# Scientific Software Design

*Presented by*

**COLABS: Collaboration for Better Software for Science**

Anshu Dubey (she/her)
Argonne National Laboratory

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

*In collaboration with*

Contributors: Anshu Dubey (ANL), Mark C. Miller (LLNL), David E. Bernholdt (ORNL)

*With prior support from*

# License, Citation and Acknowledgements

## License and Citation

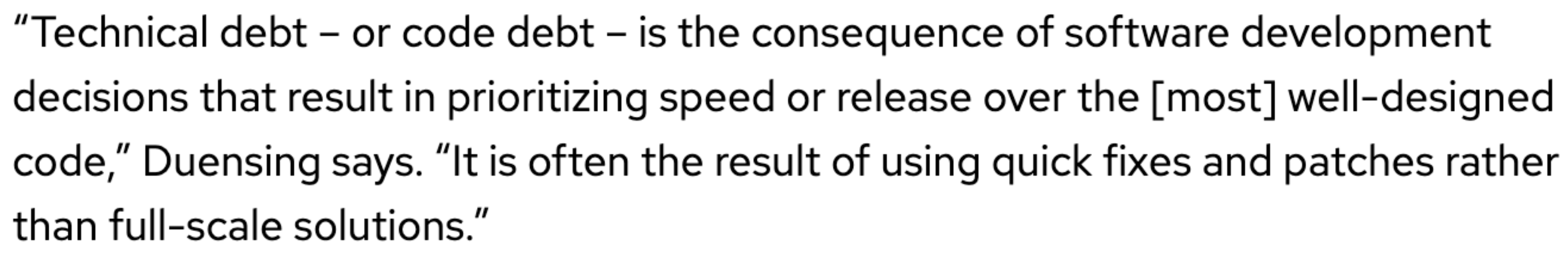
- **The requested citation the overall tutorial is:** Anshu Dubey, David E. Bernholdt, Todd Gamblin, and Jared O’Neal, Software Productivity and Sustainability track, in Argonne Training Program on Extreme-Scale Computing, St. Charles, Illinois, 2024. DOI: 10.6084/m9.figshare.26384188.
- Individual modules 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 (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 supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Next-Generation Scientific Software Technologies (NGSST) program.
- 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](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

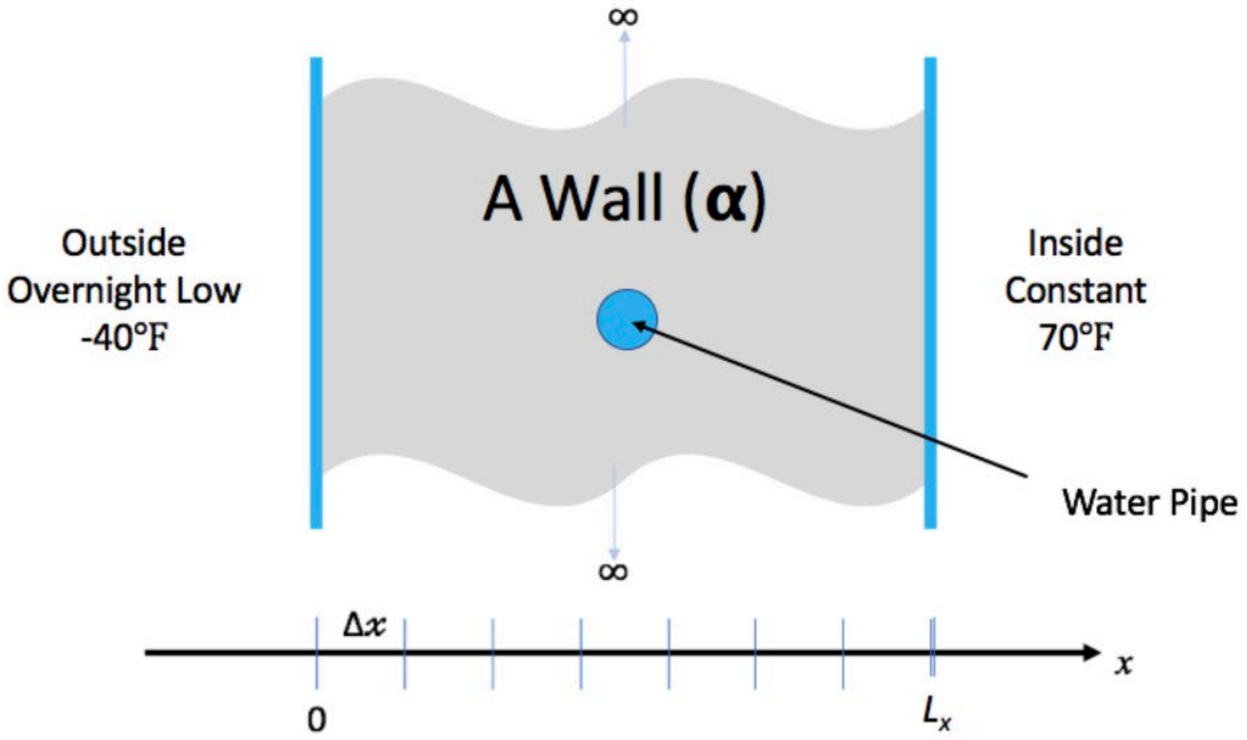
- ☐ Understand component hierarchy
- ☐ Figure out connectivity among components
- ☐ Articulate dependencies

# Example 1 – Problem Description

We have a house with exterior walls made of single material of thickness $L_x$
The wall has some water pipes shown in the picture.

The inside temperature is kept at 70 degrees. But outside temperature is expected to be -40 degrees for 15.5 hours.

Will the pipes freeze before the storm is over



# Requirements gathering

- To solve heat equation we need:
  - a discretization scheme
  - a driver for running and book-keeping
  - an integration method to evolve solution
  - Initial conditions
  - Boundary conditions
- To make sure that we are doing it correctly we need:
  - Ways to inspect the results
  - Ways of verification

# Decomposition

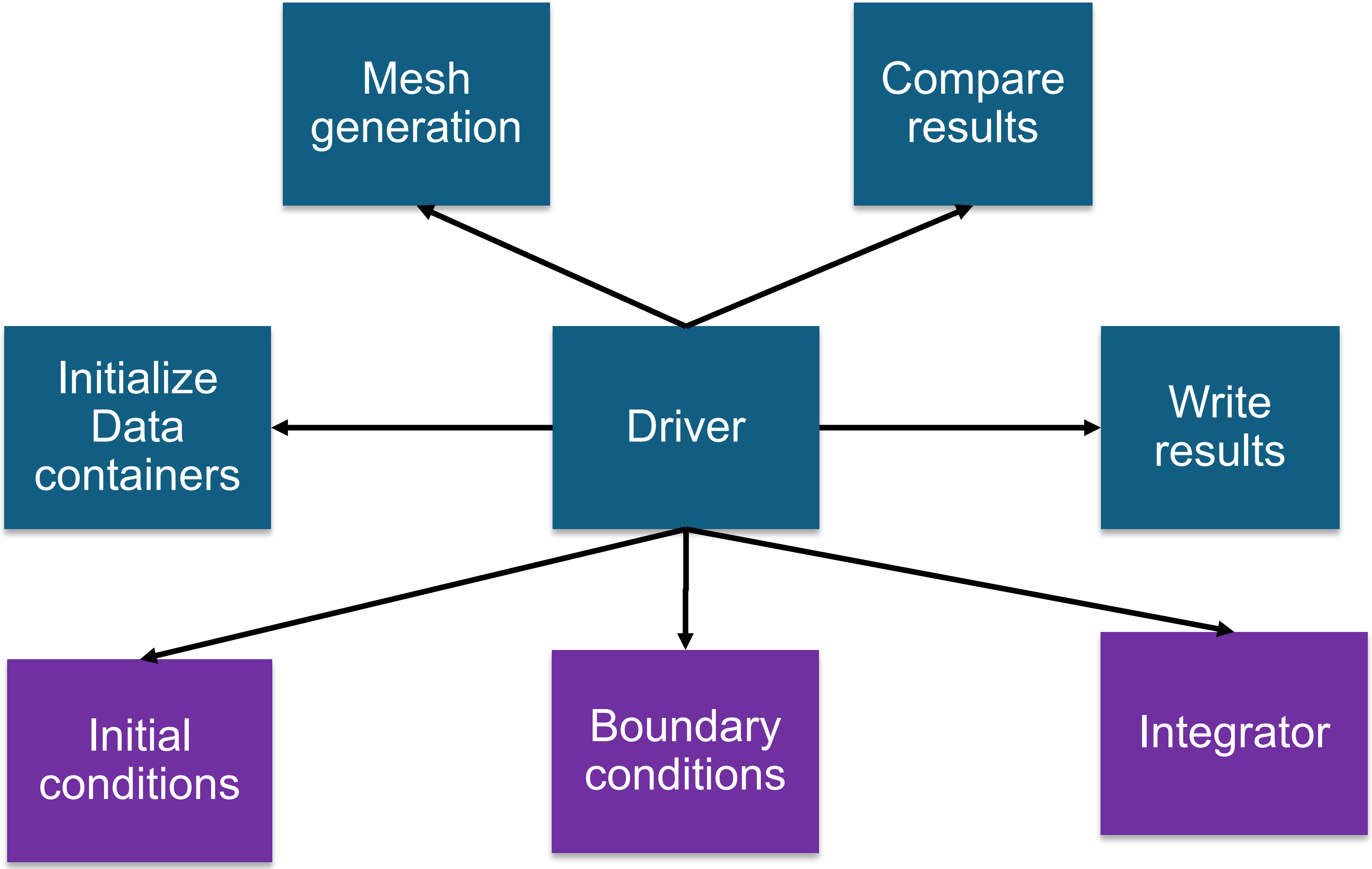
## This is a small design space

- Several requirements can directly map to components – in this instance functions
  - Driver
  - Initialization – data containers
  - Mesh initialization – applying initial conditions
  - Integrator
  - I/O
  - Boundary conditions
  - Comparison utility

## Binning components

- Components that will work for any application of heat equation
  - Driver
  - Initialization – data containers
  - I/O
  - Comparison utility
- Components that are
  - Mesh initialization – applying initial conditions
  - Integrator
  - Boundary conditions

# Connectivity


Mesh generation

Compare results

Initialize Data containers

Driver

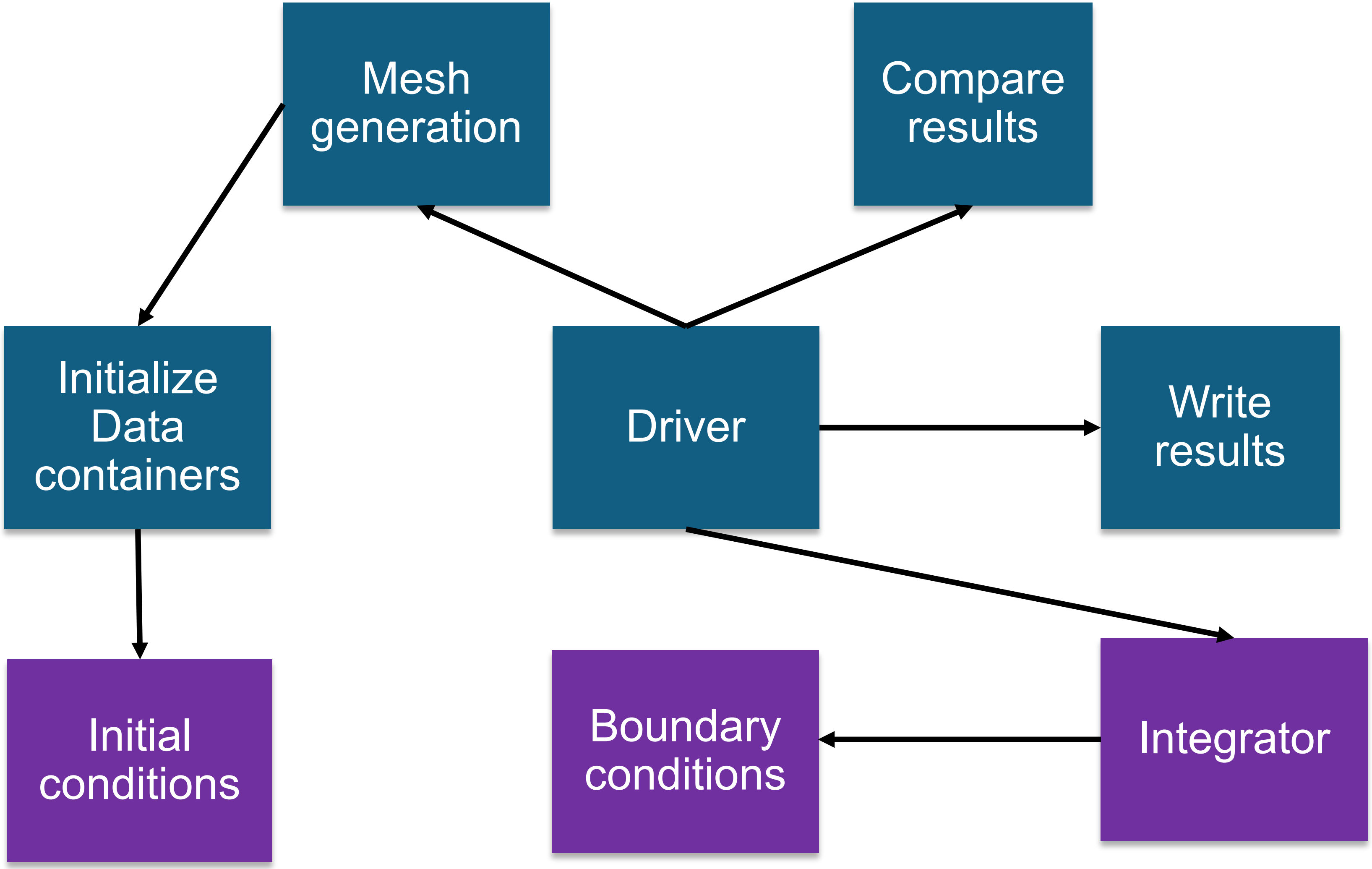
Write results

Initial conditions

Boundary conditions

Integrator

# Connectivity – alternative possibility



Mesh generation

Compare results

Initialize Data containers

Driver

Write results

Initial conditions

Boundary conditions

Integrator

# Resources for Independent Exploration

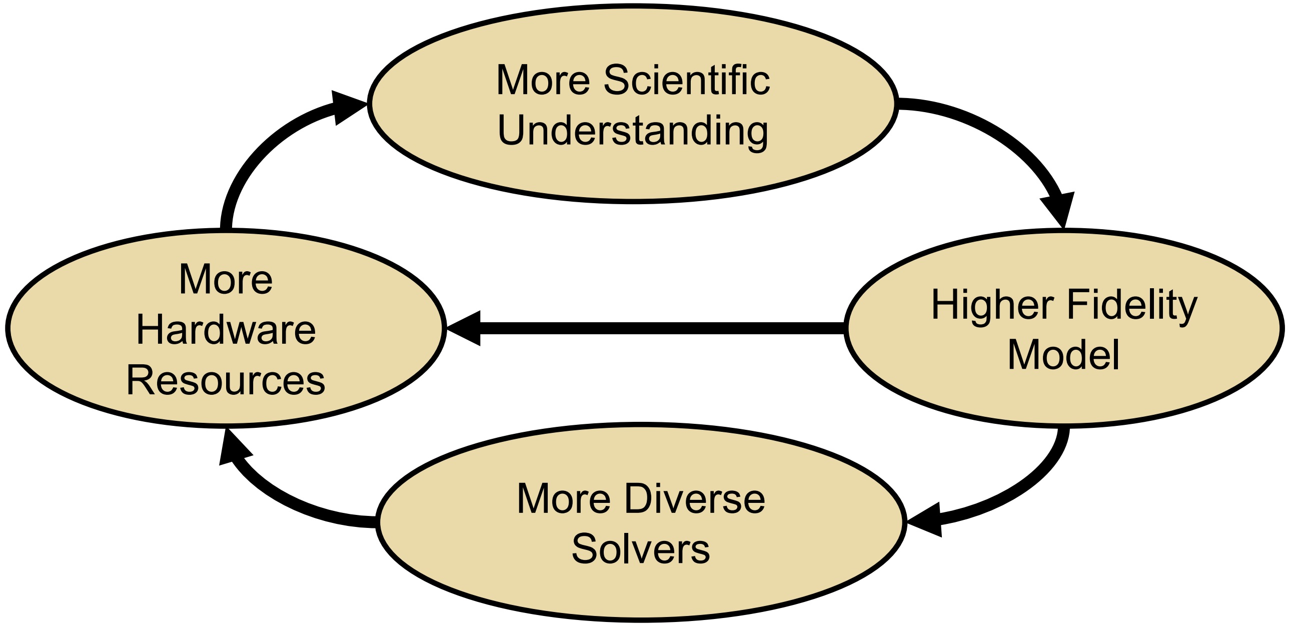
- Code repository in python

https://github.com/abiswas-odu/heateq-design-intersect-2023

- A few possibilities of design exploration
  - Did we need three different interfaces for update solution ?
  - What would have been needed to make it into one interface
- Explore the whole exercise in C++ on your own checkout

https://xsdk-project.github.io/MathPackagesTraining2020/lessons/hand_coded_heat/

# 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

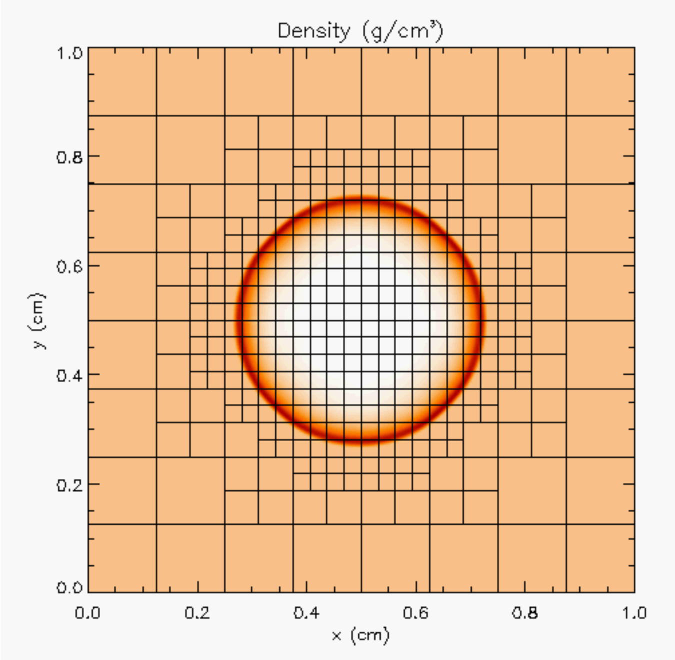
- Work with different lifecycles
  - Long-lasting vs quick changing
  - Logically vs mathematically complex

- 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
- 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
  - Remesh when refinement patterns change
- I/O
  - Getting runtime parameters and possibly initial conditions
  - Writing checkpoint and analysis data

# Connectivity

Initial conditions

Compare results

Mesh

Driver

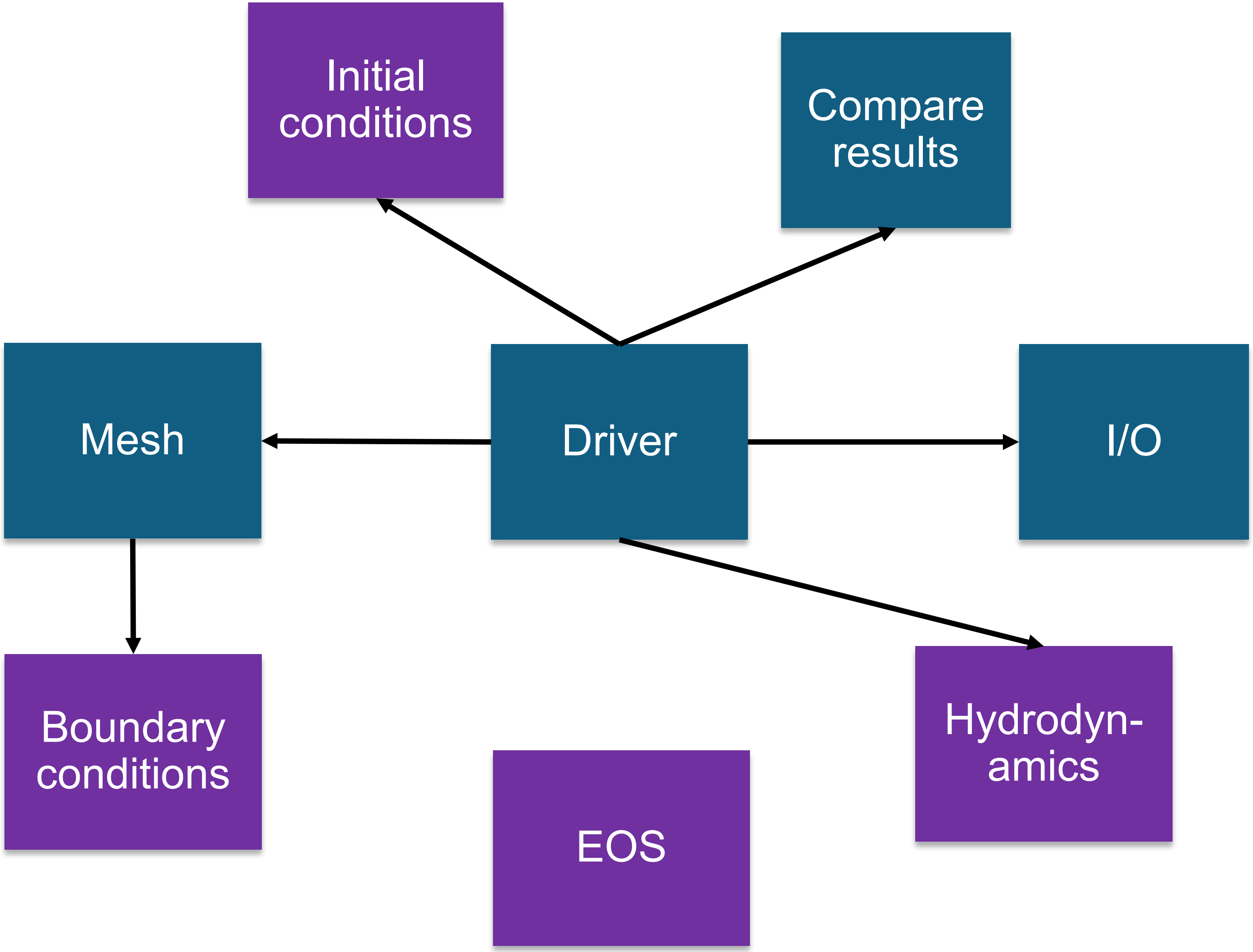
I/O

Boundary conditions

Hydrodynamics

EOS

# Connectivity


Initial conditions

Compare results

Mesh

Driver

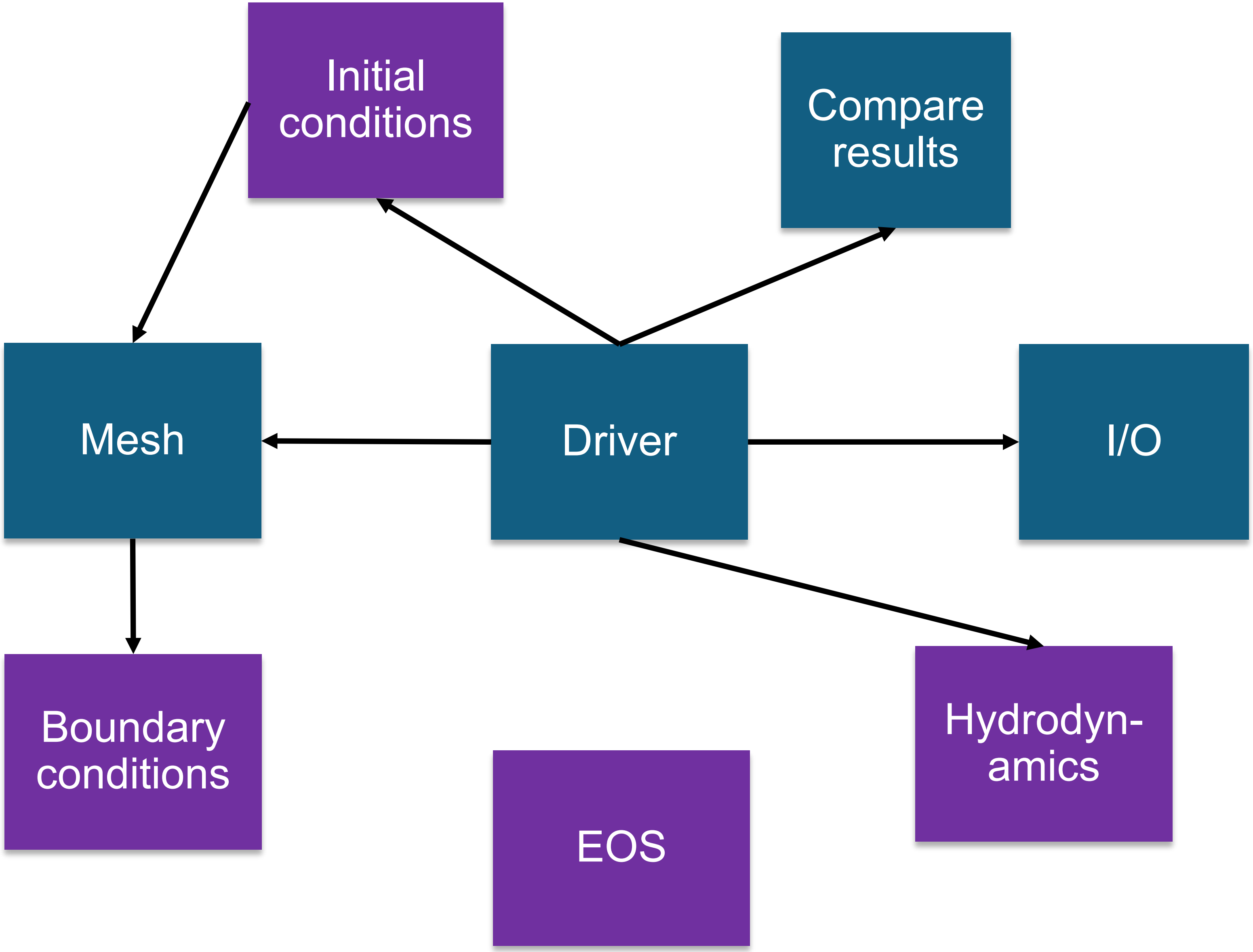
I/O

Boundary conditions

EOS

Hydrodyn-amics

# Connectivity



Initial conditions

Compare results

Mesh

Driver

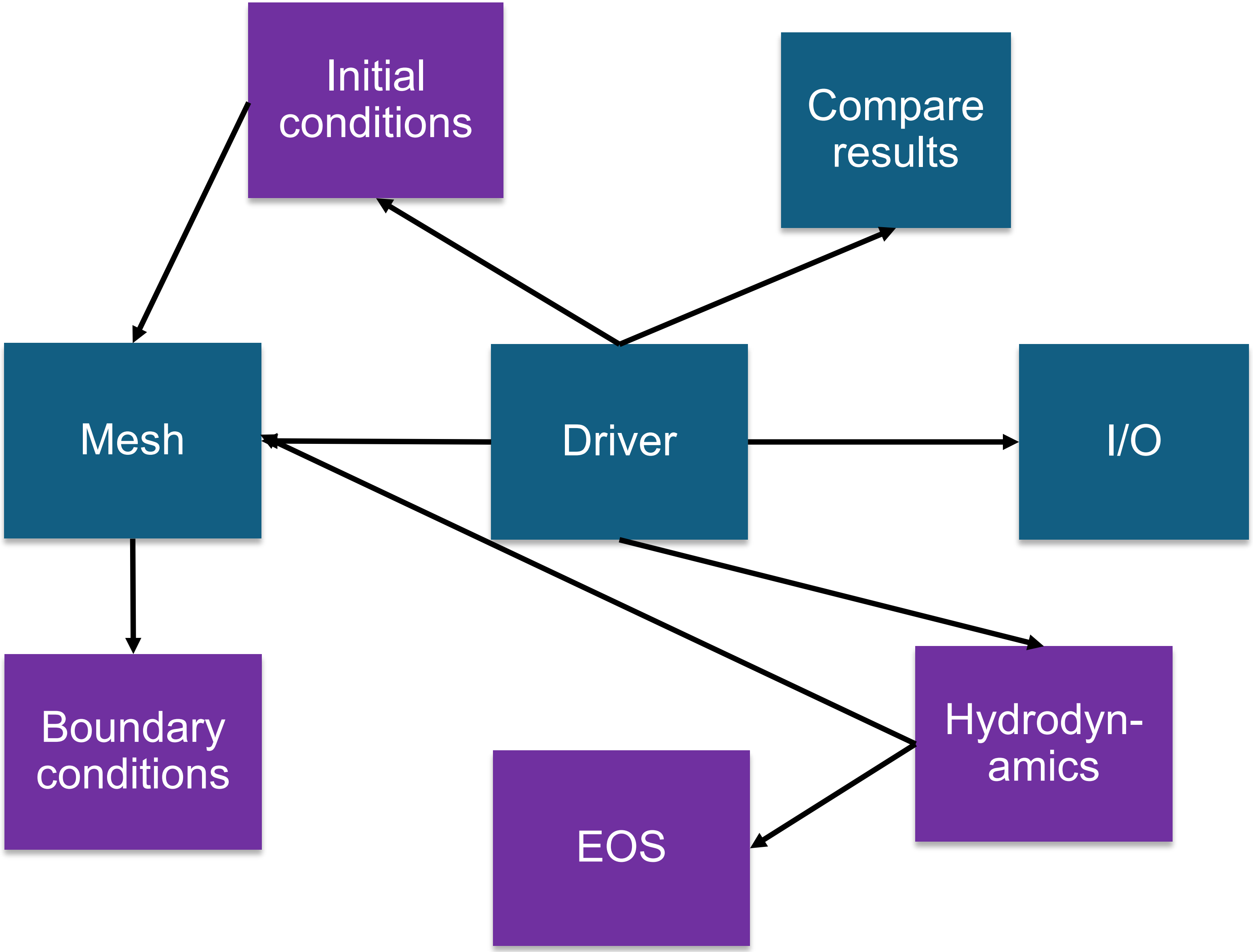
I/O

Boundary conditions

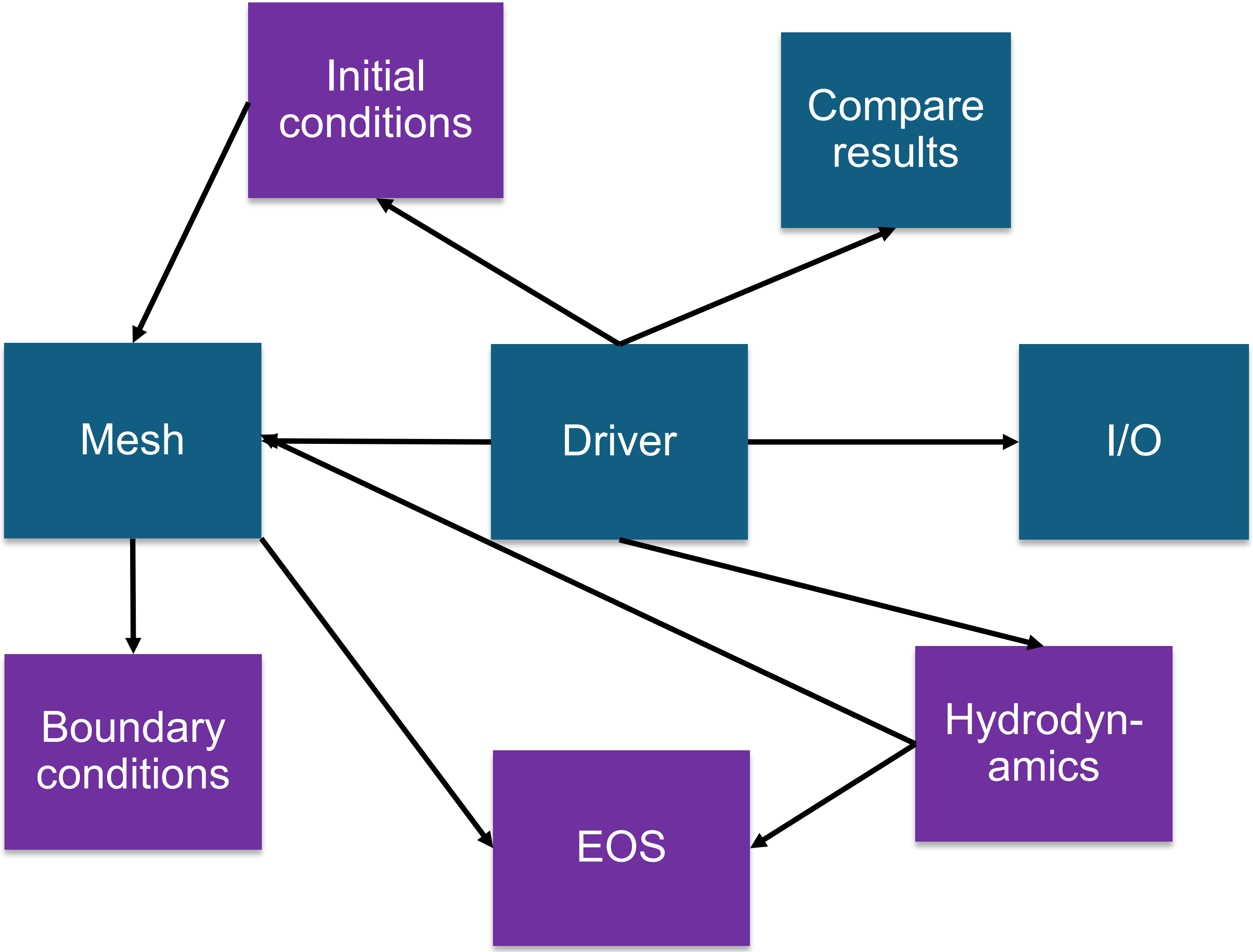
EOS

Hydrodynamics

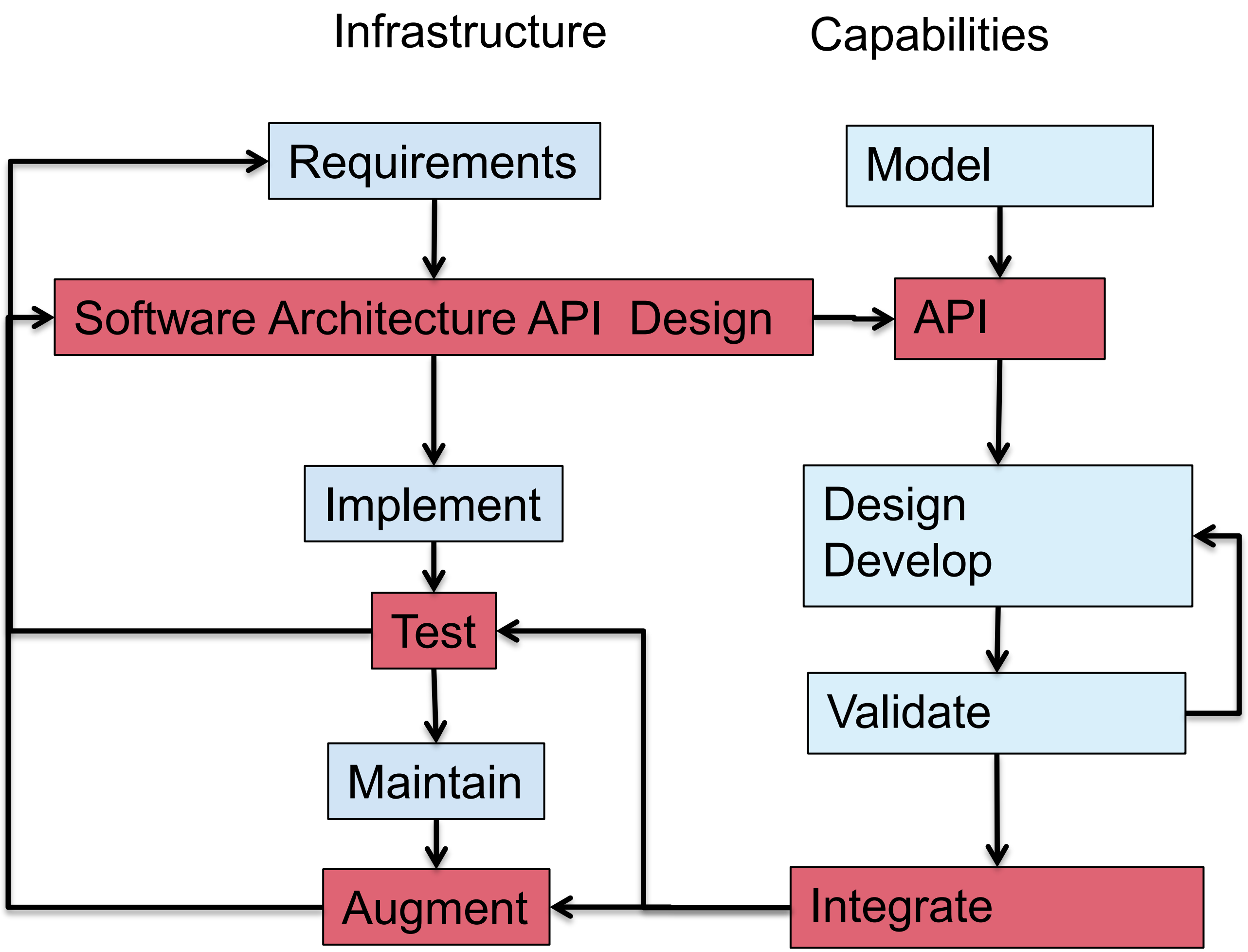
# Connectivity



# Connectivity



# A Design Model for Separation of Concerns



Infrastructure

Capabilities

Requirements

Software Architecture API Design

API

Model

Implement

Design
Develop

Test

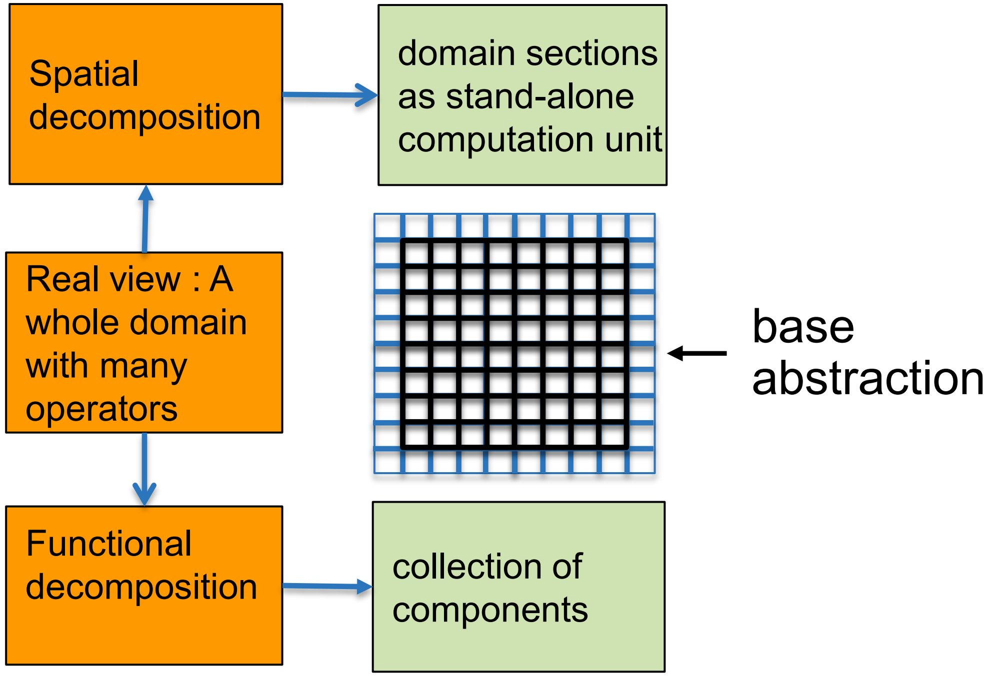
Validate

Maintain

Augment

Integrate

# Exploring design space – Abstractions



Spatial decomposition

domain sections as stand-alone computation unit

Real view : A whole domain with many operators

base abstraction

Functional decomposition

collection of components

## Constraints

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

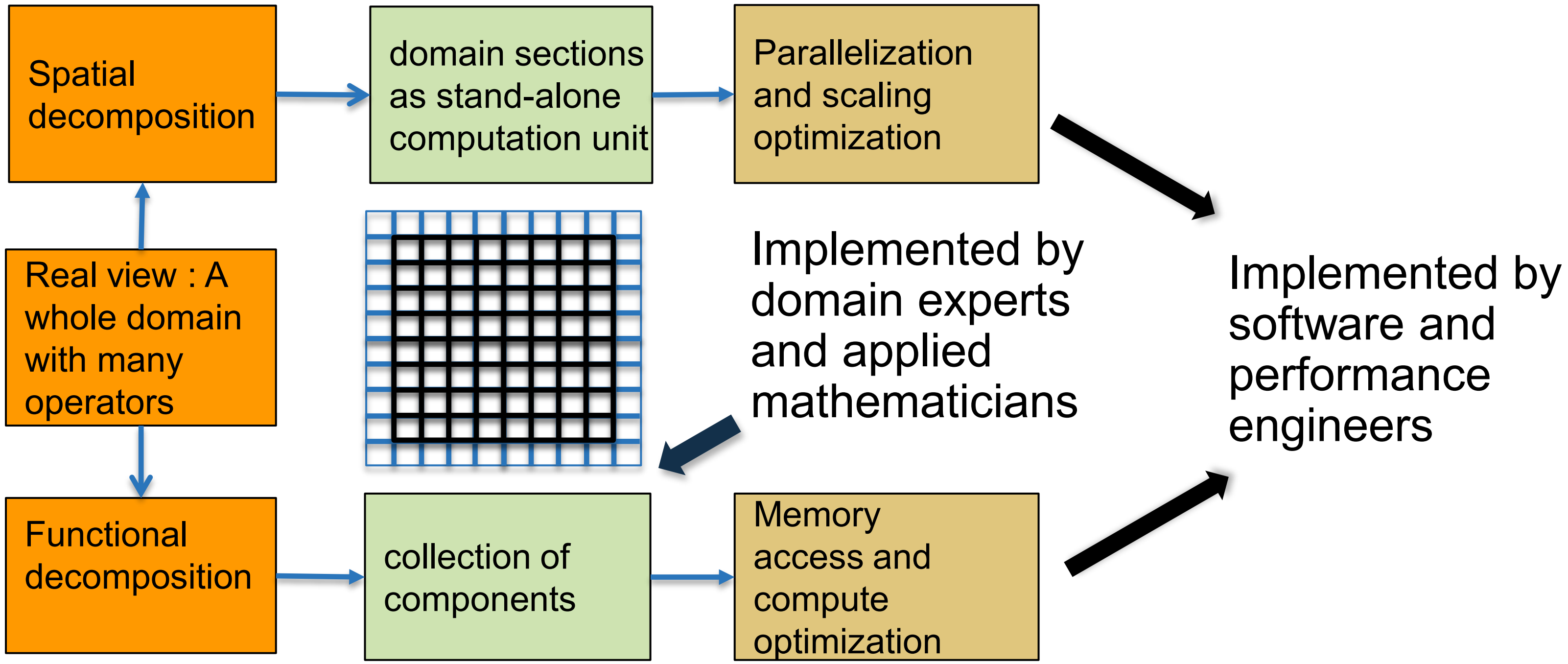
## Other Design Considerations

- Data scoping
- Interfaces in the API

**Minimal Mesh API**

- Initialize_mesh
- Halo_fill
- Access_to_data_containers
- Reconcile_fluxes
- Regrid

# Separation of Concerns Applied



Spatial decomposition

domain sections as stand-alone computation unit

Parallelization and scaling optimization

Real view : A whole domain with many operators

Implemented by domain experts and applied mathematicians

Implemented by software and performance engineers

Functional decomposition

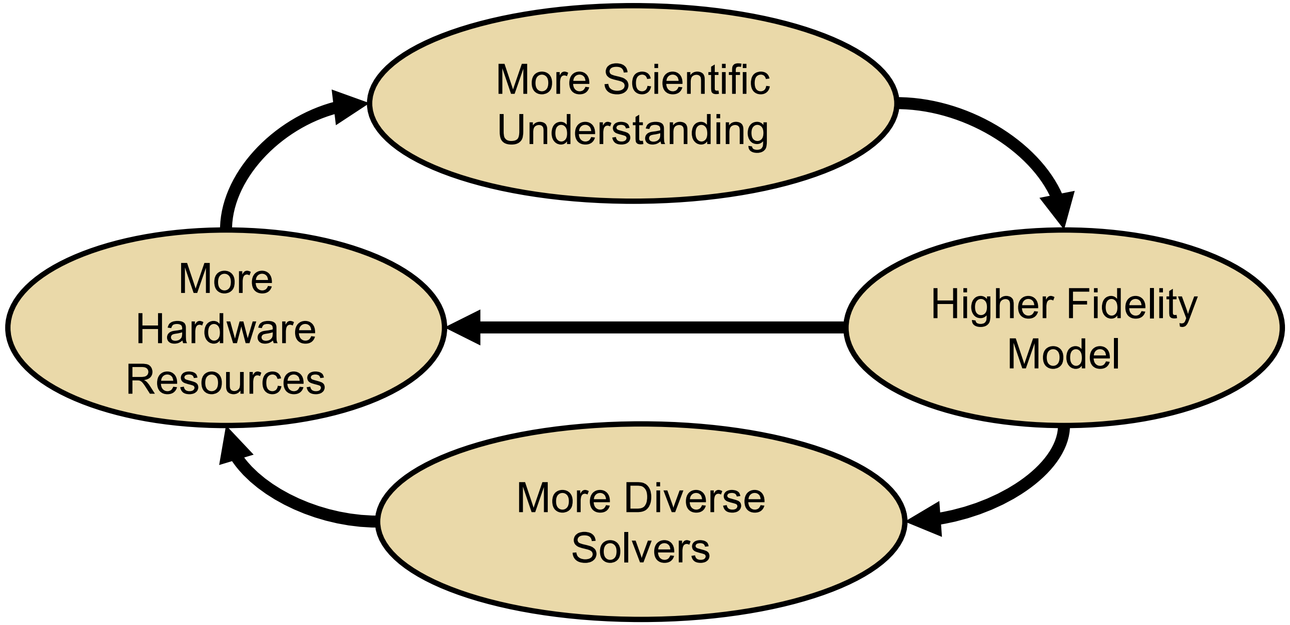
collection of components

Memory access and compute optimization

# 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

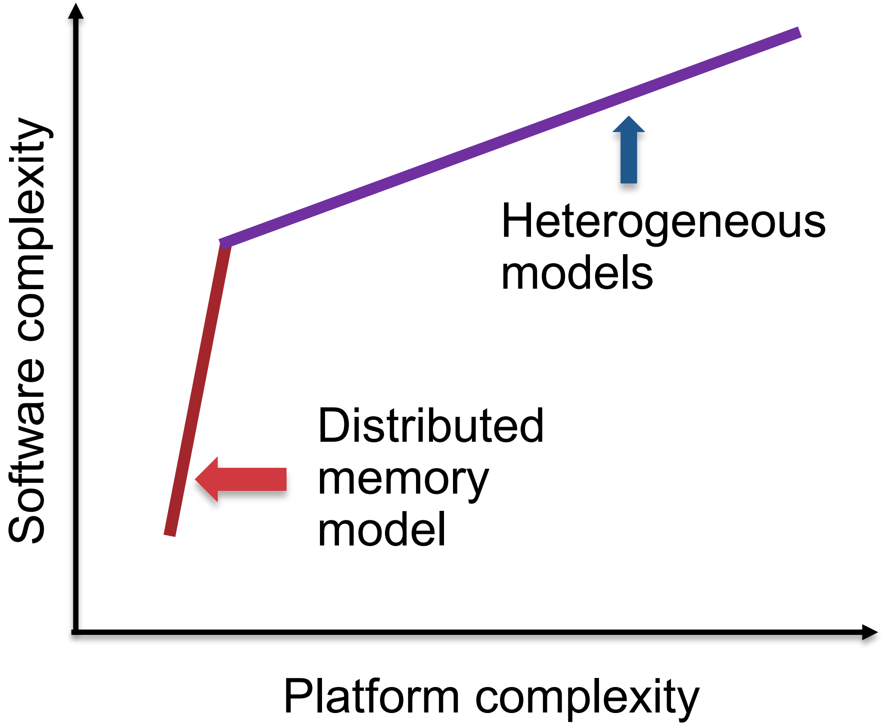


More Scientific Understanding

Higher Fidelity Model

More Diverse Solvers

More Hardware Resources



# 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

Is there another way?

# State of Practice – Abstractions and Runtimes

- Still very focused on GPU
  - Majorit t work there
- Abstracti os
- Limitatio
  - No way
  - No way ons to the tools
- None of t efine data locality
  - Boutique HPC languages like chapel do – but chicken and egg problem with adoption

**We have been developing one for Flash-X – started under ECP and TEAMS, continuing with RAPIDS and ENAF**

The holy grail for scientists – write equation and generate code

Very limited success in some domains

Is there another way?

# Orthogonal Axes of Challenges and Optimization

- Separate out arithmetic and control flow
  - Make arithmetic invariant
  - Turn separate pieces into building blocks using macros

```
[hy_fluxesSec1]
args= XL, XR,limits
definition =
@M loop_begin(limits)
   if (flux(1@M indices) > 0.) then
       call doSection1(XL(:@M indices), ……)
   else
       call doSection1(XR(:@M indices), …….)
   end if
 @M loop_end
```

❑ Permit alternative definitions for all the macros as needed

❑ Build in arbitration mechanism for picking the right definition

❑ This code section can be invoked as
  @M hy_fluxesSec1(uLeft,uRight,blkLimits)

**Alternative Definitions**

**For all spatial points at once time.**

```
[indices]
definition =
  ,i,j,k

[loop_begin]
args = limits
definition=
  do k = limits(LOW,KAXIS),limits(LOW,KAXIS)
     do j = limits(LOW,JAXIS),limits(LOW,JAXIS)
        do I = limits(LOW,IAXIS),limits(LOW,IAXIS)

[loop_end]
definition =
        enddo
     enddo
  enddo
```

**For one spatial point at a**

```
indices]
definition =

[loop_begin]
 args = limits
 definition=

[loop_end]
definition =
```

# Alternatively

```
[hy_fluxesSec1]
args= XL, XR,limits
definition =
@M loop_begin(limits)
   if (flux(1@M indices) > 0.) then
       @M doSection1(XL)
   else
      @M doSection1(XR)
   end if
 @M loop_end
```

```
[doSection1]
args=uDir
definition =
  … some computation
uDir(:@M indices) =  res
```

## Alternative Definitions

**For all spatial points at once**

```
[indices]
definition =
  ,i,j,k

[loop_begin]
args = limits
definition=
  do k = limits(LOW,KAXIS),limits(LOW,KAXIS)
      do j = limits(LOW,JAXIS),limits(LOW,JAXIS)
         do I = limits(LOW,IAXIS),limits(LOW,IAXIS)

[loop_end]
definition =
       enddo
     enddo
   enddo
```

**For one spatial point at a time**

```
[indices]
definition =

[loop_begin]
  args = limits
 definition=

[loop_end]
definition =
```

With macros it is possible to use any arbitrary code section as a building block

# Orthogonal Axes of Challenges and Optimization

- Have a method for expressing algorithmic variants
  - Without delving into the details of the arithmetic
- Example -- Flash-X supports two block-structured AMR grid backends
  - Paramesh: Octree-based, AMReX: Level-based
- Each has different preferences for flux correction at fine-coarse boundaries
- For higher order RK integration Communication avoidance – telescoping mode

```
do all_blocks
  ! hydrodynamics updates
end do
call communicate_fluxes() ! p2p communication
do all_blocks
  ! flux correction
end do
```

```
do lev = max_level, 1, -1
  call communicate_fluxes() ! p2p communication
  do blocks_on(level = lev)
    ! hydrodynamics updates
    ! flux correction
  end do
end do
```

```
do stage = 1, max_stage
  call fill_guardcells() ! p2p communication
  do all_blocks
    ! block initializations
    ! intra stage calculations
  end do
end do
```

```
call fill_guardcells() ! p2p communication
do all_blocks
  ! block initializations
  do stage = 1, max_stage
    ! intra stage calculations
  end do
end do
```

# Orthogonal Axes of Challenges and Optimization

- ☐ Have a way of rearranging data locality and moving data and computation
  - ☐ Let the human-in-the-loop dictate this

CG-Kit – recipes in python
-- templates for different variants
-- express where to compute what
-- emit code in Fortran/C/C++

Milhoja – flatten/decompose data and move it to the target
-- combine data into one data packet
-- decompose into smaller computational sections if needed

- If tools only execute what they are told to, they are simpler
- Code generation is our friend – especially when it is simple forward map
  - And is not entangled with the details of the arithmetic

- ☐ If N blocks are sent to the device we need N copies of all block-wise scratch
- ☐ For all data items we need device pointers
- ☐ Code internally decorated with directives

# Code Generators

- Two Classes
  - Data packet generators
    - Parse the interface files
    - Collect all data to be put into a data packet
    - Generate code that will flatten all data into data packets
  - Task function generators
    - Consolidate functions to be invoked
    - Bookended by internode communication
    - Unpack data packets
- Decorate interface definitions with needed metadata

Example -- *this link will work only if you have access to the Flash-X code repository. Please email flash-x@lists.cels.anl.gov with your github username to get access*

# 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