Block-Coupled Simulations Using OpenFOAM

6th OpenFOAM Workshop
13-16 June 2011

Ivor Clifford
Outline

• Introduction (5 min)
• Theory of the Block Matrix Solver (10min)
• Matrix Classes in OpenFOAM (15min)
• Two Approaches to Implementation (45min)
  – Manual matrix construction - dissimilar equation sets
  – Inline matrix construction – similar equation sets
• Further Discussion (15min)
  – Parallel processing
  – Coupled interfaces and multi-region simulations
Introduction

• Background
  – Block-coupled matrix solver included in OF-1.6-ext
  – Intended for use in implicit solutions of strongly coupled variables sharing a common mesh
  – Primary development by Hrvoje Jasak
  – Still under development, however has been used successfully in several research projects
    • ECCOMAS 2010 by Kathrin Kissling and Julia Springer (VOF Multiphase Flows)
    • M&C 2009 and HTR 2010 by Ivor Clifford (Neutronics in Nuclear Reactors)

• Workshop Aims and Objectives
  – Introduce new block-coupled matrix solver
  – Provide theoretical background
  – Introduce the basic classes of the block-coupled matrix solver
  – Demonstrate how to implement new solvers using the block-coupled matrix solver
  – Discuss issues of parallelization and coupled interfaces
Outline

• Introduction (5 min)
• Theory of the Block Matrix Solver (10min)
• Matrix Classes in OpenFOAM (15min)
• Two Approaches to Implementation (45min)
  – Manual matrix construction - dissimilar equation sets
  – Inline matrix construction – similar equation sets
• Further Discussion (15min)
  – Parallel processing
  – Coupled interfaces and multi-region simulations
Theory of the Block Matrix Solver

Finite-volume discretization of block-coupled equation set
- \((u_x, u_y, u_z)\) in cell P is dependent on \(u_P\) in cell P and \((u_x, u_y, u_z)_N\) in neighbours N
- Off-diagonal entries only for cells sharing a face
- Resulting discretization for cell P

\[
a_P u_P + \sum_N a_N u_N = b
\]

- In current framework, \(au\) written as a tensorial product

\[
au = \begin{bmatrix} a_{xx} & a_{xy} & a_{xz} \\ a_{yx} & a_{yy} & a_{yz} \\ a_{zx} & a_{zy} & a_{zz} \end{bmatrix} \begin{bmatrix} u_x \\ u_y \\ u_z \end{bmatrix}
\]

- Assembled sparse linear system

\[
[A][u] = [b]
\]
Theory of the Block Matrix Solver

\[
\begin{pmatrix}
    a_{xx} & a_{xy} & a_{xz} \\
    a_{yx} & a_{yy} & a_{yz} \\
    a_{zx} & a_{zy} & a_{zz}
\end{pmatrix}
\cdots
\begin{pmatrix}
    a_{xx} & a_{xy} & a_{xz} \\
    a_{yx} & a_{yy} & a_{yz} \\
    a_{zx} & a_{zy} & a_{zz}
\end{pmatrix}
\cdots
\begin{pmatrix}
    a_{xx} & a_{xy} & a_{xz} \\
    a_{yx} & a_{yy} & a_{yz} \\
    a_{zx} & a_{zy} & a_{zz}
\end{pmatrix}

\begin{pmatrix}
    u_x \\
    u_y \\
    u_z
\end{pmatrix}

= 

\begin{pmatrix}
    b_x \\
    b_y \\
    b_z
\end{pmatrix}

Levels of Block Coupling

Three special cases

- **Segregated** - no coupling between variables
  - Diagonal and off-diagonal coefficients in form of diagonal or spherical tensors
    \[
    a_P u_p + \sum_N a_N u_N = \begin{bmatrix}
    a_{xx} & a_{yy} \\
    a_{yy} & a_{zz}
    \end{bmatrix}
    \begin{bmatrix}
    u_p \\
    u_N
    \end{bmatrix}
    \]

- **Point-implicit** - coupling between variables only in owner (e.g. chemical reactions)
  - Diagonal coefficient in form of full tensor
    \[
    a_P u_p + \sum_N a_N u_N = \begin{bmatrix}
    a_{xx} & a_{xy} & a_{xz} \\
    a_{yx} & a_{yy} & a_{yz} \\
    a_{zx} & a_{zy} & a_{zz}
    \end{bmatrix}
    \begin{bmatrix}
    u_p \\
    u_p \\
    u_N
    \end{bmatrix}
    \]
Levels of Block Coupling

- Fully coupled - coupling between variables in owner and neighbour cells (e.g. stress analysis, adjoint convection)

\[
a_p \mathbf{u}_p + \sum_{N} a_N \mathbf{u}_N = \begin{bmatrix} a_{xx} & a_{xy} & a_{xz} \\ a_{yx} & a_{yy} & a_{yz} \\ a_{zx} & a_{zy} & a_{zz} \end{bmatrix}_{p} \mathbf{u}_p + \sum_{N} \begin{bmatrix} a_{xx} & a_{xy} & a_{xz} \\ a_{yx} & a_{yy} & a_{yz} \\ a_{zx} & a_{zy} & a_{zz} \end{bmatrix}_N \mathbf{u}_N
\]

- In spirit of generic programming, the aim is to support all levels of coupling using same underlying functionality

- The size of the block-coupled system is arbitrary (2x2, 3x3, …, NxN)
Linear Solver Algorithms

- Sparseness pattern of block matrix is unchanged from scalar matrix

- Iterative solution algorithms use simple operations
  - Vector-matrix multiplication
  - Gauss-Seidel sweep
  - Matrix decomposition

- All readily generalize for matrix with tensor coefficients
  - Simply define primitive operations for NxN coefficients and N-length vectors (coefficient-vector multiplication, coefficient inversion, dot product, etc.)
Example: ILU Factorization

```
for r := 1 step 1 until n-1 do
    d := 1/a_{rr}
    for i := (r+1) step 1 until n do
        e := d a_{i,r};
        a_{i,r} := e;
        for j := (r+1) step 1 until n do
            a_{i,j} := a_{i,j} - e a_{r,j}
        end (j-loop)
    end (i-loop)
end (r-loop)
```

***Tony F. Chan and Hank A. Van Der Vorst, "Approximate and Incomplete Factorizations"
Introduction (5 min)

Theory of the Block Matrix Solver (10 min)

Matrix Classes in OpenFOAM (15 min)

Two Approaches to Implementation (45 min)
  - Manual matrix construction - dissimilar equation sets
  - Inline matrix construction – similar equation sets

Further Discussion (15 min)
  - Parallel processing
  - Coupled interfaces and multi-region simulations
IduMatrix vs. blockLduMatrix

**IduMatrix**
- Matrix stored in 3 parts in *arrow format*
  - Diagonal coefficients
  - Off-diagonal coefficients, upper triangle
  - Off-diagonal coefficients, lower triangle
- Addressing stored as `lduAddressing` object (owner/neighbour information) within `lduMesh`
- Coefficients stored as `scalarFields` (even for vector equations)

**blockLduMatrix**
- See `$FOAM_SRC/OpenFOAM/matrices/blockLduMatrix`
- Sparse matrix structure unchanged, `lduAddressing` still applies
- Templated to allow for arbitrary number of coupled equations
  - Templating allows us to optimize coefficient operations and storage
  - Allows more esoteric definitions of coefficients, e.g. rank 3 tensors
  - Makes it more difficult to vary the number of equations programmatically
- Coefficient storage more complex (new `CoeffField` class)
template<class Type> class BlockLduMatrix
{
private:

    // LDU mesh reference
    const lduMesh& lduMesh_;

    // Block matrix elements
    // Diagonal coefficients
    CoeffField<Type>* diagPtr_;

    // Upper triangle coefficients. Also used for symmetric matrix
    CoeffField<Type>* upperPtr_;

    // Lower triangle coefficients
    CoeffField<Type>* lowerPtr_;

    // Coupling
    // List of coupled interfaces
    typename BlockLduInterfaceFieldPtrsList<Type>::Type interfaces_;

    // Coupled interface coefficients, upper
    FieldField<CoeffField, Type> coupleUpper_;

    // Coupled interface coefficients, lower
    FieldField<CoeffField, Type> coupleLower_;
BlockLduMatrix Class

- Combines lduMatrix and some features from fvMatrix into a single class in the block context
  - Since lduMatrix is not templated, not all info for vector/tensor equations can be stored here. These are included in fvMatrix instead
- Examples: Coupled interfaces, coupled coefficients and source
  - Coupled interfaces and coupled coefficients now lie in BlockLduInterface (previous approach was less than ideal)
  - For now source still lies outside BlockLduMatrix
- Amul, Tmul, H, add, subtract and negate operators, etc.
- Includes functionality for decoupled/segregated solution
template<class Type> class CoeffField;

- **See** $FOAM_SRC/OpenFOAM/fields/CoeffField
- **Special type of field that can take on four forms**
  - `BlockCoeffBase::UNALLOCATED`; coefficients are unallocated or zero
  - `BlockCoeffBase::SCALAR`; spherical tensors
  - `BlockCoeffBase::LINEAR`; diagonal tensors
  - `BlockCoeffBase::SQUARE`, full NxN tensors
- **For CoeffField<vector>, data is stored internally as a scalarField, vectorField, or tensorField respectively for each form**
  - Underlying fields retrieved using `asScalar()`, `asLinear()` and `asSquare()` member functions
  - Underlying type can be expanded or contracted using `toScalar()`, `toLinear()` and `toSquare()` member functions
- **Provides efficient storage and on-the-fly coefficient expansion**
Arbitrary Size Coefficients

• For N coupled equations
  – N-length vector → VectorN<cmpt, length>
  – NxN tensor → TensorN<cmpt, length>
  – Diagonal and Spherical NxN tensors are also provided (DiagTensorN and SphericalTensorN). These come in useful later

• Current explicit instantiations for 2, 4, 6 and 8 components
  – Length 3 vectors and 3x3 tensors are part of standard OpenFOAM

• Custom specializations can be implemented for optimization of primitive operations and tensor inversion
  – E.g. vector2 uses analytical expression for inversion, while vector4 and up use Gauss-Jordan elimination
Putting it all Together

• Example: 4 coupled equations
Outline

- Introduction (5 min)
- Theory of the Block Matrix Solver (10 min)
- Matrix Classes in OpenFOAM (15 min)
- Two Approaches to Implementation (45 min)
  - Manual matrix construction - dissimilar equation sets
  - Inline matrix construction – similar equation sets
- Further Discussion (15 min)
  - Parallel processing
  - Coupled interfaces and multi-region simulations
Two basic types of equation sets

- **Arbitrary equations for different physical processes (mass, momentum, energy, thermal diffusion)**
  - Equation structure differs (parabolic, hyperbolic, elliptical)
  - Segregated matrix structure differs (diagonal, symmetric, asymmetric)
  - Dimensions of solution variables differ
  - Discretization of coupling-terms is problem specific
  - Manual approach to block matrix construction

- **Similar equations (chemical reactions, neutron diffusion)**
  - Equation identical for each species (only coefficients change)
  - Matrix structure identical
  - Dimensions of solution variables identical
  - Coupling terms can be treated in generic fashion
  - Matrix construction can be done inline (with some work)
blockCoupledScalarTransportFoam

Block-Coupled Simulations Using OpenFOAM
Dissimilar Equation Sets

Coupled two-phase fluid/solid (porous medium) heat transfer

\[
\nabla \cdot \phi T_f - \nabla \cdot D_{T_f} \nabla T_f = \alpha (T_s - T_f) \\
- \nabla \cdot D_{T_s} \nabla T_s = \alpha (T_f - T_s)
\]

- Traditional segregated solution approach

```c
solve
(
    fvm::div(phi, Tf) - fvm::laplacian(DTf, Tf)
    == alpha*Ts - fvm::Sp(alpha, Tf)
);

solve
(
    - fvm::laplacian(DTs, Ts) == alpha*Tf - fvm::Sp(alpha, Ts)
);
```

- May require relaxation and several outer iterations for stiff problems
Constructing the Block Matrix

```cpp
#include "blockVectorNMatrices.H"

// Prepare block system
BlockLduMatrix<vector2> blockM(mesh);

// Grab block diagonal and set it to zero
Field<tensor2>& d = blockM.diag().asSquare();
d = tensor2::zero;

// Grab linear off-diagonal and set it to zero
Field<vector2>& u = blockM.upper().asLinear();
Field<vector2>& l = blockM.lower().asLinear();
u = vector2::zero;
l = vector2::zero;

// Create the source term
vector2Field blockB(mesh.nCells(), vector2::zero);

// Create the working solution variable
vector2Field blockX(mesh.nCells(), vector2::zero);
```
#include "blockMatrixTools.H"

//-- Construct scalar matrices
fvScalarMatrix TEqn
{
   fvm::div(phi, T) - fvm::laplacian(DT, T) == alpha*Ts - fvm::Sp(alpha, T)
};

fvScalarMatrix TsEqn
{
   - fvm::laplacian(DTs, Ts) == alpha*T - fvm::Sp(alpha, Ts)
};

//-- Insert equations into block Matrix
blockMatrixTools::insertEquation(0, TEqn, blockM, blockX, blockB);
blockMatrixTools::insertEquation(1, TsEqn, blockM, blockX, blockB);

//-- Add off-diagonal terms and remove from Block source
forAll(d, i)
{
   d[i](0,1) = -alpha.value() * mesh.V()[i];
d[i](1,0) = -alpha.value() * mesh.V()[i];

   blockB[i][0] = alpha.value() * blockX[i][1] * mesh.V()[i];
   blockB[i][1] = alpha.value() * blockX[i][0] * mesh.V()[i];
}

Matrix Coefficients
/--- Block coupled solver call
BlockSolverPerformance< vector2 > solverPerf =
   BlockLduSolver< vector2 >::New
   ( word("blockVar"),
     blockM,
     mesh.solver("blockVar")
   )->solve(blockX, blockB);

   solverPerf.print();

/--- Retrieve solution
blockMatrixTools::blockRetrieve(0, T.internalField(), blockX);
blockMatrixTools::blockRetrieve(1, Ts.internalField(), blockX);

T.correctBoundaryConditions();
Ts.correctBoundaryConditions();
Results - swirlTest

2-eq swirlTest

Maximum relative error vs. iteration for different solutions:
- Segregated solution T
- Segregated solution Ts
- Block coupled solution T
- Block coupled solution Ts
Real World Applications
Volume-of-Fluid CFD

Pressure-based volume fraction equation
\[
\frac{\partial [\alpha_i]}{\partial t} + \nabla \cdot \left( \left( \frac{A^i}{A_D} \right)_f \cdot S + \sum_{i=1}^N \sum_{k=4+1}^N \left( \frac{1}{A_D} \right)_f (\sigma_{ik} R_{ik})_f \left| S \right| \left( \nabla^T_i \alpha \right)_k \right)
- \left( \frac{1}{A_D} \right)_f (F \cdot \nabla)_f \left| S \right| \left( \nabla^T_i \rho \right)_f \left[ \alpha_i \right]_f \right] 
- \nabla \cdot \left( \left( \frac{1}{A_D} \right)_f \nabla \left[ p^* \alpha_i \right]_f \right) + \nabla \cdot \left( \left[ \alpha_i \right]_f \sum_{k=1,k\neq i}^N \alpha_{ik} \phi_{r,ik} \right) = 0
\]

Pressure equation
\[
\nabla \cdot \left( \left( \frac{1}{A_D} \right)_f \nabla \left[ p^* \right] \right) = \nabla \cdot \sum_{i=1}^N \left[ \alpha_i \right] \phi^*
\]

Closure relationship
\[
\sum_{i=1}^N \left[ \alpha_i \right] = 1
\]

Outline

• Introduction (5 min)
• Theory of the Block Matrix Solver (10 min)
• Matrix Classes in OpenFOAM (15 min)
• Two Approaches to Implementation (45 min)
  – Manual matrix construction - dissimilar equation sets
  – Inline matrix construction – similar equation sets
• Further Discussion (15 min)
  – Parallel processing
  – Coupled interfaces and multi-region simulations
The Fission Process
Multi-group Neutron Diffusion

- Set of $G$ equations representing neutron behaviour at different energies (groups)

$$\frac{1}{v_g} \frac{\partial \phi_g}{\partial t} - \nabla \cdot D_g \nabla \phi_g + \Sigma_{r,g} \phi_g = \sum_{h \neq g} \Sigma_h \phi_h + \chi_g \sum_h \nu \Sigma_{f,h} \phi_h \quad g = 1, \ldots, G$$

- In vector form

$$v^{-1} \frac{\partial \vec{\phi}}{\partial t} - \nabla \cdot D \nabla \vec{\phi} + \Sigma_r \cdot \vec{\phi} = \Sigma_s \cdot \vec{\phi} + \chi v \Sigma_f \cdot \vec{\phi}$$

- Number of groups $G$ typically lies between 2 and 30
- Energy varies from MeV to fractions of eV (highly stiff)
  - Segregated solution diverges quickly
- Can use manual approach discussed previously – Initially done this way
  - Quickly gets messy for many groups
  - 8 groups $\rightarrow$ 64 entries for $\Sigma_s$ and $\chi v \Sigma_f$
  $\Rightarrow$ Data handling alone becomes an issue
• Take advantage of similarity between equations
  - Define $\vec{\phi}$ as a volField of arbitrary-length vectors (VectorN<scalar,G>)
  - Define equation coefficients using fields of tensors, diagonal tensors and spherical tensors (TensorN<scalar,G>, DiagTensorN<scalar,G>, SphericalTensorN<scalar,G>)

• Additional work needed
  - Need a comprehensive library for working with fields of arbitrary-length vectors
    ⇒ Already done, see $\$FOAM\_SRC/VectorN$
  - Define BlockFvMatrix<Type> which performs same functionality as fvMatrix for block systems
  - Define custom fvm::ddt, fvm::laplacian, and fvm::Sup operators for BlockFvMatrix<VectorN<scalar,G> >
#include "VectorN.H"
#include "TensorN.H"
#include "VectorNFieldTypes.H"
#include "dimensionedVectorTensorN.H"
#include "volVectorNFields.H"

#include "blockFvCFD.H"
#include "blockFvMatrices.H"
#include "EulerBlockVectorNDDtScheme.H"
#include "gaussBlockVectorNLaplacianScheme.H"

#define nGroups 4

typedef Foam::VectorN<scalar, nGroups> vectorType;
typedef Foam::TensorN<scalar, nGroups> tensorType;
typedef Foam::DiagTensorN<scalar, nGroups> diagTensorType;

typedef dimensioned<vectorType> dimensionedVectorType;
typedef dimensioned<tensorType> dimensionedTensorType;
typedef dimensioned<diagTensorType> dimensionedDiagTensorType;

typedef GeometricField<vectorType, fvPatchField, volMesh> volVectorTypeField;
typedef GeometricField<tensorType, fvPatchField, volMesh> volTensorTypeField;
typedef GeometricField<diagTensorType, fvPatchField, volMesh> diagTensorTypeField;
//-- Solution Fields
volVectorTypeField phi
(
    IOobject(
        "phi",
        runTime.timeName(),
        mesh,
        IOobject::MUST_READ,
        IOobject::AUTO_WRITE
    ),
    mesh
);

//-- Cross sections
dimensionedDiagTensorType rV(crossSections.lookup("inverseNeutronVelocity"));
dimensionedVectorType D(crossSections.lookup("diffusionCoeff"));
dimensionedDiagTensorType sigma_a(crossSections.lookup("absorption"));
dimensionedDiagTensorType sigma_f(crossSections.lookup("fission"));
dimensionedTensorType sigma_s(crossSections.lookup("scattering"));

dimensionedVectorType chi(crossSections.lookup("fissionSpectrum"));
dimensionedVectorType nu(crossSections.lookup("neutronsPerFission"));
crossSections Dictionary

inverseNeutronVelocity  rV  [ 0 -1  1  0  0  0  0 ] ( 8e-10 2e-08 2e-07 3.5e-06 );
diffusionCoeff          D   [ 0 1  0  0  0  0  0 ] ( 3 2 1.5 1 );
absorption              sigma_a [ 0 -1  0  0  0  0  0 ] ( 6e-05 0.0004 0.001 0.0005 );
fission                 sigma_f [ 0 -1  0  0  0  0  0 ] ( 0.0001 0.0003 0.003 0.05 );
scattering              sigma_s [ 0 -1  0  0  0  0  0 ] ( -0.01 0 0 0
                                    0.01 -0.01 0 0
                                    0 0.01 -0.015 0
                                    0 0 0.015 0
                                    );
fissionSpectrum         chi  [ 0 0  0  0  0  0  0 ] ( 0.95 0.05 0 0 );
neutronsPerFission      nu   [ 0 0  0  0  0  0  0 ] ( 2.5 2.5 2.5 2.5 );
Solution Variable Dictionary

FoamFile
{
  class volVector4Field;
  object phi;
}

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
dimensions [[0 -2 -1 0 0 0 0]];
internalField uniform (0 0 0 0);
boundaryField
{
  frontAndBack
  {
    type empty;
  }
  topAndBottom
  {
    type fixedValue;
    value uniform (0 0 0 0);
  }
  leftAndRight
  {
    type fixedValue;
    value uniform (1e10 0 0 0);
  }
}
• Notes
  – No need for packing and retrieval of matrix, source and solution fields. We work directly with these
  – Finite-volume operators are problem specific
  – Minimal support for runtime selection of schemes. Essentially requires duplication of all `fvMatrix` operators to introduce this
  – Working with vector and tensor variables can be tricky. Functionality still under development

```cpp
class solve {
    fv::EulerBlockVectorNDdtScheme<scalar, nGroups>(mesh)
    .fvmDdt(rV, phi)
    - fv::gaussBlockVectorNLaplacianScheme<scalar, nGroups>(mesh)
    .fvmLaplacian(D, phi)
    + blockFvm::Sp(sigma_a - sigma_s - (chi * (nu & sigma_f)), phi)
};
```
Results - slab

Slower response of intermediate & thermal neutrons

Short timescales of fast neutrons
Real World Application
PBMR400 Nuclear Reactor

Outline

• Introduction (5 min)

• Theory of the Block Matrix Solver (10 min)

• Matrix Classes in OpenFOAM (15 min)

• Two Approaches to Implementation (45 min)
  – Manual matrix construction - dissimilar equation sets
  – Inline matrix construction – similar equation sets

• Further Discussion (15 min)
  – Parallel processing
  – Coupled interfaces and multi-region simulations
Parallel Processing

• No mention of parallel processing up to now... Why?
  – There are still mixed ideas on how to handle this
  – Parallel processing in OpenFOAM intrinsically linked to boundary conditions through coupled patches
  – We can pack coefficients into the block matrix, we cannot simply do the same with boundary conditions
  – Up to now, block matrix functionality has been added on top of existing OpenFOAM classes; We’ve had to fiddle a little deeper to enable parallel processing

• Base functionality for exchanging coupled data uses lduInterface and lduInterfaceField
  – lduInterface associated with mesh and addressing; No change required for block matrix
  – lduInterfaceField designed for scalar equations, and hacked to work for vector equations (not templated); Needs to be fixed from the ground up
virtual void lduInterfaceField::updateInterfaceMatrix
(
    const scalarField& psiInternal,
    scalarField& result,
    const lduMatrix&,
    const scalarField& boundaryCoeffs,
    const direction cmpt,
    const Pstream::commsTypes commsType
) const;

template< class Type >
virtual void BlockLduInterfaceField< Type >::updateInterfaceMatrix
(
    const Field< Type >& psiInternal,
    Field< Type >& result,
    const BlockLduMatrix< Type >&,
    const CoeffField< Type >& coeffs,
    const Pstream::commsTypes commsType
) const = 0;
Parallel Processing

• To enable parallel processing, or any other coupled boundary (cyclic, regionCouple, etc.)
  – Loop over all the boundaries and identify the patches that are coupled
  – Manually pack the boundary coefficients into BlockLduMatrix
  – Provide definitions for initInterfaceMatrixUpdate and updateInterfaceMatrix for each coupled interface
  – Create a list of interface pointers
  – Hook the list of interfaces in BlockLduMatrix

• All the difficult bits are already done by OpenFOAM when constructing volFields
  – We use a dummy volField to give us the interface list directly
    • OpenFOAM automatically creates processor, cyclic, etc. boundaries based on the underlying patch type. No need for us to manually check the boundary type
  – Added bonus; We automatically get a working field to store the solution on
  – Some wasted storage for the boundary fields
  – Ignore the dimensions; They have not meaning in this context

• For the inline approach, this is inherent; No need to do anything special
// Create the dummy volField
volVector2Field blockT
{
    IOobject
    (
        "blockT",
        runTime.timeName(),
        mesh,
        mesh,
        dimensionedVector2(word(), dimless, vector2::zero)
    );
}

// Create a reference to the working solution field
// vector2Field blockX(mesh.nCells(), vector2::zero);
vector2Field& blockX = blockT.internalField();

// Transfer the coupled interface list for processor/cyclic/etc. boundaries
blockM.interfaces() = blockT.boundaryField().blockInterfaces();

// Transfer the coupled interface coefficients
forAll(mesh.boundaryMesh(), patchI)
{
    if (blockM.interfaces().set(patchI))
    {
        Field<vector2>& coupledLower = blockM.coupleLower()[patchI].asLinear();
        Field<vector2>& coupledUpper = blockM.coupleUpper()[patchI].asLinear();

        blockMatrixTools::blockInsert(0, TEqn.internalCoeffs()[patchI], coupledLower);
        blockMatrixTools::blockInsert(1, TsEqn.internalCoeffs()[patchI], coupledLower);
        blockMatrixTools::blockInsert(0, TEqn.boundaryCoeffs()[patchI], coupledUpper);
        blockMatrixTools::blockInsert(1, TsEqn.boundaryCoeffs()[patchI], coupledUpper);
    }
}
Remarks

- Approach currently works for processor and cyclic boundaries
  - See blockCoupledCyclicSwirlTest
  - AMG and GGI interfaces are on the cards
- Potential problems with any transform boundary
  - For block coupled problems the vector isn’t defined by components in N-dimensional space, i.e. no transformation
  - Think carefully before using VectorN if you want a transformation
  - Think carefully before using vector if you don’t want a transformation
- regionCouple – You’re on your own here since we cannot know how many variables you want coupled on each side of the interface
  - Requires regionCoupleFvPatchField to be derived off some base class and is problem dependent
  - Requires a templated version of coupledMatrix
  - However this can be, and has been, done
Coupled Interfaces
Time-dependent Simplified $P_3$ Equations

\[
\frac{1}{v} \frac{\partial F_0}{\partial t} - \nabla \cdot D_0 \nabla F_0 + \Sigma_r F_0 - \frac{2}{v} \frac{\partial F_1}{\partial t} - 2 \Sigma_r F_1 = S_0 - \frac{3}{\Delta t} \nabla \cdot \left( \frac{D_0}{v} \phi_1^0 \right)
\]
\[
3 \frac{1}{v} \frac{\partial F_1}{\partial t} - \nabla \cdot D_1 \nabla F_1 + \left( \frac{5}{3} \Sigma_{tr,2} + \frac{4}{3} \Sigma_r \right) F_1 - \frac{2}{3} \frac{1}{v} \frac{\partial F_0}{\partial t} - \frac{2}{3} \Sigma_r F_0 = - \frac{2}{3} S_0 - \frac{7}{3 \Delta t} \nabla \cdot \left( \frac{D_1}{v} \phi_3^0 \right)
\]

Marshak (vacuum) boundary

\[
\frac{1}{2} F_0 - \frac{3}{8} F_1 = D_0 \left( \nabla F_0 - \frac{3}{\nu \Delta t} \phi_1^0 \right) \cdot \vec{n}
\]
\[
- \frac{1}{8} F_0 + \frac{7}{8} F_1 = D_1 \left( \nabla F_1 - \frac{7}{3 \nu \Delta t} \phi_3^0 \right) \cdot \vec{n}
\]

***I. Clifford & K. Ivanov, “PBMR 400MW Benchmark Calculations Using the Simplified $P_3$ Approach”, HTR 2010***
Results

PBMR400 Nuclear Reactor

Scalar Neutron Flux

2nd Legendre Moment

Fast
Thermal

Fast
Thermal

***I. Clifford & K. Ivanov, “PBMR 400MW Benchmark Calculations Using the Simplified $\text{P}_3$ Approach”, HTR 2010
Closing Remarks

• The block-coupled matrix solver has been released as part of OpenFOAM-1.6-ext
  – Still under development but the core functionality is fairly well established

• To date used in limited number of projects
  – We want to remedy this
  – This is a excellent piece of code that has many potential uses
    • material stress analysis
    • implicit block-coupled density-based solver
    • multi-phase flows
    • chemically reactive flows

• Ultimate goal is to replace existing scalar matrix class with a flexible, efficient block-coupled solver

• Issues around parallelization and coupled boundaries still exist, although a workable solution is in place

• Dig in and get your hands dirty...