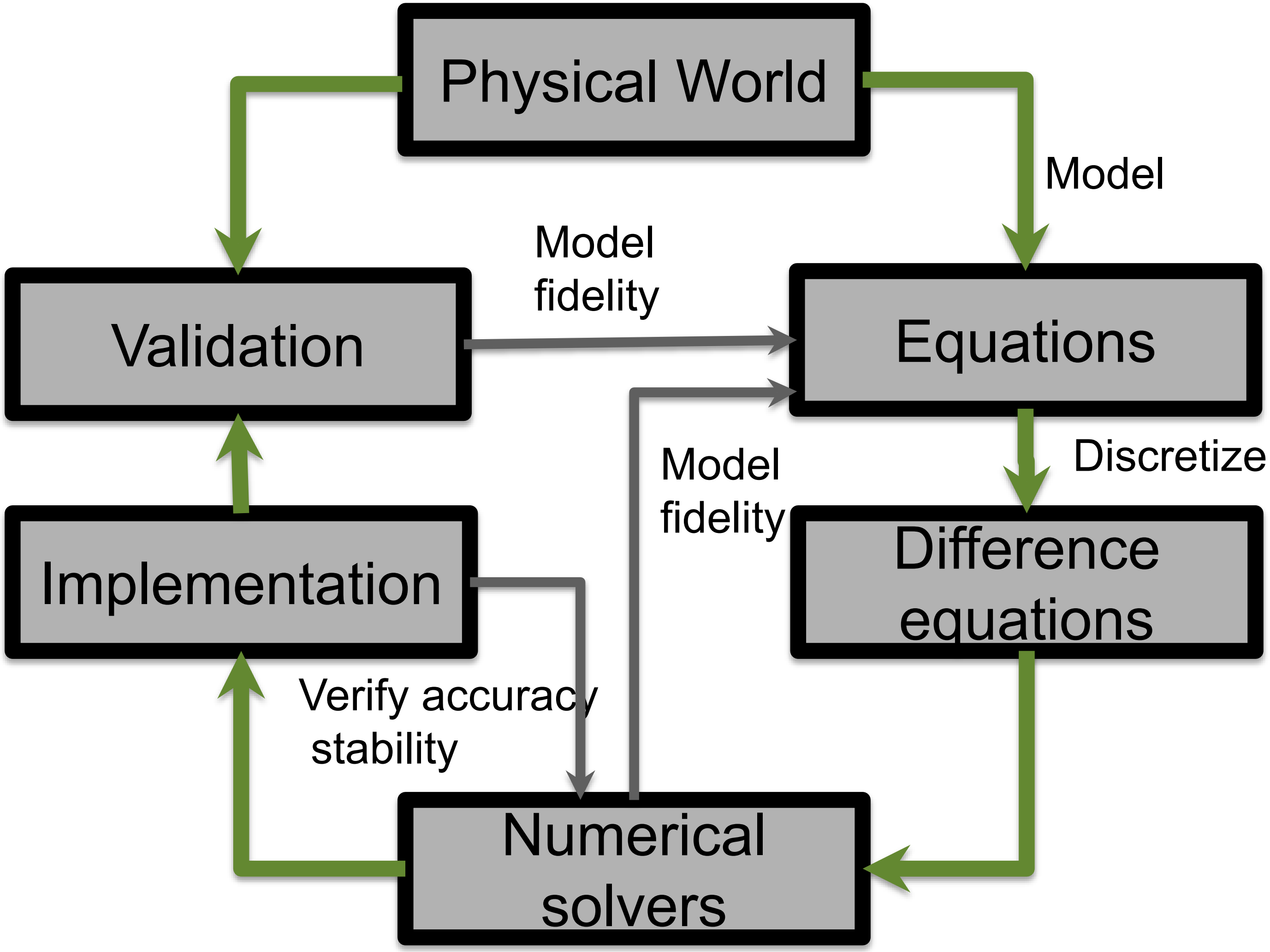
Sociological

- Competing priorities and incentives
- Limited resources
- Perception of overhead without benefit
- Need for interdisciplinary interactions

Customizations For Science Applications

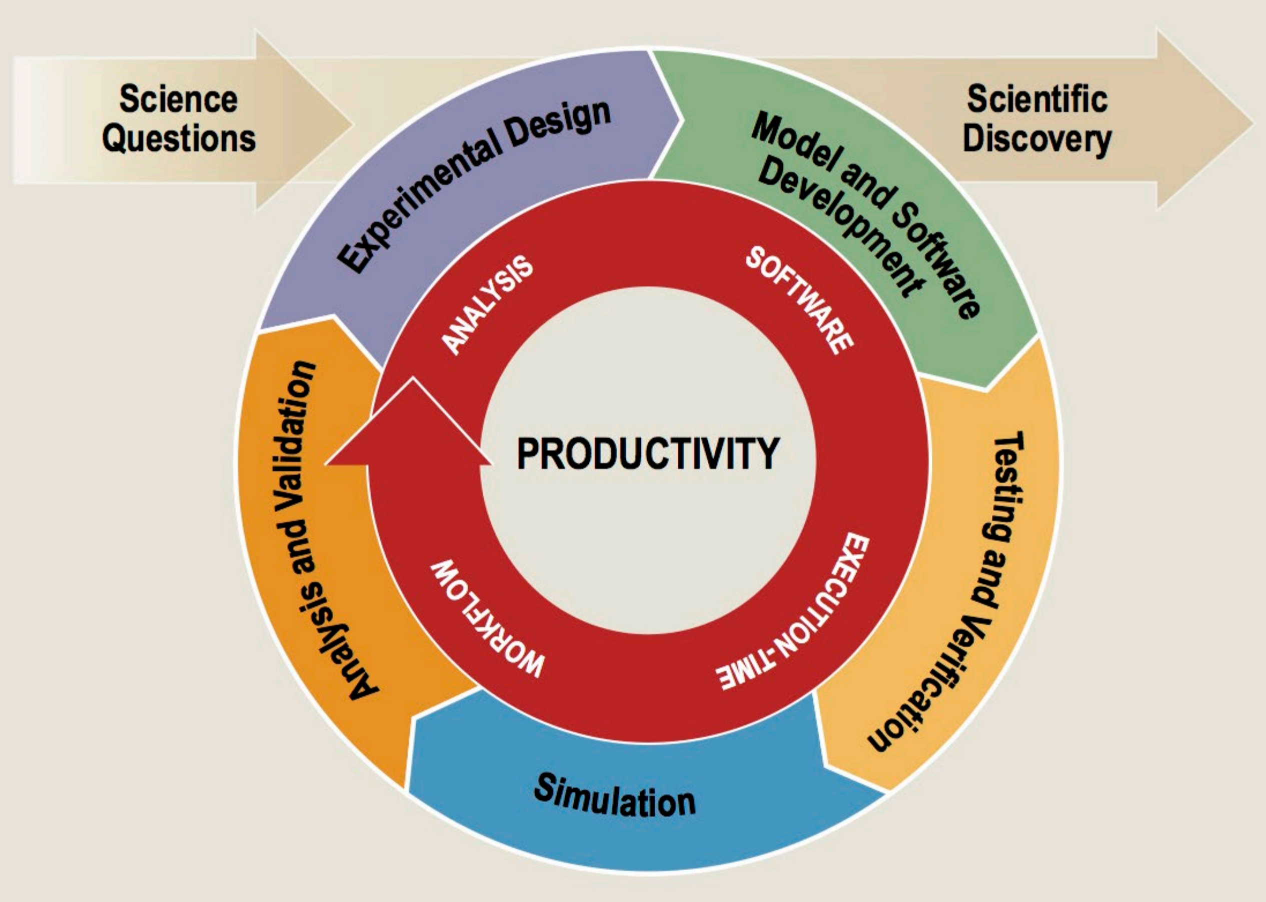
- Testing does not follow specific methods as understood by the software engineering research community
  - The extent and granularity reflective of project priorities and team size
  - Larger teams have more formalization
- Lifecycle of science compare to lifecycle of development
- Development model
  - Mostly ad-hoc, some are close to agile model, but none follows it explicitly
  - Much more responsive to the needs of the lifecycle

Lifecycle of Scientific Application



- Modeling
  - Approximations
  - Discretizations
  - Numerics
    - Convergence
    - Stability
- Implementation
  - Verification
    - Expected behavior
  - Validation
    - Experiment/observation

Software productivity cycle



http://www.orau.gov/swproductivity2014/SoftwareProductivityWorkshopReport2014.pdf

Software Process Best Practices

Baseline

- Invest in extensible code design
- Use version control and automated testing
- Institute a rigorous verification and validation regime
- Define coding and testing standards
- Clear and well defined policies for
  - Auditing and maintenance
  - Distribution and contribution
  - Documentation

Desirable

- Provenance and reproducibility
- Lifecycle management
- Open development and frequent releases

A Useful Resource

https://ideas-productivity.org/resources/howtos/

- **‘What Is’ docs**: 2-page characterizations of important topics for SW projects in computational science & engineering (CSE)
- **‘How To’ docs**: brief sketch of best practices
  - Emphasis on ``bite-sized'' topics enables CSE software teams to consider improvements at a small but impactful scale
- We welcome feedback from the community to help make these documents more useful

Other resources

http://www.software.ac.uk/

http://software-carpentry.org/

http://flash.uchicago.edu/cc2012/

http://journals.plos.org/plosbiology/article?id=10.1371/journal.pbio.1001745

http://ieeexplore.ieee.org/xpls/icp.jsp?arnumber=4375255

http://www.orau.gov/swproductivity2014/SoftwareProductivityWorkshopReport2014.pdf

http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6171147

Why Community Codes?

- Scientists can focus on developing for their algorithmic needs instead of getting bogged down by the infrastructural development
- Graduate students do not start developing codes from scratch
  - Look at the available public codes and converge on the ones that most meet their needs
  - Look at the effort of customization for their purposes
  - Select the public code, and build upon it as they need

Important to remember that they still need to understand the components developed by others that they are using, they just don’t have to actually develop everything themselves. And this is particularly true of pesky detailed infrastructure/solvers that are too well understood to have any research component, but are time consuming to implement right

Why Community Codes Continued

- Researchers can build upon work of others and get further faster, instead of reinventing the wheel
  - Code component re-use
  - No need to become an expert in every numerical technique
- More reliable results because of more stress tested code
  - Enough eyes looking at the code will find any errors faster
  - New implementations take several years to iron out the bugs and deficiencies
  - Different users use the code in different ways and stress it in different ways
- Open-source science results in more reproducible results
- Generally good for the credibility

Communities Do Use Community Codes

- Astrophysics, Molecular Dynamics, Chemistry, Climate, etc
- Community/open-source approach more common in areas which need multi-physics and/or multi-scale
- A visionary sees the benefit of software re-use and releases the code
- Sophistication in modeling advances more rapidly in such communities
- Others keep their software close for perceived competitive advantage
  - Repeated re-invention of wheel
  - General advancement of model fidelity slower

Summary

- Good software practices are needed for scientific productivity
- Science at extreme-scales is complex and requires multiple expertise
- Software process does need to address reality
- Open codes, community contribution, are a powerful tool

It is extremely important to recognize that science through computing is only as good as the software that produces it