Simplifying Multiphysics Through Application Composition

Derek Gaston

Idaho National Laboratory
MIT Computational Reactor Physics Group

Cody Permann, Derek Gaston, David Andrs,
John Peterson, Andrew Slaughter, Dmitry Karpeyev,
Rich Martineau

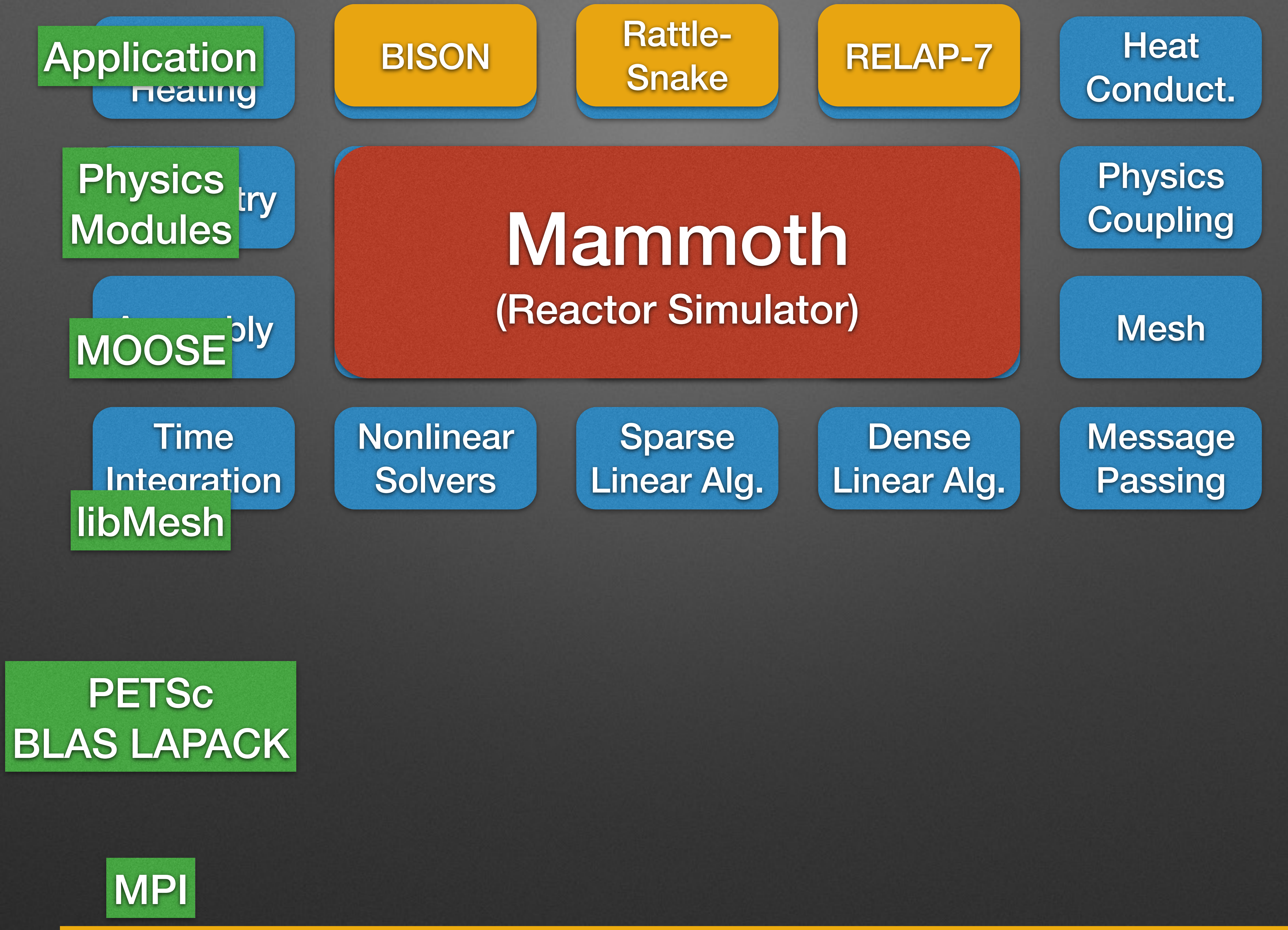
DRY

Don’t Repeat Yourself!

Local Application

- Application of DRY within one application is obvious:
  - Functions
  - Object-oriented design
  - Macros
  - etc.
- DRY for really common activities?
  - Libraries
  - Native Language Support (i.e. threading support in C++11)
- What about leveraging multiple applications across research groups and disciplines?
  - Head in the sand?
  - Development of “coupling” codes?

Finite-Element Reactor Fuel Simulation



Modularity is Key

- Software engineers tell us that data should only be accessed through strict interfaces with code having good separation of responsibilities.
  - Allows for “decoupling” of code
  - Leads to more reuse and less bugs
- They’ve never coded FEM!
  - Shape functions, DoFs, Elements, QPs, Material Properties, Analytic Functions, Global Integrals, Transferred Data and More are needed in FEM assembly.
  - Makes computational science codes brittle and hard to reuse
- A consistent set of “modules” are needed that carry out common actions
- These modules should be separated by interfaces

MOOSE “Systems”

- Actions
- Auxiliary Kernels
- Auxiliary Variables
- BCs
- Constraints
- Dampers
- DGKernels
- DiracKernels
- Executioners
- Functions
- GeomSearch
- ICs
- Indicators
- Kernels
- Markers
- Materials
- Mesh
- MeshModfiers
- MooseApps
- MultiApps
- Outputs
- Oversampling
- Postprocessors
- Preconditioners
- Predictors
- Restart
- Splits
- TimeIntegrators
- TimeSteppers
- Transfers
- UserObjects
- Variables

Systems (cont.)

- Systems break apart responsibility
- No direct communication between Systems
  - Everything flows through MOOSE interfaces
- Objects can be mixed and matched to achieve simulation goals
  - They "operate in a vacuum"
  - Incoming data can be changed dynamically
  - Outputs can be manipulated (e.g. multiplication by *r* for cylindrical coordinates)
- **Objects from one Application are no different than those from another.**
  - **An object, by itself, can be lifted from one Application and used by another.**

DarcyConvection Kernel

```
DarcyConvection::DarcyConvection(const std::string & name, InputParameters parameters) :
    Kernel(name, parameters),

   _pressure_gradient(coupledGradient("darcy_pressure")),
   _pressure_var(coupled("darcy_pressure")),

    _permeability(getMaterialProperty<Real>("permeability")),
    _porosity(getMaterialProperty<Real>("porosity")),
    _viscosity(getMaterialProperty<Real>("viscosity")),
    _density(getMaterialProperty<Real>("density")),
    _heat_capacity(getMaterialProperty<Real>("heat_capacity"))
{
}
```

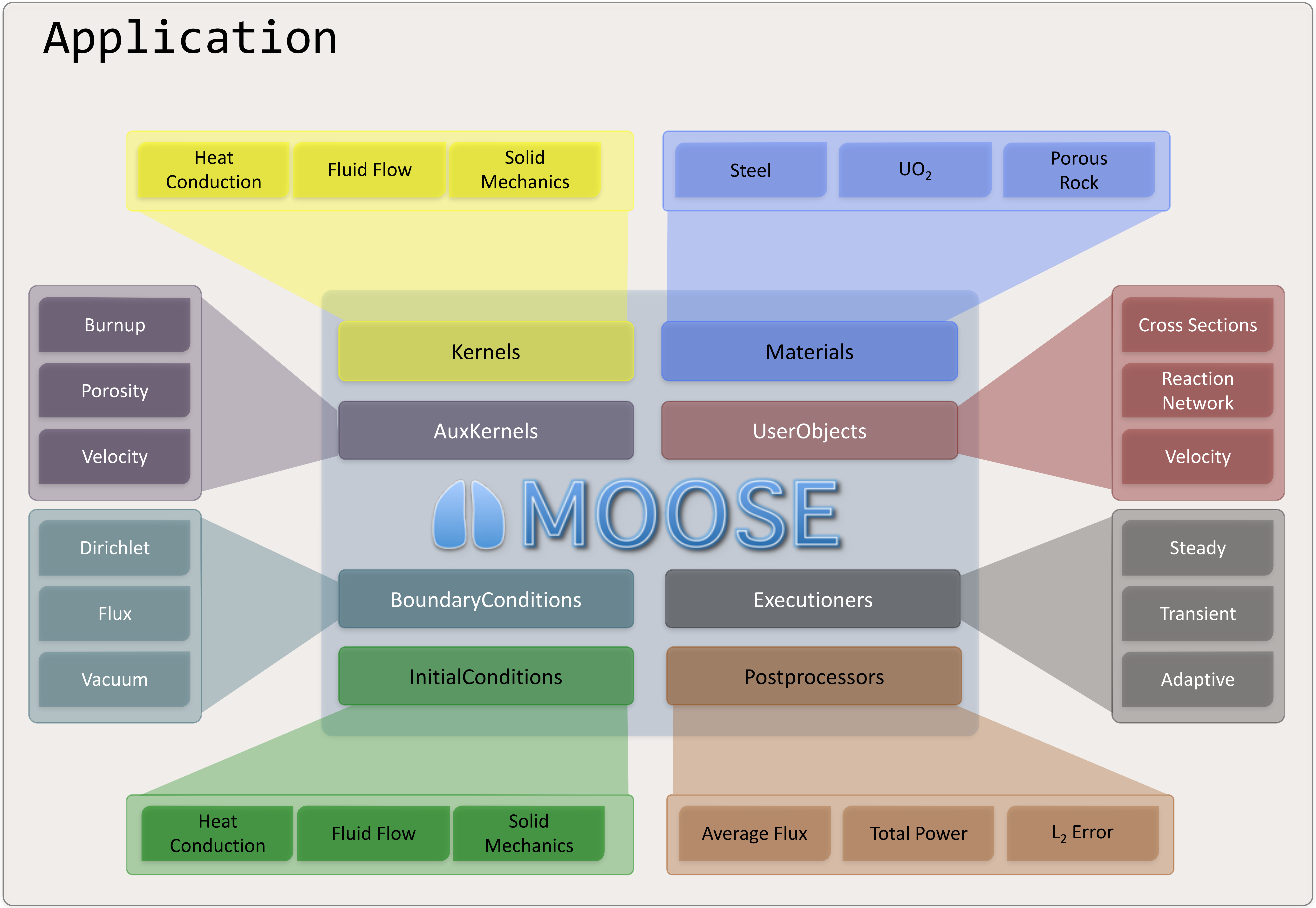
$$\nabla \cdot \vec{u} = 0$$

$$\vec{u} = -\frac{\mathbf{K}}{\mu}\nabla p$$

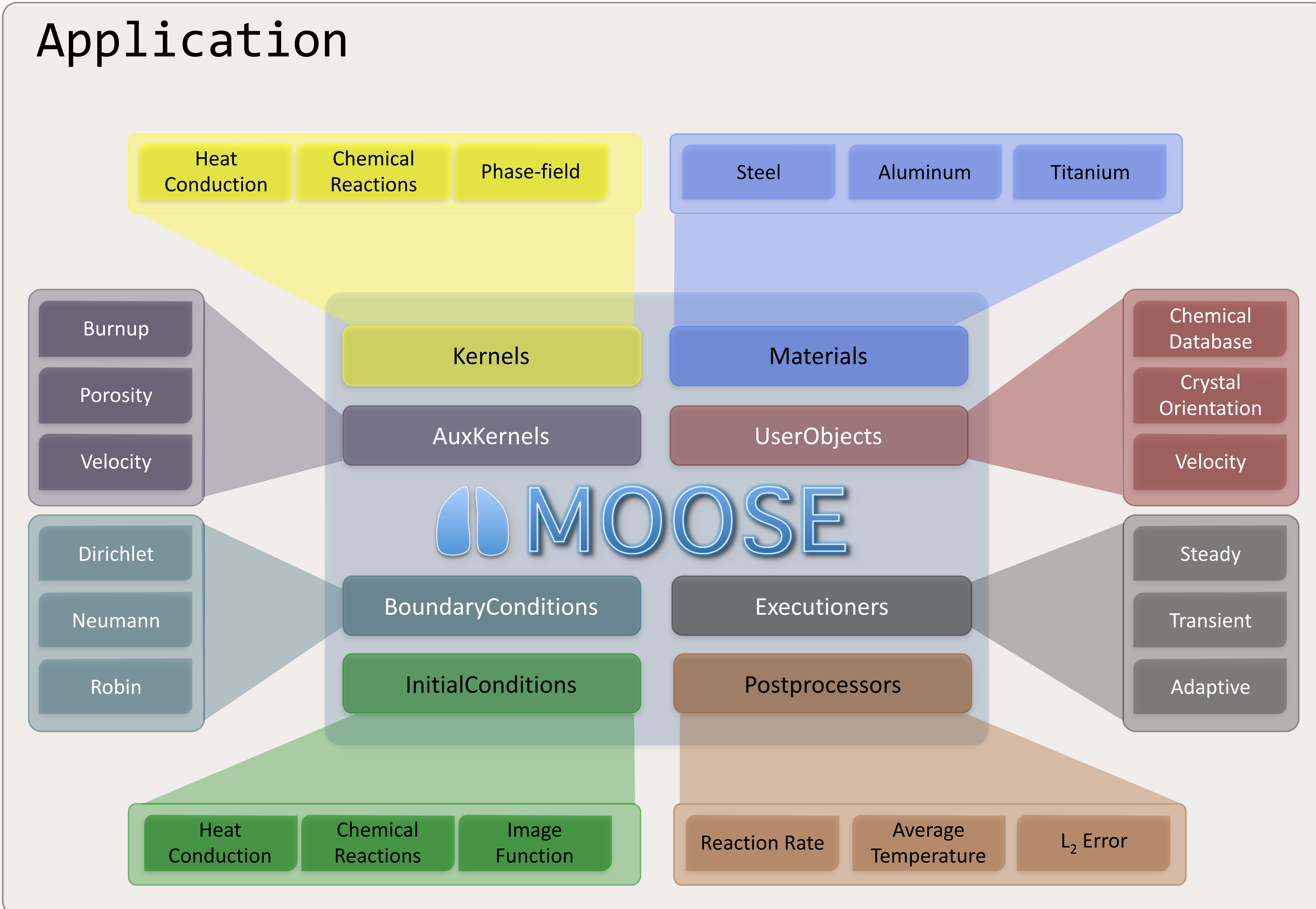
$$C\frac{\partial T}{\partial t} + C\epsilon\vec{u}\cdot\nabla T - \nabla\cdot k\nabla T = 0$$

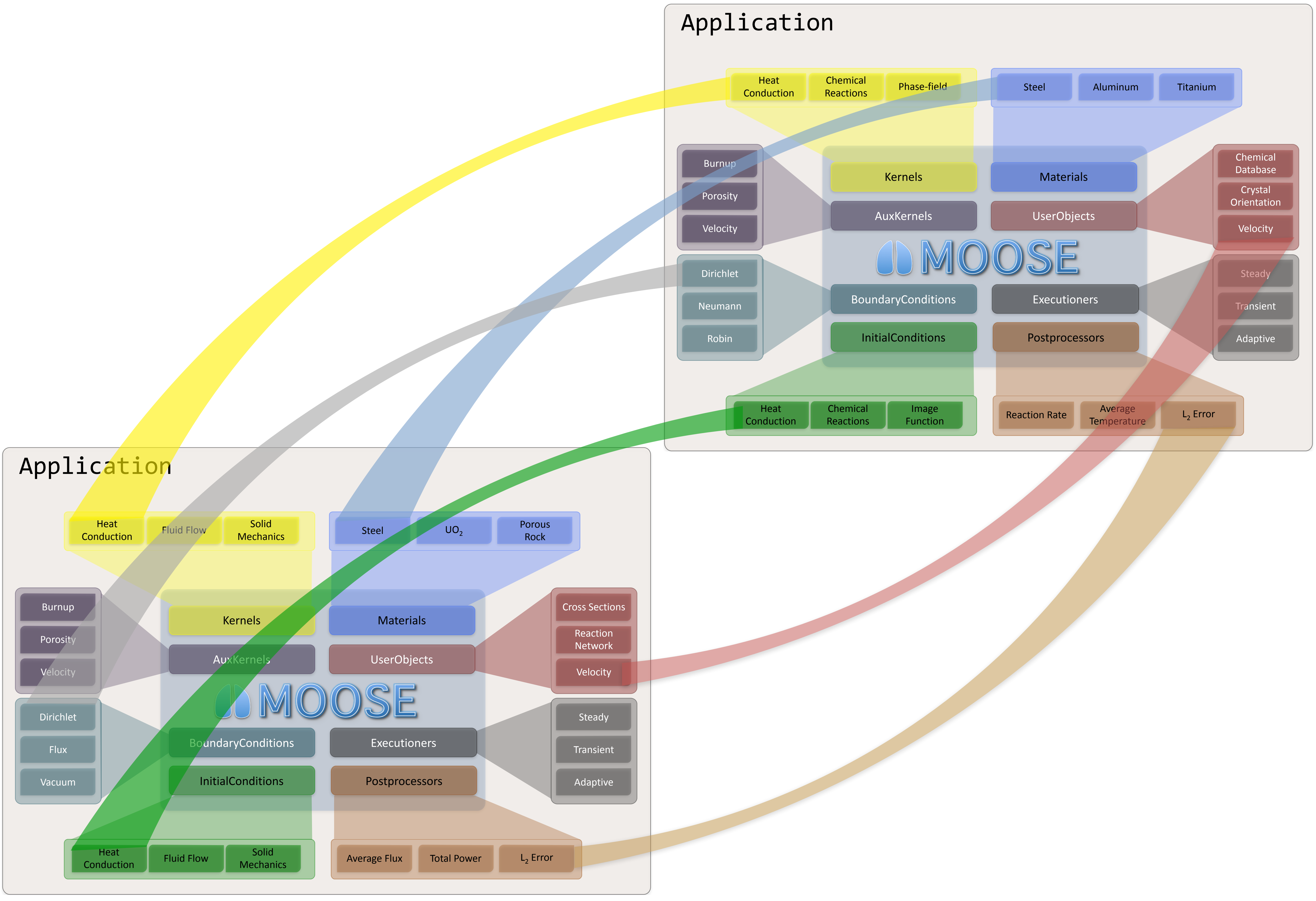
```
Real
DarcyConvection::computeQpResidual()
{
  RealVectorValue superficial_velocity = _porosity[_qp]*-(_permeability[_qp]/
_viscosity[_qp])*_pressure_gradient[_qp];

  return _heat_capacity[_qp] * superficial_velocity * _grad_u[_qp] * _test[_i][_qp];
}
```



Application
Heat Conduction
Fluid Flow
Solid Mechanics
Steel
UO_2
Porous Rock
Burnup
Porosity
Velocity
Kernels
Materials
AuxKernels
UserObjects
Cross Sections
Reaction Network
Velocity
MOOSE
Dirichlet
Flux
Vacuum
BoundaryConditions
Executioners
Steady
Transient
Adaptive
InitialConditions
Postprocessors
Heat Conduction
Fluid Flow
Solid Mechanics
Average Flux
Total Power
L_2 Error


Application
Heat Conduction
Chemical Reactions
Phase-field
Steel
Aluminum
Titanium
Burnup
Porosity
Velocity
Kernels
Materials
AuxKernels
UserObjects
Chemical Database
Crystal Orientation
Velocity
MOOSE
Dirichlet
Neumann
Robin
BoundaryConditions
Executioners
Steady
Transient
Adaptive
InitialConditions
Postprocessors
Heat Conduction
Chemical Reactions
Image Function
Reaction Rate
Average Temperature
L_2 Error


Application
Heat Conduction
Chemical Reactions
Phase-field
Steel
Aluminum
Titanium
Burnup
Porosity
Velocity
Kernels
Materials
AuxKernels
UserObjects
Chemical Database
Crystal Orientation
Velocity
MOOSE
Dirichlet
Neumann
Robin
BoundaryConditions
Executioners
InitialConditions
Postprocessors
Steady
Transient
Adaptive
Heat Conduction
Chemical Reactions
Image Function
Reaction Rate
Average Temperature
L_2 Error
Application
Heat Conduction
Fluid Flow
Solid Mechanics
Steel
UO_2
Porous Rock
Burnup
Porosity
Velocity
Kernels
Materials
AuxKernels
UserObjects
Cross Sections
Reaction Network
Velocity
MOOSE
Dirichlet
Flux
Vacuum
BoundaryConditions
Executioners
InitialConditions
Postprocessors
Steady
Transient
Adaptive
Heat Conduction
Fluid Flow
Solid Mechanics
Average Flux
Total Power
L_2 Error

DRYP

Don’t Repeat Your Physics!

Application Composition

- Enables reusable Applications
- Two methods within the MOOSE System:
- Static Registration:
    - One Application links the other in
    - Pros: “Make cascade”, seamless
    - Cons: Inflexible
- Dynamic Registration:
    - At runtime an Application can pull in objects from another application
    - Pros: **Extremely** flexible
    - Cons: Build system doesn’t see links

MOOSE-App Makefile

```
# Use the MOOSE submodule if it exists and MOOSE_DIR is not set
MOOSE_SUBMODULE    := $(CURDIR)/moose
ifneq ($(wildcard $(MOOSE_SUBMODULE)/framework/Makefile),)
  MOOSE_DIR        ?= $(MOOSE_SUBMODULE)
else
  MOOSE_DIR        ?= $(shell dirname `pwd`)/moose
endif

# framework
FRAMEWORK_DIR      := $(MOOSE_DIR)/framework
include $(FRAMEWORK_DIR)/build.mk
include $(FRAMEWORK_DIR)/moose.mk

################################## MODULES ####################################
ALL_MODULES := yes
include $(MOOSE_DIR)/modules/modules.mk
###############################################################################

# dep apps
APPLICATION_DIR    := $(CURDIR)
APPLICATION_NAME   := frog
BUILD_EXEC         := yes
DEP_APPS           := $(shell $(FRAMEWORK_DIR)/scripts/find_dep_apps.py $(APPLICATION_NAME))
include            $(FRAMEWORK_DIR)/app.mk
```

Static Registration

```
# Use the MOOSE submodule if it exists and MOOSE_DIR is not set
MOOSE_SUBMODULE    := $(CURDIR)/moose
ifneq ($(wildcard $(MOOSE_SUBMODULE)/framework/Makefile),)
  MOOSE_DIR        ?= $(MOOSE_SUBMODULE)
else
  MOOSE_DIR        ?= $(shell dirname `pwd`)/moose
endif

# framework
FRAMEWORK_DIR      := $(MOOSE_DIR)/framework
include $(FRAMEWORK_DIR)/build.mk
include $(FRAMEWORK_DIR)/moose.mk

################################## MODULES ####################################
ALL_MODULES := yes
include $(MOOSE_DIR)/modules/modules.mk
###############################################################################

# dep apps
BISON_DIR          ?= $(CURDIR)/bison
APPLICATION_DIR    := $(BISON_DIR)
APPLICATION_NAME   := bison
include            $(FRAMEWORK_DIR)/app.mk

APPLICATION_DIR    := $(CURDIR)
APPLICATION_NAME   := frog
BUILD_EXEC         := yes
DEP_APPS           := $(shell $(FRAMEWORK_DIR)/scripts/find_dep_apps.py $(APPLICATION_NAME))
include            $(FRAMEWORK_DIR)/app.mk
```

Static Registration (cont.)

```
void
FrogApp::registerObjects(Factory & factory)
{
  …
  BisonApp::registerObjects(factory);
  …
}
```

- All objects from the other application now available
- “make” will result in building all dependent applications
- Seamless for users

Dynamic Registration

- First: Add to the MOOSE_LIBRARY_PATH…

```
export MOOSE_LIBRARY_PATH=$MOOSE_LIBRARY_PATH:$HOME/projects/bison/lib
```

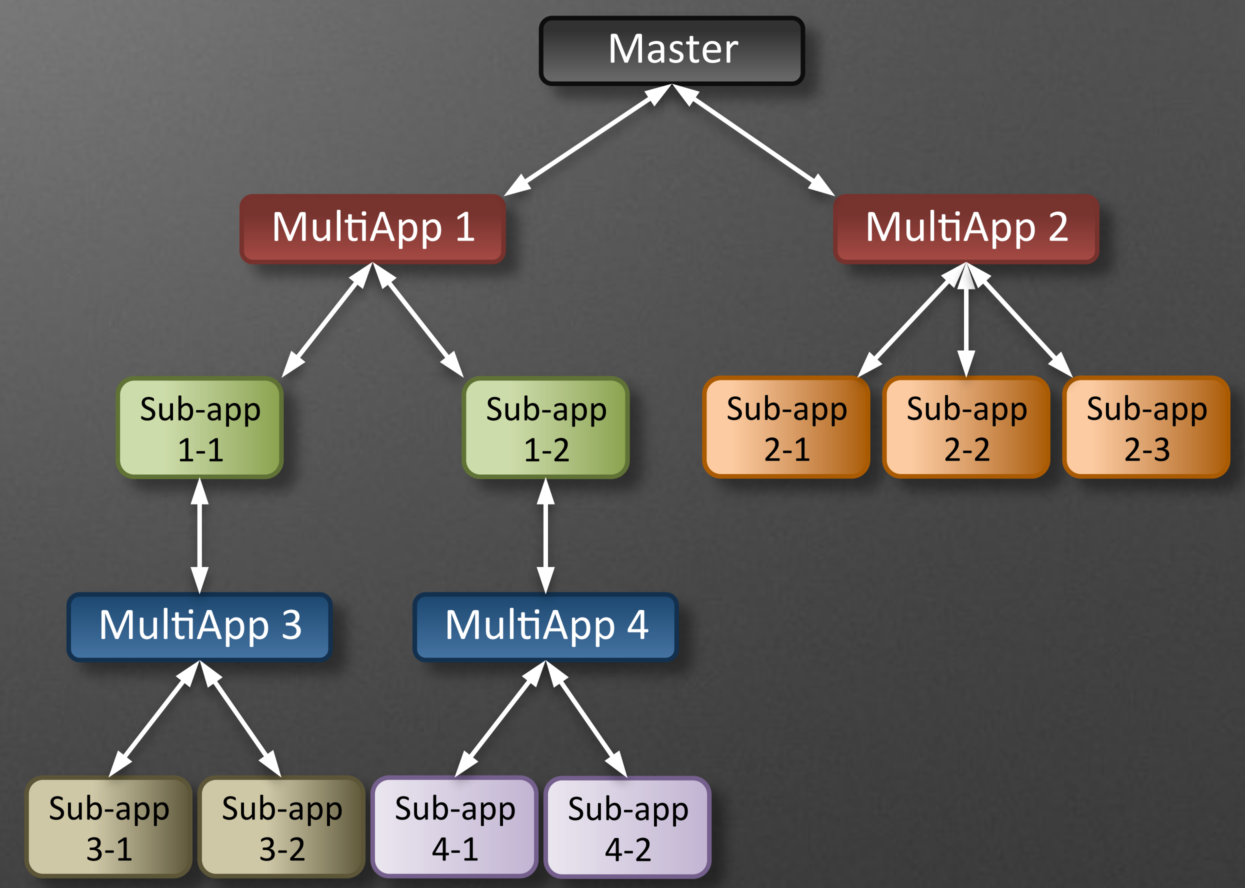
- Next: Add input file syntax to pull objects from the other Application

```
[Problem]
  register_objects_from = 'BisonApp'
  object_names = 'CladMat FuelMat FissionHeating'
[]
```

- Note: The build system will NOT build dependent Apps

MultiApps

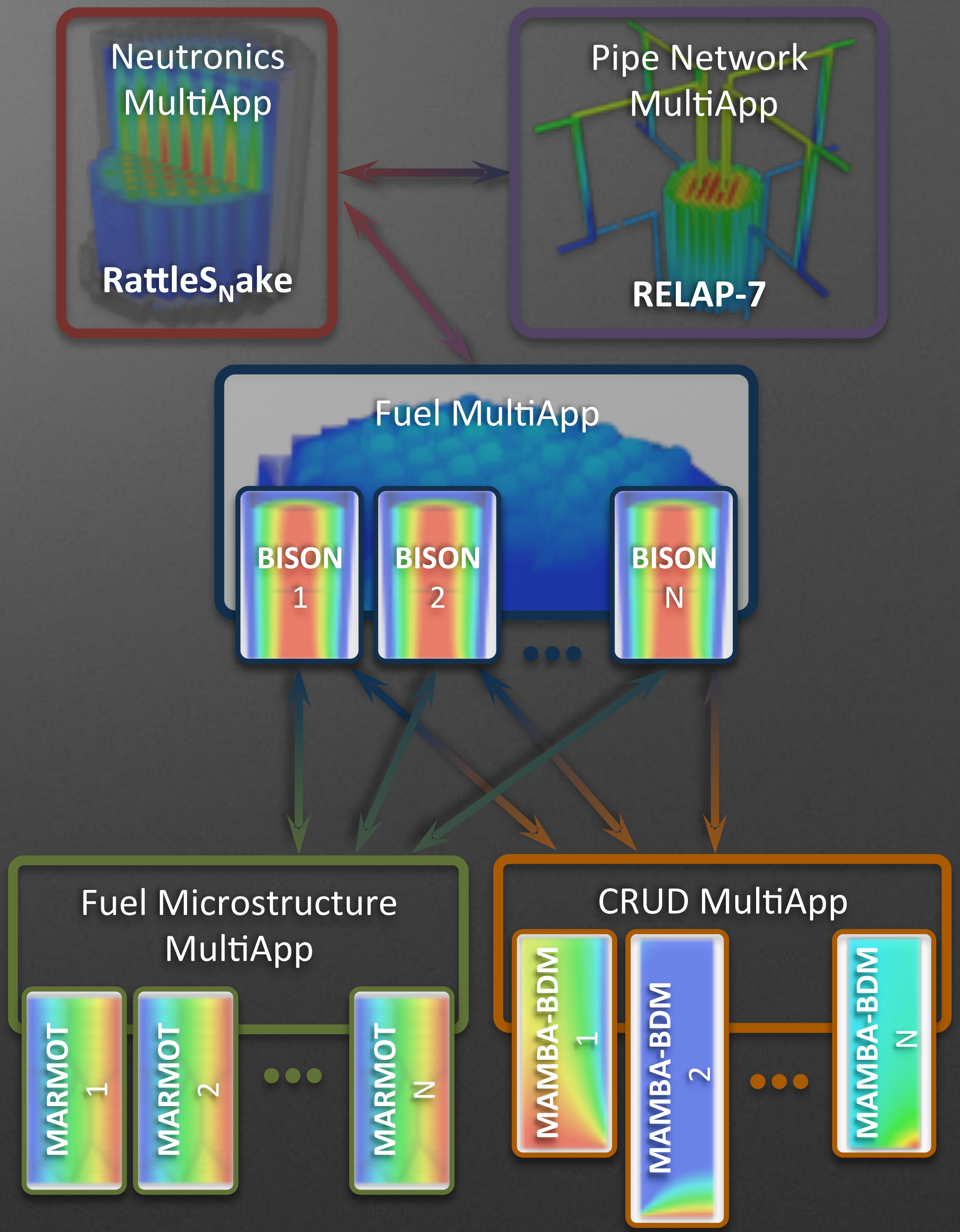
- Sometimes you want to reuse an entire application:
  - Multiscale (in space or time)
  - Loose coupling
  - Different meshes
- MultiApps allow you to run multiple MOOSE-based applications simultaneously in Parallel
- Transfers move data between the Main App and SubApps
- A “MooseApp” is an object just like any other in MOOSE
- Static or Dynamic registration allows immediate access to running another MOOSE-based Application as a SubApp



Gaston, Derek R., et al. "Physics-based multiscale coupling for full core nuclear reactor simulation." Annals of Nuclear Energy (2015).

Mammoth Setup

```
[MultiApps]
  [./bison]
    type = TransientMultiApp
    app_type = BisonApp
    positions_file = positions
    input_files = bison.i
    output_in_position = true
    catch_up = true
    max_catch_up_steps = 32
  [../]
  [./relap]
    type = TransientMultiApp
    app_type = Relap7App
    execute_on = timestep
    positions = '0 0 0'
    input_files = relap-7.i
    max_procs_per_app = 1
    max_failures = 1000
    sub_cycling = true
    steady_state_tol = 1e-6
    detect_steady_state = true
  [../]
[]
```



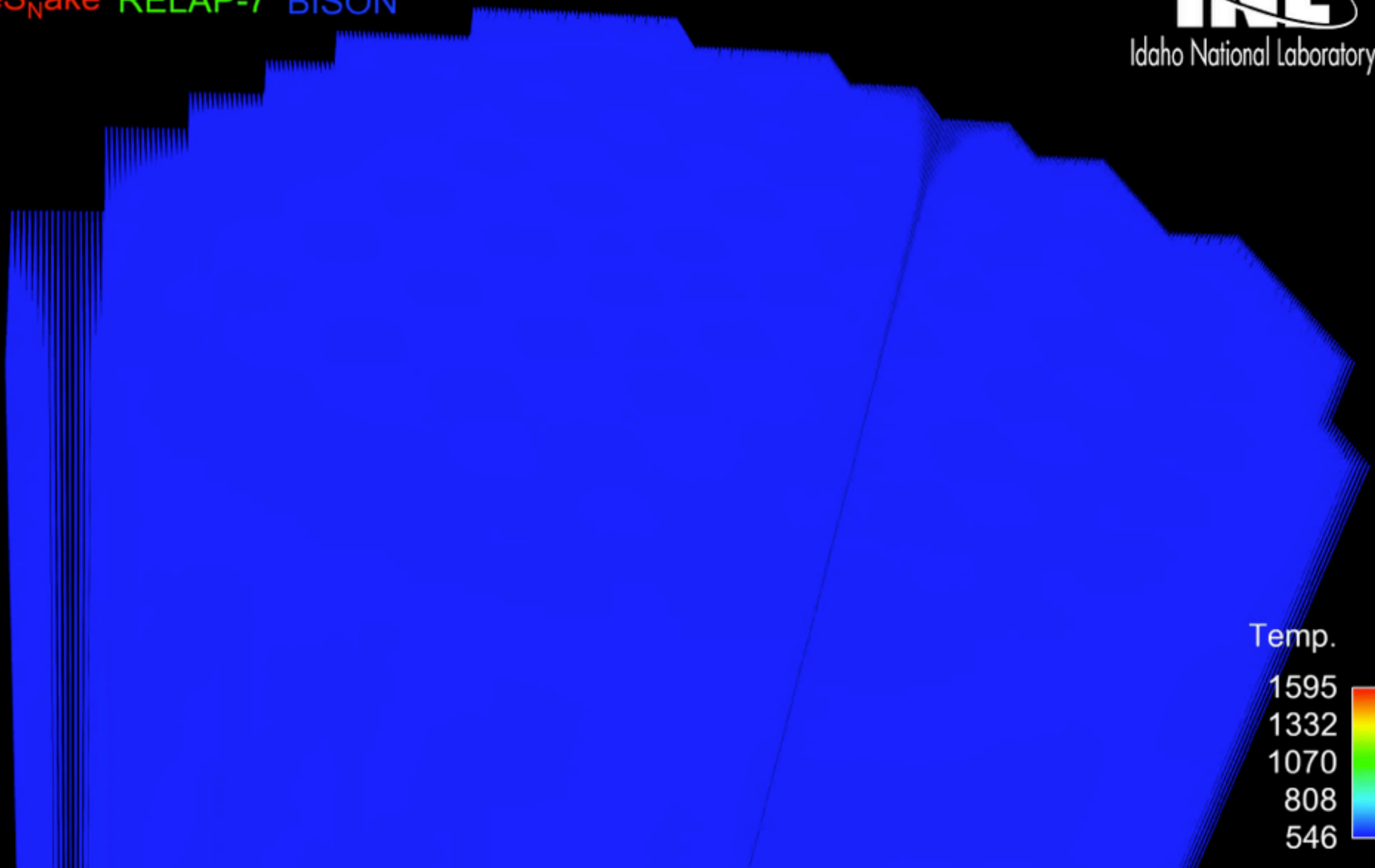
Gaston, Derek R., et al. "Physics-based multiscale coupling for full core nuclear reactor simulation." Annals of Nuclear Energy (2015).

MOOSE

RattleS$_N$ake RELAP-7 BISON

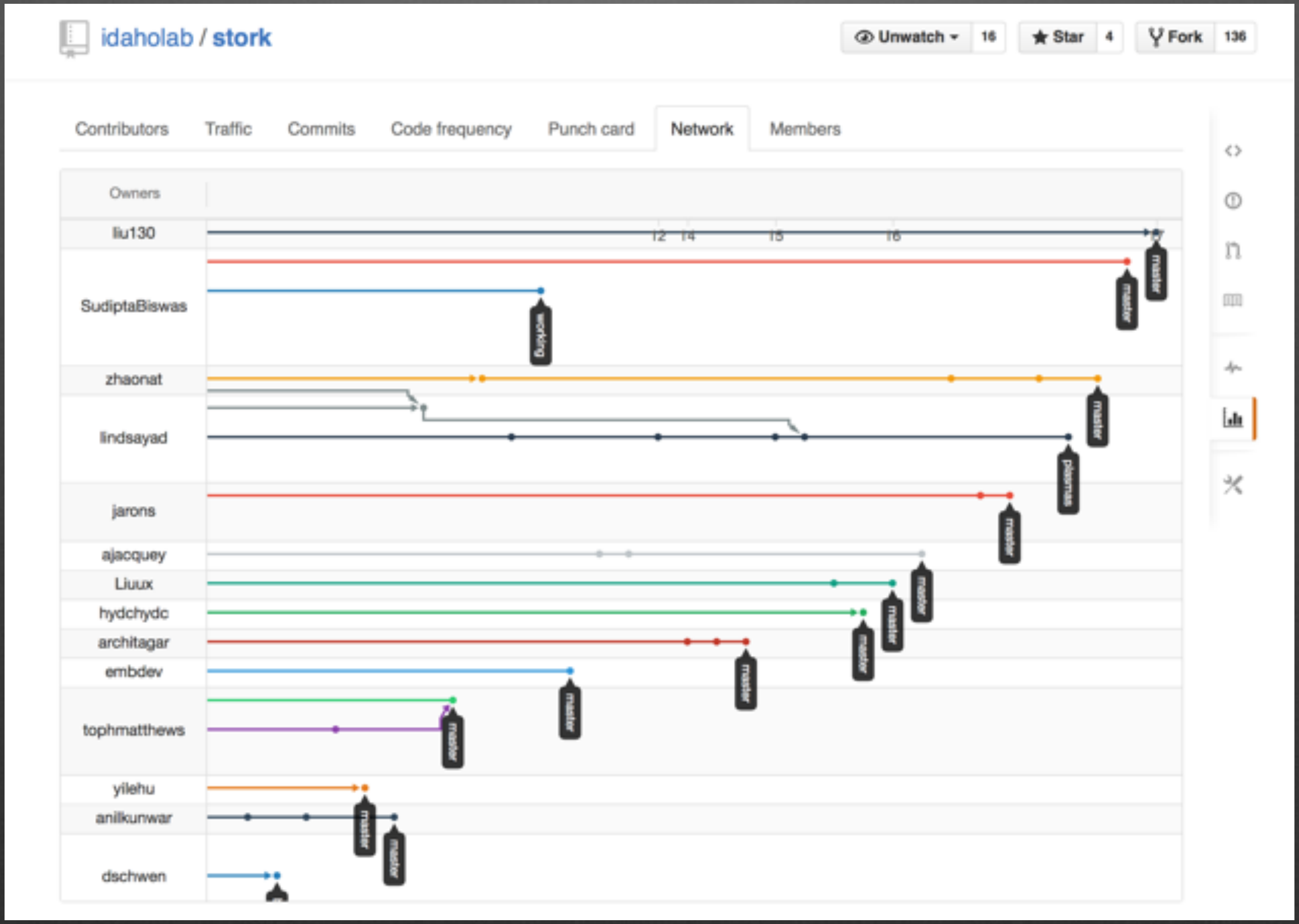
Time = 0.1 Days

Idaho National Laboratory



Finding Apps

- With all MOOSE-based Applications also being libraries: how do we keep track of them?



Summary

- Advances in computational science have lead to more code reuse over time:
  - MPI, PETSc, libMesh, MOOSE and MANY others…
- Data dependencies inherent to computational science can limit Application reusability
- Separating capabilities into modules with communication through interfaces can decouple scientific code
- Application composition can enable DRYP
  - **Static registration allows for seamless integration**
  - **Dynamic registration is more flexible**
- **By simplifying Application composition new areas of multiphysics can be explored using MultiApps**
- **Final Take Away: Software architecture can turn application developers into unwitting library developers!**