# Software Design and Testing

Anshu Dubey
Argonne National Laboratory

Better Scientific Software Tutorial
SC19, Denver, Colorado

# License, Citation and Acknowledgements

## License and Citation



- **The requested citation the overall tutorial is: David E. Bernholdt, Anshu Dubey, Michael A. Heroux, and Jared O'Neal, Better Scientific Software tutorial, in SC '19: International Conference for High Performance Computing, Networking, Storage and Analysis, Denver, Colorado, 2019. DOI: 10.6084/m9.figshare.10114880**
- Individual modules may be cited as *Module Authors, Module Title*, in Better Scientific Software Tutorial…

## Acknowledgements


- Additional contributors to this this tutorial include: Alicia Klinvex, Katherine Riley, and James Willenbring
- This work was supported by the U.S. Department of Energy Office of Science, Office of Advanced Scientific Computing Research (ASCR), and by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.
- This work was performed in part at the Argonne National Laboratory, which is managed managed by UChicago Argonne, LLC for the U.S. Department of Energy under Contract No. DE-AC02-06CH11357.

# Architecting scientific codes

## Desirable Characteristics

**Extensibility**
Well defined structure and modules
Encapsulation of functionalities

**Portability**
General solutions that work without significant manual intervention across platforms

**Performance**
Spatial and temporal locality of data
Minimizing data movement
Maximizing scalability

**Maintainability and Verifiability**
Clean code
Documentation
Comprehensive testing

# Architecting scientific codes

## Why it is challenging

**Extensibility**
Well defined structure and modules
Encapsulation of functionalities

**Portability**
General solutions that work without significant manual intervention across platforms

**Performance**
Spatial and temporal locality of data
Minimizing data movement
Maximizing scalability

**Maintainability and Verifiability**
Clean code
Documentation
Comprehensive testing

# Architecting scientific codes

## Why it is challenging

**Extensibility**
Same data layout not good for all solvers. Many corner cases. Necessary lateral interactions

**Portability**
General solutions that work without significant manual intervention across platforms

**Performance**
Spatial and temporal locality of data
Minimizing data movement
Maximizing scalability

**Maintainability and Verifiability**
Clean code
Documentation
Comprehensive testing

# Architecting scientific codes

## Why it is challenging

**Extensibility**
Same data layout not good for all solvers. Many corner cases. Necessary lateral interactions

**Portability**
Tremendous platform heterogeneity
A version for each class of device => combinatorial explosion

**Performance**
Spatial and temporal locality of data
Minimizing data movement
Maximizing scalability

**Maintainability and Verifiability**
Clean code
Documentation
Comprehensive testing

# Architecting scientific codes

## Why it is challenging

**Extensibility**
Same data layout not good for all solvers. Many corner cases. Necessary lateral interactions

**Portability**
Tremendous platform heterogeneity
A version for each class of device => combinatorial explosion

**Performance**
Low arithmetic intensity solvers with hard dependencies. Proximity and work distribution at cross purposes

**Maintainability and Verifiability**
Clean code
Documentation
Comprehensive testing

# Architecting scientific codes

## Why it is challenging

**Extensibility**
Same data layout not good for all solvers. Many corner cases. Necessary lateral interactions

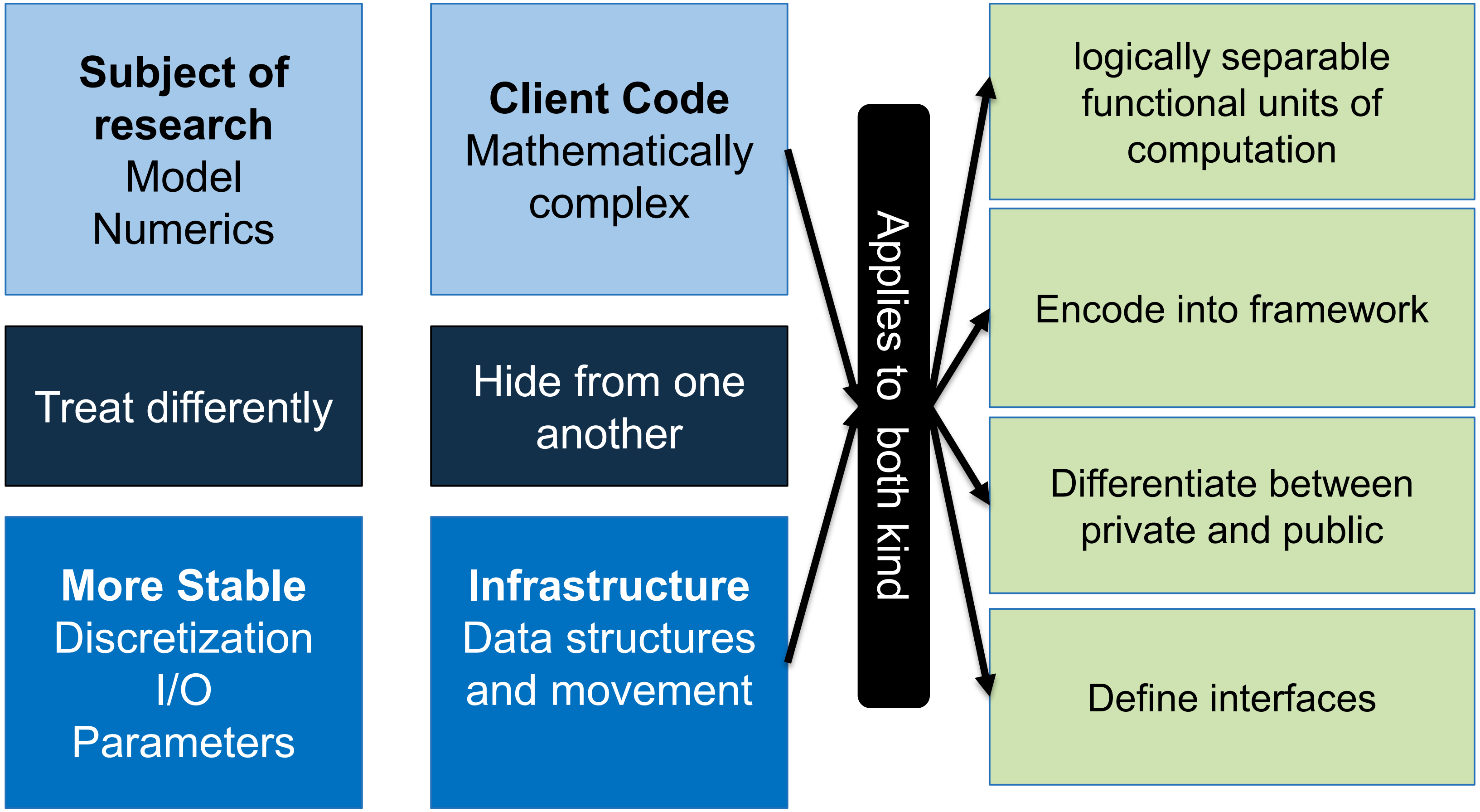
**Portability**
Tremendous platform heterogeneity
A version for each class of device => combinatorial explosion

**Performance**
Low arithmetic intensity solvers with hard dependencies. Proximity and work distribution at cross purposes

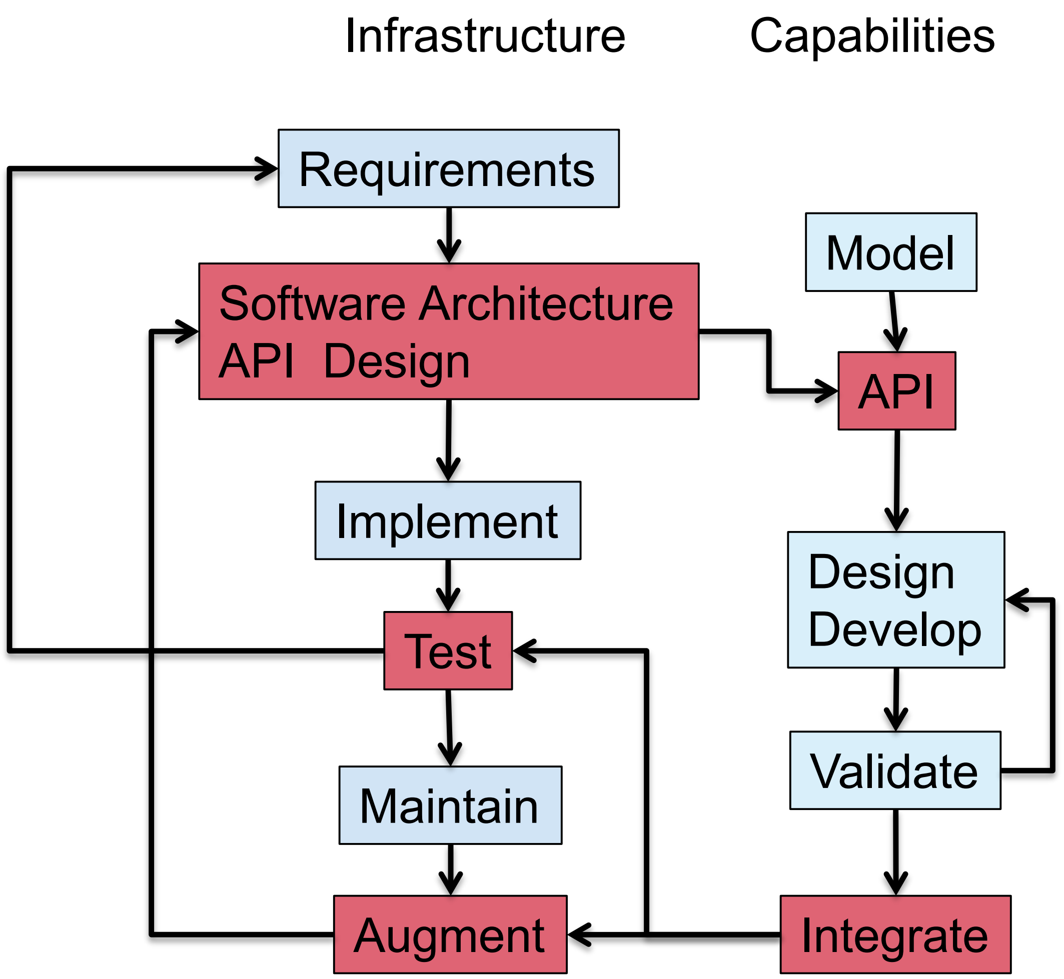
**Maintainability and Verifiability**
Wrong incentives
Designing good tests is hard

# Architecting scientific codes

## Taming the Complexity: Separation of Concerns



# A successful model



# Example From FLASH: EOS interface Design

- Hierarchy in complexity of interfaces
  - For collection of points
  - For sections of a block

- 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

```
physics/Eos/Eos
physics/Eos/Eos_finalize
physics/Eos/Eos_getAbarZbar
physics/Eos/Eos_getData
physics/Eos/Eos_getParameters
physics/Eos/Eos_getTempData
physics/Eos/Eos_guardCells
physics/Eos/Eos_init
physics/Eos/Eos_logDiagnostics
physics/Eos/Eos_nucDetectBounce
physics/Eos/Eos_nucOneZone
physics/Eos/Eos_putData
physics/Eos/Eos_unitTest
physics/Eos/Eos_wrapped
```

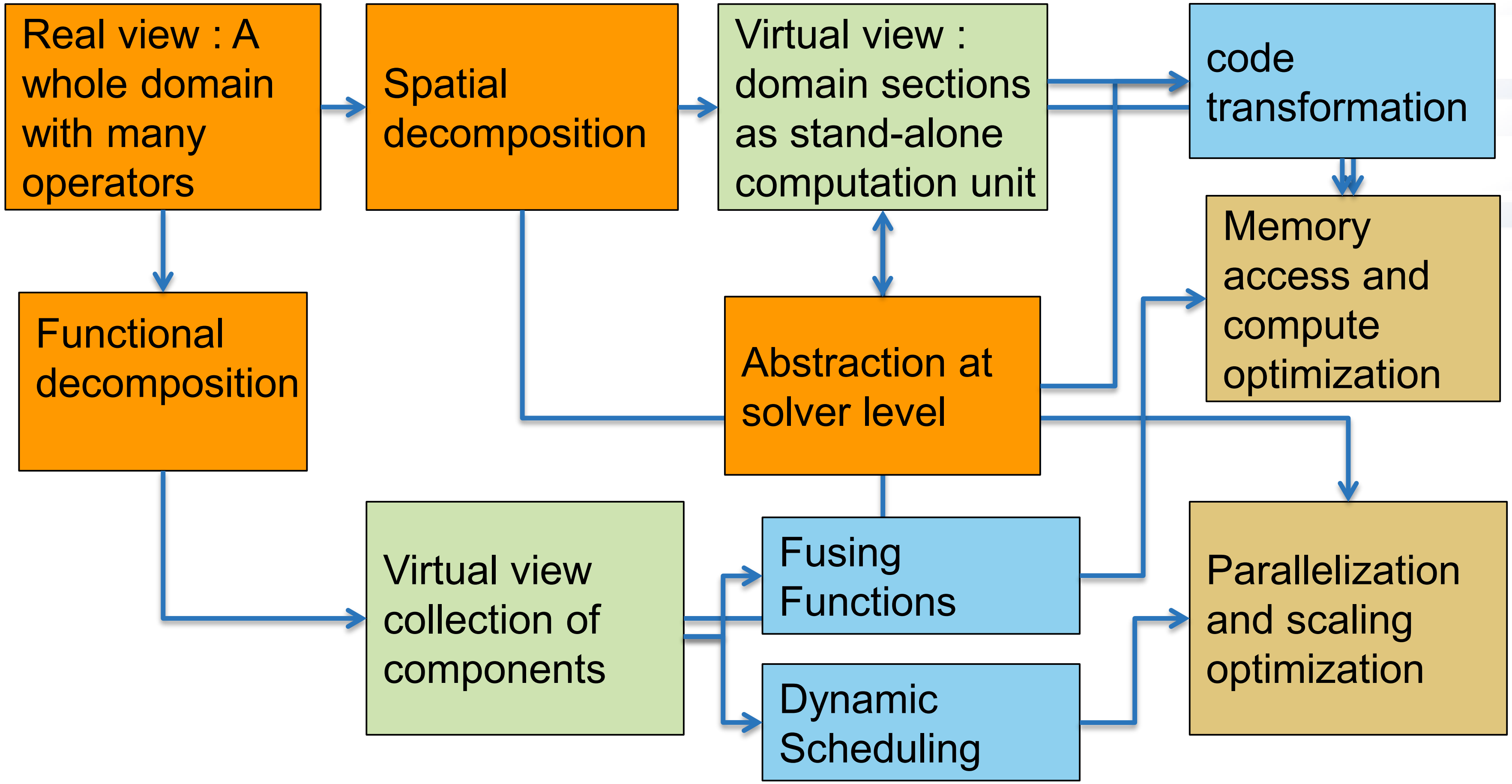
# Preparing for future

- 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.

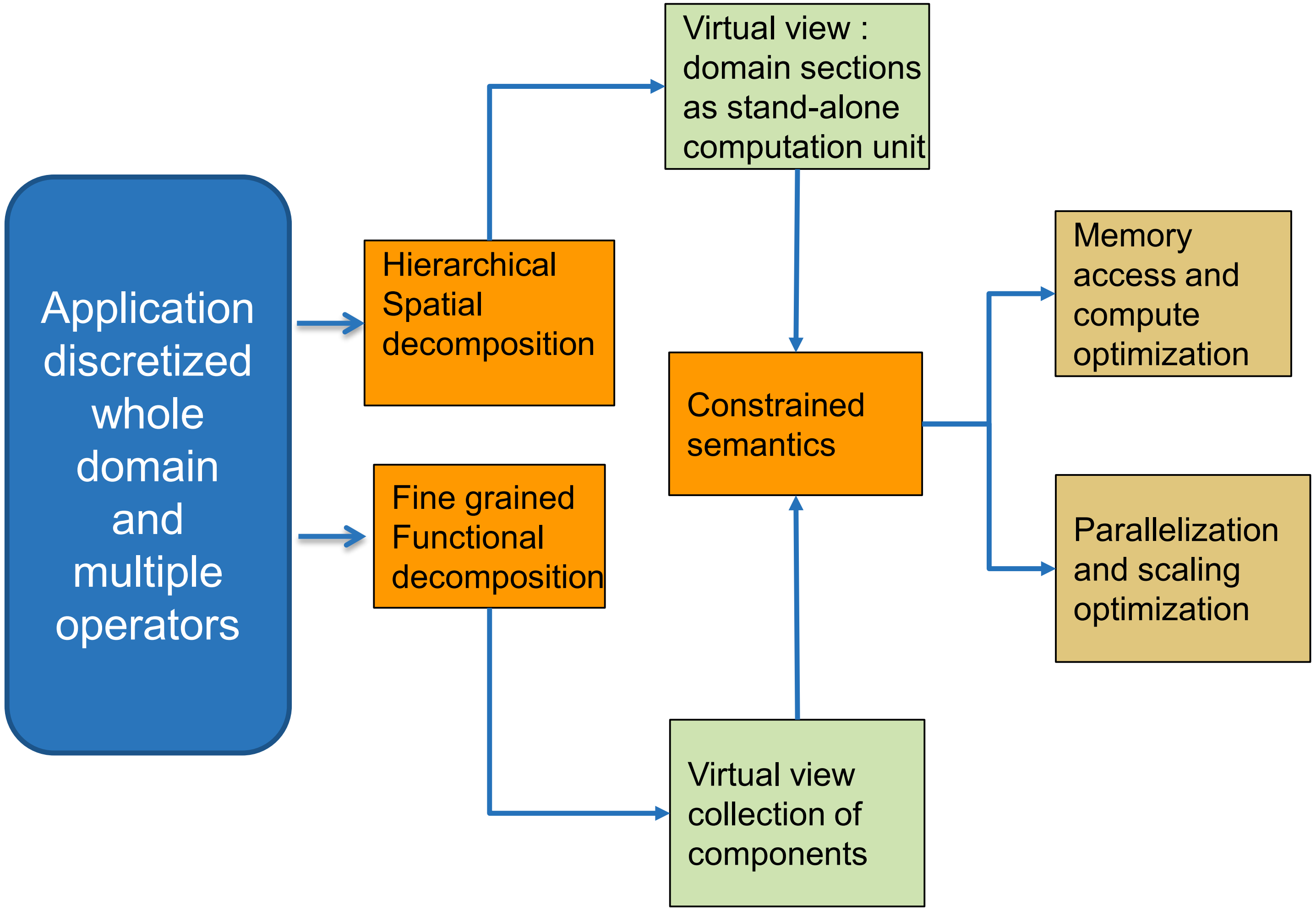
# Design considerations

- Composing tasks
  - Components or kernels
- Task orchestration
  - Mapping tasks to devices
    - CPU, accelerators, specialized devices
  - Managing data movement between devices

# Example: PDE's



# Example: FLASH5 approach

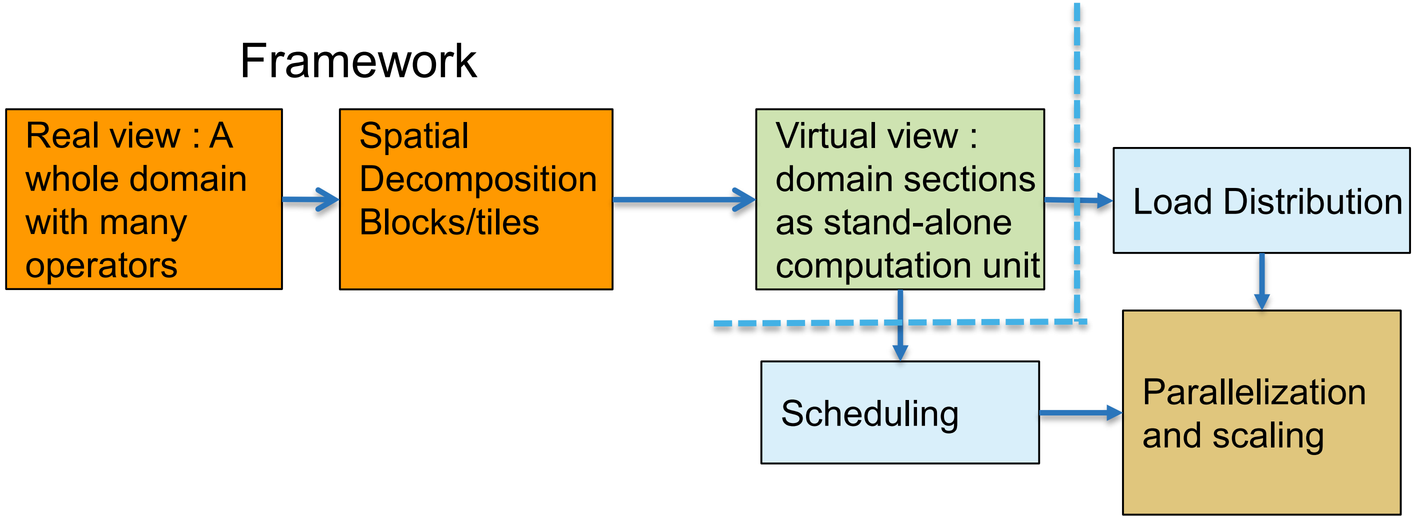





IDEAS productivity

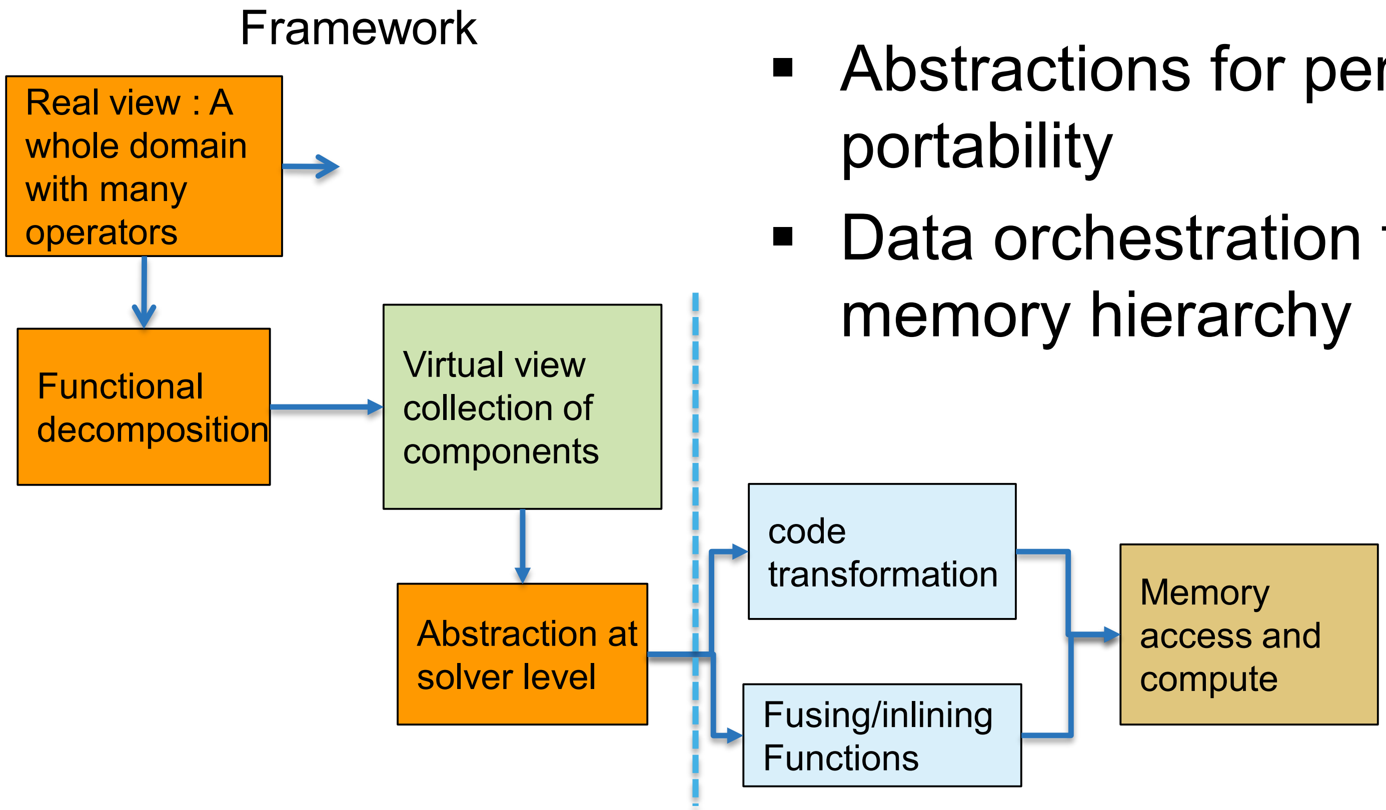
ECP EXASCALE COMPUTING PROJECT

# Components in play: infrastructure




- AMR infrastructure: refinement, load balancing, work redistribution
- Scheduling and data movement at block and operator level

# Components in Play: operators




- Abstractions for performance portability
- Data orchestration for memory hierarchy

# 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**

# TESTING AND VERIFICATION

# Verification

- Code verification uses tests
  - It is much more than a collection of tests
- It is the holistic process through which you ensure that
  - Your implementation shows expected behavior,
  - Your implementation is consistent with your model,
  - Science you are trying to do with the code can be done.

# Stages and types of verification

- During initial code development
  - Accuracy and stability
  - Matching the algorithm to the model
  - Interoperability of algorithms
- In later stages
  - While adding new major capabilities or modifying existing capabilities
  - Ongoing maintenance
  - Preparing for production

# Verification Challenges

- Functionality coverage
- Particularly true of codes that allow composability in their configuration
- Codes may incorporate some legacy components
  - Its own set of challenges
    - No existing tests at any granularity
- Examples – multiphysics application codes that support multiple domains

# Challenges with legacy codes

## Checking for coverage

- Legacy codes can have many gotchas
  - Dead code
  - Redundant branches
- Interactions between sections of the code may be unknown
- Can be difficult to differentiate between just bad code, or bad code for a good reason
  - Nested conditionals

**Code coverage tools are of limited help**

# Components of Verification

- Testing at various granularity
  - Individual components
  - Interoperability of components
  - Convergence, stability and accuracy
- Validation of individual components
- Testing practices
- Error bars
  - Necessary for differentiating between drift and round-off
- Selection of tests for coverage

# Regular Testing

- Part of ongoing verification
- Automating is helpful
- Can be just a script
- Or a testing harness

**Jenkins**
**C-dash**
**Custom**
(FlashTest)

- Essential for large code
  - Set up and run tests
  - Evaluate test results
- Easy to execute a logical subset of tests
  - Pre-push
  - Nightly
- Automation of test harness is critical for
  - Long-running test suites
  - Projects that support many platforms

# Good Testing Practices

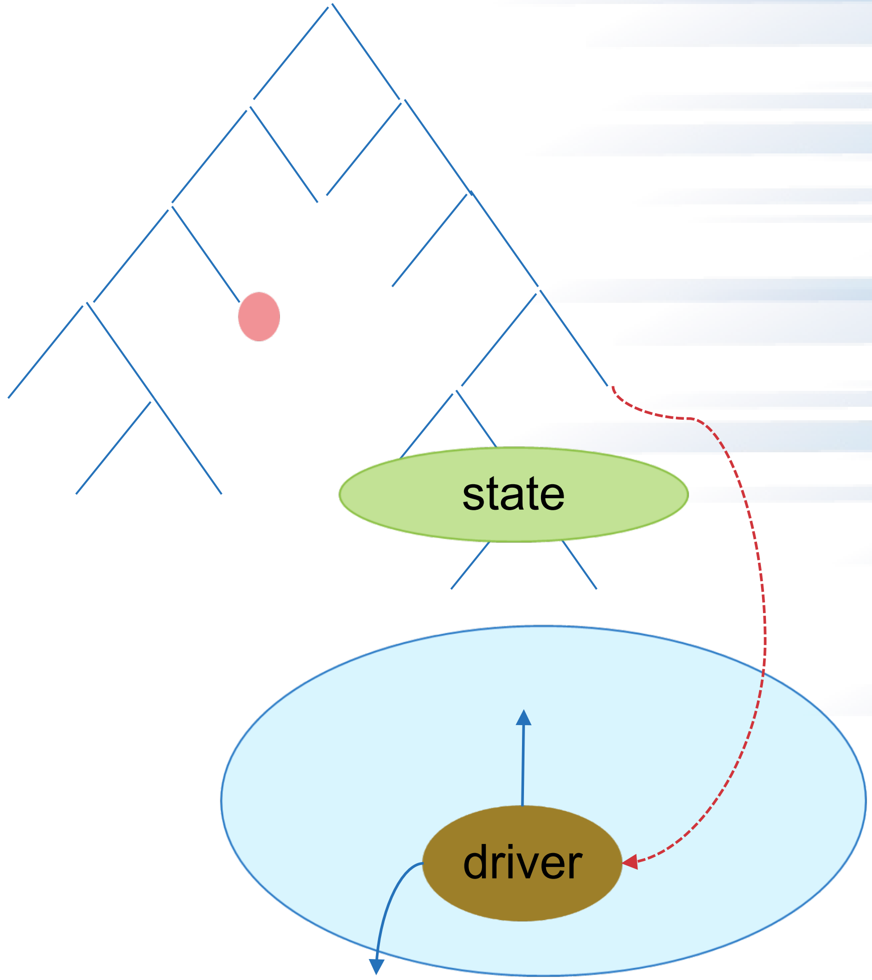
- Must have consistent policy on dealing with failed tests
  - Issue tracking
    - How quickly does it need to be fixed?
    - Who is responsible for fixing it?
- Someone should be watching the test suite
- When refactoring or adding new features, run a regression suite before check in
  - Add new regression tests or modify existing ones for the new features
- Code review before releasing test suite is useful
  - Another person may spot issues you didn't
  - Incredibly cost-effective

# Test Development

- Development of tests and diagnostics goes hand-in-hand with code development
  - Non-trivial to devise good tests, but extremely important
  - Compare against simpler analytical or semi-analytical solutions
- When faced with legacy codes with no existing tests
  - Isolate a small area of the code
  - Dump a useful state snapshot
  - Build a test driver
    - Start with only the files in the area
    - Link in dependencies
      - Copy if any customizations needed
  - Read in the state snapshot
  - Verify correctness
    - Always inject errors to verify that the test is working

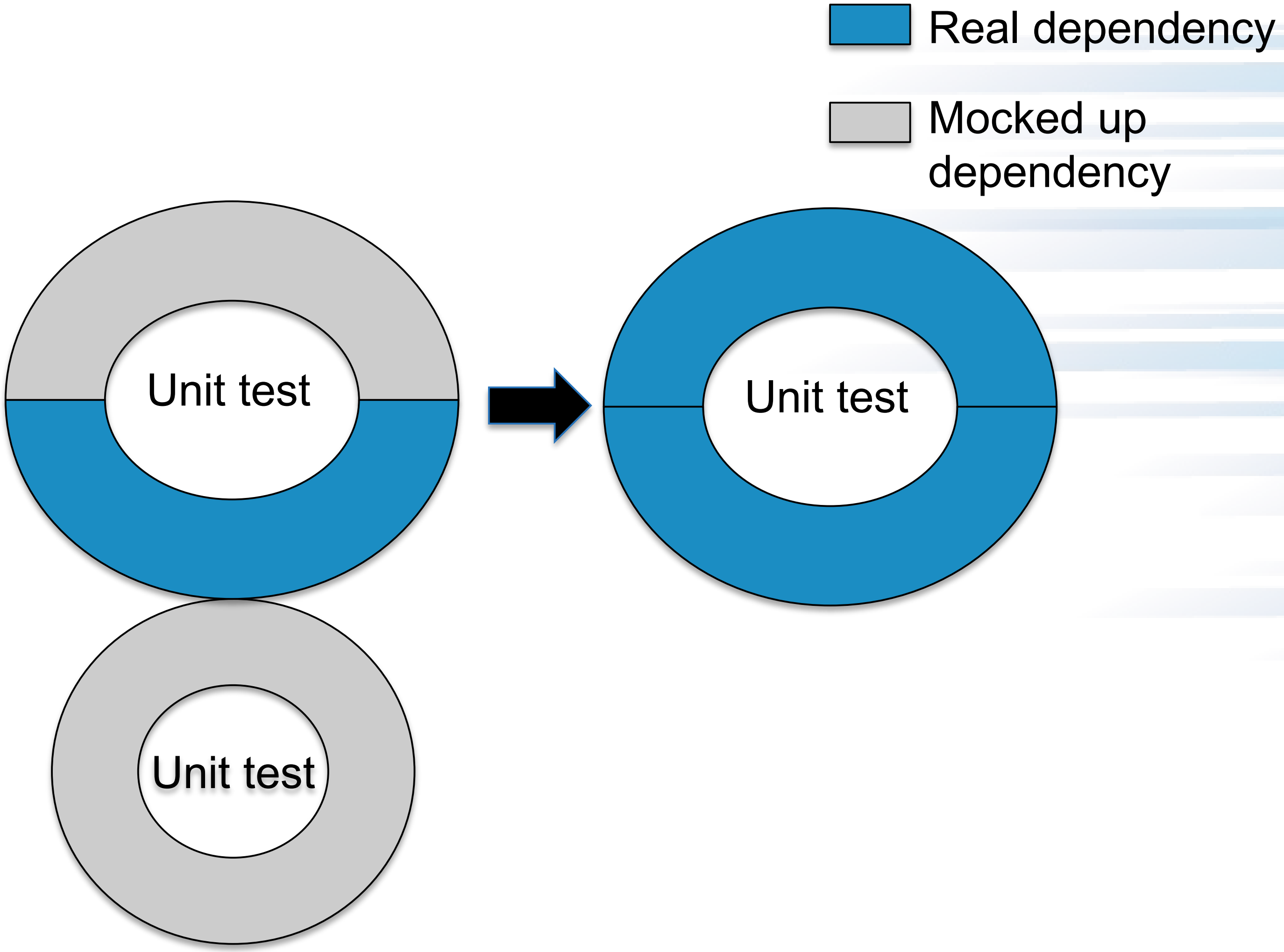
# Example from E3SM

- Isolate a small area of the code
- Dump a useful state snapshot
- Build a test driver
  - Start with only the files in the area
  - Link in dependencies
    - Copy if any customizations needed
- Read in the state snapshot
- Restart from the saved state



# Workarounds for Granularity

- Approach the problem sideways
  - Components can be exercised against known simpler applications
  - Same applies to combination of components
- Build a scaffolding of verification tests to gain confidence



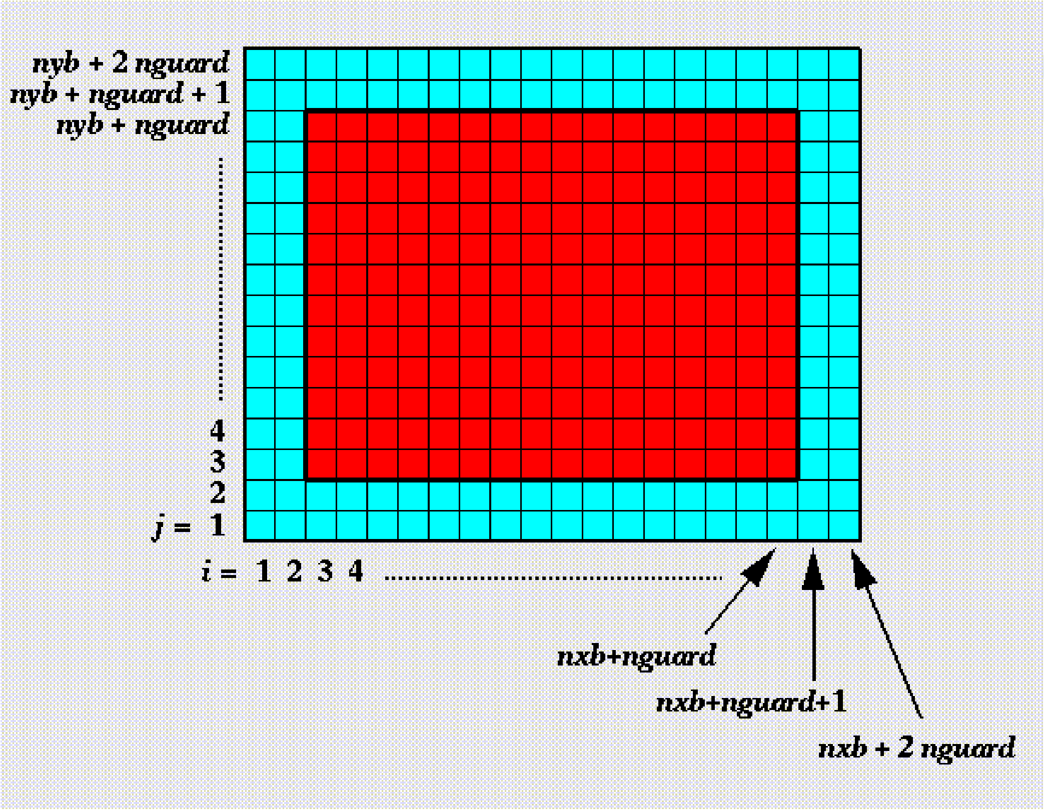


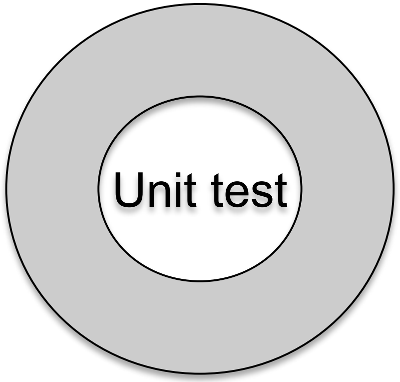

IDEAS productivity ECP EXASCALE COMPUTING PROJECT

# Example from FLASH

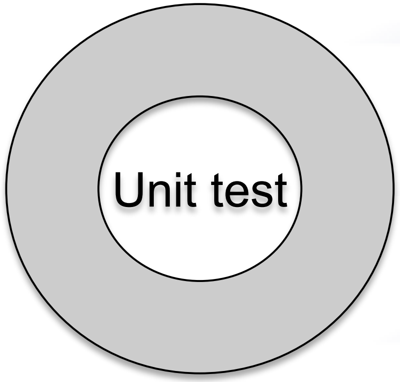
## Unit test for Grid

- Verification of guard cell fill
- Use two variables A & B
- Initialize A in all cells and B only in the interior cells (red)
- Apply guard cell fill to B



# Example from Flash



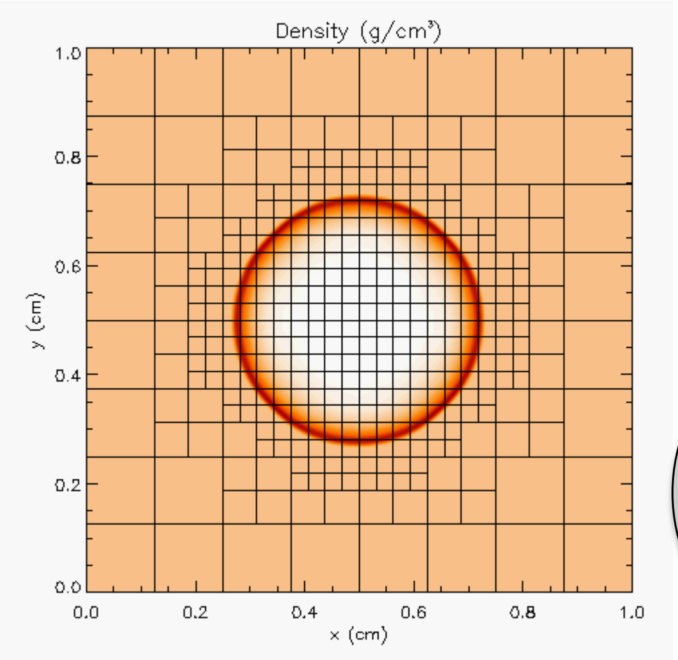

## Unit test for Equation of State (EOS)

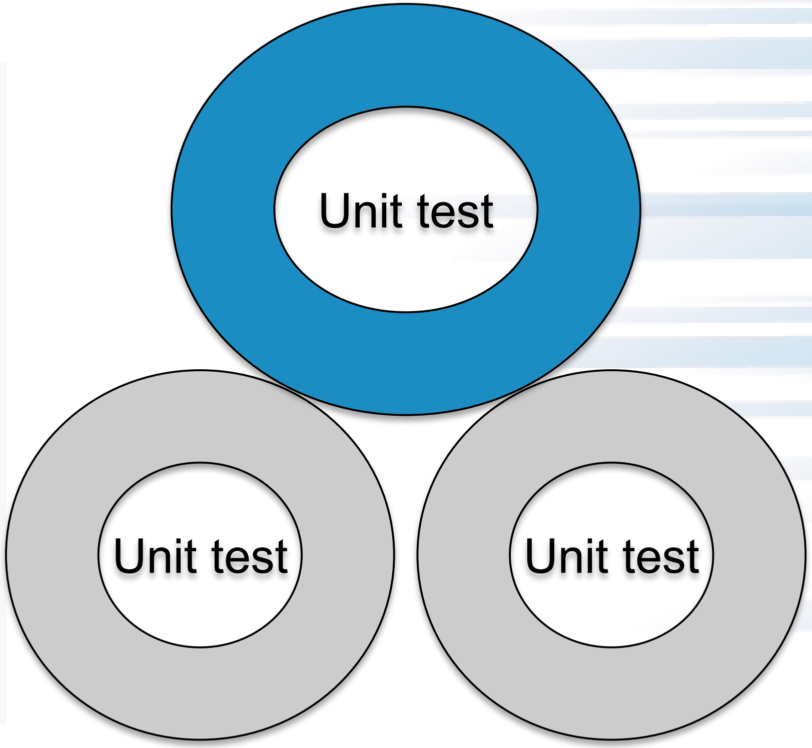
- Three modes for invoking EOS
  - MODE1: Pressure and density as input, internal energy and temperature as output
  - MODE2: Internal energy and density as input temperature and pressure as output
  - MODE3: Temperature and density as input pressure and internal energy as output
- Use initial conditions from a known problem, initialize pressure and density
- Apply EOS in MODE1
- Using internal energy generated in the previous step apply EOS in MODE2
- Using temperature generated in the previous step apply EOS in MODE3
- At the end all variables should be consistent within tolerance

# Example from FLASH

## Unit test for Hydrodynamics

- Sedov blast wave
- High pressure at the center
- Shock moves out spherically
- FLASH with AMR and hydro
- Known analytical solution




Though it exercises mesh, hydro and eos, if mesh and eos are verified first, then this test verifies hydro

**More testing needed for Grid using AMR**
**Flux correction and regridding**

# Example from FLASH

## Reason about correctness for testing Flux correction and regridding

IF Guardcell fill and EOS unit tests passed

- Run Hydro without AMR
  - If failed fault is in Hydro
- Run Hydro with AMR, but no dynamic refinement
  - If failed fault is in flux correction
- Run Hydro with AMR and dynamic refinement
  - If failed fault is in regridding

# Selection of tests

- Two purposes
  - Regression testing
    - May be long running
    - Provide comprehensive coverage
  - Continuous integration
    - Quick diagnosis of error
- A mix of different granularities works well
  - Unit tests for isolating component or sub-component level faults
  - Integration tests with simple to complex configuration and system level
  - Restart tests
- Rules of thumb
  - Simple
  - Enable quick pin-pointing

# Why not always use the most stringent testing?

- Effort spent in devising tests and testing regime are a tax on team resources
- When the tax is too high…
  - Team cannot meet code-use objectives
- When is the tax is too low…
  - Necessary oversight not provided
  - Defects in code sneak through
- Evaluate project needs
  - Objectives: expected use of the code
  - Team: size and degree of heterogeneity
  - Lifecycle stage: new or production or refactoring
  - Lifetime: one off or ongoing production
  - Complexity: modules and their interactions

# Commonalities

- Unit testing is always good
  - It is never sufficient
- Verification of expected behavior
- Understanding the range of validity and applicability is always important
  - Especially for individual solvers

# Test Selection

- First line of defense – code coverage tools (demo later)
- Necessary but not sufficient – don't give any information about interoperability

- Build a matrix
  - Physics along rows
  - Infrastructure along columns
  - Alternative implementations, dimensions, geometry
- Mark <i,j> if test covers corresponding features
- Follow the order
  - All unit tests – including full module tests
  - Tests representing ongoing productions
  - Tests sensitive to perturbations
  - Most stringent tests for solvers
  - Least complex test to cover remaining spots

# Example

| | Hydro | EOS | Gravity | Burn | Particles |
|---|---|---|---|---|---|
| AMR | CL | CL | | CL | CL |
| UG | SV | SV | | | SV |
| Multigrid | WD | WD | WD | WD | |
| FFT | | | PT | | |

- A test on the same row indicates interoperability between corresponding physics
- Similar logic would apply to tests on the same column for infrastructure
- More goes on, but this is the primary methodology

| Tests | Symbol |
|---|---|
| Sedov | SV |
| Cellular | CL |
| Poisson | PT |
| White Dwarf | WD |

# TAKEAWAYS

- UNDERSTAND YOUR NEEDS
- DO THE COST-BENEFIT ANALYSIS
- ADOPT WHAT WORKS FOR YOU WITHOUT INCURRING TECHNICAL DEBT
- DESIGN WITH PORTABILITY, EXTENSIBILITY, REPRODUCIBILITY AND MAINTAINABILITY IN MIND
- VERIFY … VERIFY … VERIFY

…….QUESTIONS ?