Flash Center for Computational Science

The software development process of FLASH, a Multiphysics Simulation Code

Anshu Dubey, Katie Antypas, Alan Calder, Bruce Fryxell, Don Lamb, Paul Ricker, Lynn Reid, Katherine Riley, Robert Rosner, Andrew Siegel, Francis Timmes, Natalia Vladimirova, Klaus Weide,

SE-CSE May 18, 2013
FLASH’s Beginnings

- ASCI Center with delivery of a multi-physics code as a stated objective
- Intent to develop a single code usable for multiple applications
  - Thermonuclear runaways
    - Compressible reactive hydrodynamics
    - Specialized equation of state
    - Nuclear burning networks
  - AMR because of different scales in the physics
- Intent to release the code publicly
- Prometheus, PARAMESH and other research codes smashed together into one code
The Good
- Desire to use the same code for many different applications necessitated some thought to infrastructure and architecture
- Concept of alternative implementations, with a script for plugging different EOS – the setup tool
- Beginning of inheriting directory structure
- First release FLASH 1.6

The Bad
- F77 style of programming; Common blocks for data sharing
- Inconsistent data structures, divergent coding practices and no coding standards
And the ugly

Two camps
  Camp 1 – do it right, think about design and then build
  Camp 2 – do it right, enable science as soon as possible

For a while there were parallel efforts
  The two camps did not communicate

The resources were not enough for parallel efforts
  The science centric view won out
  Till today the scientists and developers involved only in that phase view only that as the right model

The saving grace – among the science centric developers there were some who were passionate about the open source model, and had a great deal of influence
Version 2: Data Inventory

- Address the worst of the bad in version 1
  - Eliminate common blocks
  - Inventory the data
  - Identify different variable types and classify them
  - Resulted in centralized database

- Enhance the good
  - Setup tool got enhanced
  - Config files got formalized

- New in this version – testing got formalized
  - Test-suite version 1
  - Run on multiple platforms

- Not much else changed in the architecture
Central Database Disadvantages

- Navigating the source tree became more confusing and Config file dependencies became more verbose.
- No possibility of data scoping; every data item was equally accessible to every routine in the code.
- When parsing a function, one could not tell the source of data.
- Lateral dependencies were further hidden.
- Overhead of database querying slowed the code by about 10-15%.
- The queries caused huge amount of code replication and source files became ugly.
- Encapsulation became nearly impossible.
Version 3: the Current Architecture

- Kept inheriting directory structure, configuration and customization mechanisms from earlier versions
- Defined naming conventions
  - Differentiate between namespace and organizational directories
  - Differentiate between API and non-API functions in a unit
  - Prefixes indicating the source and scope of data items
- Formalized the unit architecture
  - Defined API for each unit with null implementation at the top level
- Resolved data ownership and scope
- Resolved lateral dependencies for encapsulation
- Introduced subunits and built-in unit test framework
Version Transitions – 1 to 2

- The bias at the time – keep the scientists in control
- Keep the development and production branches synchronized
  - Enforced backward compatibility in the interfaces
  - Precluded needed deep changes
  - Hugely increased developer effort
  - High barrier to entry for a new developer
- Did not get adopted for production in the center for more than two years
  - Development continued in FLASH1.6, and so had to be brought simultaneously into FLASH2 too.
  - Database caused performance hit and IPA could not be done, so slower
Version Transitions 2 to 3

- Controlled by the developers
- Sufficient time and resources made available to design and prototype
- No attempt at backward compatibility
- No attempt to keep development synchronized with production
- All focus on a forward looking modular, extensible and maintainable code

Two very important factors to remember:
The scientists had a robust enough production code
The developers had internalized the vagaries of the solvers
The Methodology

- Build the framework in isolation from the production code base
- Infrastructure units first implemented with a homegrown Uniform Grid.
  - Helped define the API and data ownership
- Unit tests for infrastructure built before any physics was brought over
- Hydro and ideal gas EOS were next with one application
- Next was AMR: the application and the IO implementation were verified
- Test-suite was started on multiple platforms with various configurations (1/2/3D, UG/PARAMESH, HDF5/PnetCDF)
- This took about a year and a half, the framework was very well tested and robust by this time
The Methodology Continued …

- In the next stage the mature solvers (ones that were unlikely to have incremental changes) were transitioned to the code
  - Once a code unit became designated for FLASH3, no users could make a change to that unit in FLASH2 without consulting the code group.
- The next transition was the simplest production application (with minimal amount of physics)
- Scientists were in the loop for verification and in prioritizing the units to be transitioned at this stage
- FLASH3 was in production in the Center long before its official 3.0 release
  - The ugly had been addressed: the science centric view had given way to a more balanced one; took tremendous effort on the part of the center’s leaders
  - More mutual trust and respect
  - More reliable code; unit tests provided more confidence, and it was easier to add capabilities
Version 4

- Did not need any change in the architecture
- Primarily a capabilities addition exercise
- Mesh replication was easily introduced for multigroup radiation
- Expanded to other communities such as fluid-structure interaction because of existing Lagrangian framework and elliptic solver
- Has Chombo as an alternative mesh package, but for hydro only applications
Interdisciplinary Interactions

Prioritization

- whether good long term design or meet short term science objectives
- Both have their place
- Initial stages should be driven by science objectives
  - Too early for long term software design
  - Quick and dirty solutions with an eye to learning about code components and their interplay
- Once there is useable code, long term planning and design should occur
  - Willingness to make wholesale changes to the code at least once is necessary
  - At no stage should one lose sight of science objectives
Interdisciplinary Interactions

Partnership model

- Science users who recognize the code as a research instrument that needs its own research
- Even better if they are interested in the code
  - Flash early scientists were
- Developers and computer scientists interested in a product and the science being done with the code
  - Helps to have people with multidisciplinary training
- Comparable resources and autonomy for code group
  - And recognition of their intellectual contribution to scientific discovery
- Careful balance between long term and short term objectives
Lessons Learned

- Public Releases – every 8-10 months – forces discipline
  - Brings the code up to coding standards
  - Reconciles and refreshes the test suite
- Documentation – transient developer population
  - User support documentation
  - Extensive inline documentation
- Backward compatibility is overrated
- Uncluttered infrastructure is the best
- Supporting users is good, letting users drive the capability addition is even better
- Testing the code on multiple platforms is indispensable
- Allowing branches to diverge is a really bad idea
Some useful links

- http://flash.uchicago.edu/site/flashcode
- http://flash.uchicago.edu/site/flashcode/user_support/
- http://flash.uchicago.edu/site/publications/flash_pubs.shtml
- http://flash.uchicago.edu/site/testsuite/home.py