MATH 676

–

Finite element methods in scientific computing

Wolfgang Bangerth, Texas A&M University

Lecture 43:

Beyond computational methods

Part 2:

Issues with developing large software

Software issues in HPC

Ultimately, HPC is about *applications*, not just algorithms and their analysis.

Thus, we need to consider the issue of *software that implements* these applications:

- How complex is the software?
- How do we write software? Are there tools?
- How do we verify the correctness of the software?
- How do we validate the correctness of the model?

- Testing
- Documentation
- Social issues

Complexity of software

Many HPC applications are *several orders of magnitude* larger than everything you have probably ever seen!

For example, a crude measure of complexity is the number of lines of code in a package (as of 2011):

- Deal.II has 550k
- PETSc has 500k
- Trilinos has 3.1M

At this scale, software development does not work the same as for small projects:

- No single person has a global overview
- There are many years of work in such packages
- No person can remember even the code they wrote

Complexity of software

The only way to deal with the complexity of such software is to:

- *Modularize:* Different people are responsible for different parts of the project.
- *Define interfaces:* Only a small fraction of functions in a module is available to other modules
- *Document:* For users, for developers, for authors, and at different levels

- *Test, test, test*

How do we write software

Successful software must follow *the prime directive of software*:

- Developer time is the single most scarce resource!

As a consequence (part 1):

- Do not reinvent the wheel: use what others have already implemented
- Use the best tools
- Do not become the bottleneck (e.g. by not writing documentation)

- Delegate. You can't do it all.

How do we write software

Successful software must follow *the prime directive of software*:

- Developer time is the single most scarce resource!

As a consequence (part 2):

- Re-use code, don't duplicate
- Use strategies to *avoid* introducing bugs

- Test, test, test:
  - The earlier a bug is detected the easier it is to find
  - Even good programmers spend more time debugging code than writing it

Verification & validation (V&V): Verification

Verification refers to the process of ensuring that the software solves the problem it is supposed to solve:

"The program solves the problem correctly"

A common strategy to achieve this is to...

Verification & validation (V&V): Verification

Verification refers to the process of ensuring that the software solves the problem it is supposed to solve:

"The program solves the problem correctly"

A common strategy to achieve this is to *test test test:*

- *Unit tests* verify that a function/class does what it is supposed to do (assuming that correct result is known)
- *Integration tests* verify a whole algorithm (e.g. using what is known as the Method of Manufactured Solutions)
- Write *regression tests* that verify that the output of a program does not change over time

Software that is not tested does not produce the correct results!

(Note that I say "does not", and not "may not"!)

Verification & validation (V&V): Validation

Validation refers to the process of ensuring that the software solves a formulation that accurately represents the application:

"The program solves the correct problem"

The details of this go beyond this class.

Testing

Let me repeat the fundamental truth about software with more than a few 100 lines of code:

Software that is not tested does not produce the correct results!

No software that does not run lots of automatic tests can be good/usable.

As just one example (numbers as of 2011):

- deal.II runs ~2300 tests after every single change
- This takes ~10 CPU hours every time
- The test suite has another 250,000 lines of code.

Documentation

Documentation serves different purposes:

- It spells out to the developer what the *implementation* of a function/class is supposed to do (it's a *contract*)
- It tells a user what a function does
- It must come at different levels (e.g. functions, classes, modules, tutorial programs)

Also:

- Later reminds the author what she had in mind with a function
- Avoids that everyone has to ask the developer for information (bottleneck!)
- Document the history of a code by using a version control system

Social issues

Most HPC software is a collaborative effort. Some of the most difficult aspects in HPC are social:

- Can I modify this code?
- X just modified the code but didn't update the documentation and didn't write a test!
- Y1 has written a great piece of code but it doesn't conform to our coding style and he's unwilling to adjust it.
- Y2 seems clever but still has to learn. How do I interest her to collaborate without accepting subpar code?
- Z agreed to fix this bug 3 weeks ago but nothing has happened.
- M never replies to emails with questions about his code.

MATH 676

–

Finite element methods in scientific computing

Wolfgang Bangerth, Texas A&M University