

# Software Concepts for Multiphysics Simulation

$$\frac{\partial \rho e}{\partial t} + \nabla \cdot (\rho e \mathbf{u}) = -\nabla \cdot \mathbf{J}_h - \nabla \cdot (\boldsymbol{\tau} \cdot \mathbf{u} + p \mathbf{u}) + \sum_{i=1}^{n_s} \rho \mathbf{g} \cdot \mathbf{v}$$

## Traditional approach: focus on **algorithms**.

- Leads to *fragile code with complicated logic* to manage complex dependencies.
  - Multiplicity of transport equations, constitutive laws, state relationships, etc.

## Alternative: focus on **data dependencies**.

- Algorithm is *deduced* (automatically) from data dependencies.
- Algorithm is not unique.
- Algorithm may easily be re-generated if models change (even *during* a simulation).

Enthalpy  
diffusive flux

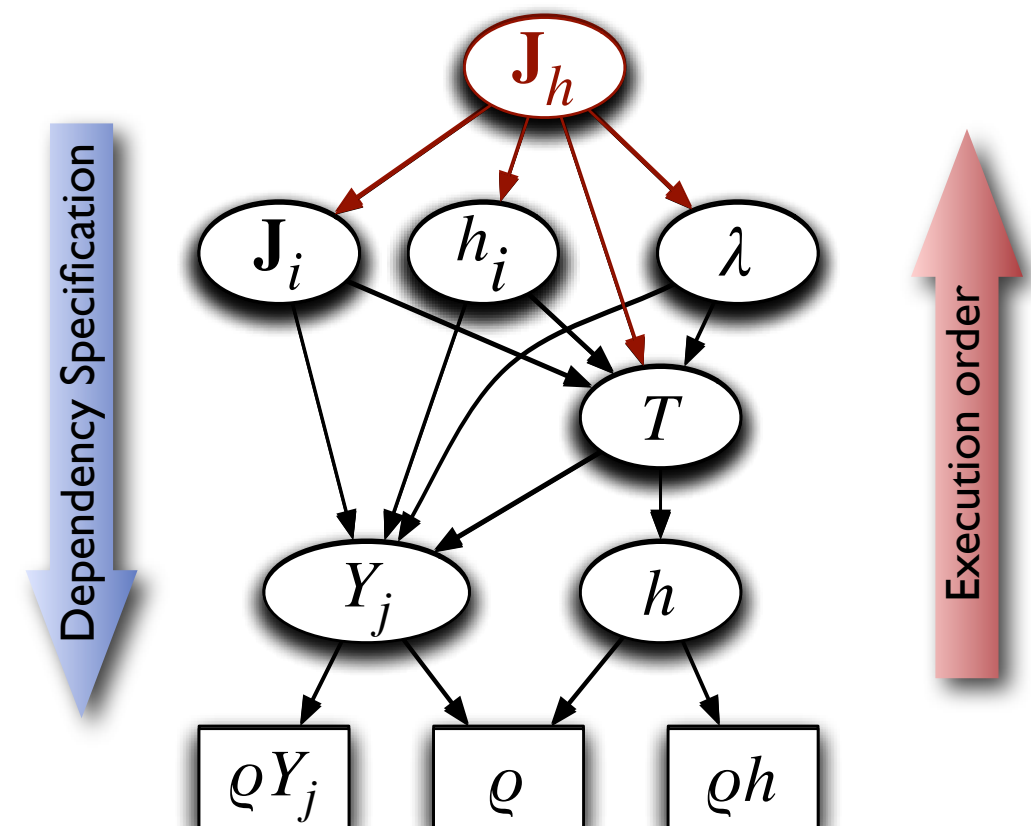
$$\mathbf{J}_h = -\lambda \nabla T + \sum_{i=1}^{n_s} h_i \mathbf{J}_i$$

$$\lambda \equiv \lambda_0(T, Y_j)$$

$$\mathbf{J}_i = -\sum_{j=1}^{n_s} D_{ij} \nabla Y_j - D_i^T \nabla T$$

$$D_{ij} = D_{ij}(Y_j, Y_j)$$

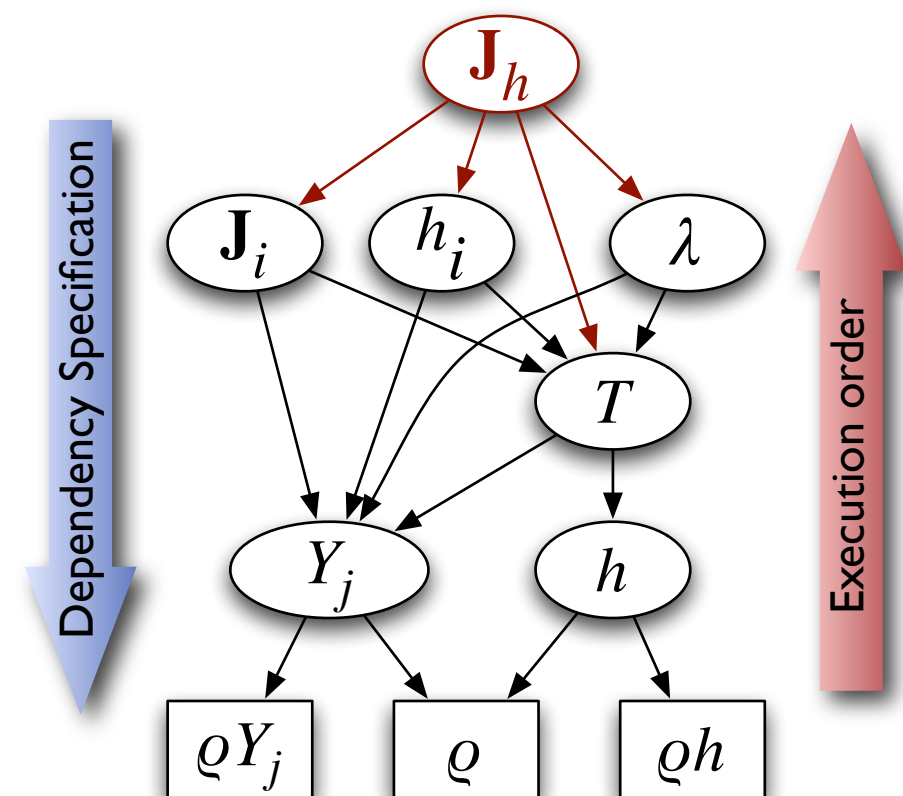
$$D_i^T = D_i^T(T, Y_j)$$



# “Expression” Concepts

- Each “Expression” is a software representation of a mathematical expression.
- An Expression evaluates the field it models.
  - Easily control model granularity.
  - General for any field type through use of C++ class templates & arbitrary spatial operators ( $\nabla$ ,  $\nabla \cdot$ , etc.)
- Each Expression indicates which expression(s) it *directly* depends on.
  - Allows us to *automatically* construct the entire dependency tree using graph theory.
    - ▶ Algorithm is deduced from data dependencies! Programmer need not deal with logic of algorithm orchestration.
  - Only a single layer of dependency is required of any expression.
    - ▶ Localized influence of changes in models.

$$\mathbf{J}_h = -\lambda \nabla T - \sum_{i=1}^{n_s} h_i \mathbf{J}_i$$



# Conceptual Code Structure

## 1. Create all required spatial operators.

- Depends on discretization scheme (FV, FD, staggered/colocated, structured/unstructured).

## 2. Register all expressions with a registry.

- Use operators, but are *independent* of the discretization scheme.
- Different constitutive model  $\Rightarrow$  different expression registered (no required changes to algorithms).

## 3. Define the root expression(s) and build the associated tree(s).

- Required expressions are *automatically constructed* using expressions in the registry.
- Required fields are *automatically* determined & registered.
- *Graph theory* determines a proper order of execution to satisfy data dependencies.

## 4. Execute the tree(s) within a time-integration loop

- Set BCs as appropriate.
- Advance solution variable(s) in time as appropriate.

$$\frac{\partial \rho h}{\partial t} = -\nabla \cdot \mathbf{J}_h$$

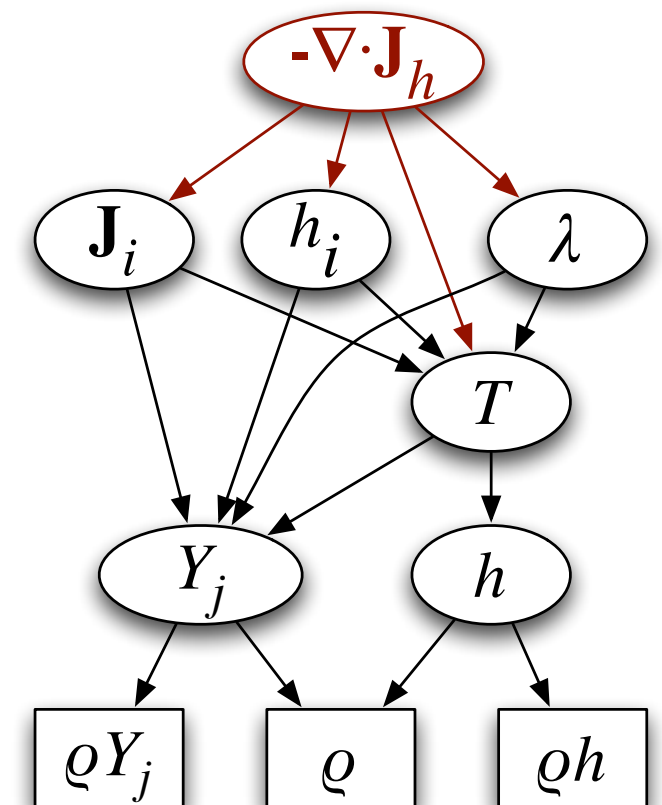
$$\mathbf{J}_h = -\lambda \nabla T - \sum_{i=1}^{n_s} h_i \mathbf{J}_i$$

$$\lambda = \lambda(T, Y_j)$$

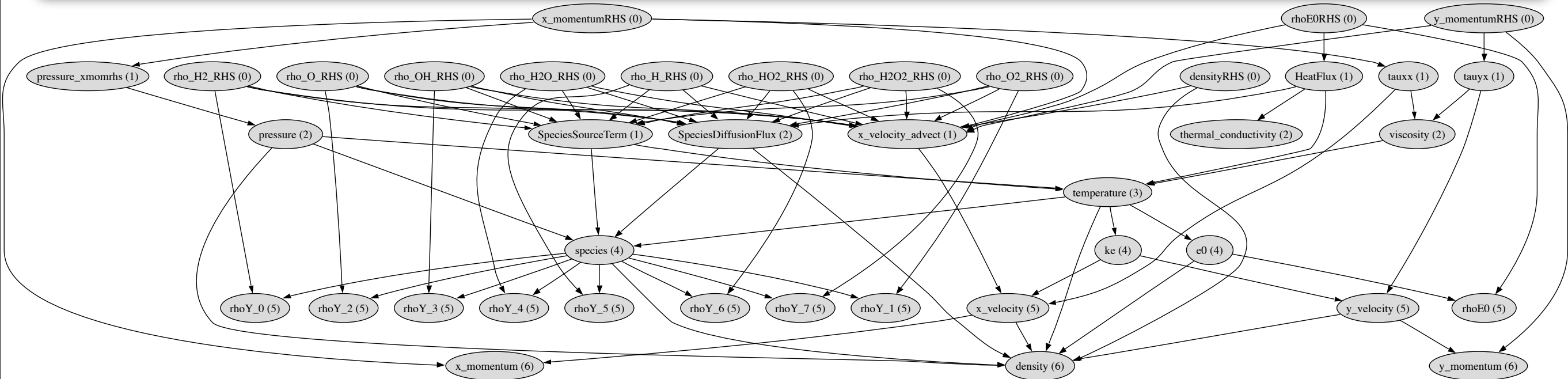
$$\mathbf{J}_i = -\sum_{j=1}^{n_s} D_{ij} \nabla Y_j - D_i^T \nabla T$$

$$D_{ij} = D_{ij}(T, Y_j)$$

$$D_i^T = D_i^T(T, Y_j)$$

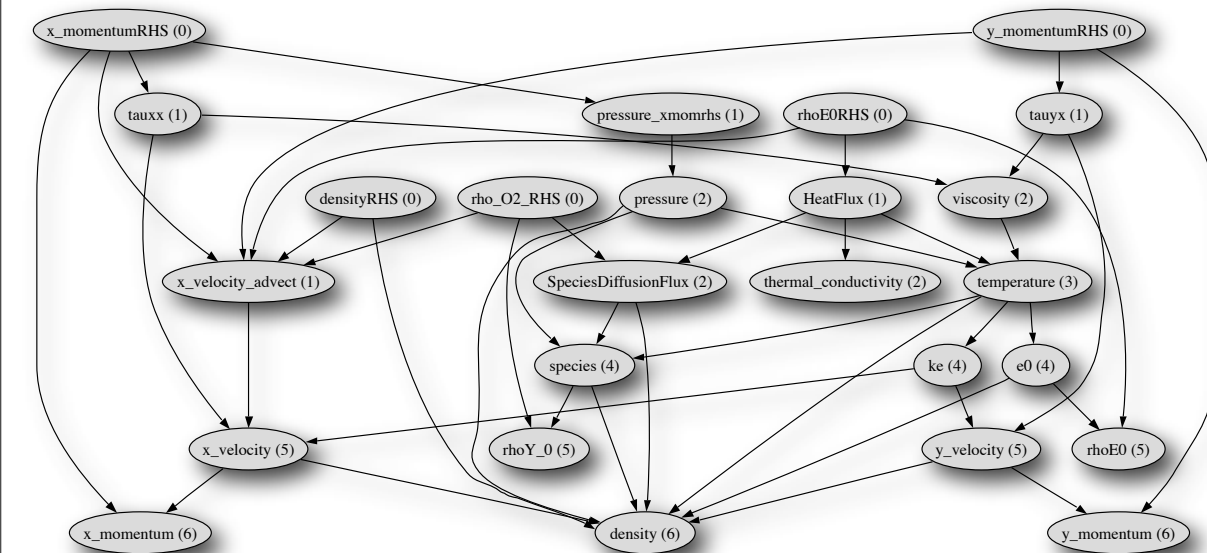


# Expression Graph



## Hydrogen-Air combustion

- 9 Species transport equations
- detailed Arrhenius kinetic mechanism
- detailed thermodynamics
- detailed transport
- energy equation
- momentum equations
- continuity equation
- (13 total PDEs)
- Full gasdynamics (compressible formulation)

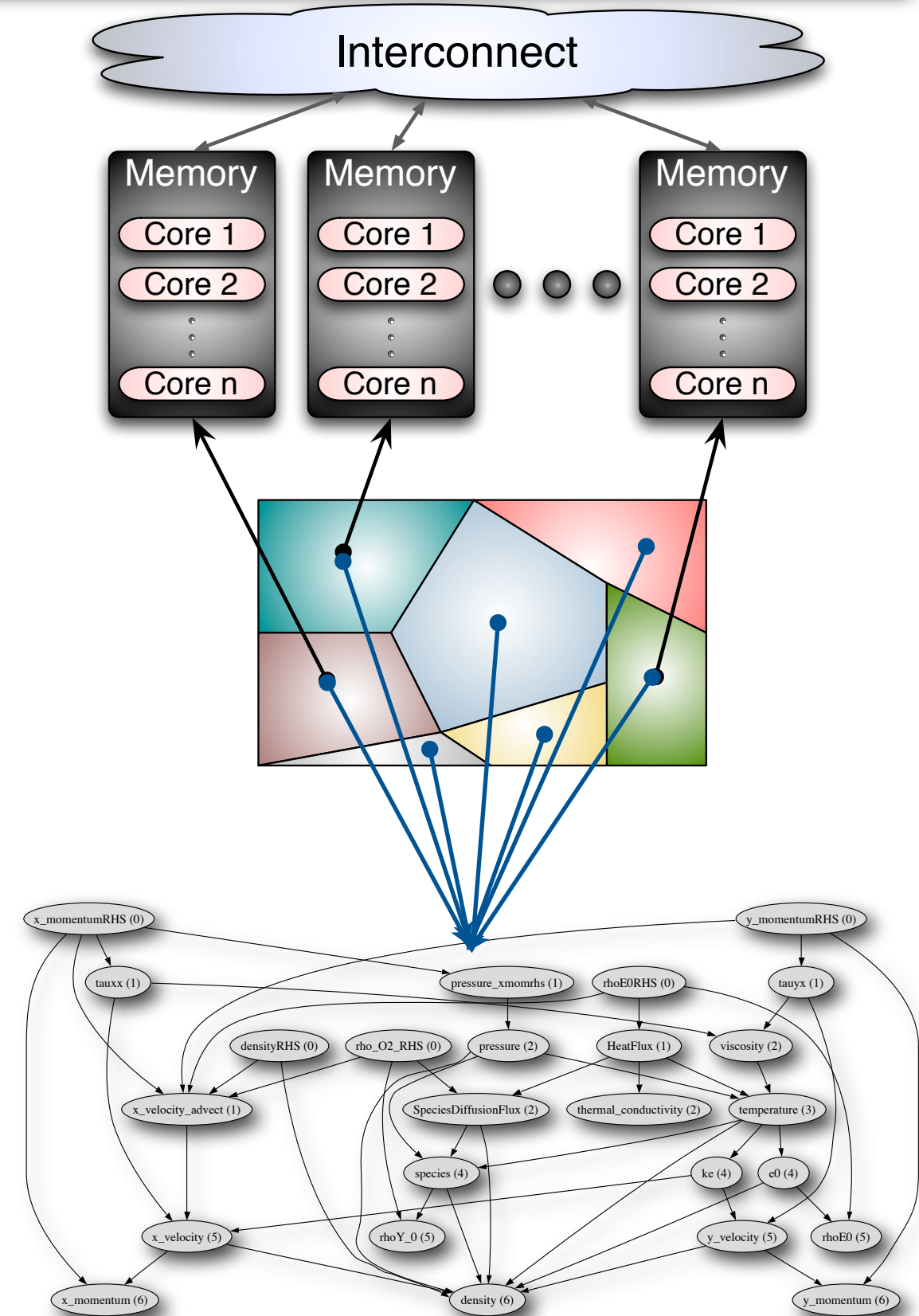


# Automatic Hybrid Parallelization

- Domain decomposition (MPI)
  - partition physical domain “patches” onto computational nodes
  - rely on computational frameworks
    - SIERRA, SAMRAI, UINTAH, etc.

- Algorithm decomposition
  - On a given patch, decompose algorithm onto cores on a node.
  - Need to develop analogous framework approach!
  - Orthogonal to domain decomposition

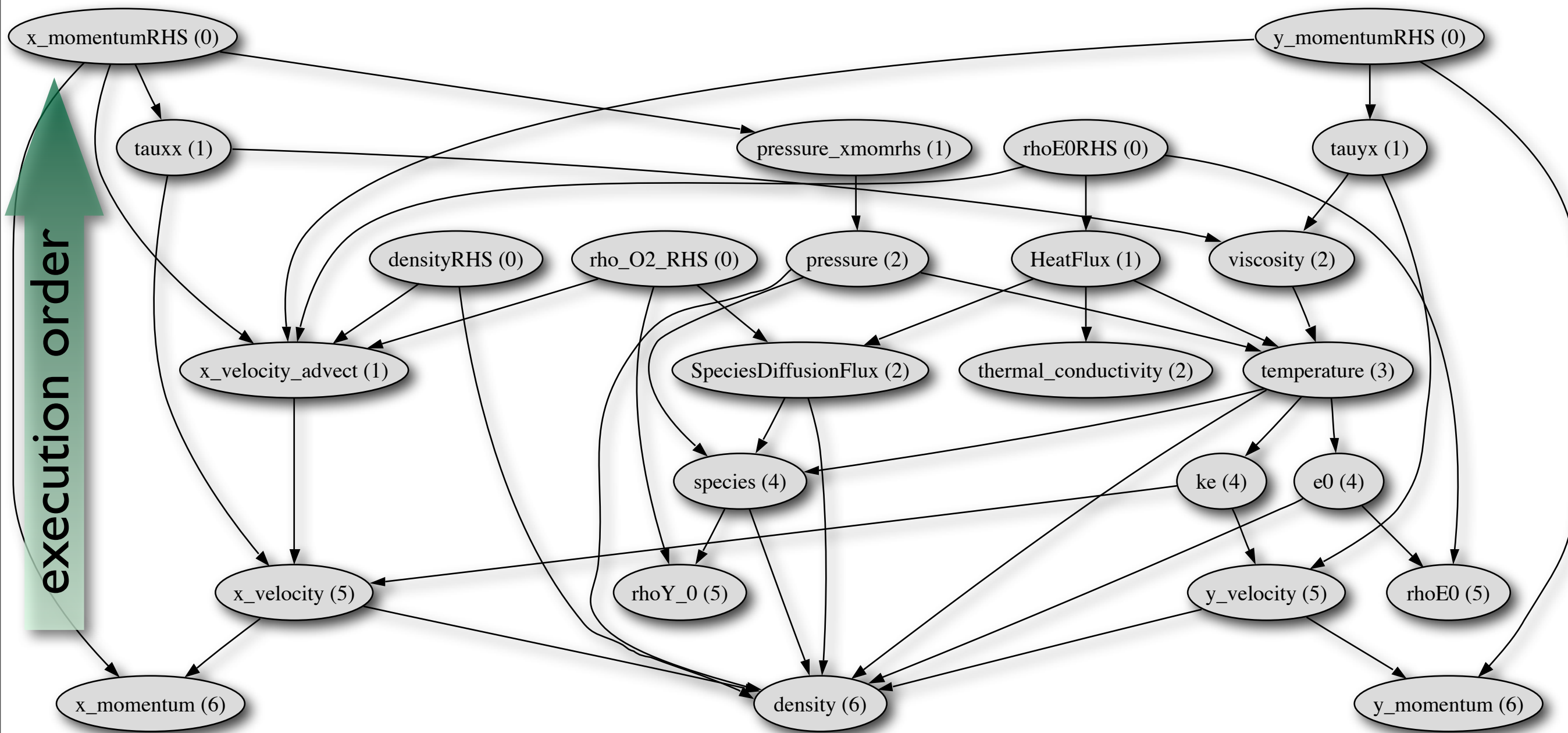
- Issues: granularity, memory pipelining (data locality, reuse), etc.





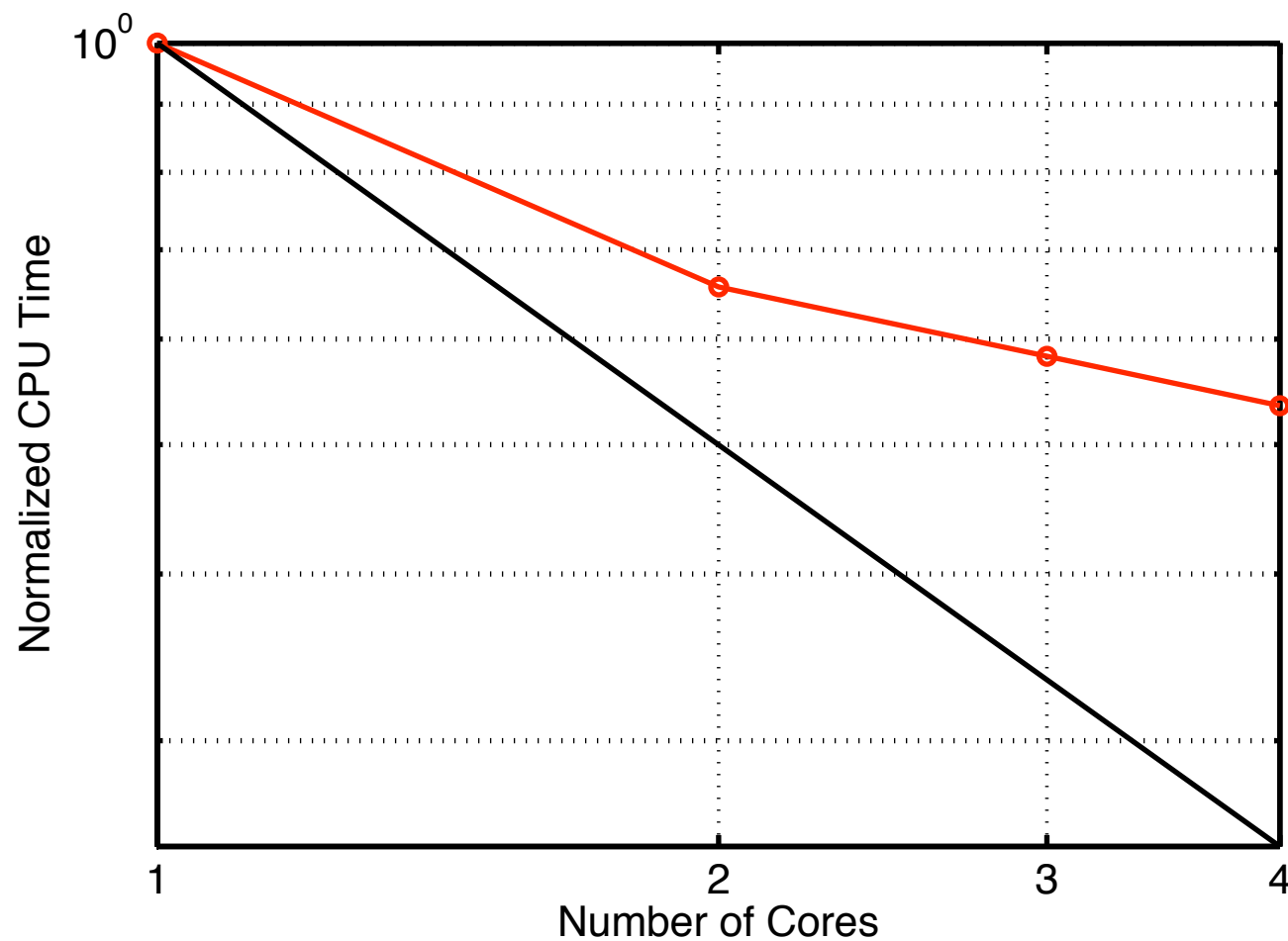
# Priority Queue Threading

Allows “backfilling”

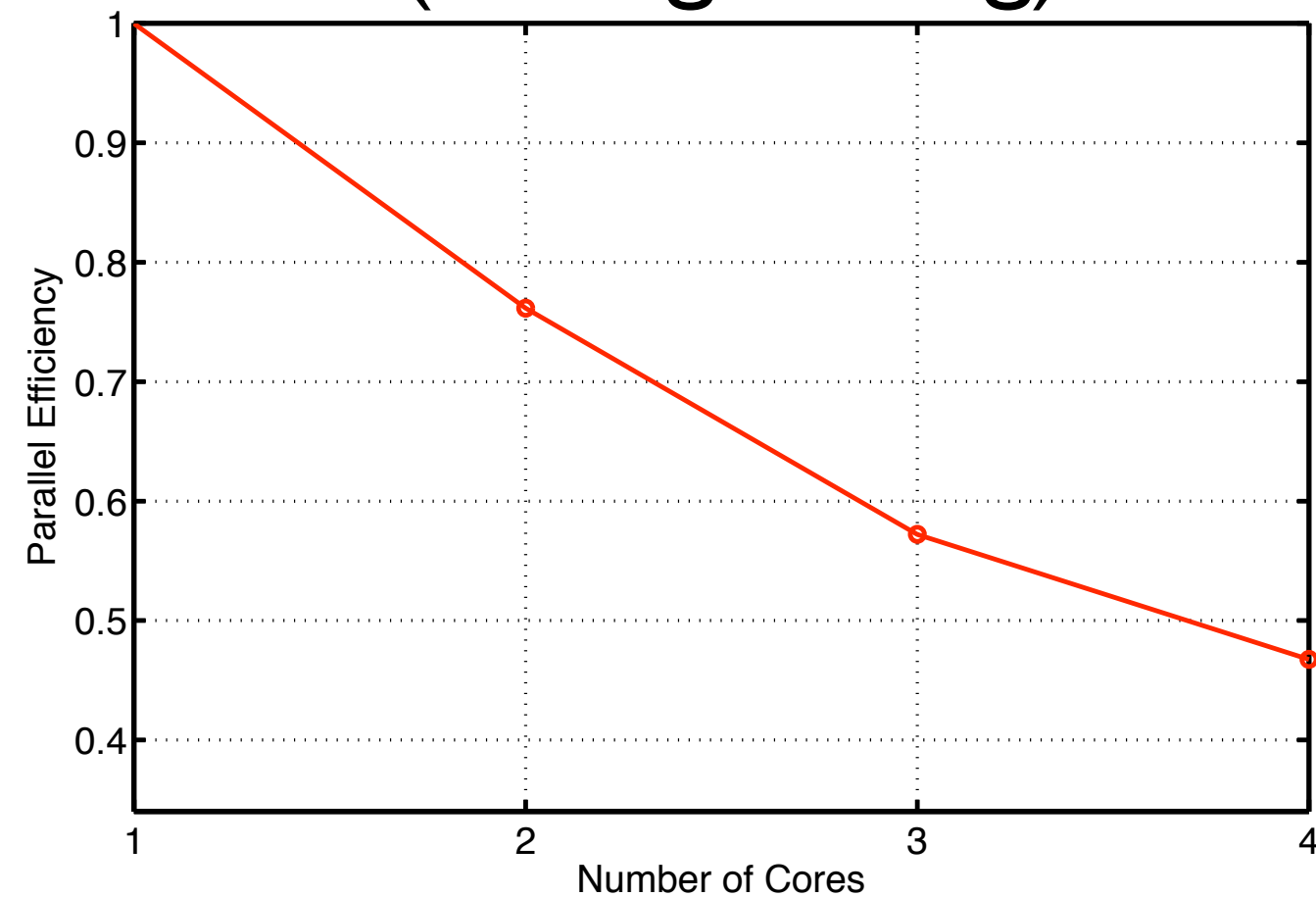


# Scalability - Preliminary Results

## Strong Scaling

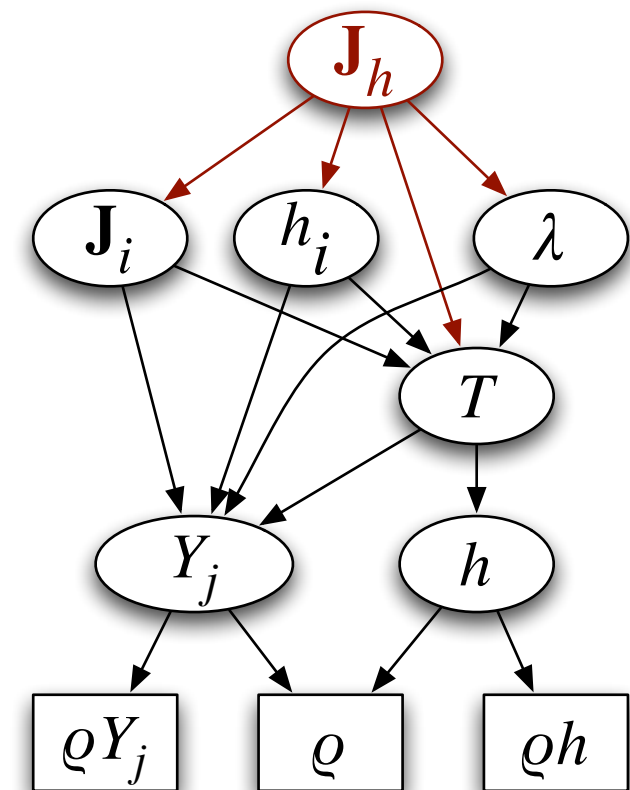


## Parallel Efficiency (strong scaling)



# Advantages of the “Expression” Approach

- Handles arbitrarily complex dependencies naturally
  - Ideal for multiscale, multiphysics simulations where a multitude of models exist for various regimes.
  - Allows programmer to easily determine and implement desired granularity in models.
- Facilitates dynamic model transitions.
  - In principle, we could *dynamically* select the model that is most appropriate for the physics occurring in the simulation.
- Allows thread-based parallelism based on independent portions of the algorithm.
  - Automatically detected and implemented!
  - Difficult to do in a multiphysics code using traditional programming models...
- Potential for automatic *analytic* calculation of *arbitrary* sensitivities - can be obtained dynamically at runtime



Represents a single term in the enthalpy transport equation. The full tree is much more complex!



# Key Expression Functionality

```
// 1. define dependencies on other expressions  
// 2. register field that this expression evaluates
```

```
void advertise_dependents( ExprDeps& exprDeps,  
                           FieldDeps& fieldDeps )
```

```
// obtain pointers to any required spatial operators
```

```
void bind_operators( const SpatialOps::OperatorDatabase& opDB )
```

```
// resolve pointers to fields used by this expression
```

```
void bind_fields( const FieldManagerList& fldMgrList )
```

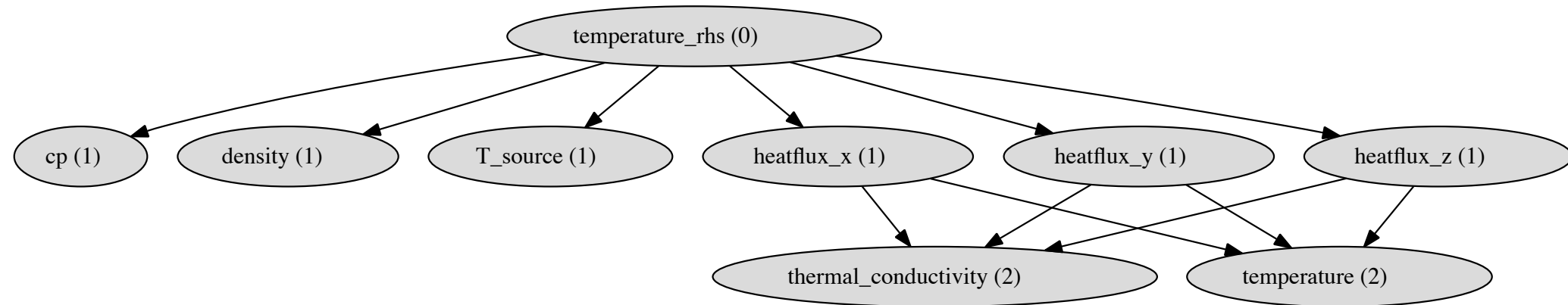
```
// evaluate this expression
```

```
void evaluate()
```

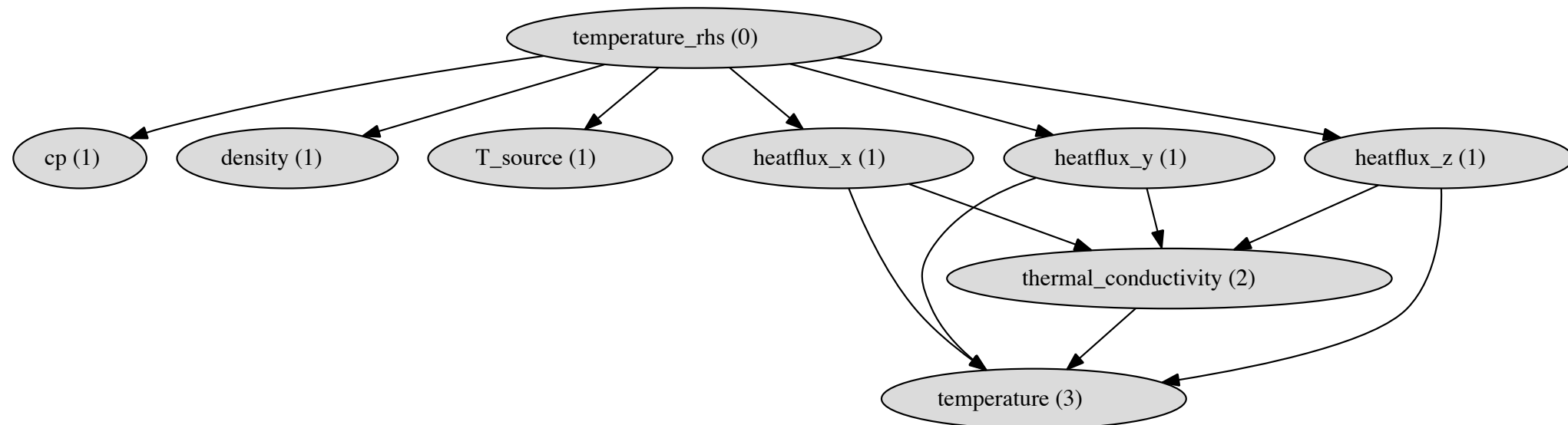
# Example: Heat Equation

$$\frac{\partial T}{\partial t} = -\frac{1}{\rho c_p} \nabla \cdot \mathbf{q} + \frac{1}{\rho c_p} s_T \quad \mathbf{q} = -\lambda \nabla T$$

$\lambda = \text{constant}$



$\lambda = \lambda(T)$



# Example: Heat Flux Calculation

$$\mathbf{q} = -\lambda \nabla T$$


- ▶  $\lambda, T$  stored at cell centers
- ▶  $\mathbf{q}$  at cell faces.

## Required operators:

- Gradient operator ( $T$ )
- Interpolant operator ( $\lambda$ )

```
template< typename GradT,      // Gradient operator type
          typename InterpT,    // Interpolant operator type
          typename PatchT >    // patch type
class HeatFlux
: public Expr::Expression< typename GradT::DestFieldType, PatchT >
{
    typedef typename GradT::SrcFieldType  ScalarT;
    typedef typename GradT::DestFieldType FluxT;
}
```

Different types of patches for different frameworks (interface to field management, etc)

The Expression base class is templated on the field type that the expression evaluates as well as the type of Patch that it lives on.

Operators define the types of fields they operate on and produce.

# HeatFlux class declaration

```
template< typename GradT,  
          typename InterpT,  
          typename PatchT >  
class HeatFlux  
    : public Expr::Expression< typename GradT::DestFieldType, PatchT >  
{  
public:  
  
    void evaluate();  
    void advertise_dependents( Expr::ExprDeps& exprDeps,  
                               Expr::FieldDeps& fieldDeps );  
    void bind_fields( const Expr::FieldManagerList& fml );  
    void bind_operators( const SpatialOps::OperatorDatabase& opDB );  
  
private:  
  
    typedef typename GradT::SrcFieldType  ScalarT;  
    typedef typename GradT::DestFieldType FluxT;  
  
    HeatFlux( const Expr::Tag& thermCondTag,  
              const Expr::Tag& tempTag,  
              const Expr::ExpressionID& id,  
              const Expr::ExpressionRegistry& reg,  
              PatchT& patch );  
  
    PatchT& patch_;  
    const Expr::Tag tcT_, tT_;  
    const GradT* gradOp_;  
    const InterpT* interpOp_;  
  
    const ScalarT* temp_;  
    const ScalarT* lambda_;  
};
```

# Field Dependencies

```
void
advertise_dependents( ExprDeps& exprDeps,
                     FieldDeps& fieldDeps )
{
    // specify dependencies on other expressions
    exprDeps.requires_expression( thermCondTag_ );
    exprDeps.requires_expression( temperatureTag_ );

    // we require a field to store the flux variable.
    fieldDeps.requires_field<PatchT,FluxT>( patch_, this->name() );
}
```

This facilitates construction of the dependency graph to generate the algorithm.

```
void
bind_fields( const Expr::FieldManagerList& fml )
{
    const FieldManager<PatchT,ScalarT>& scalarFM = fml.field_manager<PatchT,ScalarT>(patch_);
    const FieldManager<PatchT,FluxT>& fluxFM = fml.field_manager<PatchT,FluxT>(patch_);

    this->exprValue_ = &fluxFM.field_ref( this->name() );

    temp_ = &scalarFM.field_ref( tT_ );
    lambda_ = &scalarFM.field_ref( tcT_ );
}
```

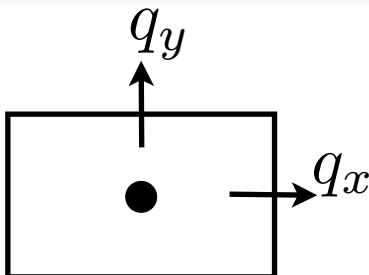


# Resolving Operators

An operator database is provided for an expression to resolve the operators it requires.

```
void  
bind_operators( const SpatialOps::OperatorDatabase& opDB )  
{  
    gradOp_    = opDB.retrieve_operator<GradT  >();  
    interpOp_  = opDB.retrieve_operator<InterpT>();  
}
```

# Evaluating the Expression

$$\mathbf{q} = -\lambda \nabla T$$


```
void
evaluate()
{
    FluxT& heatFlux = this->value();

    // grab a work field - prevents allocation of temporaries
    SpatFldPtr<FluxT> fluxTmp = SpatialFieldStore<FluxT>::self().get( heatFlux );

    // calculate: q = -lambda grad(T)
    gradOp_->apply_to_field ( *temp_,   heatFlux ); // calculate grad(T)
    interpOp_->apply_to_field( *lambda_, *fluxTmp ); // interpolate thermal cond
    heatFlux *= *fluxTmp;
    heatFlux *= -1.0;
}
```

Note: usage of expression templates  
would allow us to write:

```
heatFlux *= -fluxTmp
```