SOFTWARE PROCESS FOR FLASH, A CODE SERVING MULTIPLE SCIENTIFIC COMMUNITIES

ANSHU DUBEY
FLASH’S BEGINNINGS

- ASCI Center with delivery of a multi-physics code as a stated objective - 1998
- 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
FLASH ARCHITECTURE

- Philosophy:
  - Encode the logical separation of functional units into a framework
  - Encapsulate module, define data scope, arbitrate on data ownership
  - Define interfaces

- Requirements for infrastructure support:
  - AMR and Uniform Grid, with replication
  - IO and runtime parameters
  - Lagrangian Framework

- Solver requirements
  - Mixed hyperbolic, elliptic, and parabolic systems
  - 0D computations (EOS, burn)
  - Time-stepping
  - Lagrangian entities in various flavors
UNIT HIERARCHY

UnitMain
Common API implementation

UnitSomething
API implementation

Impl_1
Remaining API impl

Impl_2
Remaining API impl

Impl_3
Remaining API impl

Common Impl

kernel
FLASH SETUP SCRIPT: IMPLEMENTS ARCHITECTURE

Python code links together needed physics and tools for a problem

- Parses Config files to
  - Determine a self consistent set of units to include
  - If a unit has multiple implementations, finds out which implementation to include
  - Get list of parameters from units
  - Determines solution data storage

- Configures Makefiles properly
  - For a particular platform
  - For included Units

- Implements inheritance with unix directory structure
- Provides a mechanism for customization
Software architecture and framework research for CSE

- Inheriting directory structure
- Concept of alternative implementations, with a script for plugging different EOS – the setup tool
  - Together they formed the basis of object oriented infrastructure
- The Challenges
  - F77 style of programming; Common blocks for data sharing
  - Inconsistent data structures, divergent coding practices and no coding standards
- Testing introduced but not formalized
Data Inventory with centralized database

- Eliminate common blocks
- Identify different variable types and classify them
- Object oriented approached formalized
- Testing got formalized
  - Mostly regression tests, compared against approved results
  - Automating through python script
  - Run on multiple platforms
- Attempt to modularize, encapsulation still a challenge
- Beginning of documentation formalization
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

Defined contributions policies
Redesign of software architecture

- FLASH4 architecture unsuitable for heterogeneity
- Two axes of heterogeneity: platform and model
  - For details see (Dubey et al, Heteropar 2015 (with Europar 2015))
- Modernize software process
  - Move from SVN to Git with a defined workflow
  - Use Git tools for project management
  - Using continuous integration in addition to other forms of testing
- Inherit the remainder of the software process from earlier versions
VERSION 5 SOFTWARE ENGINEERING

Greater attention to software Process

- Primarily a capabilities and domain expansion exercise
- Policies about distribution, contribution
- Managing code consistency across multiple production branches and development branch
  - Merge schedule
  - Testing schedule
  - Division of responsibilities
AUDITING PROCESS

- Version Control
  - SVN until FLASH 4, current release
  - FLASH 5 development on github
- Test Suite in-house harness, predates Jenkins
- Online Coding Violation Tracking and issue tracking
  - Unfinished tasks, bugs, bad code, developer queries
- Profiling Tools
  - Built-in timers, logging
  - Hooks for external tools
- Documentation
  - Online documentation for Unit APIs -- ROBODOC
  - User’s guide in HTML and PDF
  - “Howto” available for developers, various platforms
  - Email users’ group for FLASH 4
VERIFICATION: THREE FLAVORS

- Continuous Integration on master in FLASH 5
- Nightly comprehensive testing in FLASH 4 and FLASH 5
  - Can also track daily code performance
- Developer run test-suite before merge to staging
- Targeted verification for simulation campaign
  - Pathfinder runs
  - Correctness and feasibility, resource estimate

**FlashTest Invocations**

- **Platform**
- **Date of run**
- **Floating statistics box** gives immediate overview of results
- **Green light** indicates all runs were successful
- **Red light** indicates 1 or more tests failed
DESIGN INVESTMENT IMPACT

Table 4: Correlation between effort, duration released, and publications.

<table>
<thead>
<tr>
<th>Domain</th>
<th>initial effort (person years)</th>
<th>total effort (person years)</th>
<th>release date (year)</th>
<th>first publication (year)</th>
<th>total publications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Supernova, Astro</td>
<td>16</td>
<td>60</td>
<td>2000</td>
<td>2000</td>
<td>427</td>
</tr>
<tr>
<td>Cosmology</td>
<td>2.5</td>
<td>20</td>
<td>2001</td>
<td>2001</td>
<td>257</td>
</tr>
<tr>
<td>Solar physics</td>
<td>0.5</td>
<td>5</td>
<td>2000</td>
<td>2002</td>
<td>70</td>
</tr>
<tr>
<td>HEDP</td>
<td>8</td>
<td>28</td>
<td>2010</td>
<td>2010</td>
<td>38</td>
</tr>
<tr>
<td>Core-collapse</td>
<td>1</td>
<td>5</td>
<td>2012</td>
<td>2008</td>
<td>20</td>
</tr>
<tr>
<td>CFD/FSI</td>
<td>3</td>
<td>5</td>
<td>2012</td>
<td>2012</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>capabilities</th>
<th>categories</th>
<th>new community reached</th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>base</td>
<td>all</td>
<td>thermonuclear astrophysics</td>
<td>2000</td>
</tr>
<tr>
<td>MHD</td>
<td>physics</td>
<td>reconnection, solar plasma</td>
<td>2002</td>
</tr>
<tr>
<td>particles</td>
<td>physics and infrastructure</td>
<td>cosmology</td>
<td>2003</td>
</tr>
<tr>
<td>multigrid</td>
<td>infrastructure</td>
<td>CFD</td>
<td>2008</td>
</tr>
<tr>
<td>Lagrangian Markers</td>
<td>infrastructure</td>
<td>FSI</td>
<td>2009</td>
</tr>
<tr>
<td>PIC</td>
<td>physics and infrastructure</td>
<td>plasma</td>
<td>2010</td>
</tr>
<tr>
<td>nuclear EOS, neutrino source terms and leakage</td>
<td>physics</td>
<td>core-collapse supernovae</td>
<td>2010-2012</td>
</tr>
<tr>
<td>3-T, conductivity Radiation, laser</td>
<td>physics and infrastructure</td>
<td>HEDP</td>
<td>2010-2011</td>
</tr>
<tr>
<td>sink particles</td>
<td>physics</td>
<td>star formation</td>
<td>2012</td>
</tr>
</tbody>
</table>
LESSONS LEARNED

- Baking separation of concerns into the design is an investment that keeps giving
  - Version transitions can be localized
- Uncluttered infrastructure is the best
- Code pruning is an extremely good idea
- Public Releases – every 8-10 months – forces discipline
- Documentation – transient developer population
  - User support documentation
  - Extensive inline documentation
- 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
Questions ?