Scientific Software Design

Presented by

https://pesoproject.org

Member of

Consortium for the Advancement of Scientific Software
https://cass.community

With prior support from

David E. Bernholdt (he/him)
Oak Ridge National Laboratory

Software Sustainability track @ Argonne Training Program on Extreme-Scale Computing summer school

Contributors: Anshu Dubey (ANL), David E. Bernholdt (ORNL)

License, Citation and Acknowledgements

License and Citation

- **The requested citation the overall tutorial is:** David E. Bernholdt, Software Sustainability track, in Argonne Training Program on Extreme-Scale Computing, St. Charles, Illinois, 2026. DOI: 10.6084/m9.figshare.33139910.
- Individual modules may be cited as *Speaker, Module Title*, in *Tutorial Title*, …

Acknowledgements

- This work was supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Next-Generation Scientific Software Technologies (NGSST) and Scientific Discovery through Advanced Computing (SciDAC) programs.
- 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 by UChicago Argonne, LLC for the U.S. Department of Energy under Contract No. DE-AC02-06CH11357.
- This work was performed in part at the Lawrence Livermore National Laboratory, which is managed by Lawrence Livermore National Security, LLC for the U.S. Department of Energy under Contract No. DE-AC52-07NA27344.
- This work was performed in part at the Los Alamos National Laboratory, which is managed by Triad National Security, LLC for the U.S. Department of Energy under Contract No.89233218CNA000001
- This work was performed in part at the Oak Ridge National Laboratory, which is managed by UT-Battelle, LLC for the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.
- This work was performed in part at Sandia National Laboratories. Sandia National Laboratories is a multi-mission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Introduction

- Investing some thought in design of software makes it possible to maintain, reuse and extend it
- Even if some research software begins its life as a one-off use case, it often gets reused
  - Without proper design it is likely to accrete features haphazardly and become a monstrosity
    - Acquires a lot of technical debt in the process

> "Technical debt – or code debt – is the consequence of software development decisions that result in prioritizing speed or release over the [most] well-designed code," Duensing says. "It is often the result of using quick fixes and patches rather than full-scale solutions."

definition from https://enterprisersproject.com/article/2020/6/technical-debt-explained-plain-english

  - Many projects have had this happen
  - Most end up with a hard reset and start over again
- In this module we will cover general design principles and those that are tailored for scientific software
- We will also work through two use cases

Designing Software – High Level Phases

Requirements gathering

- ❑ Features and capabilities
- ❑ Constraints
- ❑ Limitations
- ❑ Target users
- ❑ Other …..

Decomposition

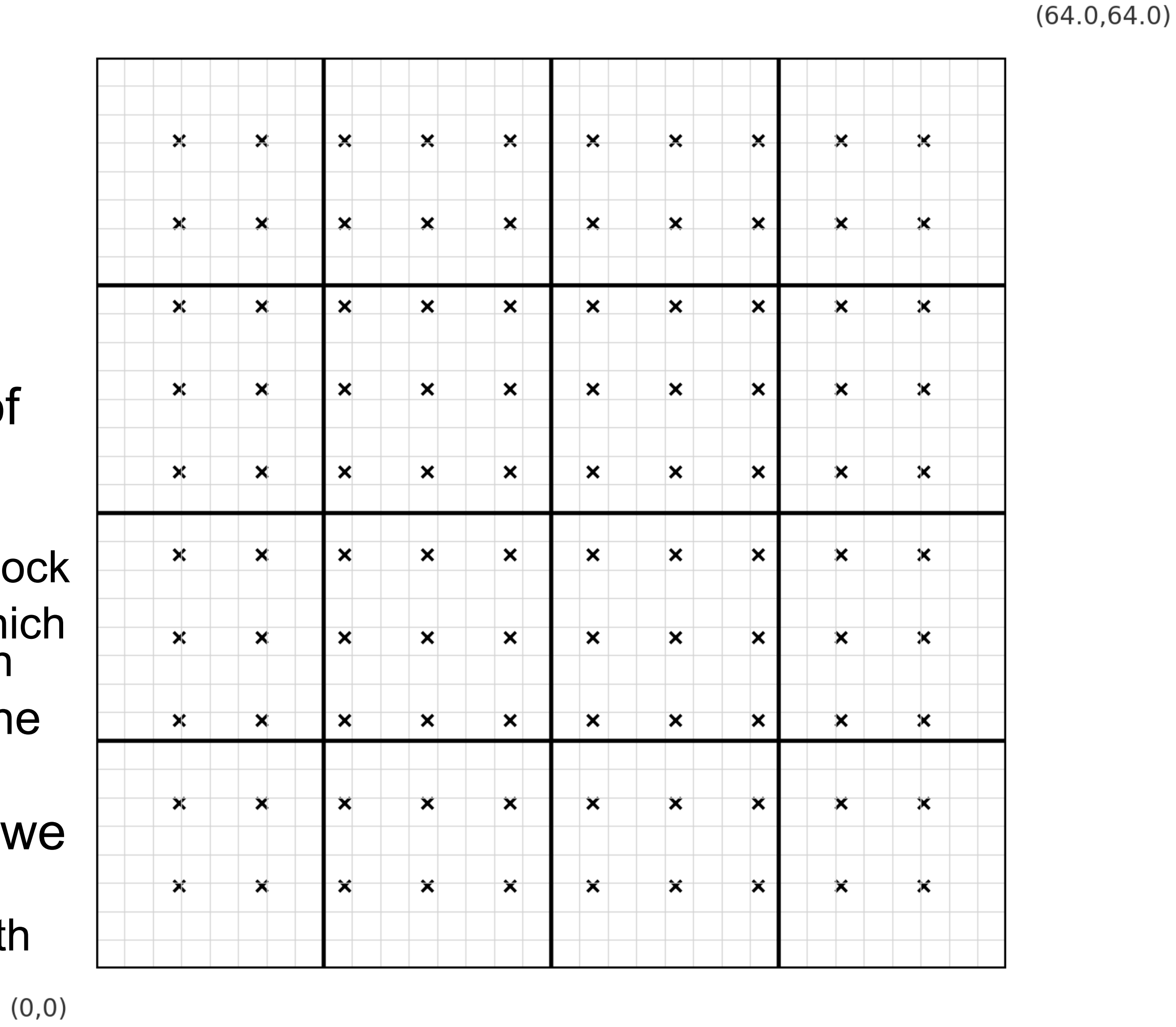
- ❑ Understand design space
- ❑ Decompose into high level components
- ❑ Bin components into types

Connectivity

- ❑ Understand component hierarchy
- ❑ Figure out connectivity among components
- ❑ Articulate dependencies

Simple Use Case

- Many applications combine mesh and particles
  - Particles have positions associated with them
  - They interchange physical quantities with the mesh
  - They change positions based on interactions
- We will build a uniformly discretized 2D cartesian mesh and initialize with a lattice of particles
  - We will divide the mesh into blocks
  - We will associate each particle with a single block
  - We will move particles to different locations which may change the block they are associated with
  - We will need to handle particles that leave the domain
- To make sure that we are doing it correctly we need
  - Ways of verifying that particle is associated with the right block



Requirements gathering

- Components we need
  - Mesh generator – with mesh divided into blocks
  - Particles generator – in a 2D lattice
  - For each particle ability to locate the block on which the particle resides
  - Making the particles change position
- To verify that we are doing it correctly we need to make sure that
  - The mesh is generated correctly
  - The particles are initialized in a lattice and associated with blocks correctly
  - After moving the particles change their associations correctly

Decomposition

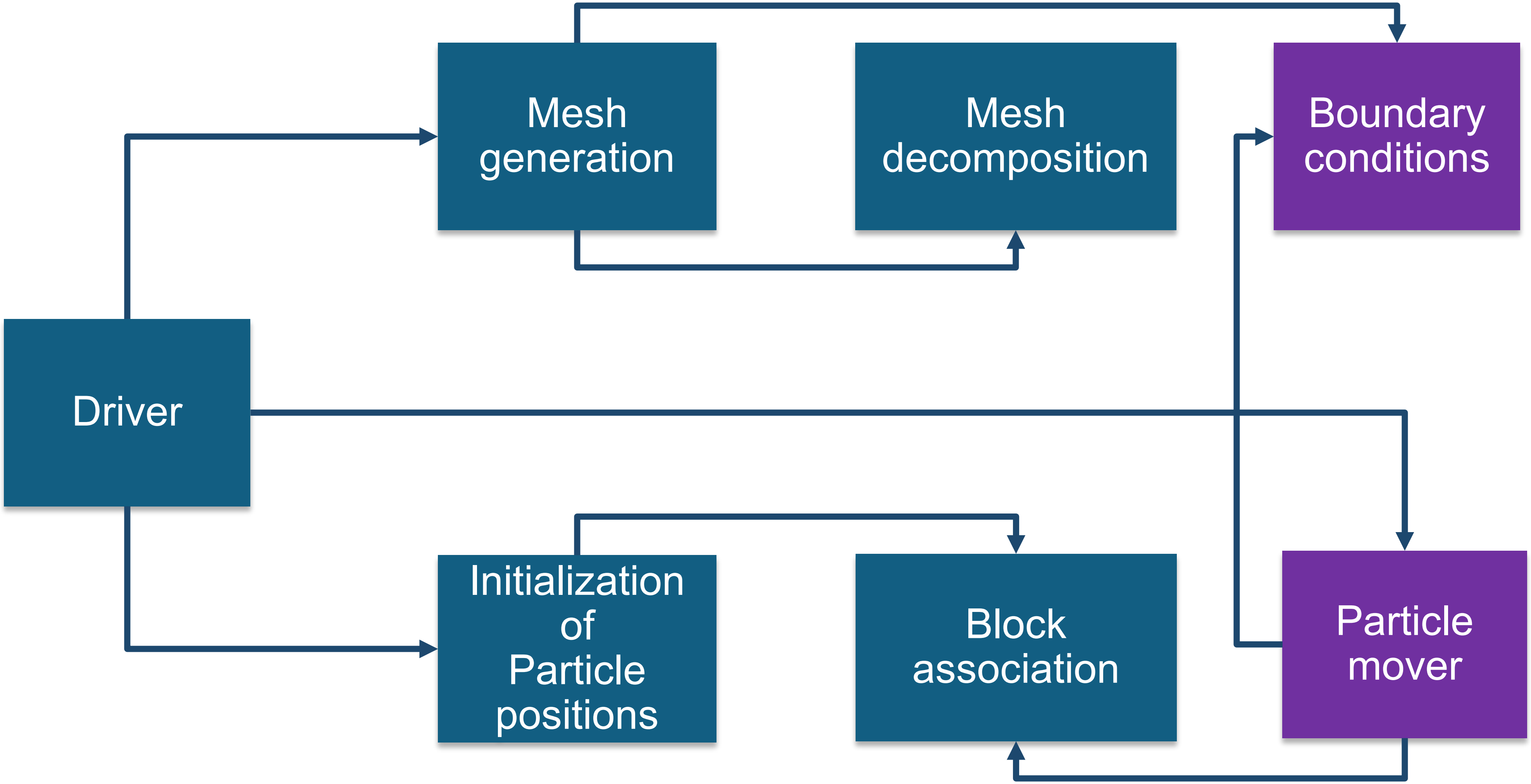
This is a small design space

- Several requirements can directly map to components
  - Driver
  - Mesh initialization
    - Data structure
    - Division into blocks
  - Particle initialization
    - Position
    - Association with a mesh block
  - Particle mover
  - Tests

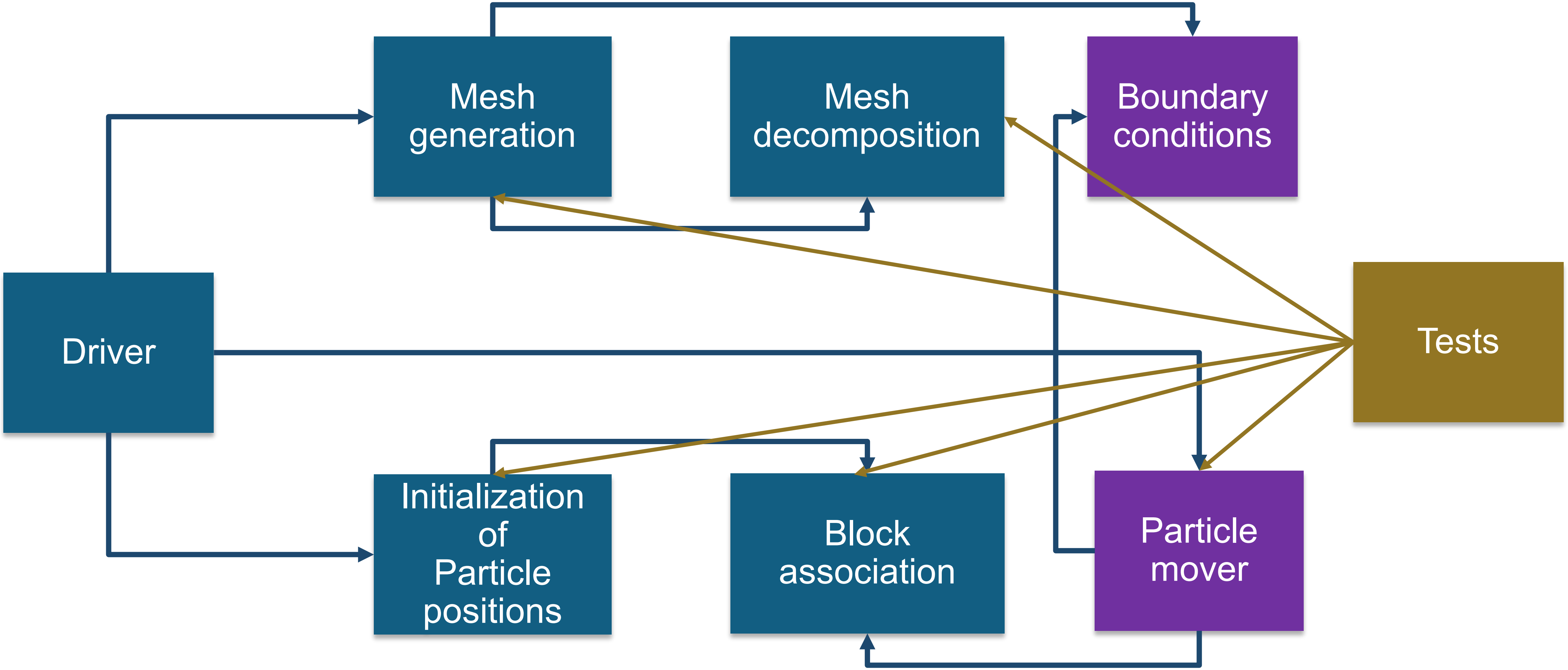
Binning components

- Infrastructure Components
  - Driver
  - Initialization
    - Mesh with division into blocks
    - Particles position
  - Association with a mesh block
- Components that are specific to one instance
  - Boundary conditions
  - Particle mover
- Components for verification
  - Tests

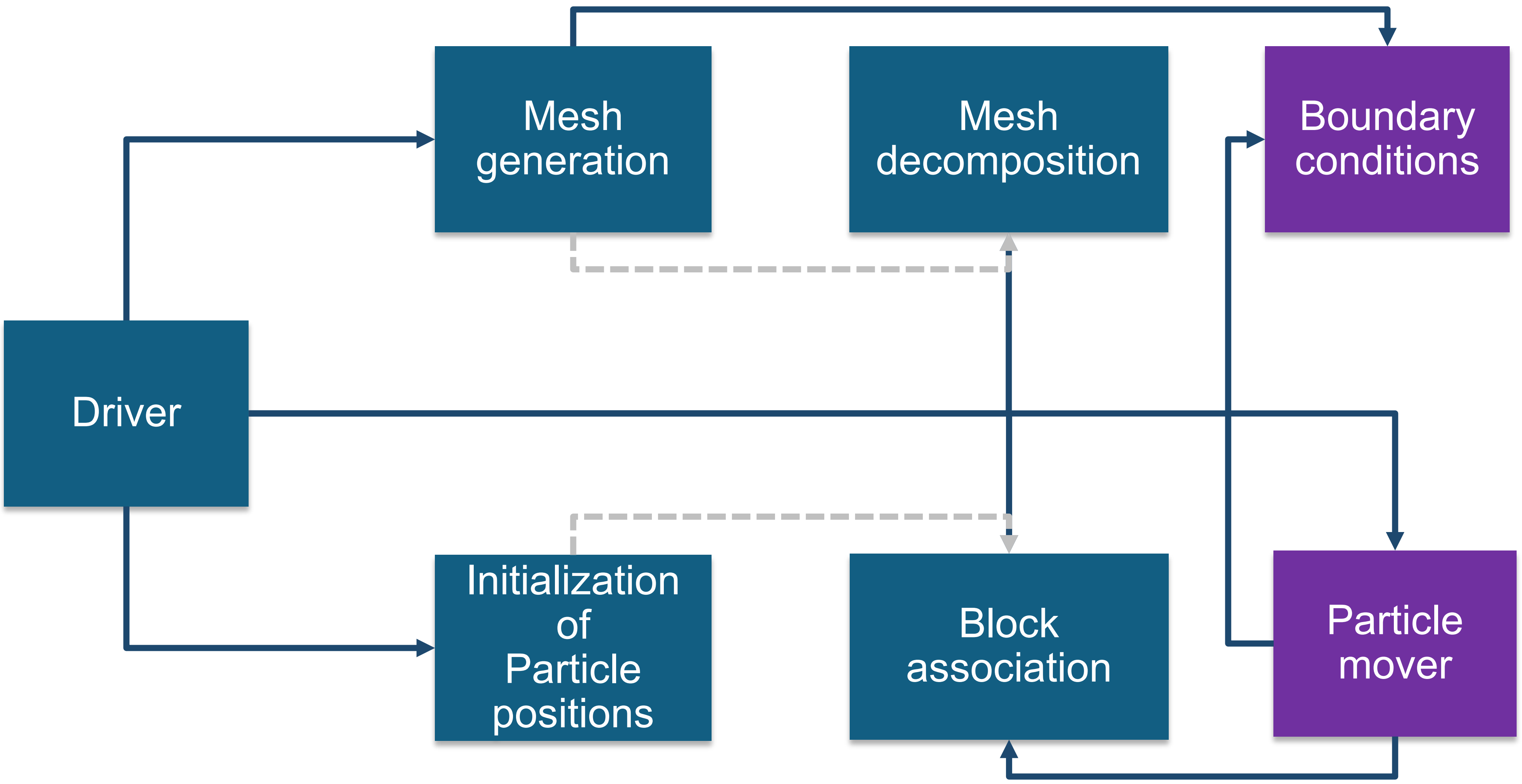
Connectivity



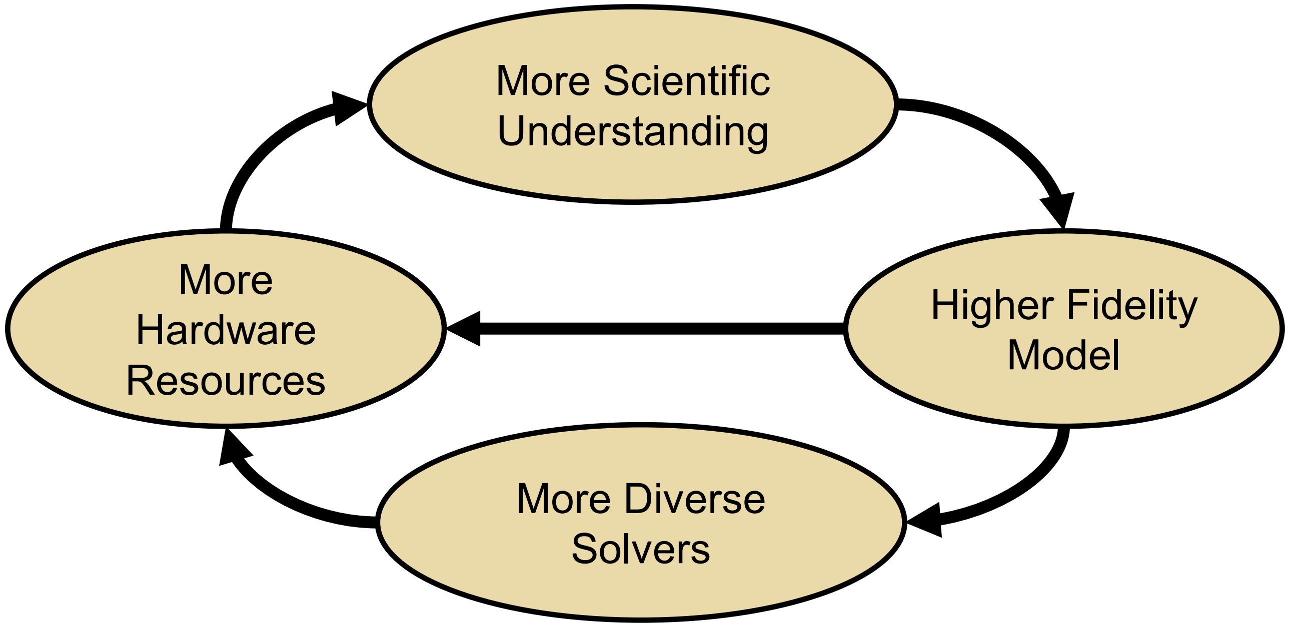
Connectivity



Connectivity



Research Software Challenges



- Many parts of the model and software system can be under research
- Requirements change throughout the lifecycle as knowledge grows
- Verification complicated by floating point representation
- Real world is messy

Additional Considerations for Research Software

Considerations

- ❑ Multidisciplinary
  - ❑ Many facets of knowledge
  - ❑ To know everything is not feasible
- ❑ Two types of code components
  - ❑ Infrastructure (mesh/IO/runtime …)
  - ❑ Science models (numerical methods)
- ❑ Codes grow
  - ❑ New ideas => new features
  - ❑ Code reuse by others

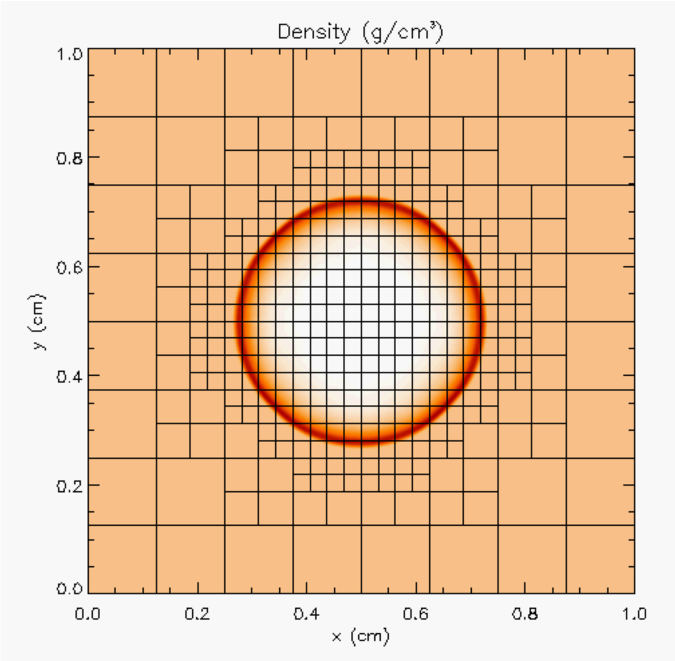
Design Implications

- ❑ Separation of Concerns
  - ❑ Shield developers from unnecessary complexities
- ❑ Work with different lifecycles
  - ❑ Long-lasting vs quick changing
  - ❑ Logically vs mathematically complex
- ❑ Extensibility built in
  - ❑ Ease of adding new capabilities
  - ❑ Customizing existing capabilities

More Complex Application Design – Sedov Blast Wave

Description

High pressure at the center cause a shock to moves out in a circle. High resolution is needed only at and near the shock



Requirements

- Adaptive mesh refinement
  - Easiest with finite volume methods
- Driver
- I/O
- Initial condition
- Boundary condition
- Shock Hydrodynamics
- Ideal gas equation of state
- Method of verification

Deeper Dive into Requirements

- Adaptive mesh refinement → divide domain into blocks
  - Blocks need halos to be filled with values from neighbors or boundary conditions
    - At fine-coarse boundaries there is interpolation and restriction
  - Blocks are dynamic, go in and out of existence
  - Conservation needs reconciliation at fine-coarse boundaries
- Shock hydrodynamics
  - Solver for Euler's equations at discontinuities
  - EOS provides closure
  - Riemann solver
  - Halo cells are fine-coarse boundaries need EOS after interpolation
- Method of verification
  - An indirect way of checking – shock distance traveled can be computed analytically

Components

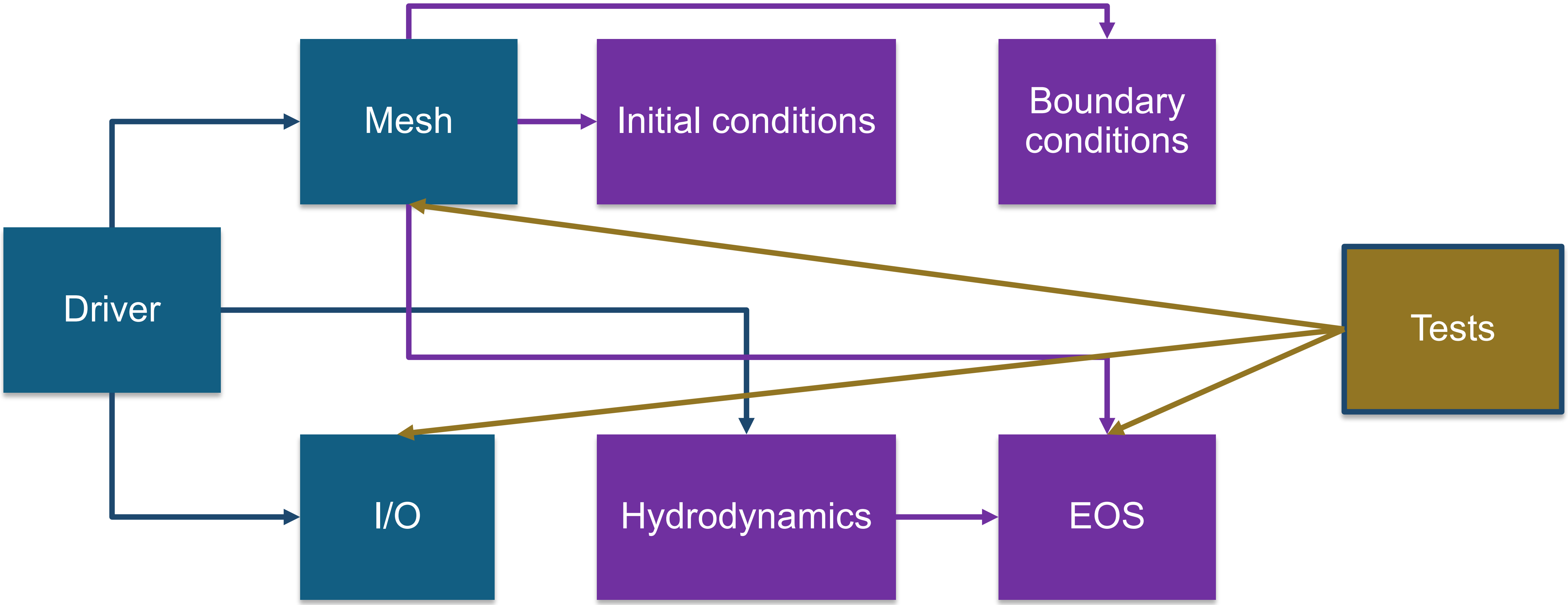
Binned Components

- Unchanging or slow changing infrastructure
  - Mesh
  - I/O
  - Driver
  - Comparison utility -- testing
- Components evolving with research – physics solvers
  - Initial and boundary conditions
  - Hydrodynamics
  - EOS

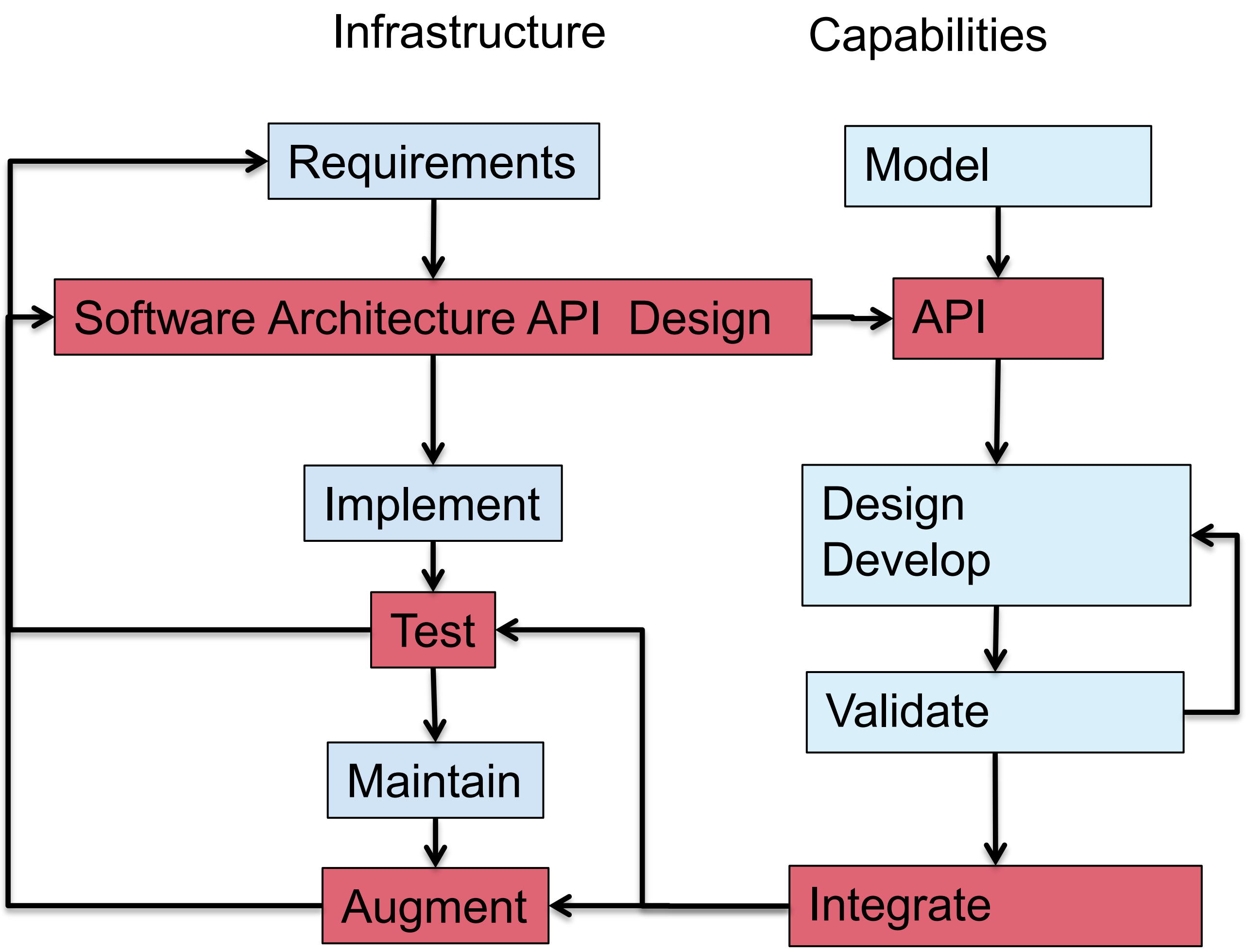
Deeper Dive into some Components

- Driver
  - Iterate over blocks
  - Implement connectivity
- Mesh
  - Data containers
  - Halo cell fill, including application of boundary conditions
  - Reconciliation of quantities at fine-coarse block boundaries
  - Re-mesh when refinement patterns change
- I/O
  - Getting runtime parameters and possibly initial conditions
  - Writing checkpoint and analysis data

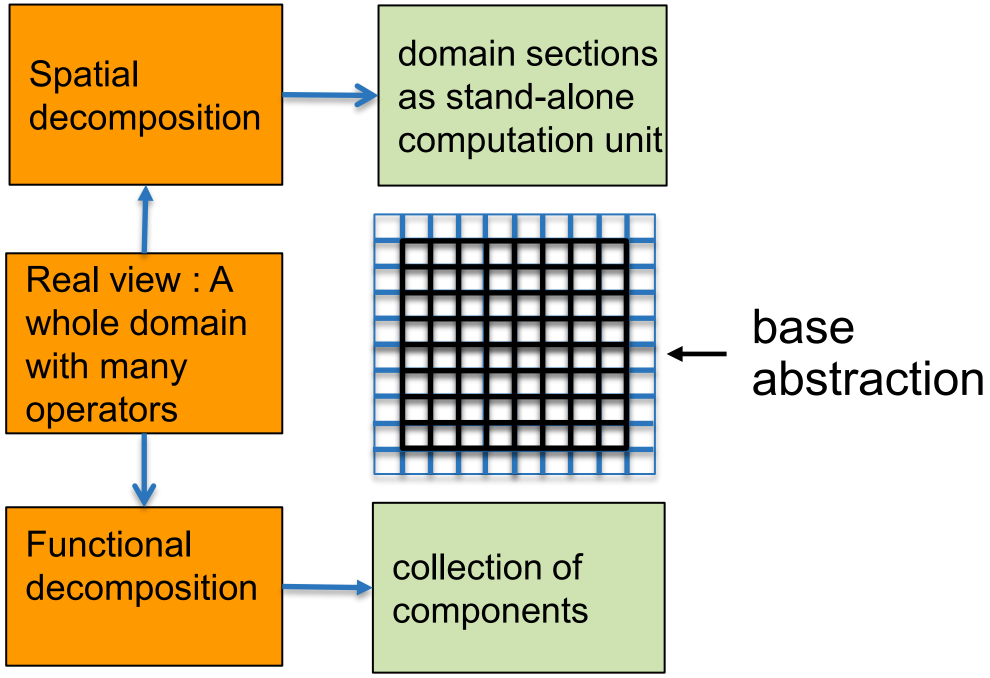
Connectivity



A Design Model for Separation of Concerns



Exploring design space – Abstractions



Constraints

- Only infrastructure components have global view
- All physics solvers have block view only

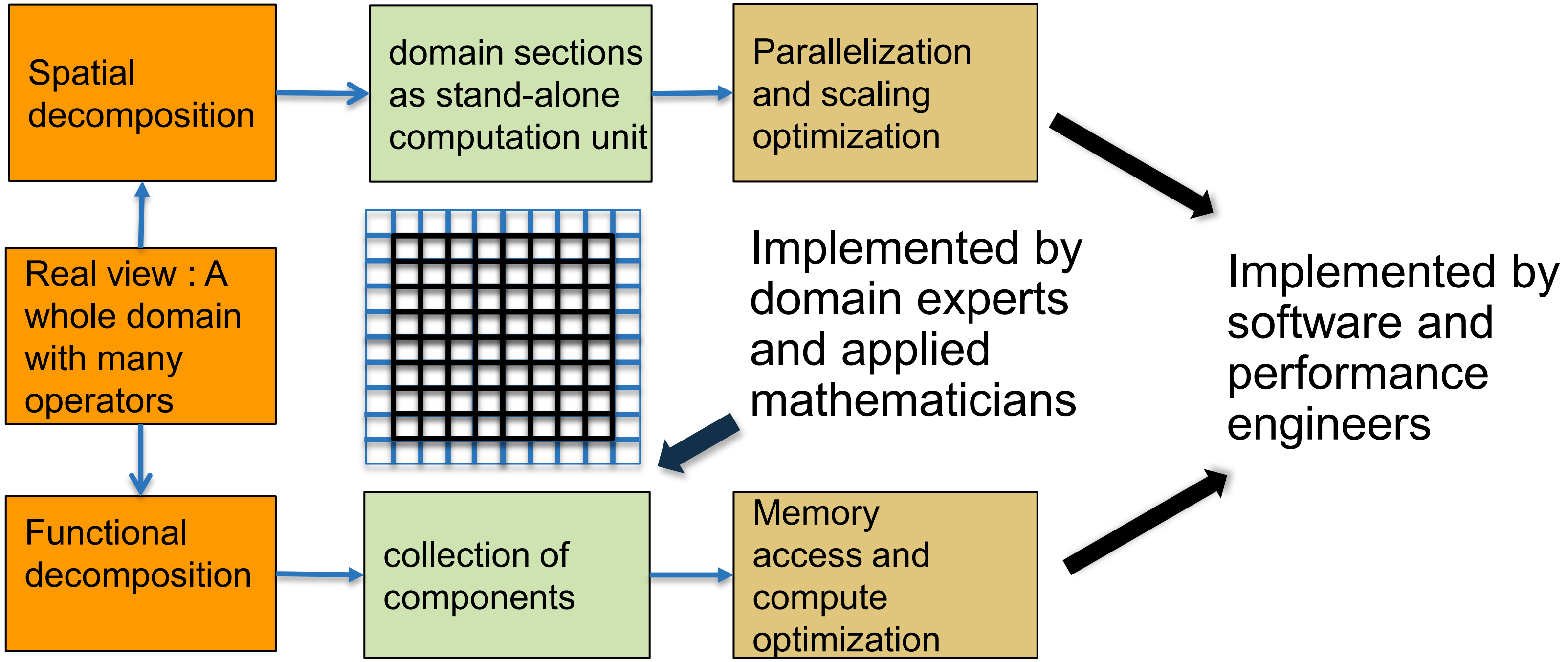
Other Design Considerations

- Data scoping
- Interfaces in the API

Minimal Mesh API

- Initialize_mesh
- Halo_fill
- Access_to_data_containers
- Reconcile_fluxes
- Regrid

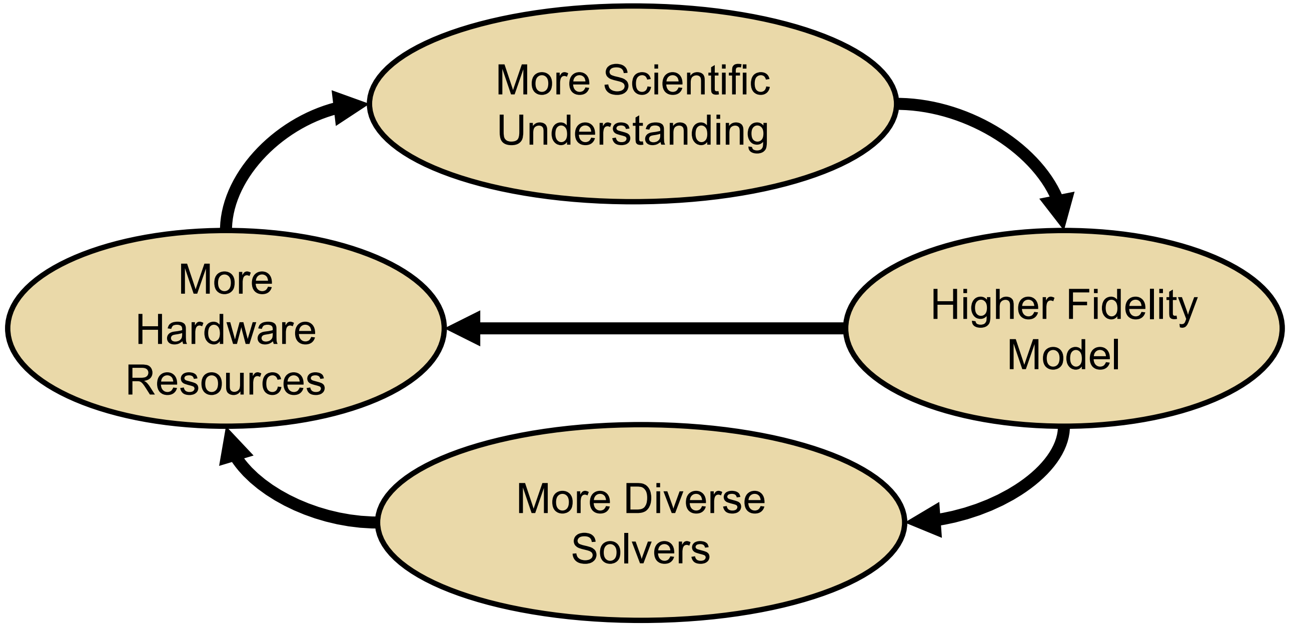
Separation of Concerns Applied

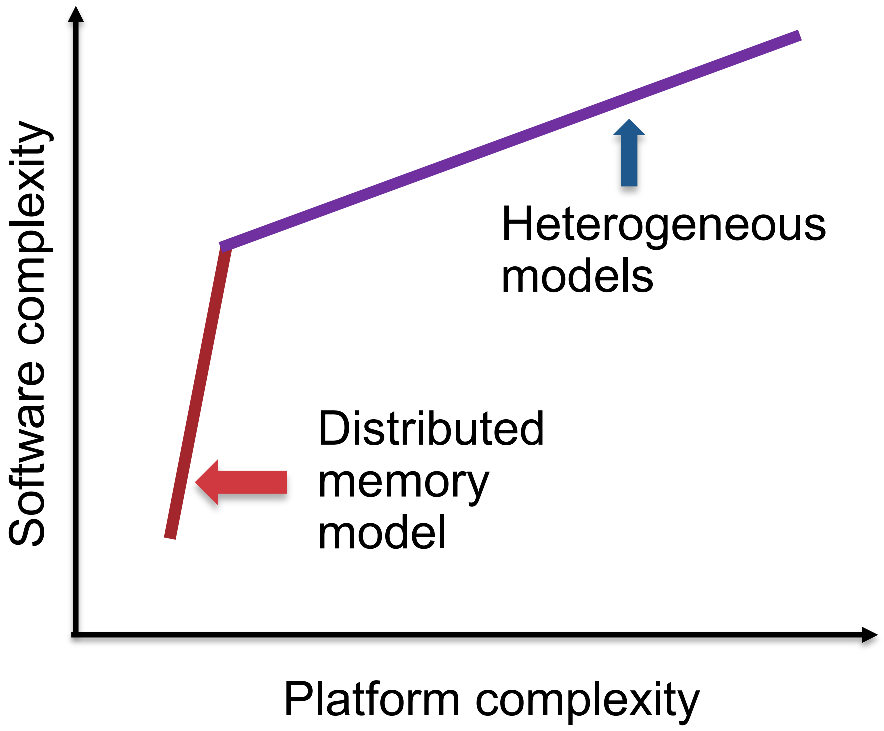


Takeaways so far

- Differentiate between slow changing and fast changing components of your code
- Understand the requirements of your infrastructure
- Implement separation of concerns
- Design with portability, extensibility, reproducibility and maintainability in mind

New Paradigm Because of Platform Heterogeneity





Mechanisms Needed by the Code

Mechanisms to unify expression of computation
- Minimize maintained variants of source suitable for all computational devices
- Reconcile differences in data structures

Mechanisms to move work and data to computational targets
- Moving between devices
  - Launching work at the destination
  - Hiding latency of movement
- Moving data off node

Mechanisms to map work to computational targets
- Figuring out the map
  - Expression of dependencies
  - Cost models
- Expressing the map

So, what do we need?

- Abstractions layers
- Code transformation tools
- Data movement orchestrators

Mechanisms Needed by the Code: Example of Flash-X

Mechanisms to unify expression of computation

Macros with inheritance

Mechanisms to move work and data to computational targets

Domain specific runtime

Mechanisms to map work to computational targets

DSL for recipes with code generator

Composability in the source
A toolset of each mechanism
Independent tool sets

State of Practice – Abstractions and Runtimes

- Still very focused on GPU
  - Majority of ECP applications park their data on the GPU and just work there
- Abstractions -- data structures and parallelization of loops
- Limitations
  - No way to handle algorithmic variants in a unified way
  - No way to transfer domain knowledge based possible optimizations to the tools
- None of the prevalent languages allow a good way to define data locality
  - Boutique HPC languages like chapel do – but chicken and egg problem with adoption

State of Practice – Abstractions and Runtimes

- Still very focused on GPU
  - Majority of ECP applications park their data on the GPU and just work there
- Abstractions -- data structures and parallelization of loops
- Limitations
  - No way to handle algorithmic variants in a unified way
  - No way to transfer domain knowledge based possible optimizations to the tools
- None of the prevalent languages allow a good way to define data locality
  - Boutique HPC languages like chapel do – but chicken and egg problem with adoption

The holy grail for scientists – write equation and generate code

Very limited success in some domains

Final takeaways

- Requirements gathering and intentional design are indispensable for sustainable software development
- Many books and online resources available for good design principles
- Research software poses additional constraints on design because of its exploratory nature
  - Scientific research software has further challenges
  - High performance computing research software has even more challenges
  - That are further exacerbated by the ubiquity of accelerators in platforms
- Separation of concerns at various granularities, and abstractions enable sustainable software design

References

- Dubey Anshu, “**Insights from the software design of a multiphysics multicomponent scientific code**” Computing in Science & Engineering, 2021. DOI:10.1109/MCSE.2021.3069343
- Dubey, Anshu, et al. "**Flash-X: A multiphysics simulation software instrument.**" *SoftwareX* 19 (2022): 101168. DOI:10.1016/j.softx.2022.101168
- Rudi, Johann, et al. "**CG-Kit: Code Generation Toolkit for Performant and Maintainable Variants of Source Code Applied to Flash-X Hydrodynamics Simulations.**" *arXiv preprint arXiv:2401.03378* (2024).
- O’Neal, Jared, et al. "**Domain-specific runtime to orchestrate computation on heterogeneous platforms.**" *European Conference on Parallel Processing*. Cham: Springer International Publishing, 2021. DOI:10.1007/978-3-031-06156-1_13
- Dubey, Anshu, et al. "**A tool and a methodology to use macros for abstracting variations in code for different computational demands.**" *Future Generation Computer Systems* (2023). DOI:10.1016/j.future.2023.07.014