Trilinos in Coolfluid 3

Bart Janssens
Royal Military Academy (Belgium)

3 Mar 2015

Overview

Presentation objective

1. Present an overview of the Coolfluid workflow
2. Focus on FEM problems using an Embedded Domain Specific Language and the interaction with Trilinos
3. Discussion of other Trilinos aspects:
   - Configuration through Python
   - Mesh partitioning

Main message: Coolfluid 3 provides a high-level interface for generating and solving Trilinos linear systems for FEM problems

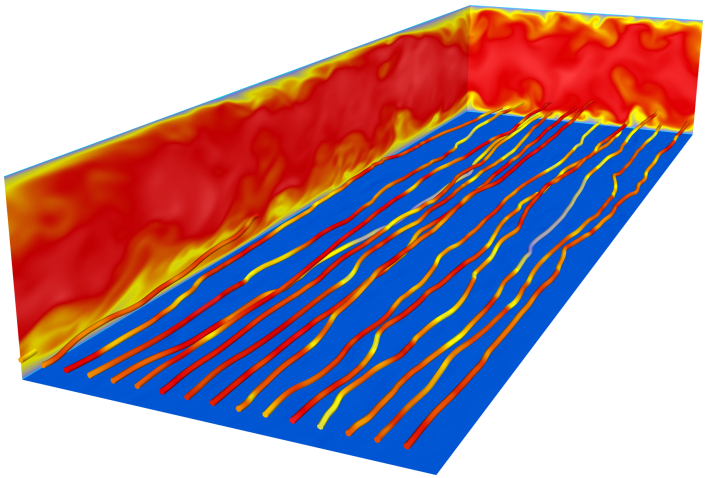
What is Coolfluid?

- Framework for numerical simulation
- Collaborative effort
- Mostly CFD
- Open source (GPL v3): http://coolfluid.github.io/

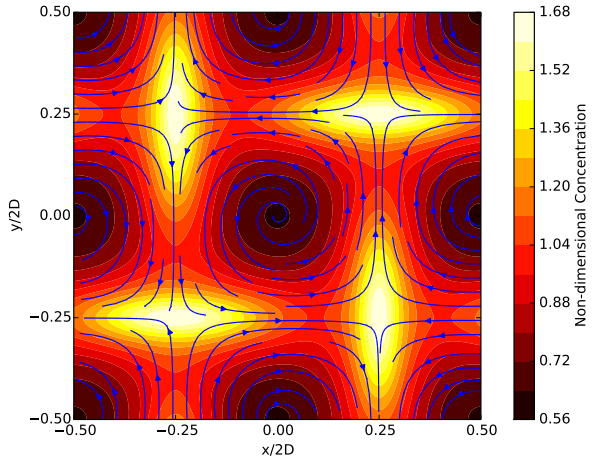
Vrije Universiteit Brussel

VON KARMAN INSTITUTE

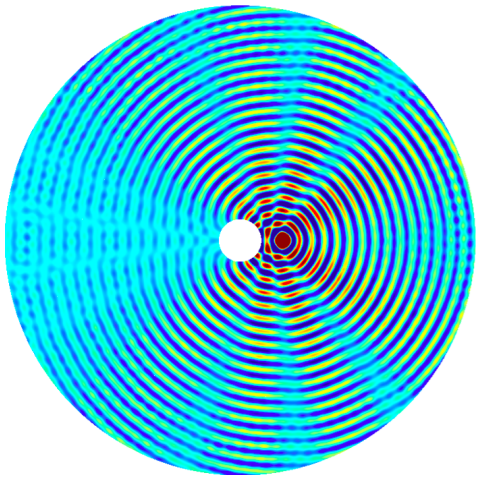
Application examples



Incompressible flow



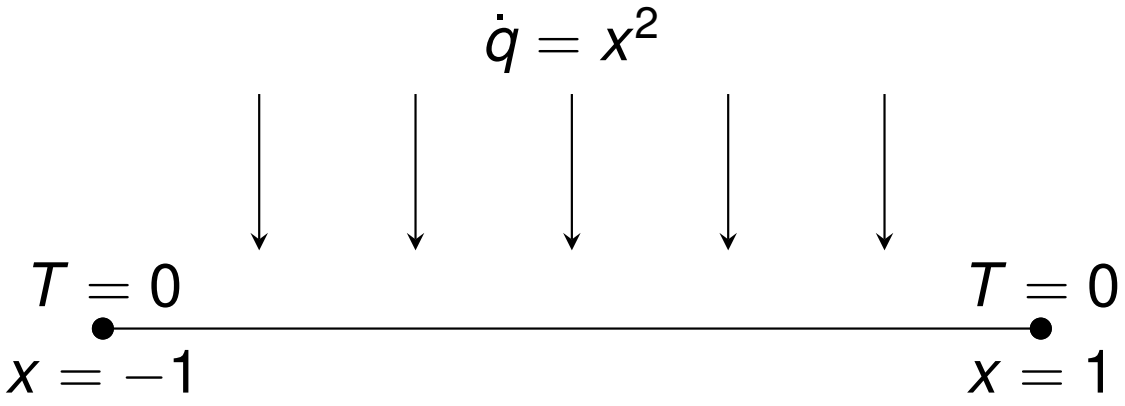
Particle dispersion



Acoustics

Overview

Heat conduction example



$$k\nabla^2 T + \dot{q} = 0$$

Simulation file = Python script

Set up imports and constants

```
1  import coolfluid as cf
2  import pylab as pl
3  import numpy as np
4
5  k = 1./12.
6  Tb = 0.
7
8  # Set up the simulation
9  model = cf.root.create_component('Model', 'cf3.solver.Model')
10 domain = model.create_domain()
11 physics = model.create_physics('cf3.UFEM.NavierStokesPhysics')
12 solver = model.create_solver('cf3.UFEM.Solver')
13 hc = solver.add_direct_solver('cf3.UFEM.HeatConductionSteady')
14 physics.thermal_conductivity = k
15 hc.options.heat_space_name = 'cf3.mesh.LagrangeP2'
```

Simulation file = Python script

Create Coolfluid Components

```
1  import coolfluid as cf
2  import pylab as pl
3  import numpy as np
4
5  k = 1./12.
6  Tb = 0.
7
8  # Set up the simulation
9  model = cf.root.create_component('Model', 'cf3.solver.Model')
10 domain = model.create_domain()
11 physics = model.create_physics('cf3.UFEM.NavierStokesPhysics')
12 solver = model.create_solver('cf3.UFEM.Solver')
13 hc = solver.add_direct_solver('cf3.UFEM.HeatConductionSteady')
14 physics.thermal_conductivity = k
15 hc.options.heat_space_name = 'cf3.mesh.LagrangeP2'
```

Simulation file = Python script

Set options

```
1  import coolfluid as cf
2  import pylab as pl
3  import numpy as np
4
5  k = 1./12.
6  Tb = 0.
7
8  # Set up the simulation
9  model = cf.root.create_component('Model', 'cf3.solver.Model')
10 domain = model.create_domain()
11 physics = model.create_physics('cf3.UFEM.NavierStokesPhysics')
12 solver = model.create_solver('cf3.UFEM.Solver')
13 hc = solver.add_direct_solver('cf3.UFEM.HeatConductionSteady')
14 physics.thermal_conductivity = k
15 hc.options.heat_space_name = 'cf3.mesh.LagrangeP2'
```

Simulation file = Python script

Set initial conditions

```
# Set initial conditions
ic_heat = solver.InitialConditions.create_initial_condition(
    builder_name = 'cf3.UFEM.InitialConditionFunction',
    field_tag = 'source_terms')
ic_heat.variable_name = 'Heat'
ic_heat.options.field_space_name = hc.options.heat_space_name
ic_heat.value = ['x^2']

# Generate a 1D line mesh
mesh = domain.create_component('mesh','cf3.mesh.Mesh')
mesh_generator = cf.root.create_component("mesh_generator","cf3.
    mesh.SimpleMeshGenerator")
mesh_generator.mesh = mesh.uri()
mesh_generator.nb_cells = [10]
mesh_generator.lengths = [2.]
mesh_generator.offsets = [-1.]
mesh_generator.execute()
```

Simulation file = Python script

Generate a mesh

```
1  # Set initial conditions
2  ic_heat = solver.InitialConditions.create_initial_condition(
       builder_name = 'cf3.UFEM.InitialConditionFunction',
       field_tag = 'source_terms')
3  ic_heat.variable_name = 'Heat'
4  ic_heat.options.field_space_name = hc.options.heat_space_name
5  ic_heat.value = ['x^2']
6
7  # Generate a 1D line mesh
8  mesh = domain.create_component('mesh','cf3.mesh.Mesh')
9  mesh_generator = cf.root.create_component("mesh_generator","cf3.
       mesh.SimpleMeshGenerator")
10 mesh_generator.mesh = mesh.uri()
11 mesh_generator.nb_cells = [10]
12 mesh_generator.lengths = [2.]
13 mesh_generator.offsets = [-1.]
14 mesh_generator.execute()
```

Simulation file = Python script

```
# Set the region for the simulation
hc.regions = ic_heat.regions = [mesh.topology.uri()]

# Boundary conditions
hc.BoundaryConditions.add_constant_bc(region_name = 'xneg',
    variable_name = 'Temperature').value = Tb
hc.BoundaryConditions.add_constant_bc(region_name = 'xpos',
    variable_name = 'Temperature').value = Tb

# Parameters for linear system
hc.LSS.SolutionStrategy.Parameters.linear_solver_type = 'Amesos'

# Run the simulation
model.simulate()

# Plot the result
x_p2 = mesh.geometry.coordinates
T_p2 = mesh.geometry.heat_conduction_solution
pl.plot(x_p2, T_p2, 'kx', mfc='none', label='Simulation')
```

Simulation file = Python script

```
# Set the region for the simulation
hc.regions = ic_heat.regions = [mesh.topology.uri()]

# Boundary conditions
hc.BoundaryConditions.add_constant_bc(region_name = 'xneg',
    variable_name = 'Temperature').value = Tb
hc.BoundaryConditions.add_constant_bc(region_name = 'xpos',
    variable_name = 'Temperature').value = Tb

# Parameters for linear system
hc.LSS.SolutionStrategy.Parameters.linear_solver_type = 'Amesos'

# Run the simulation
model.simulate()

# Plot the result
x_p2 = mesh.geometry.coordinates
T_p2 = mesh.geometry.heat_conduction_solution
pl.plot(x_p2, T_p2, 'kx', mfc='none', label='Simulation')
```

Simulation file = Python script

```
# Set the region for the simulation
hc.regions = ic_heat.regions = [mesh.topology.uri()]

# Boundary conditions
hc.BoundaryConditions.add_constant_bc(region_name = 'xneg',
    variable_name = 'Temperature').value = Tb
hc.BoundaryConditions.add_constant_bc(region_name = 'xpos',
    variable_name = 'Temperature').value = Tb

# Parameters for linear system
hc.LSS.SolutionStrategy.Parameters.linear_solver_type = 'Amesos'

# Run the simulation
model.simulate()

# Plot the result
x_p2 = mesh.geometry.coordinates
T_p2 = mesh.geometry.heat_conduction_solution
pl.plot(x_p2, T_p2, 'kx', mfc='none', label='Simulation')
```

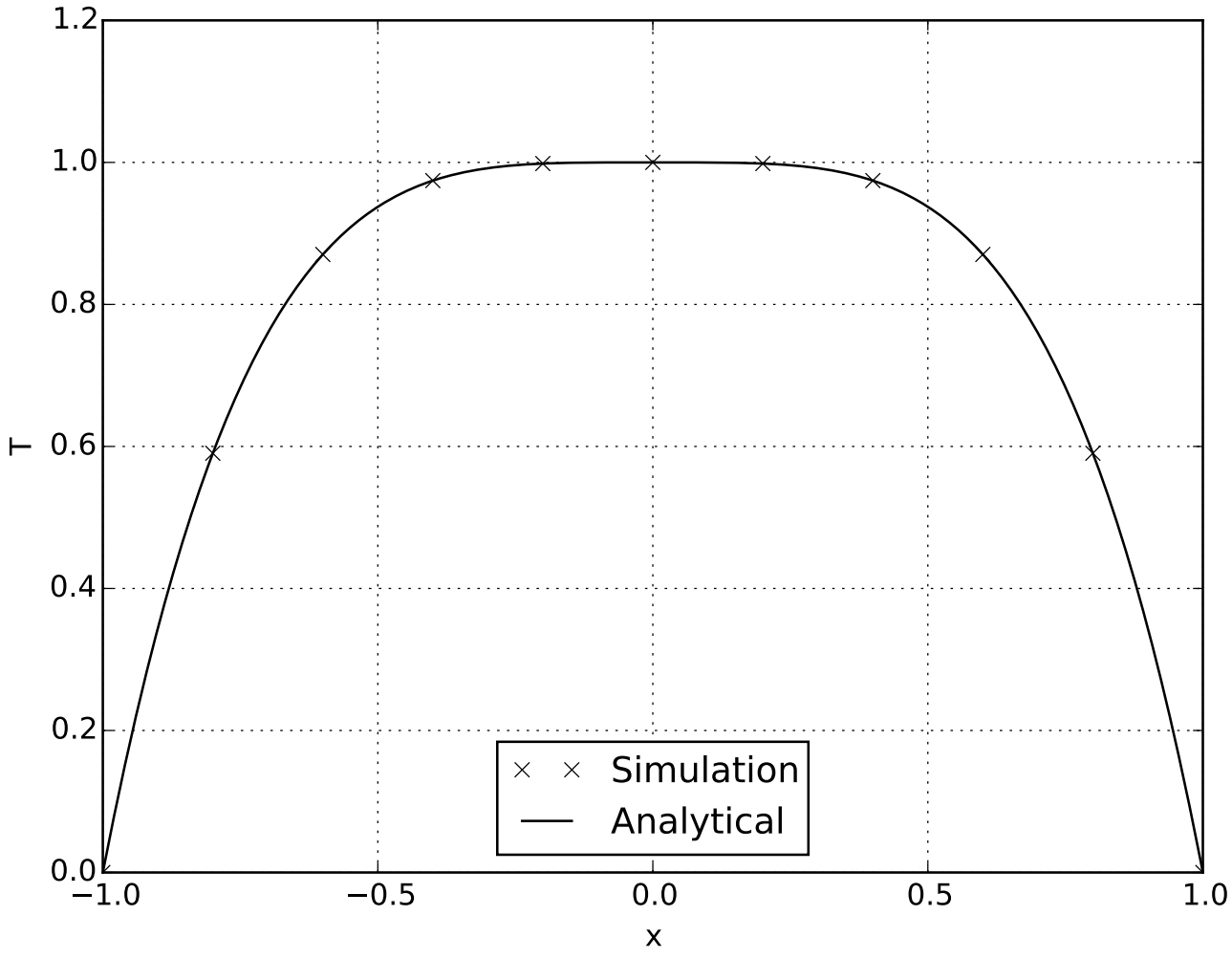
Simulation file = Python script

```
# Set the region for the simulation
hc.regions = ic_heat.regions = [mesh.topology.uri()]

# Boundary conditions
hc.BoundaryConditions.add_constant_bc(region_name = 'xneg',
    variable_name = 'Temperature').value = Tb
hc.BoundaryConditions.add_constant_bc(region_name = 'xpos',
    variable_name = 'Temperature').value = Tb

# Parameters for linear system
hc.LSS.SolutionStrategy.Parameters.linear_solver_type = 'Amesos'

# Run the simulation
model.simulate()

# Plot the result
x_p2 = mesh.geometry.coordinates
T_p2 = mesh.geometry.heat_conduction_solution
pl.plot(x_p2, T_p2, 'kx', mfc='none', label='Simulation')
```

Simulation file = Python script

```
# Set the region for the simulation
hc.regions = ic_heat.regions = [mesh.topology.uri()]

# Boundary conditions
hc.BoundaryConditions.add_constant_bc(region_name = 'xneg',
    variable_name = 'Temperature').value = Tb
hc.BoundaryConditions.add_constant_bc(region_name = 'xpos',
    variable_name = 'Temperature').value = Tb

# Parameters for linear system
hc.LSS.SolutionStrategy.Parameters.linear_solver_type = 'Amesos'

# Run the simulation
model.simulate()

# Plot the result
x_p2 = mesh.geometry.coordinates
T_p2 = mesh.geometry.heat_conduction_solution
pl.plot(x_p2, T_p2, 'kx', mfc='none', label='Simulation')
```

Result



The component tree

```
Model (cf3.solver.Model)
  tools (cf3.common.Group)
  Domain (cf3.mesh.Domain)
  NavierStokesPhysics (cf3.UFEM.NavierStokesPhysics)
    VariableManager (cf3.math.VariableManager)
      heat_conduction_solution (cf3.math.VariablesDescriptor)
      source_terms (cf3.math.VariablesDescriptor)
  Solver (cf3.UFEM.Solver)
    FieldManager (cf3.mesh.FieldManager)
    InitialConditions (cf3.UFEM.InitialConditions)
      heat_conduction_solution (cf3.UFEM.
          InitialConditionConstant)
      source_terms (cf3.UFEM.InitialConditionFunction)
    HeatConductionSteady (cf3.UFEM.HeatConductionSteady)
      ZeroLSS (cf3.math.LSS.ZeroLSS)
      Assembly (cf3.solver.ProtoAction)
      BoundaryConditions (cf3.UFEM.BoundaryConditions)
      SolveLSS (cf3.math.LSS.SolveLSS)
      Update (cf3.solver.ProtoAction)
```

Overview

Building a component

```
1  class UFEM_API HeatConductionSteady : public LSSAction
2  {
3  public:
4    // Component basics:
5    HeatConductionSteady ( const std::string& name );
6    static std::string type_name() {
7      return "HeatConductionSteady"; }
8  private:
9    // Set default initial conditions
10   virtual void on_initial_conditions_set(InitialConditions&
         initial_conditions);
11   // Triggered on option changes
12   void trigger();
13   // System assembly
14   Handle<solver::actions::Proto::ProtoAction> m_assembly;
15   // Solution update
16   Handle<solver::actions::Proto::ProtoAction> m_update;
17   PhysicsConstant k; // thermal conductivity
18
19 };
```

Building a component

```
1  class UFEM_API HeatConductionSteady : public LSSAction
2  {
3  public:
4    // Component basics:
5    HeatConductionSteady ( const std::string& name );
6    static std::string type_name() {
7      return "HeatConductionSteady"; }
8  private:
9    // Set default initial conditions
10   virtual void on_initial_conditions_set(InitialConditions&
         initial_conditions);
11   // Triggered on option changes
12   void trigger();
13   // System assembly
14   Handle<solver::actions::Proto::ProtoAction> m_assembly;
15   // Solution update
16   Handle<solver::actions::Proto::ProtoAction> m_update;
17   PhysicsConstant k; // thermal conductivity
18
19 };
```

Linear system assembly

Linear heat conduction differential equation:

$$k\nabla^2 T + \dot{q} = 0$$

Finite element formulation:

$$\sum_i \underbrace{\int_{\Omega_i} k\nabla N_T^{\mathrm{T}} \nabla N_T}_{A_e} \mathbf{T_e} = \sum_i \underbrace{\int_{\Omega_i} N_T^{\mathrm{T}} N_q \mathbf{q_e}}_{b_e} \, \mathrm{d}\Omega_i$$

Solve for difference:

$$A_e \left(\mathbf{T_e}^{n+1} - \mathbf{T_e}^{n} \right) = -A_e \mathbf{T_e}^{n} + b_e$$

Linear system assembly

```
1  m_assembly = create_component<ProtoAction>("Assembly");
2  m_assembly->set_expression(elements_expression
3  (
4    boost::mpl::vector<LagrangeP1::Line1D, LagrangeP2::Line1D>(),
5    group
6    (
7      _A = _0,
8      element_quadrature(_A(T) += k*transpose(nabla(T))*nabla(T)),
9      system_matrix += _A,
10     system_rhs += -_A * _x + integral<2>(transpose(N(T))*N(q)*
           jacobian_determinant) * nodal_values(q)
11   )
12 ));
```

Linear system assembly

```
1  m_assembly = create_component<ProtoAction>("Assembly");
2  m_assembly->set_expression(elements_expression
3  (
4    boost::mpl::vector<LagrangeP1::Line1D, LagrangeP2::Line1D>(),
5    group
6    (
7      _A = _0,
8      element_quadrature(_A(T) += k*transpose(nabla(T))*nabla(T)),
9      system_matrix += _A,
10     system_rhs += -_A * _x + integral<2>(transpose(N(T))*N(q)*
           jacobian_determinant) * nodal_values(q)
11   )
12 ));
```

Linear system assembly

$$A_e = \int_{\Omega_i} k \nabla N_T^{\mathrm{T}} \nabla N_T$$

```
1  m_assembly = create_component<ProtoAction>("Assembly");
2  m_assembly->set_expression(elements_expression
3  (
4    boost::mpl::vector<LagrangeP1::Line1D, LagrangeP2::Line1D>(),
5    group
6    (
7      _A = _0,
8      element_quadrature(_A(T) += k*transpose(nabla(T))*nabla(T)),
9      system_matrix += _A,
10     system_rhs += -_A * _x + integral<2>(transpose(N(T))*N(q)*
           jacobian_determinant) * nodal_values(q)
11   )
12 ));
```

Linear system assembly

$$A_e \left(\mathbf{T_e}^{n+1} - \mathbf{T_e}^{n} \right) = -A_e \mathbf{T_e}^{n} + \int_{\Omega_i} N_T^{\mathrm{T}} N_q \mathbf{q_e}$$

```
1  m_assembly = create_component<ProtoAction>("Assembly");
2  m_assembly->set_expression(elements_expression
3  (
4    boost::mpl::vector<LagrangeP1::Line1D, LagrangeP2::Line1D>(),
5    group
6    (
7      _A = _0,
8      element_quadrature(_A(T) += k*transpose(nabla(T))*nabla(T)),
9      system_matrix += _A,
10     system_rhs += -_A * _x + integral<2>(transpose(N(T))*N(q)*
           jacobian_determinant) * nodal_values(q)
11   )
12 ));
```

Solution update

Solution was for the difference, so:

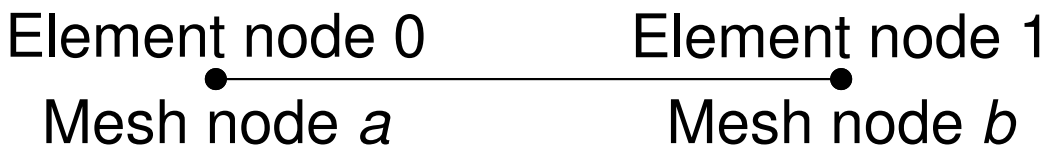
$$T^{n+1} = T^n + \text{solution}$$

This can be written as an expression over the nodes:

```
1 m_update = create_component<ProtoAction>("Update");
2 m_update->set_expression(nodes_expression(T += solution(T)));
```

Element matrix

Line element with two nodes:



2×2 element matrix $\rightarrow$ global system matrix:

$$\begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} \rightarrow \begin{bmatrix} \cdots & \cdots & \text{col } a & \cdots & \text{col } b \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \text{row } a & \vdots & A_{00} & \vdots & A_{01} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \text{row } b & \cdots & A_{10} & \cdots & A_{11} \end{bmatrix}$$

Element matrix encapsulation

- The element matrix is a dense matrix from the Eigen library
- It is contained in a structure that contains an index mapping
- All element-level computations happen in Eigen

```
1  struct BlockAccumulator
2  {
3    // Element matrix
4    Eigen::Matrix<...> mat;
5    // Element solution vector
6    Eigen::Matrix<...> sol;
7    // Element right hand side
8    Eigen::Matrix<...> rhs;
9    // Index mapping
10   std::vector<Uint> indices;
11 };
```

System matrix assembly

Coolfluid matrix interface:

```
1 void add_values(const BlockAccumulator& values);
```

- `Epetra_CrsMatrix`: For each row: call `SumIntoMyValues`
- `Epetra_FEVbrMatrix`: Extract view using `ExtractMyBlockRowView` and then copy each entry

Solution and RHS assembly

Coolfluid vector interface:

```
1 void add_rhs_values(const BlockAccumulator& values);
2 void add_sol_values(const BlockAccumulator& values);
```

- `Epetra_Vector` is a view over an `std::vector`
- Rationale: Store “ghost node” values invisibly to Epetra
- Assembly into the raw data using `std::vector` interface

Boundary conditions

Matrix interface:

```
1 symmetric_dirichlet(const Uint blockrow, const Uint ieq, const
      Real value, Vector& rhs);
```

- Maintain symmetry for e.g. Conjugate Gradient methods
- Set the column and the row to 0, except 1 on the diagonal
- Requires loop over all columns in connected rows to find the correct index
- Caching mechanism for repeated BC application

Linear system solution parameterization

- Linear system solution: controlled by a `SolutionStrategy` object
- Use wrapped `Teuchos::ParameterList`

```
1 lss.SolutionStrategy.Parameters.LinearSolverTypes.Belos.
      solver_type = 'Block GMRES'
2 lss.SolutionStrategy.Parameters.LinearSolverTypes.Belos.
      SolverTypes.BlockGMRES.maximum_iterations = 300
3 lss.SolutionStrategy.Parameters.LinearSolverTypes.Belos.
      SolverTypes.BlockGMRES.num_blocks = 100
4 lss.SolutionStrategy.Parameters.preconditioner_type = 'ML'
5 lss.SolutionStrategy.Parameters.PreconditionerTypes.ML.
      MLSettings.add_parameter(name = 'aggregation: aux: enable',
       value = True)
6 lss.SolutionStrategy.Parameters.PreconditionerTypes.ML.
      MLSettings.add_parameter(name = 'aggregation: aux:
      threshold', value = 0.0001)
```

Linear system solution

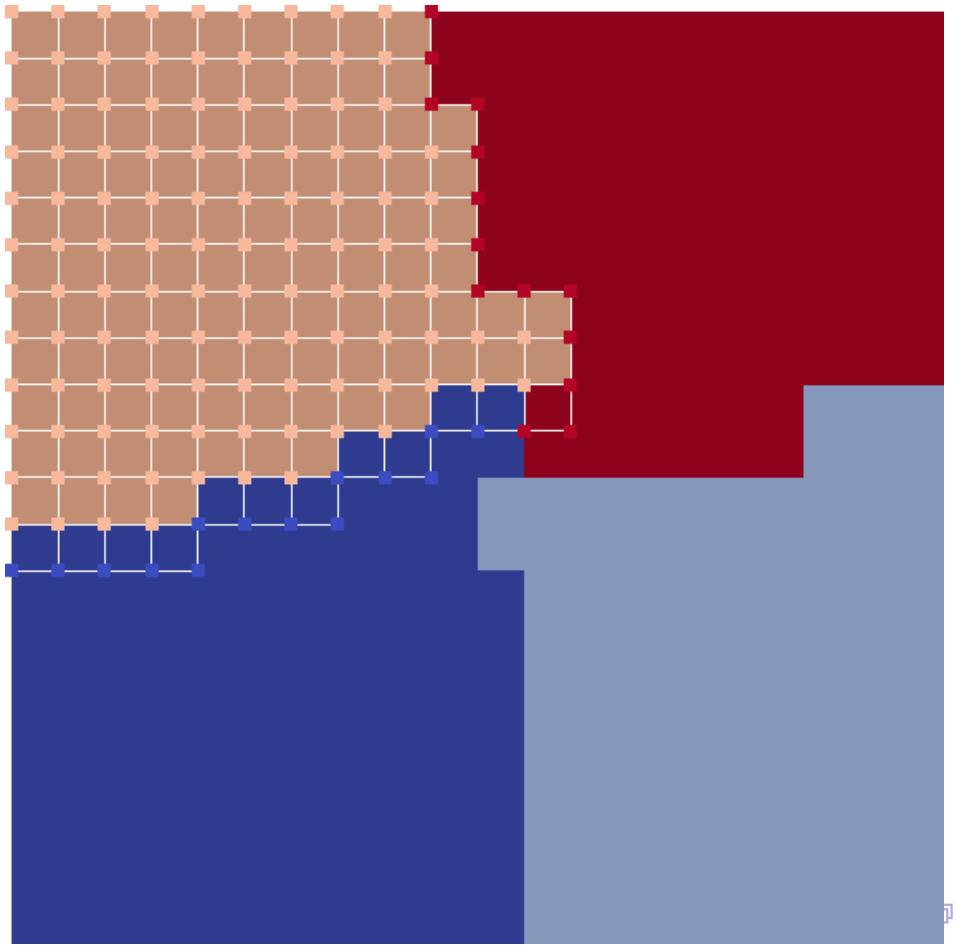
```
1  struct TrilinosStratimikosStrategy
2  {
3  TrilinosStratimikosStrategy() :
4    m_parameter_list(Teuchos::createParameterList()),
5    m_preconditioner_reset(1),
6    m_iteration_count(0)
7  { m_linear_solver_builder.setParameterList(m_parameter_list); }
8
9  void setup_solver() {
10   m_lows_factory = Thyra::createLinearSolveStrategy(
         m_linear_solver_builder);
11 }
12
13 Teuchos::RCP<Teuchos::ParameterList> m_parameter_list;
14 Stratimikos::DefaultLinearSolverBuilder m_linear_solver_builder;
15
16 Teuchos::RCP<Thyra::LinearOpWithSolveFactoryBase<double> >
       m_lows_factory;
17 Teuchos::RCP<Thyra::LinearOpWithSolveBase<double> > m_lows;
18 // CF3 data...
19 };
```

Linear system solution

```
1  void solve() {
2   if(m_lows.is_null())
3     m_lows = m_lows_factory->createOp();
4
5   // Reuse the preconditioner
6   if(m_iteration_count % m_preconditioner_reset == 0)
7     Thyra::initializeOp(*m_lows_factory, m_matrix->thyra_operator
          (), m_lows.ptr());
8   else
9     Thyra::initializeAndReuseOp(*m_lows_factory, m_matrix->
          thyra_operator(), m_lows.ptr());
10
11  RCP< Thyra::VectorBase<Real> const > b = m_rhs->thyra_vector();
12  RCP< Thyra::VectorBase<Real> > x = m_solution->thyra_vector();
13  Thyra::SolveStatus<double> status = Thyra::solve<double>(*
        m_lows, Thyra::NOTRANS, *b, x.ptr());
14
15  ++m_iteration_count;
16 }
```

Mesh partitioning

Zoltan Hypergraph partitioning (experimental), overlap to complete elements



Overview

Stabilized Navier-Stokes

Unsteady linear system:

$$\sum_{e=1}^{N} \left(\frac{1}{\Delta t} T_e + \theta A_e \right) \left(\mathbf{x}_e^{n+1} - \mathbf{x}_e^n \right) = -A_e \mathbf{x}_e^n$$

Unknowns: pressure and velocity

$$\mathbf{x}_e^n = \left[p_0^n \cdots p_m^n \, (u_0^n)_0 \cdots (u_0^n)_m \cdots (u_2^n)_m \right]$$

Element matrices A_e and T_e:

$$A_e = \begin{bmatrix} A_{pp} & A_{pu} \\ A_{up} & A_{uu} \end{bmatrix} = \begin{bmatrix} A_{pp} & A_{pu_0} & A_{pu_1} & A_{pu_2} \\ A_{u_0 p} & A_{u_0 u_0} & A_{u_0 u_1} & A_{u_0 u_2} \\ A_{u_1 p} & A_{u_1 u_0} & A_{u_1 u_1} & A_{u_1 u_2} \\ A_{u_2 p} & A_{u_2 u_0} & A_{u_2 u_1} & A_{u_2 u_2} \end{bmatrix}$$

Element integrals

$$A_{pp} = \int_{\Omega_e} \tau_{\mathrm{PS}} \nabla N_p^{\mathrm{T}} \nabla N_p \mathrm{d}\Omega_e$$

$$A_{pu_i} = \int_{\Omega_e} \left(\left(N_p + \frac{\tau_{\mathrm{PS}} \tilde{\mathbf{u}}_{\mathrm{adv}} \cdot \nabla N_p}{2} \right)^{\mathrm{T}} (\nabla N_u)_i + \tau_{\mathrm{PS}} \left(\nabla N_p \right)_i^{\mathrm{T}} \tilde{\mathbf{u}}_{\mathrm{adv}} \nabla N_u \right) \mathrm{d}\Omega_e$$

$$T_{pu_i} = \int_{\Omega_e} \tau_{\mathrm{PS}} \left(\nabla N_p \right)_i^{\mathrm{T}} N_u \mathrm{d}\Omega_e$$

Element integrals

$$A_{u_i u_j} = \int_{\Omega_e} \left(\tau_{\mathrm{BU}} \left(\nabla N_u \right)_i + \frac{1}{2} \left(\tilde{\mathbf{u}}_{\mathrm{adv}} \right)_i \left(N_u + \tau_{\mathrm{SU}} \mathbf{u} \nabla N_u \right) \right)^{\mathrm{T}} \left(\nabla N_u \right)_j \mathrm{d}\Omega_e$$

$$A_{u_i u_i} = \int_{\Omega_e} \left(\nu \nabla N_u^{\mathrm{T}} \nabla N_u + \left(N_u + \tau_{\mathrm{SU}} \tilde{\mathbf{u}}_{\mathrm{adv}} \nabla N_u \right)^{\mathrm{T}} \tilde{\mathbf{u}}_{\mathrm{adv}} \nabla N_u \right) \mathrm{d}\Omega_e + A_{u_i u_j}$$

$$A_{u_i p} = \int_{\Omega_e} \left(N_u + \tau_{\mathrm{SU}} \tilde{\mathbf{u}}_{\mathrm{adv}} \nabla N_u \right)^{\mathrm{T}} \left(\nabla N_p \right)_i \mathrm{d}\Omega_e$$

$$T_{u_i u_i} = \int_{\Omega_e} \left(N_u + \tau_{\mathrm{SU}} \tilde{\mathbf{u}}_{\mathrm{adv}} \nabla N_u \right)^{\mathrm{T}} N_u \mathrm{d}\Omega_e$$

Element matrix computation

```
1  element_quadrature
2  (
3    _A(p    , u[_i]) += transpose(N(p) + tau_ps*u_adv*nabla(p)
         *0.5) * nabla(u)[_i] + tau_ps * transpose(nabla(p)[_i]) *
          u_adv*nabla(u),
4    _A(p    , p)     += tau_ps * transpose(nabla(p)) * nabla(p),
5    _A(u[_i], u[_i]) += nu_eff * transpose(nabla(u)) * nabla(u) +
         transpose(N(u) + tau_su*u_adv*nabla(u)) * u_adv*nabla(u),
6    _A(u[_i], p)     += transpose(N(u) + tau_su*u_adv*nabla(u)) *
         nabla(p)[_i],
7    _A(u[_i], u[_j]) += transpose(tau_bulk*nabla(u)[_i]
8                        + 0.5*u_adv[_i]*(N(u) + tau_su*u_adv*nabla
                             (u))) * nabla(u)[_j],
9    _T(p    , u[_i]) += tau_ps * transpose(nabla(p)[_i]) * N(u),
10   _T(u[_i], u[_i]) += transpose(N(u) + tau_su*u_adv*nabla(u)) *
         N(u),
11   _a[u[_i]] += transpose(N(u) + tau_su*u_adv*nabla(u)) * g[_i] *
          Tref / T
12 )
```

Semi-implicit solution

Parameterization for full system:

- Belos Block GMRES
- ML preconditioner (NSSA, 2 Chebyshev sweeps, full-MGV)
- Convergence can be slow, especially on fine 3D grids (e.g. turbulent channel DNS)
- Poor conditioning mainly due to advective terms

Semi-implicit solution:

- Treat advection terms explicitly
- Solve separate systems for pressure and velocity
- Time step is limited due to stability concerns!

Semi-implicit algorithm

1: $\mathbf{u}^0 = \mathbf{u}^n$
2: $p^0 = p^n$
3: $\mathbf{a}^0 = \mathbf{0}$
4: **for** $m = 0$ to $M - 1$ **do**
5: Solve velocity system for $\Delta\mathbf{a}^*$
6: Solve pressure system for Δp^{m+1}
7: Compute:
$\Delta\mathbf{a} = \Delta\mathbf{a}^* - \left(T_{uu} + \theta\Delta t A_{uu}\right)^{-1} \theta A_{up}\Delta p^{m+1}$
8: Update $\mathbf{u}^{m+1} = \mathbf{u}^m + \Delta t\Delta\mathbf{a}$
9: Update $p^{m+1} = p^m + \Delta p^{m+1}$
10: **end for**

- Matrix operations implemented using Thyra
- Need ghost nodes in result vectors for assembly computation!

Properties of the separate systems

Velocity system:

- Well conditioned, SPD
- ILU + CG, with reuse of the preconditioner

Pressure system:

- Poisson problem
- Constant coefficients during the entire simulation!
- Factor once if small enough
- ML + CG otherwise
- Usually the slowest step in the algorithm

Tests on clusters

RMA

- 42 nodes, 8 cores/node
- 1 Gb Ethernet interconnect

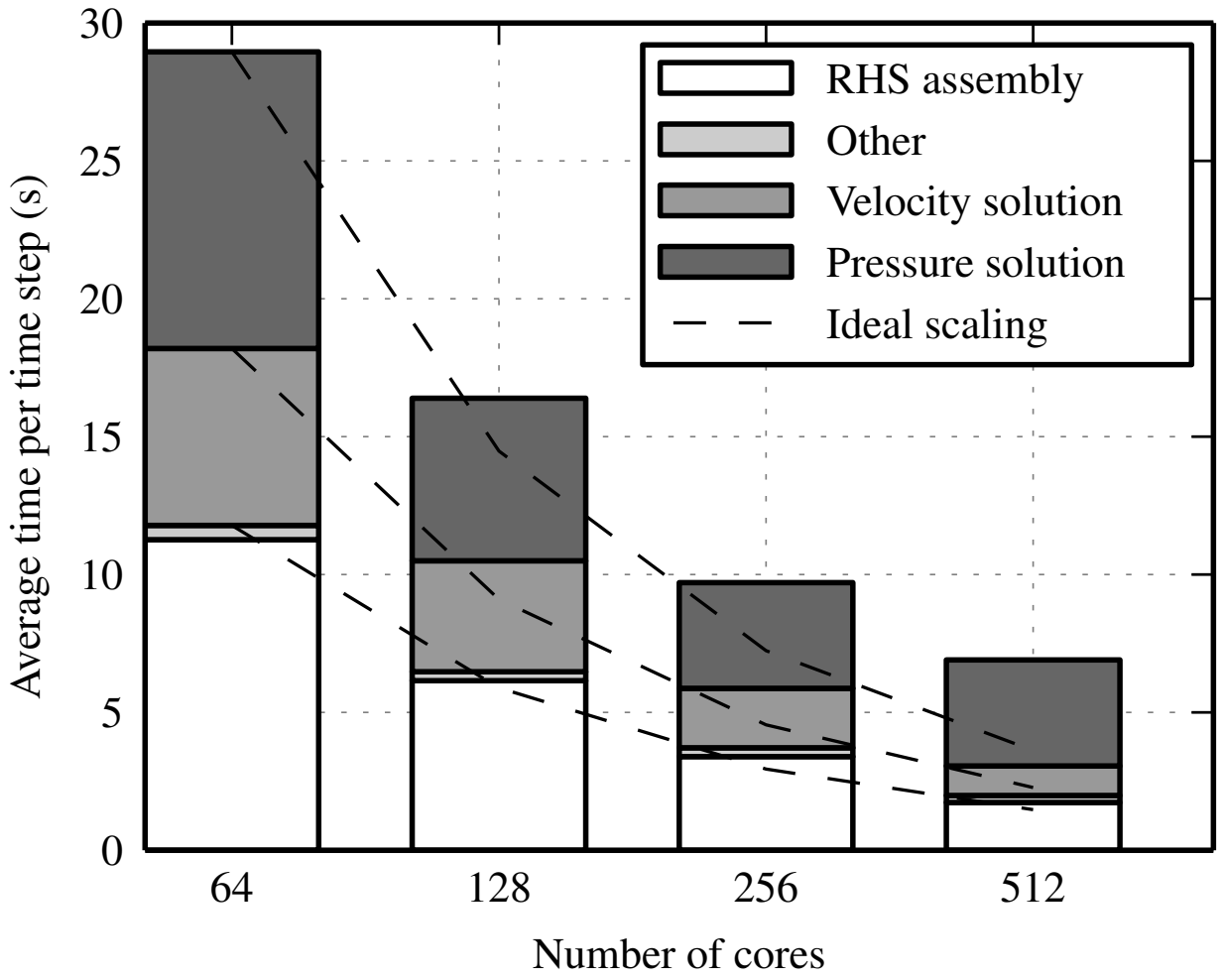
VKI

- 28 nodes, 64 cores/node
- InfiniBand interconnect

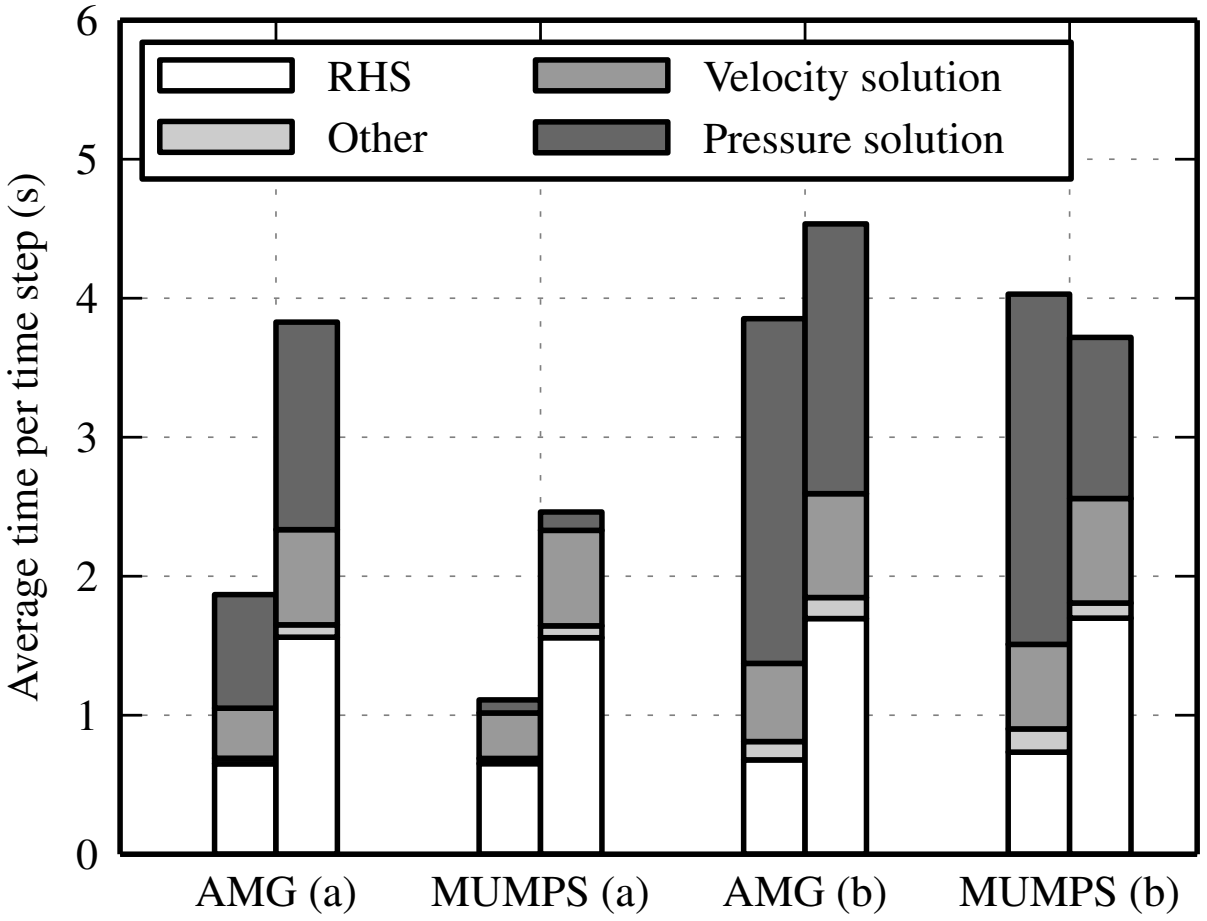
Segregated vs. Coupled

	Segregated		*Coupled*		
Mesh	t_s (s)	Scaling	t_c (s)	Scaling	t_c/t_s
$32 \times 65 \times 32$	1.44	-	7.64	-	5.31
$32 \times 129 \times 32$	4.78	3.32	35.46	4.64	7.42
$32 \times 257 \times 32$	12.26	2.57	277.96	7.84	22.67
$32 \times 65 \times 32$ random	1.70	-	15.26	-	8.98
$32 \times 129 \times 32$ random	3.60	2.12	66.00	4.33	18.35
$32 \times 257 \times 32$ random	11.99	3.33	575.30	8.72	47.98

Parallel performance ($128 \times 257 \times 128$)



Parallel performance (MUMPS vs ML)



a: $32 \times 65 \times 32$ b: $64 \times 129 \times 64$

Overview

Conclusions

- Domain Specific Language: provides abstraction for model developers
- Link with Python for dynamic configuration
- Satisfactory scalability thanks to Trilinos

Open issues

- Treatment of the pressure Poisson problem?
- Direct solver interfacing: Trilinos or direct interface?
- Mesh partitioning strategy (especially with periodic BC?)
- Load all possible defaults into ParameterList?
- Use Tpetra, Kokkos...?