Argonne NATIONAL LABORATORY

SOFTWARE AND PROCESS DESIGN

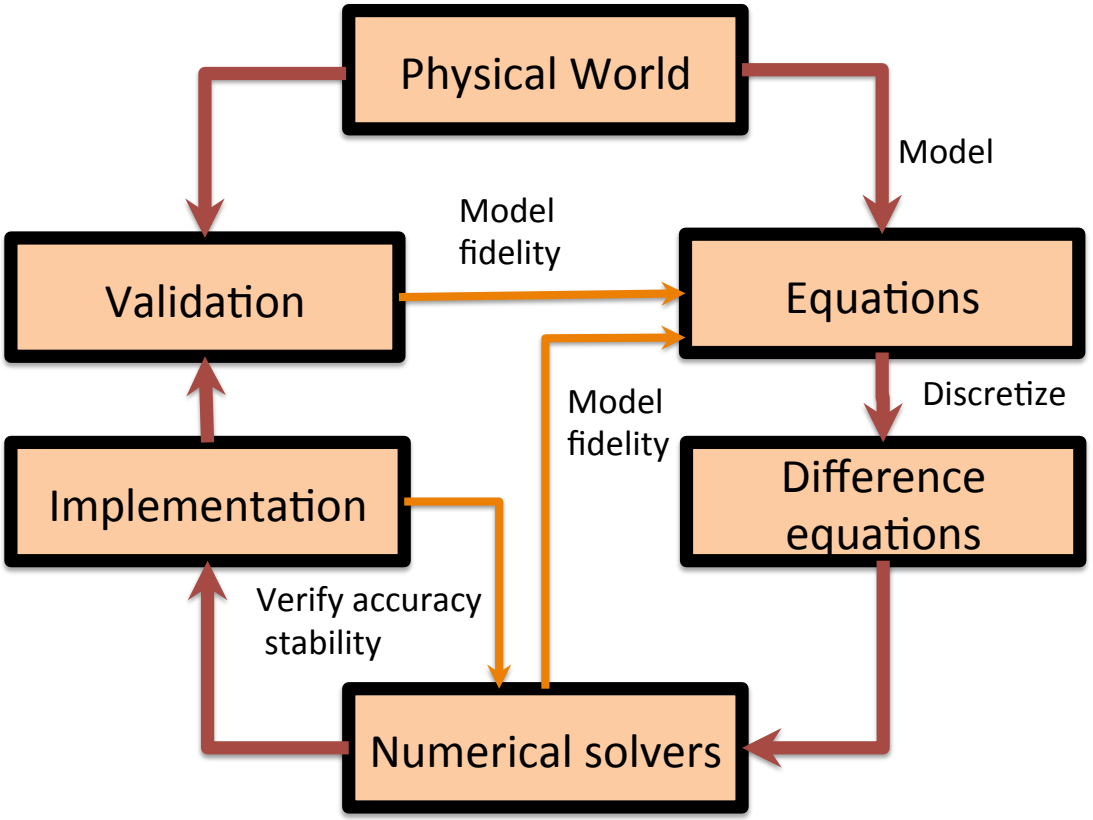
ANSHU DUBEY
Mathematics and Computer Science Division
Argonne National Laboratory

August 9, 2016
ATPESC
St. Charles, IL

CODE ARCHITECTURE DESIGN

SIMPLIFIED SCHEMATIC OF SCIENCE THROUGH COMPUTATION

This is for simulations, but the philosophy applies to other computations too.



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

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

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

- 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

- 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

How to tame the 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)

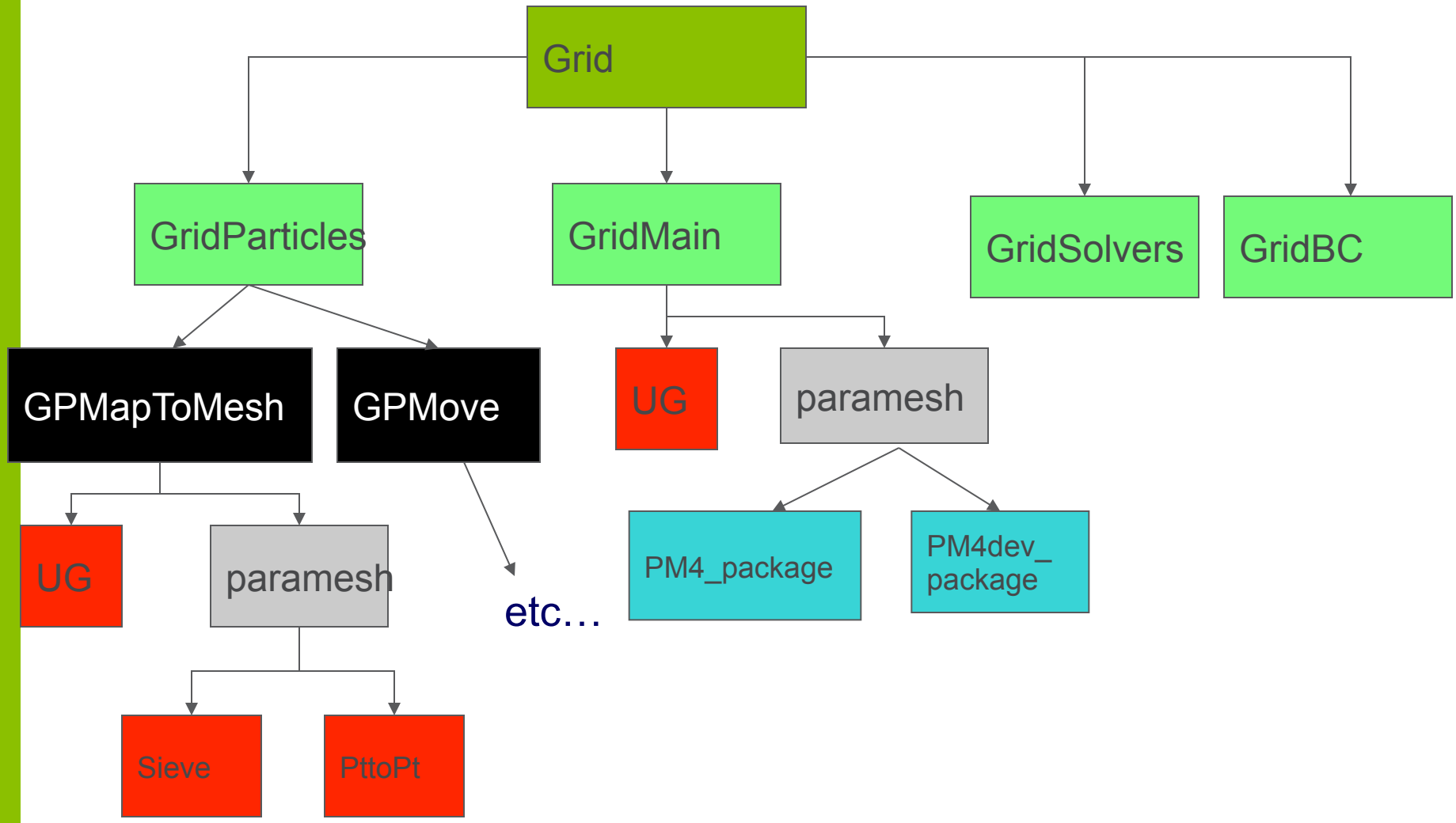
TAMING FUNCTIONAL COMPLEXITY

- 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

- FLASH has five categories of units
  - Infrastructure – Grid, I/O, Runtime ….
    - Grid owns discretized mesh, state variables
    - Manages decomposition and scaling
  - Physics units – hydro, eos, 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

EOS INTERFACE DESIGN

- Hierarchy in complexity of interfaces
  - For single point calculation scalar input and output
  - For sections of a block or full block vectorized input and output
    - wrappers to vectorize and configure the data
    - select derivative quantities to return with masking
- Different levels in the hierarchy give different degrees of control to the client routines
  - Most of the complexity is completely hidden from casual users
  - More sophisticated users can bypass the wrappers for greater control
- Done with elaborate machinery of masks and defined constants

OTHER CODES AND RESOURCES

https://www.cct.lsu.edu/research/cyber-advancement/cactus
http://flash.uchicago.edu/site/flashcode
https://computation-rnd.llnl.gov/SAMRAI
http://ambermd.org
https://www.earthsystemcog.org/projects/esmf
https://commons.lbl.gov/display/chombo

R. Armstrong, G. Kumfert, L. McInnes, S. Parker, B. Allan, M. Sottile, T. Epperly, T. Dahlgren, The CCA component model for high-performance scientific computing, Concurrency and Computation: Practice and Experience 18 (2) (2006) 215–229.

P. Hovland, K. Keahey, L.C. McInnes, B. Norris, L.F. Diachin, P. Raghavan, A quality of service approach for high-performance numerical components, in: Proceedings of Workshop on QoS in Component-Based Software Engineering, Software Technologies

Conference, Toulouse, France, 2003.

D. Worth, C. Greenough, A survey of available tools for developing quality software using Fortran 95. Technical report RAL-TR-2005, SFTC Rutherford

Appleton Laboratory, SESP Software Engineering Support Programme,

2005. <http://www.sesp.cse.clrc.ac.uk/html/Publications.html>.

PERFORMANCE PORTABILITY CHALLENGE

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

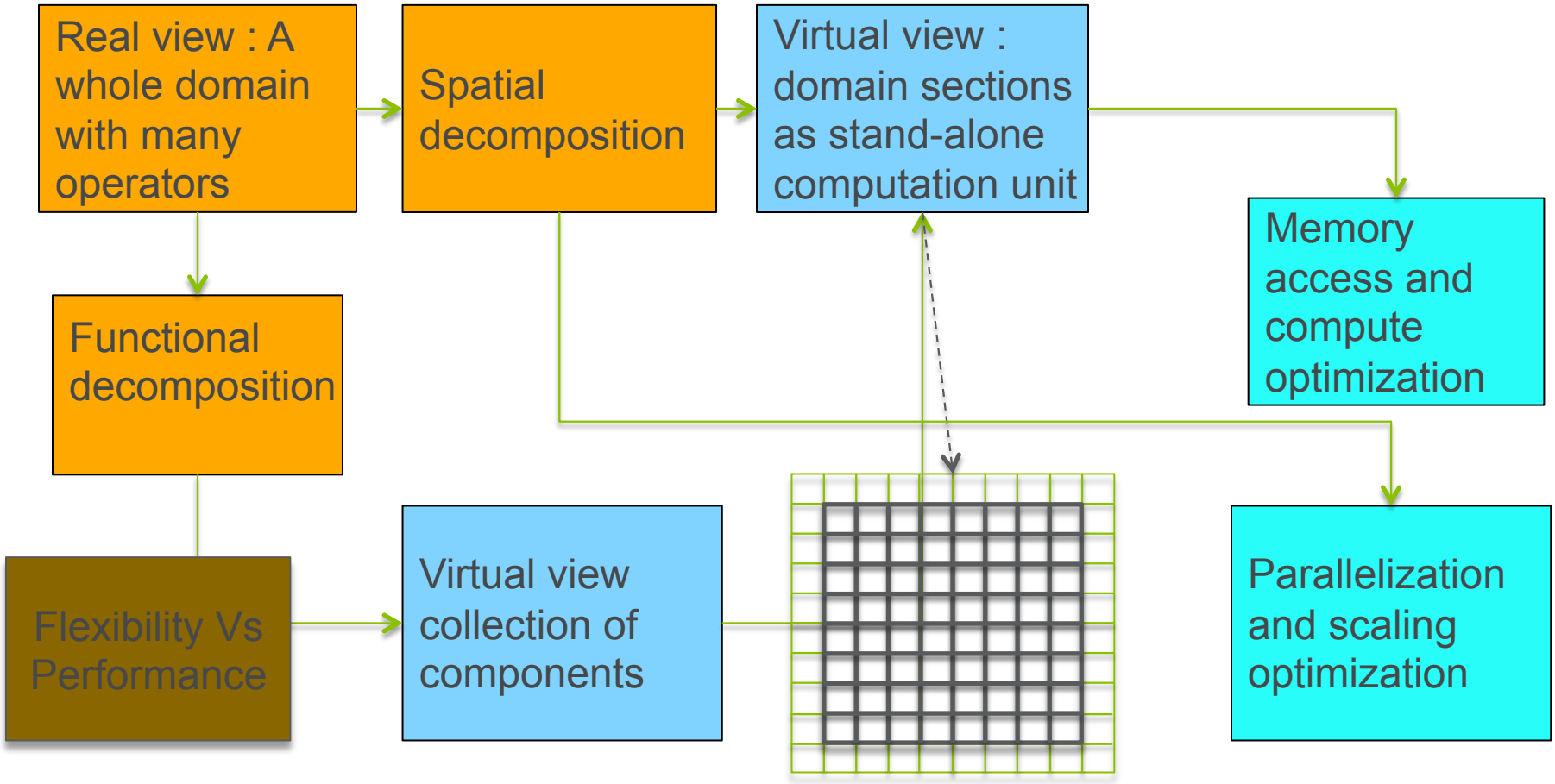
- No single machine model to program to

- 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

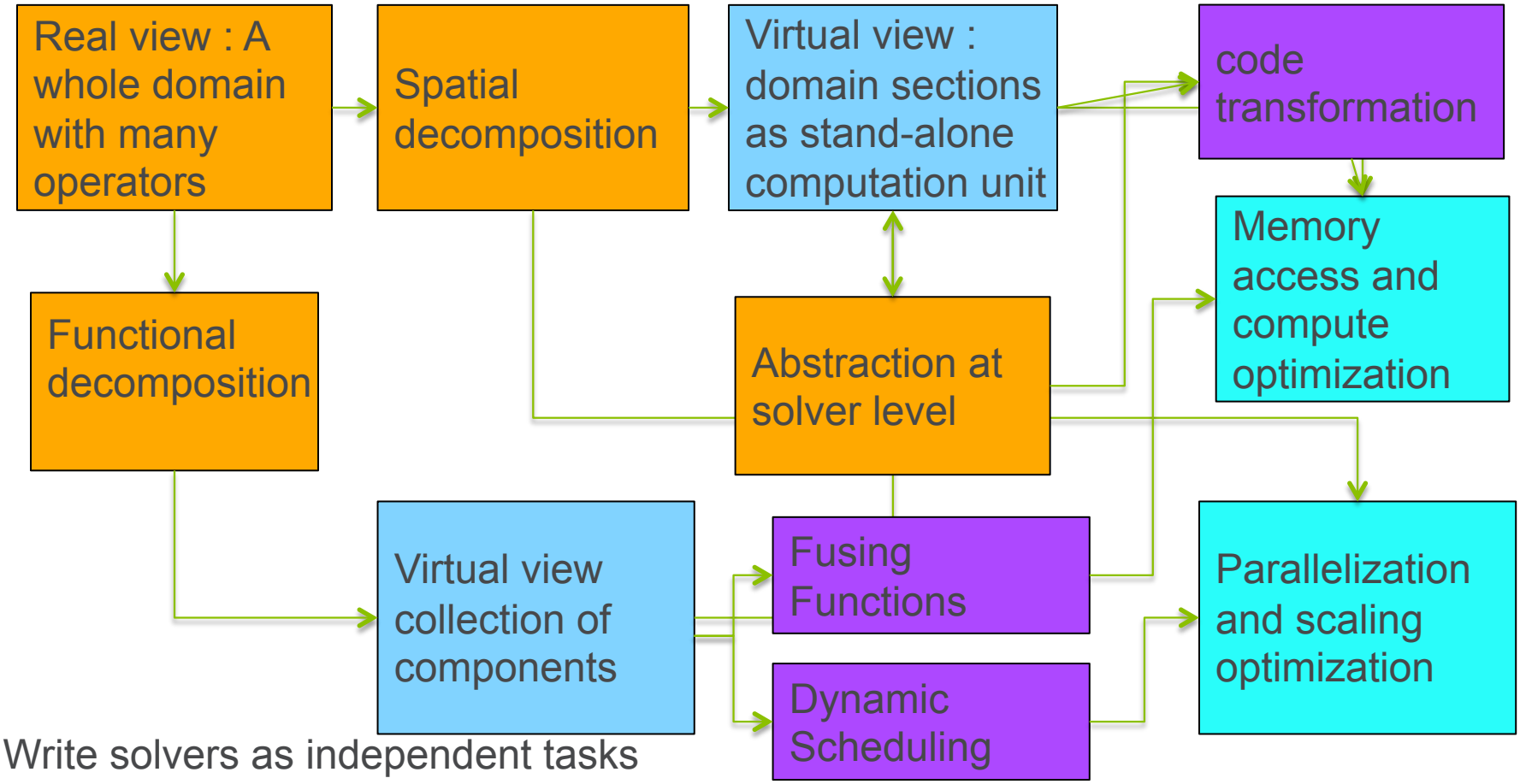
- Differentiate between physical view and virtual view
- Simpler world view at either end enables separation of concerns
- Hard-nosed trade-offs

EXAMPLE: PDE'S



Customizations can be hidden under the virtual views as needed

EXAMPLE: PDE'S

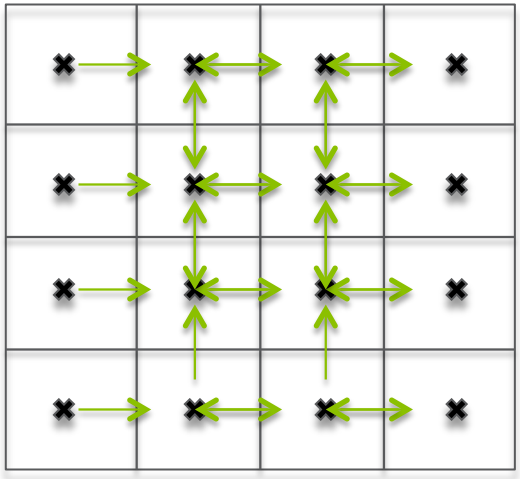


Write solvers as independent tasks
Explicitly call out dependencies
Expose fusion possibilities

Dubey and Graves, *A Design Proposal for a Next Generation Scientific Software Framework*, HeteroPar 2015

SOLVER LEVEL ABSTRACTIONS

- Stencil DSLs
  - A stencil operator is a collection of shifts with corresponding coefficients
  - Applying the stencil operator
    - Weighted sum of some points on the mesh
    - Offset specified by the shift relative to the target



<-1,0>

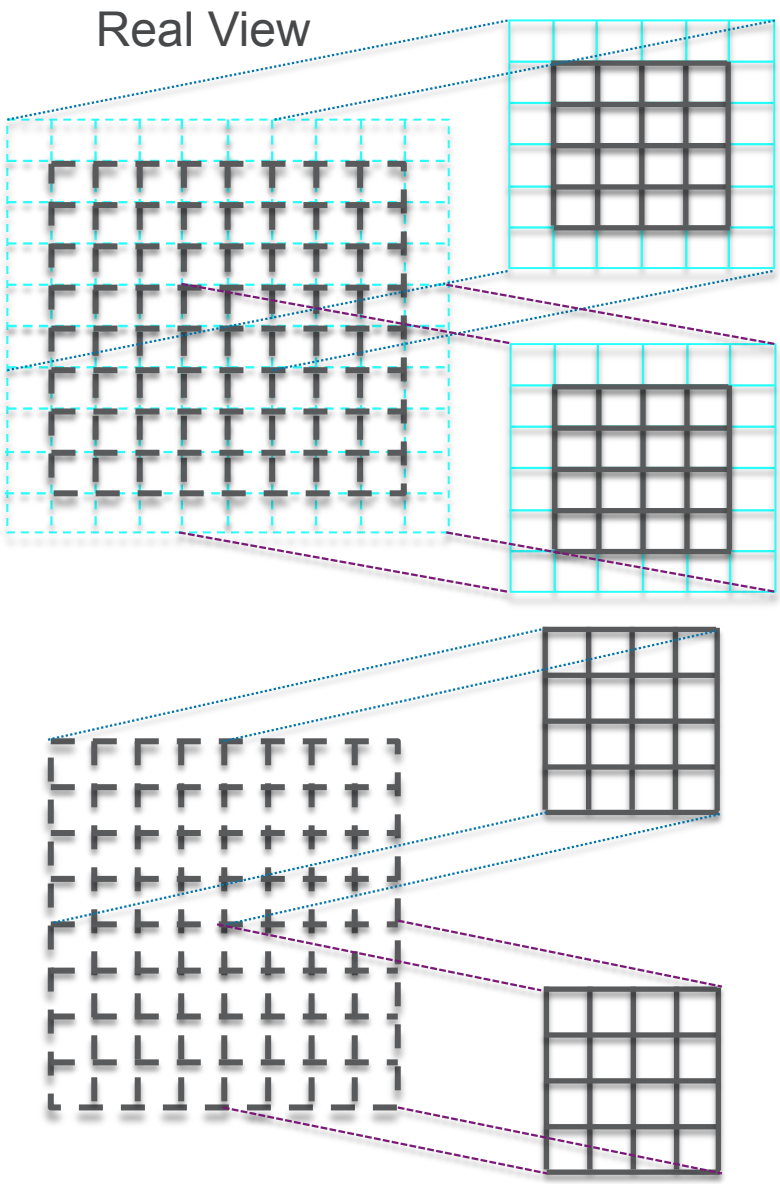
<1,0>

<0,-1>

<0,1>

<0,0>

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

SOME AVAILABLE OPTIONS

- Many efforts to provide tools to application developers
  - KoKKOs : Integrated Option with polymorphic arrays
  - Raja :
  - TiDA, HTA : managing tiling abstractions
  - GridTools : comprehensive solution from CSCS-ETH
  - Dash : managing multilevel locality
  - Task based processing – OCR, charm++, HPX, Quark etc
  - Language based solutions – Julia, Chapel, UPC++ etc
  - Domain specific languages

OTHER THINGS TO CONSIDER

- Leverage existing software
  - Libraries may have better solvers
    - Off-load expertise and maintenance
  - Examine the interoperability constraints
    - Many times the cost is justified even if there is more data movement
- More available packages are attempting to achieve interoperability
  - See if a combination meets your requirements
- May be worthwhile to let the library dictate data layout if the corresponding operations dominate

Institute an extremely rigorous verification regime at the outset

SOFTWARE PROCESS DESIGN

TAKING STOCK

Understand project needs

- 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

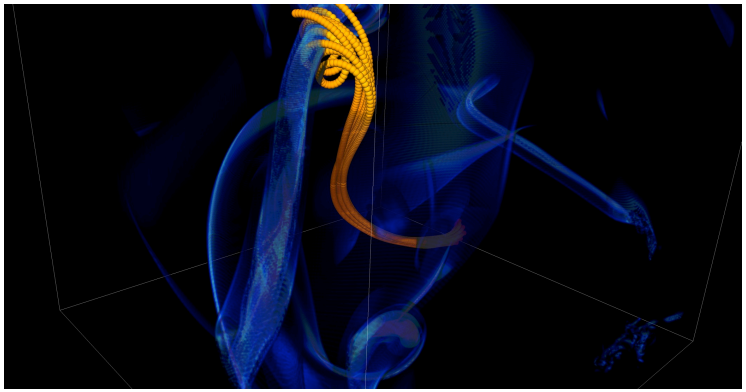
- Software for internal use of a small academic team
  - The minimum set described earlier

- 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

LAST THOUGHT: WHAT CAN HAPPEN WHEN PROCESS IS IGNORED



- In 2005 BG/L was made available at short notice
- Quick and dirty development of particles
- Many in-flight corrections of defects
- One was adding tags to track individual particles
  - **Got many duplicated tags due to round-off**
- Had to develop post-processing tools to correctly identify trajectories

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. We got ready for the run in less than a month, the run went for 1.5 weeks, and it took over 6 months before we could trust the processed results.

TAKEAWAYS

- KNOW THE TOOLS AVAILABLE
- UNDERSTAND YOUR NEEDS
- DO THE COST-BENEFIT ANALYSIS
- ADOPT WHAT WORKS FOR YOU WITHOUT INCURRING TECHNICAL DEBT
- THERE IS NO NEED TO CONSIDER SOFTWARE ENGINEERING SOLUTIONS WITH ALL OR NONE APPROACH