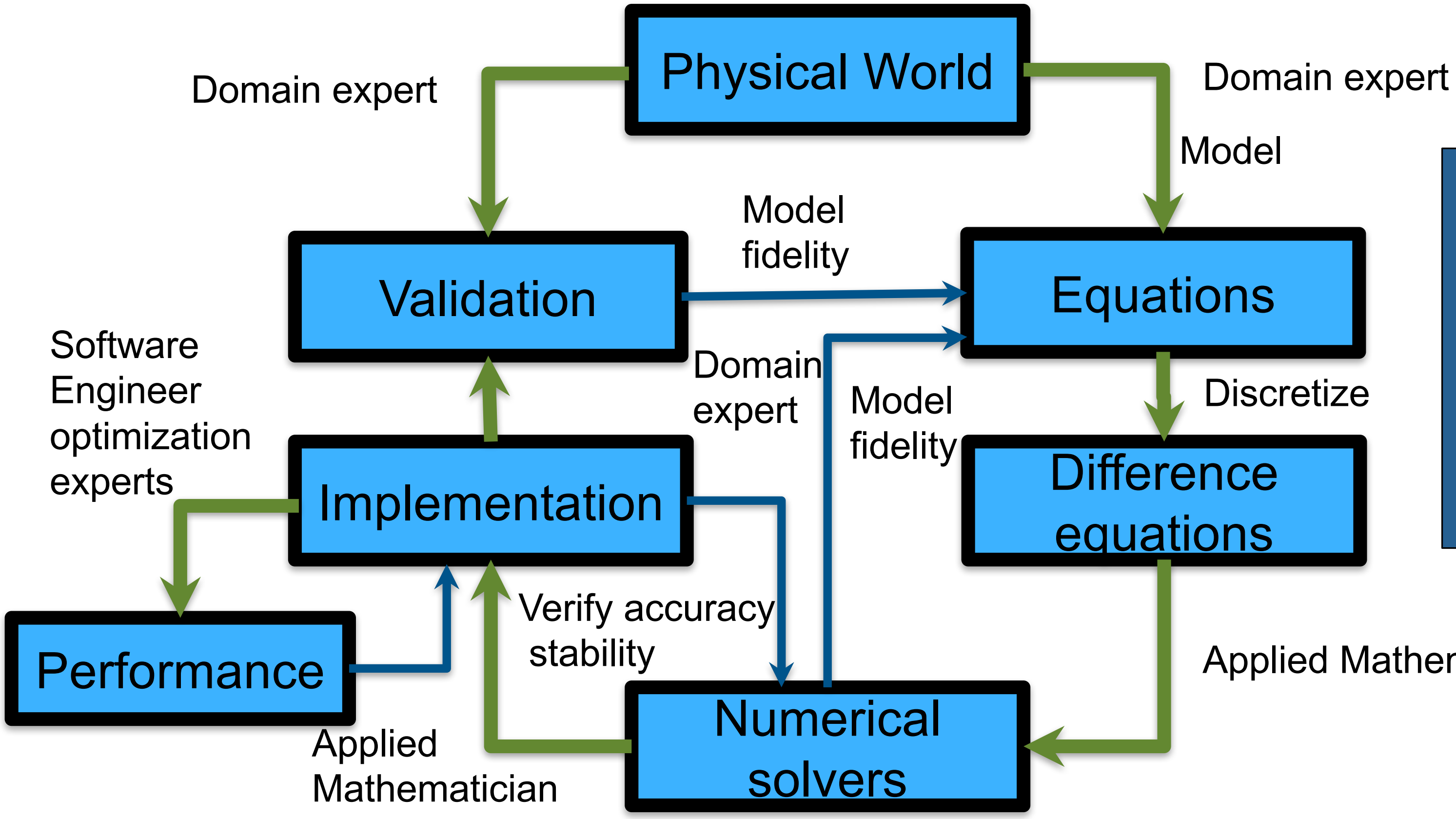
Software Lifecycle with an Example, Community Impact

Presented to
ATPESC 2017 Participants

Anshu Dubey
Computer Scientist, ANL

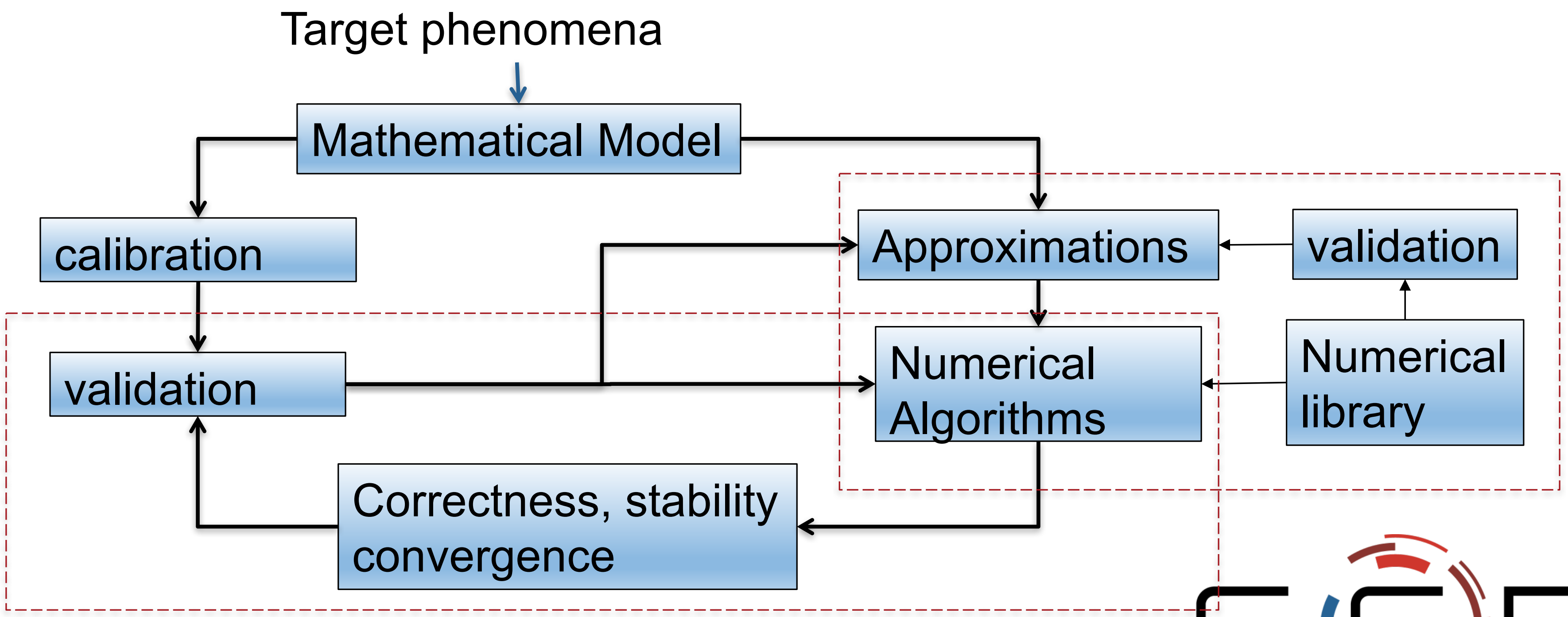
Q Center, St. Charles, IL (USA)
Date 08/09/2017

Simplified Schematic of Science Through Computation

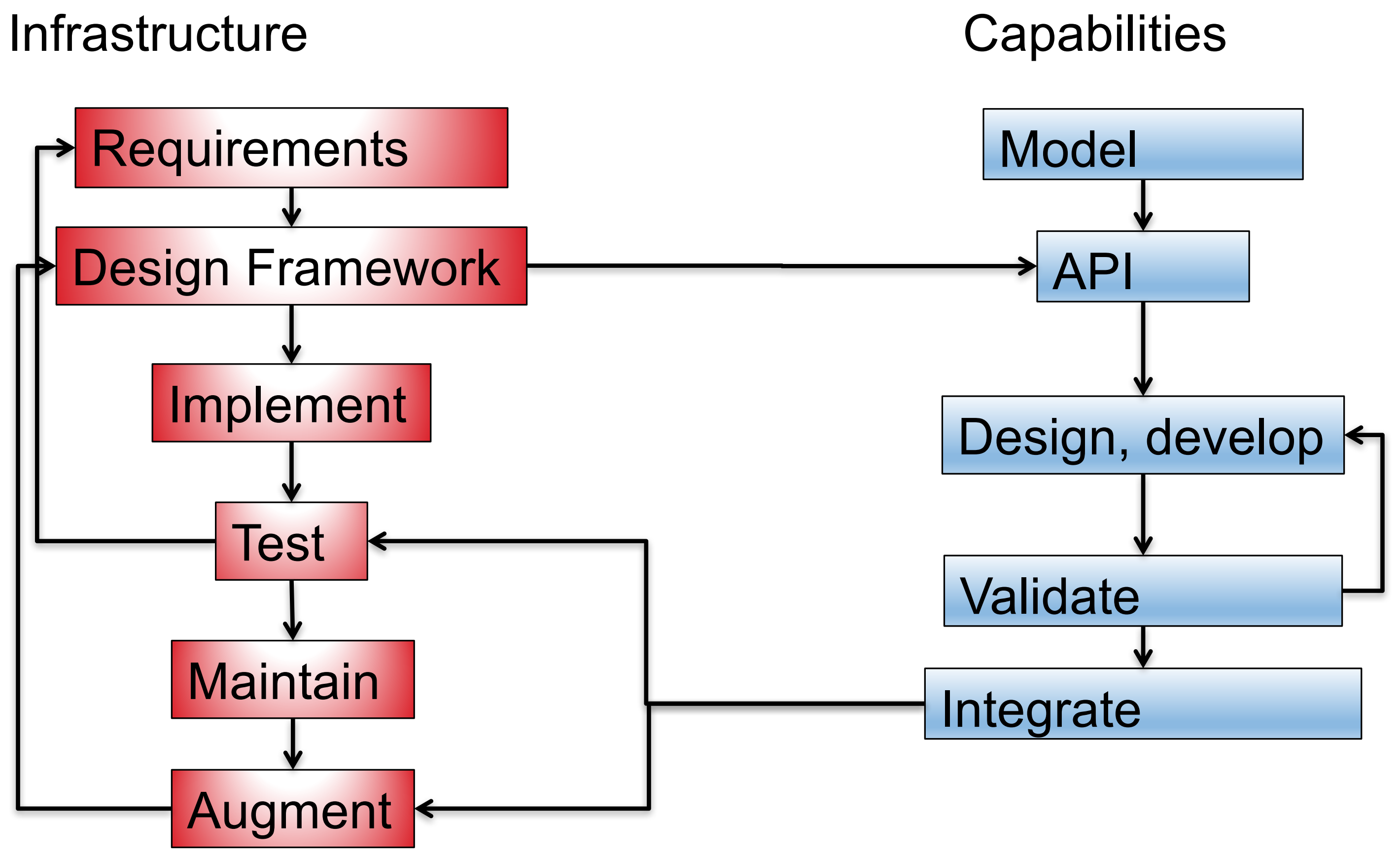


Many stages in the lifecycle have components that may themselves be under research => need modifications

One View of Lifecycle



Coupling Between Infrastructure and Physics



Architecting Scientific Codes: Taking Stock

- Software architecture and process design is an overhead
  - Value lies in avoiding technical debt (future saving)
  - Worthwhile to understand the trade-off
- The target of the software
  - Proof-of-concept
  - Verification
  - Exploration of some phenomenon
  - Experiment design
  - Product design
  - Analysis
  - Other …
- Target should dictate the rigor of the design and software process
  - Cognizant of resource constraints

In this lecture focus is on the needs of large multicomponent codes used for exploration and experiment design. Others need a subset.

Architecting Scientific Codes: Desirable Characteristics

- Extensibility
  - Most uses need additions and/or customizations
- Portability
  - Platforms change
  - Even the same generation platforms are different
- Performance and Performance portability
  - All machines need to be used well
- Maintainability and verifiability
  - For believable results

Architecting Scientific Codes : Implications

- Extensibility and maintainability require
  - Well defined structure and modules
  - Encapsulation of functionalities
  - Understanding data ownership
    - Arbitration among different functionalities
- Performance requires
  - Increasing spatial and temporal locality of data
  - Minimizing data movement
  - Maximizing scalability
- Portability requires general solutions that work across platforms
  - Performance portability requires that they work efficiently without significant manual intervention across platforms
- Verifiability requires
  - Verifiable output for critical components
  - Tests at various granularities

Architecting Scientific Codes : The Reality

- Where it gets messy
  - Well defined structure and modules
    - Same data layout not good for all solvers
    - Many corner cases (branches, other special handling)
  - Encapsulation of functionalities
    - Necessary lateral interactions
  - Minimization of data movement
    - Necessity of transposition / other form of copy
  - Maximization of locality and scalability
    - Solvers with low arithmetic intensity but hard sequential dependencies
    - Proximity and work distribution at cross purposes

Architecting Scientific Codes: Taming Complexity

- Differentiate between types of functionalities in the code
  - Model and the numerics may be subject of research
    - May need updates and extensions fairly regularly
  - Discretizations and other support services are more stable
    - I/O, parameter management etc
- Handle them differently
  - Good to hide implementation complexity from variable sections
    - Data structure and movement management
    - Where possible treat mathematically complex sections as client code
- First step in separation of concerns (more later)

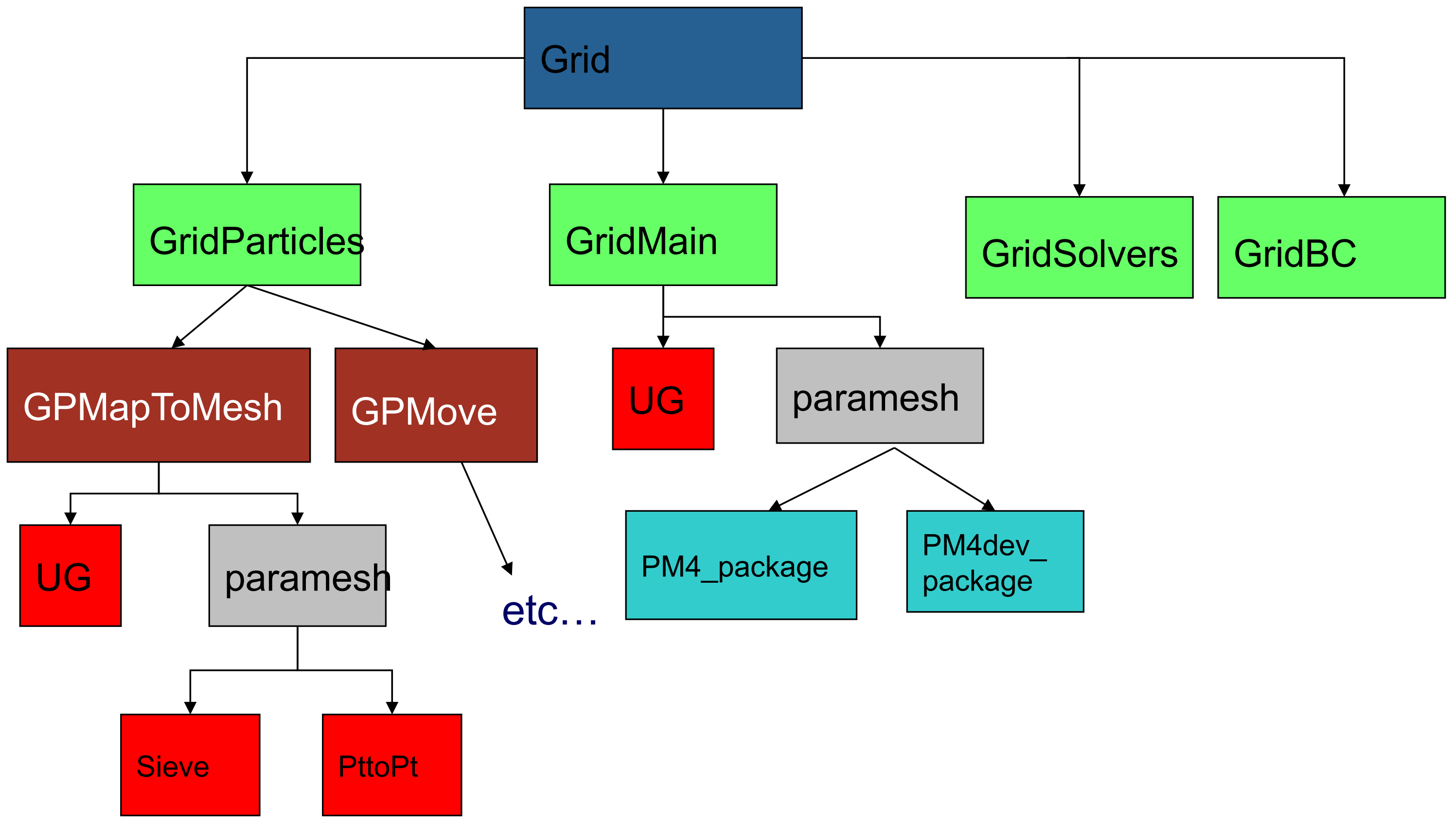
Taming Functional Complexity

- Identify logically separable functional units of computation
  - Infrastructure
  - Solvers
  - Monitors
- Encode the logical separation (modularity) into a framework
  - Infrastructure units being the backbone
  - Handle all global scope data
- Separate what is exposed from what is private to the module
- Define interfaces through which the modules can interact with each other

Example from FLASH : Five Categories of Units

- Infrastructure – Grid, I/O ….
  - Grid owns discretized mesh, state variables
  - Manages decomposition and scaling
- Physics units – hydro, gravity ….
  - Implement specific operators
  - Request Grid for data to operate on
  - Do not differentiate between whole domains and a subset of domain
- Monitoring units – logfile, timers …
  - Monitor the progress and performance
- Driver
  - Implement operator splitting and time integration
- Simulation
  - Unique to FLASH, where the application is put together
  - Also where customizations occur

Example of a Unit – Grid (From FLASH)



Example of Unit Design

- Non trivial to design several of the physics units in ways that meet modularity and performance constraints.
- Eos (equation of state) unit is a good example
  - Individual mesh points are independent of each other
  - There are several reusable calculations
  - Other physics units demand great flexibility from it
    - single grid point
    - only the interior cells, or only the ghost cells
    - a row at a time, a column at a time or the entire block at once
    - different grid data structures, and different modes at different times
  - Implementations range from simple ideal gas law to table look up and iterations for degenerate matter and plasma, with widely differing relative contribution in the overall execution time
  - Relative values of overall energy and internal energy play role in accuracy of results
  - Sometimes several derivative quantities are desired as output

Preparing For Future: The New Challenges

- Last two decades machine model stable and uniform
  - Clusters: Distributed memory machines
  - Many algorithmic optimizations applied across the board
  - Portability often guaranteed performance portability
- Before that there was another long term stable model
  - Vector machines
  - Transition from vector to parallel
    - Codes still relatively small
    - Move from one stable paradigm to another
    - Code lifecycle >>> transition time

- Now there is explosion in heterogeneity
  - Machines, models and solvers
  - Portability is not the same as performance portability
  - Optimal for one may be terrible for another

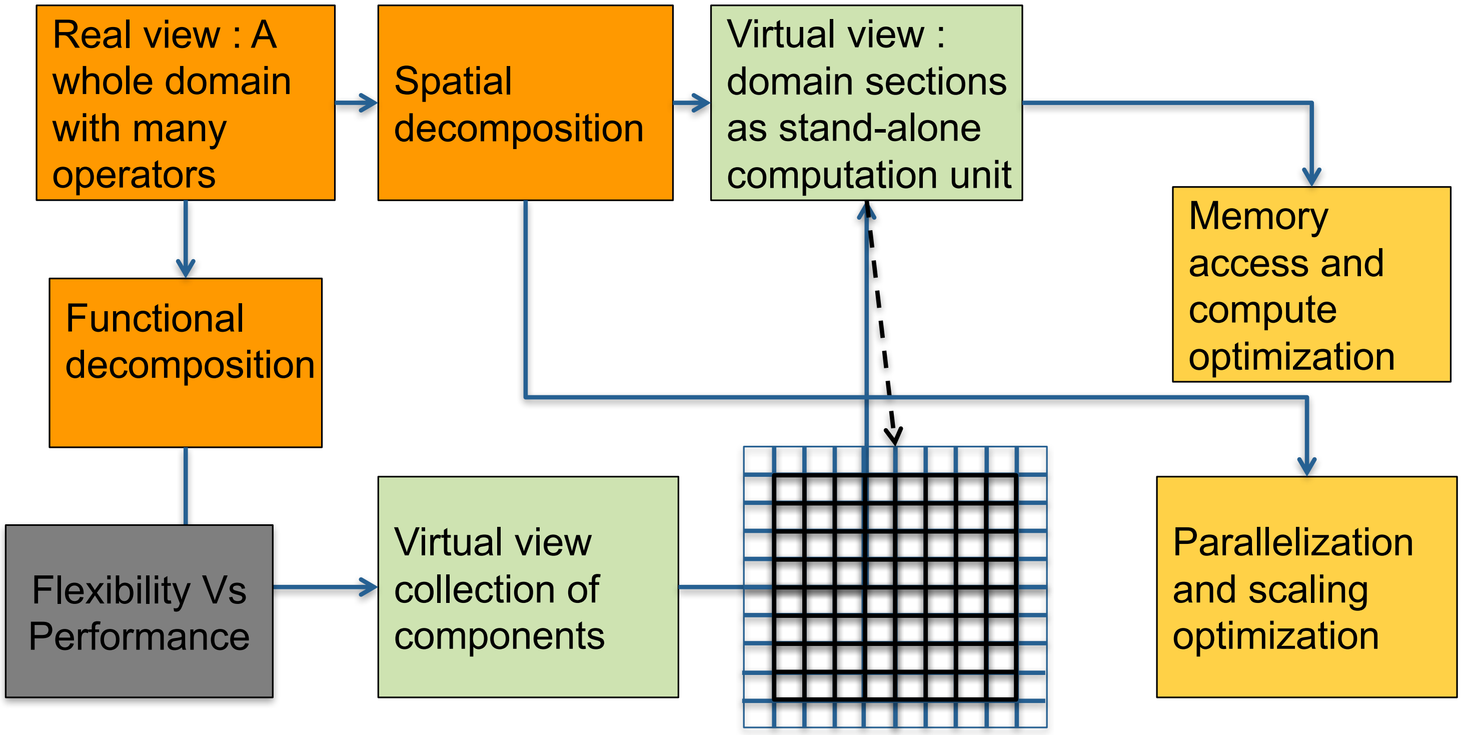
Preparing for future :Legacy challenges

- Much larger codes
  - Transition time much longer than before
  - Platform life <<< code lifecycle
  - Platform life ~= transition time
  - Same generation has different platforms
- No single machine model to target
- Need to deepen parallel hierarchy and lift abstraction
  - Let abstraction and middle layers do the heavy lifting for portability
  - Many ideas, little convergence.

Overarching Theme

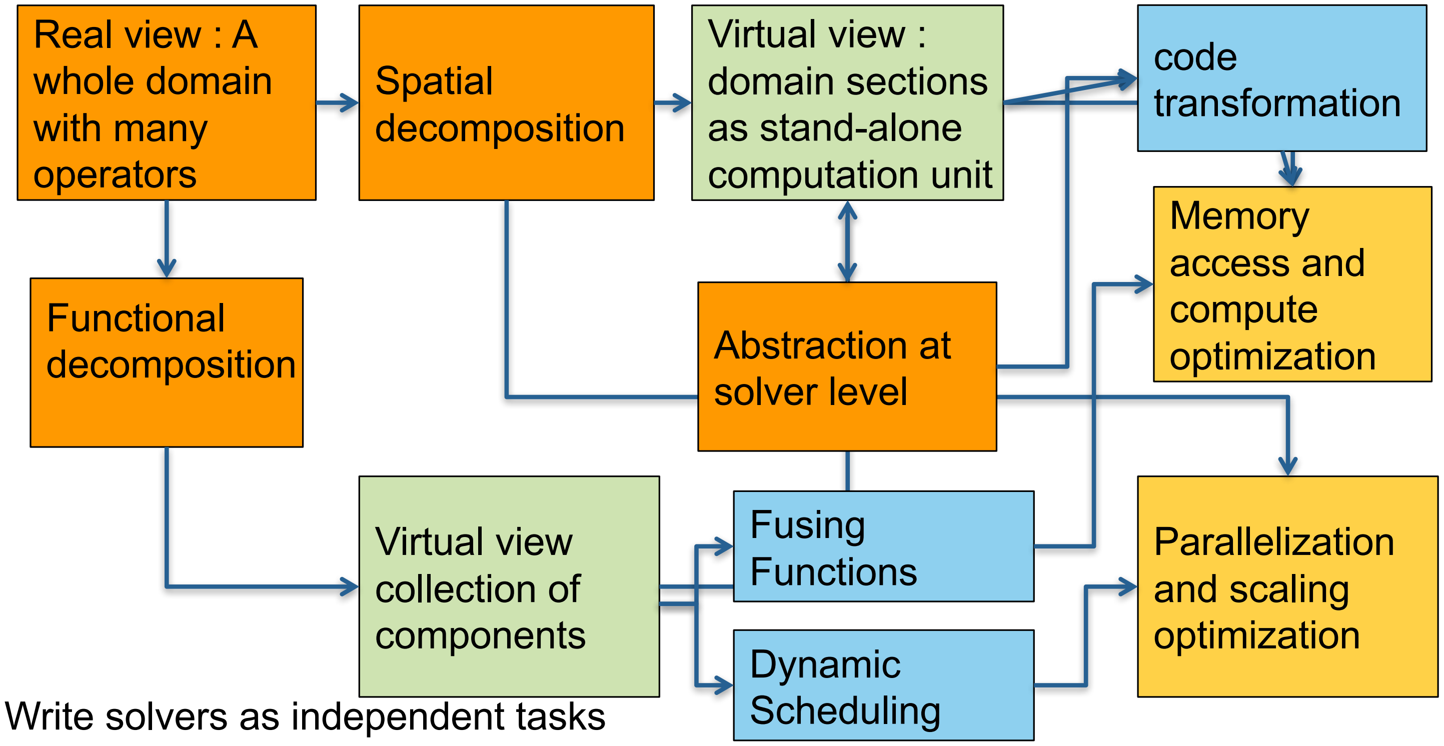
- Differentiate between physical view and virtual view
- Simpler world view at either end enables separation of concerns
- Hard-nosed trade-offs

Example: PDE's


Real view : A whole domain with many operators
Spatial decomposition
Virtual view : domain sections as stand-alone computation unit
Memory access and compute optimization
Functional decomposition
Flexibility Vs Performance
Virtual view collection of components
Parallelization and scaling optimization

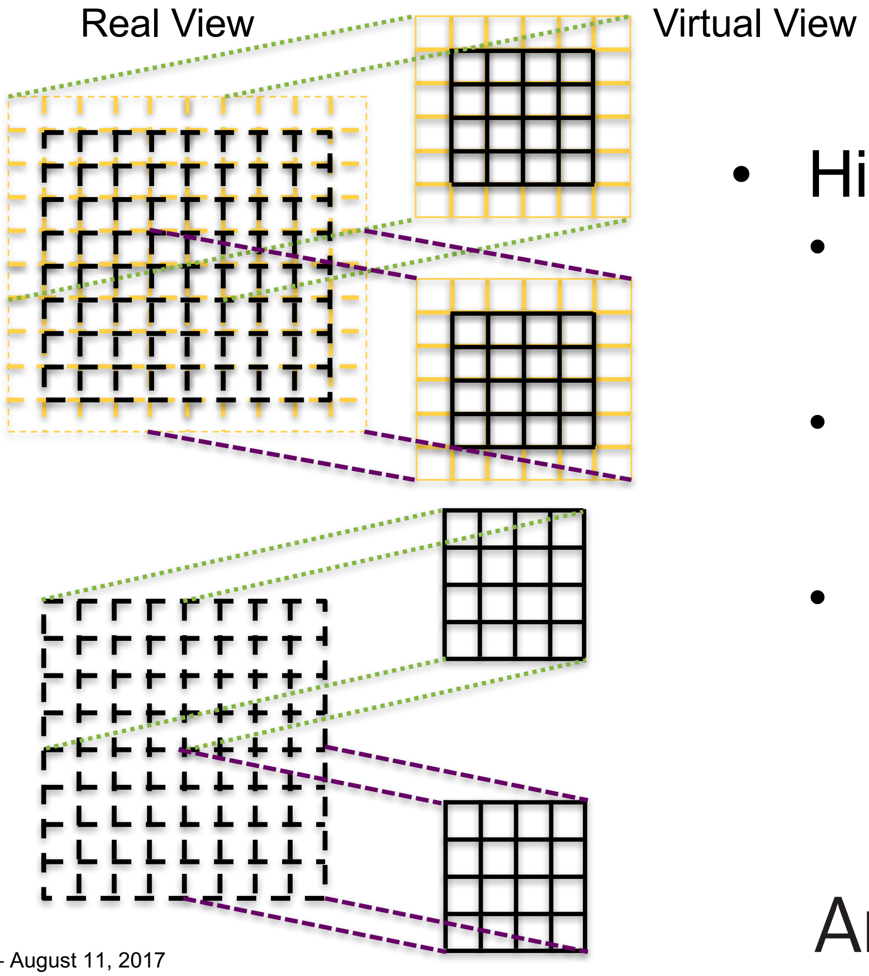
Customizations can be hidden under the virtual views as needed

Example: PDE's


Real view : A whole domain with many operators
Spatial decomposition
Virtual view : domain sections as stand-alone computation unit
code transformation
Functional decomposition
Abstraction at solver level
Memory access and compute optimization
Virtual view collection of components
Fusing Functions
Dynamic Scheduling
Parallelization and scaling optimization

Write solvers as independent tasks
Explicitly call out dependencies
Expose fusion possibilities

Hierarchical Decomposition



Tiling

- Hierarchy of tiling:
  - Larger non-overlapping tile maps to coherence domain
  - Smaller overlapping tile exposes more parallelism
  - Parameterize end-points, shape tiles as needed

Asynchronous Execution

- Barriers are the easy way to reconcile dependencies
  - Take away the option of pipelining and/or overlapping
- With hierarchical spatial and functional decomposition rich collection of tasks
  - Articulate dependencies explicitly
  - Let the framework find the unit of computation that is ready and hand it to client code with all the necessary data
    - Under the hood, framework can be managing dependencies
    - If client code assumes not-in-place update each of the tiles is a task with neighborhood dependencies
- Can be made into build or run environment specifications through appropriate parameterization

Putting it all Together

- The construction of operators
  - Express computation in the form of stencil operators or other appropriate abstraction
  - Specify the part of the domain, and the conditions under which the operators apply
    - Use masks to take care of branching
- Mix-mode parallelism
  - Parameters to control the degree of tiling or other forms of mix-mode parallelism
    - Could be handed to the compiler when technology arrives
  - Framework forms the data containers
- Dynamic tasking
  - Smarter iterators that are aware of mix-mode parallelism and dependencies
  - The iterating loops give up control and do while loops

Software Process Design : Taking stock

- Different needs for different scope projects
- Blindly applying processes is bad for productivity
- Rigor and extent of the process should reflect the needs
  - Never lose sight of overhead and trade-offs
  - Primary focus is productivity
  - If a process has no advantage, increases burden, drop it
  - If you incur technical debt from ignoring process, make sure to adopt one
- Small teams need few formal practices, large teams need more, diverse teams need most
- Some processes should be there irrespective of the size
  - Repository
  - Strong verification
  - Algorithm and implementation documentation

Map Team Needs to Process Scope: Small Projects

- Software for internal use of a small academic team
  - The minimum set described earlier

- Software for internal use of medium to large team: no public release
  - Simple software architecture
    - Encapsulation and data arbitration
    - Interfaces, coding standards
  - Ongoing verification (automatic testing)
  - Some usage documentation
  - Range of validity documentation
  - Additional policies
    - Testing responsibilities
    - Contribution policies

Map team needs to process scope:Large software with many interoperating moving parts

- Software architecture
  - Separation of concerns
  - Well defined APIs
  - Infrastructure backbone

- Software process : all from previous slide + a few more
  - Release and licensing
    - Frequency of release
    - Testing for release
    - Distribution model
  - User support, maybe training
  - Contribution policies
    - Gatekeeping
    - Resolution of tension between IP protection and open source

Community Codes : Motivation

- 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

- 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

Models: “Cathedral and the Bazaar”, Eric S. Raymond

- **The *Cathedral* model**
  - Code is available with each software release
  - Development between releases is restricted to an exclusive group of software developers.
    - GNU Emacs and GCC are presented as examples.
  - Central control models

- **The *Bazaar* model**
  - Code is developed over the Internet in view of the public.
  - Raymond credits Linus Torvalds, leader of the Linux kernel project, as the inventor of this process.
  - Distributed control models

Scientific codes

- Mostly follow the cathedral model
- Many reasons are given, some valid, others spring from bias
- The valid ones
  - The code quality becomes hard to maintain
  - Hard to find financial support for gate keeping and general maintenance
  - Typical user communities are too small to effectively support the bazaar model
  - The reward structure for majority of potential contributors is incompatible
- The not so valid ones
  - Codes are far too complex
  - Competitive advantage from owning the code

The real reason many times is simply the history of the development of the code and the pride of ownership

The Benefits of the Bazaar model

- Given a large enough beta-tester and co-developer base, almost every problem will be characterized quickly and the fix will be obvious to someone
  - More varied test cases that demonstrate bugs
  - Debugging can be effectively parallelized.
  - The infrastructure limitations are quickly exposed
- Capability addition is rapid, codes can do more
  - A corollary to that is a good extensible design
  - Users always want something more and/or something different from what is available
  - Greater knowledge pool operating together, more possibility of innovation

The Pitfalls of the Bazaar model

- Many of the benefitting reasons can equally easily go the other way
  - Bigger knowledge pool can also mean more conflicting opinions
  - Prioritizations can become extremely challenging
- Gatekeeping can become a huge challenge for maintaining software quality
  - Scientific codes have their own peculiarities for verification and validation that can be extremely challenging
  - The orchestration of capability combination is harder when there is physics involved because many times it just won't play well together

Scientific Community Codes Can Follow Several Different Paths :

- The most common path
  - Someone wrote a very useful piece of code that several people in the group started using
  - Collaborations happened
  - People moved and took the code with them
  - Critical mass of users achieved, code becomes popular
- No focused effort to build the code
  - Usually very little software process involved
  - For the whole code, limited shelf life

A More Sustained Path

- Sometimes enough like minded people take it a step further
  - Some long term planning resulting in better engineered code
  - Thought given to extensibility and for future code growth
  - As the code grows so does its community supported model
- This model is still relatively rare.
  - The occurrences are increasing

A Desirable Path

- Explicit funding to build a code for a target community
- Implied support for the design phase
- The outcome is expected to be long lasting and well engineered
- The occurrences are even rarer
- And it is getting increasingly harder
- When it works outcome is more capable and longer lasting codes

Some Flourishing Communities

- ❑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

Let us examine what does it take to build a community code

Community Building

- Popularizing the code alone does not build a community
- Neither does customizability – different users want different capabilities

So what does it take ?

- Enabling contributions from users and providing support for them
- Including policy provisions for balancing the IP protection with open source needs
- Relaxed distribution policies – giving collective ownership to groups of users so they can modify the code and share among themselves as long as they have the license

More inclusivity => greater success in community building
An investment in robust and extensible infrastructure, and a strong culture of user support is a pre-requisite

Contribution Policies

- Balancing contributors and code distribution needs
    - Contributor may want some IP protection
- Maintainable code requirements
    - – The minimum set needed from the contributor
        - Source code, build scripts, tests, documentation
- Agreement on user support
    - – Contributor or the distributor
- Add-ons: components not included with the distribution, but work with the code

Survey of ideas use-cases

IDEAS scientific software productivity project: www.ideas-productivity.org

- Five application codes and four numerical libraries
- All use version control, and all but one use distributed version control
- Builds are evenly divided between GNU make and CMake
- All provide documentation with some form of user's guide, many use automated documentation generation tools
- All have testing in some form, a couple do manual regression testing, the rest are automated
- Roughly half make use of unit testing explicitly
- Majority are publicly available

Summary From community codes workshop (2012)

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

- Codes – FLASH, Cactus, Enzo, ESMF, Lattice QCD code-suite, AMBER, Chombo, and yt
- Software architecture is almost always in the form of composable components
  - Need for extensibility
- All codes have rigorous auditing processes in place
- Gatekeeping for contributions, though models are different
- All codes have wide user communities, and the communities benefit from a common highly exercised code base

Communities headed towards community codes

- Climate modeling
  - DOE effort – ACME
    - Started in 2014
    - Community Model – many groups
    - Many practices in place

- Accelerator
  - People thinking about it
    - Whitepapers
  - Objective – avoid duplication, get some convergence,
    - Also more believable results

Common Threads

- Open source with a governance structure in place
- Trust building among teams
- Commitment to transparent communications
- Strong commitment to user support
- Either an interdisciplinary team, or a group of people comfortable with science and code development
- Attention to software engineering and documentation
- Understanding the benefit of sharing as opposed to being secretive about the code