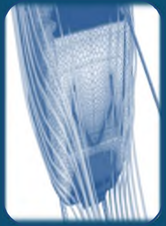
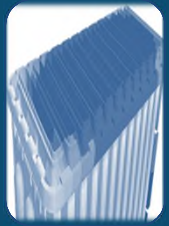
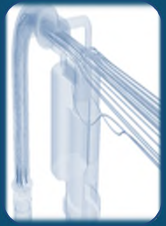
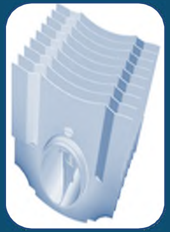
OpenFOAM®
in wastewater applications: *4 - Simulation Process*

nelson.marques@bluecape.com.pt

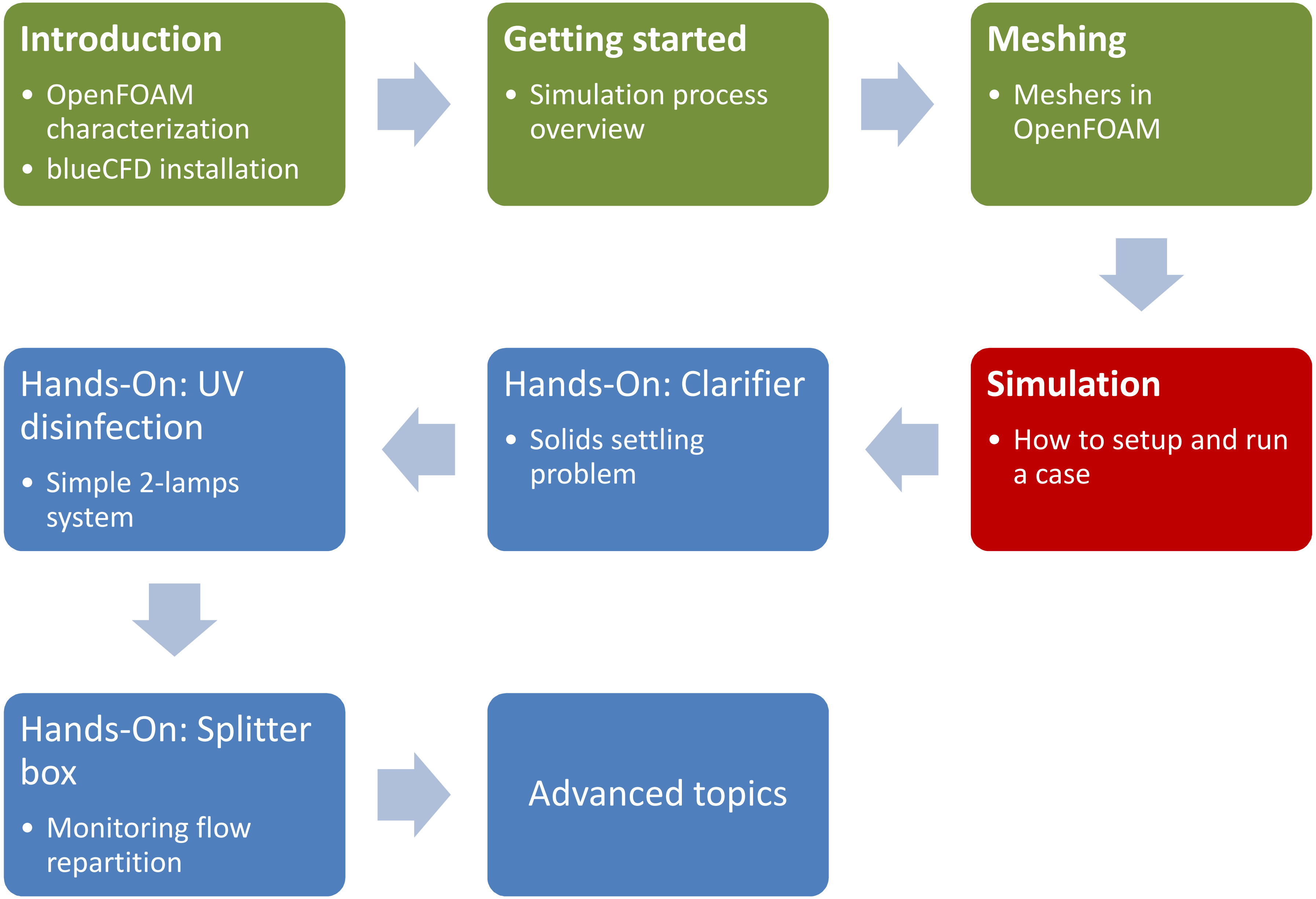
13-14th June 2015

bluecaPe

Computer Applications
in Science & Engineering

Progress



Introduction

- OpenFOAM characterization
- blueCFD installation

Getting started

- Simulation process overview

Meshing

- Meshers in OpenFOAM

Simulation

- How to setup and run a case

Hands-On: Clarifier

- Solids settling problem

Hands-On: UV disinfection

- Simple 2-lamps system

Hands-On: Splitter box

- Monitoring flow repartition

Advanced topics

Contents

Relevant solvers (1/3)

- OpenFOAM has a number of solvers available to users:

  http://www.openfoam.org/features/standard-solvers.php

- With OpenFOAM, users select solvers rather than models, as in commercial CFD codes.

- There are basic modelling functionalities which permeate several solvers at once. For example: solvers that allow for turbulence modelling generally allow users to select one model from the vast available range; the same is true for the specification of thermophysical properties.

- New solvers can be easily (relatively speaking) developed if necessary, since OpenFOAM is fundamentally a tool for solving partial differential equations.

Relevant solvers (2/3)

“Basic” CFD codes

- *laplacianFoam*
- *scalarTransportFoam*
- *potentialFoam*

Incompressible Flow

- *icoFoam*
- *nonNewtonianIcoFoam*

Compressible Flow

- *rhoPimpleFoam*
- *rhoSimpleFoam*

Multiphase flow

- *bubbleFoam*
- *interFoam*
- *MRFMultiphaseInterFoam*

Combustion

- *reactingFoam*

Particle-Tracking flows

- *UncoupledKinematicParcelFoam*
- *LTSReactingParcelFoam*

Heat transfer

- *buoyantBoussinesqPimpleFoam*
- *buoyantPimpleFoam*

Molecular dynamics

DNS

Electromagnetics

Stress analysis of solids

Finance

Relevant solvers (3/3)

Solver	Base model equations	Other physical models	Turbulence	Time domain	Fluid properties
interFoam	Navier-Stokes	VOF	All incompressible flow models	Transient	Constant
buoyantBoussinesqPimpleFoam	Navier-Stokes	Energy equation, with radiation	All incompressible flow models	Transient	
driftFluxFoam	Navier-Stokes			Transient	

Boundary conditions (1/22)

Under this topic, the following details will be addressed:

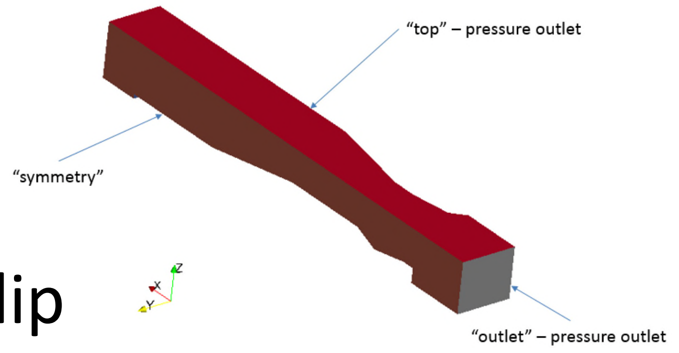
1. How to define boundary conditions for the fields at *t= 0s*
2. How to initialize the internal fields
3. How to set-up the turbulence models and initial values

The same example case from the *Getting Started* presentation will be used for addressing these topics.

The case overview and respective boundary conditions are quickly revised in the next slide:

Boundary conditions (2/22)

- Case name: halfParshall
- Boundary conditions:
  - Inlet: 375 kg/s
  - Bottom floor and side wall: no-slip
  - Outlet surfaces: pressure outlet
  - Symmetry plane surface: symmetry
- Fluid properties:
  - Water:
    - Density: 999 kg/m^3
    - Dynamic Viscosity: 1.15E-3 Pa.s
  - Air:
    - Density: 1.18 kg/m^3
    - Dynamic Viscosity: 1.855E-5 Pa.s



Boundary conditions (3/22)

In summary, the example case has:

- 6 field files, which are initially defined in the folder **0.org**:
  - **U** – velocity field
  - **p_rgh** – pressure field, without the hydrostatic term
  - **alpha.water** – phase fraction field
    - 1 = 100% water
    - 0 = 100% not water (air in our case)
  - **epsilon** – turbulent dissipation rate field
  - **k** – turbulent kinetic energy field
  - **nut** – turbulent dynamic viscosity field

(continues in the next slide…)

Boundary conditions (4/22)

- 4 major groups of boundary conditions:

  1. Inlet – assigned to the “inlet” surface
     - Velocity set to a predefined value.
     - Pressure set to zero gradient or fixed flux.
     - Phase fraction set to 1.
     - Turbulence fields *k* and *epsilon* set with appropriate values (addressed in the respective subtopic section).
     - *nut* set to *calculated*.

(continues in the next slide…)

Boundary conditions (5/22)

2. Outlet – assigned to the “outlet” and “top” surfaces
   - Velocity set to zero gradient and/or a condition that disables recirculation.
   - Pressure (*p_rgh)* set to 0 Pa.
   - Turbulence fields *k* and *epsilon* set to zero gradient and a condition that disables recirculation.
   - *nut* set to *calculated*.

(continues in the next slide…)

Boundary conditions (6/22)

3. Wall – assigned to the “backWall”, “bottomWall”, “sideWall” surfaces
   - Velocity set to no-slip, i.e. 0 m/s.
   - Pressure set to zero gradient or fixed flux.
   - Turbulence fields *k*, *epsilon* and *nut* set use wall treatments.

4. Symmetry – assigned to the “symmetry” surface
   - Boundary set to “symmetry” on all fields.

Boundary conditions (7/22)

Main parameters in a field file:

- *dimensions* – the units for the field:
    - Mass - kilogram
    - Length - metre
    - Time - second
    - Temperature - Kelvin
    - Quantity - mole
    - Current - ampere
    - Luminous intensity – candela
    - Example: [0 1 1 0 0 0 0] = m/s
- *internalField* – the value list for the internal field
- *boundaryField* – the list of boundary conditions

Boundary conditions (8/22)

For example, the *U* field file roughly looks like this:

```
dimensions       [0 1 -1 0 0 0 0];
internalField    uniform (0.0 0.0 0.0);
boundaryField
{
    backWall
    {
        type            fixedValue;
        value           uniform (0.0 0.0 0.0);
    }

    bottomWall
    {
        type            fixedValue;
        value           uniform (0.0 0.0 0.0);
    }

  …
}
```

Boundary conditions (9/22)

Figuring out what are the corresponding boundary conditions is usually done with the following strategies:

1. Looking into the tutorial cases that OpenFOAM has.
2. Checking the OpenFOAM User Guide, section “5.2 Boundaries”.
3. Looking at the complete list of boundary conditions, available in the Doxygen generated code documentation: www.openfoam.org/docs/cpp/
   - Near the bottom of the page are two links:
     - *Post-processing*
     - *Boundary Conditions*

Boundary conditions (10/22)

In the next slides are the boundary conditions used for our example case.

But first a small side note about *Regular Expressions*:

- These are search patterns that OpenFOAM supports in several dictionary files.
- For example:
    - “(backWall|bottomWall|sideWall)” → refers to the 3 patch names *backWall*, *bottomWall* and *sideWall*.
    - "procBoundary.*“ → refers to all patch names that start with “procBoundary”.

For more details: en.wikipedia.org/wiki/Regular_expression

Boundary conditions (11/22)

Inlet group (1/2):

U:

```
inlet
{
  type          flowRateInletVelocity;
  massFlowRate  375;
  rho           rho;
  rhoInlet      999.0;
}
```

alpha.water:

```
inlet
{
  type   fixedValue;
  value  uniform 1.0;
}
```

p_rgh:

```
inlet
{
  type   fixedFluxPressure;
  value  uniform 0.0;
}
```

nut:

```
inlet
{
  type   calculated;
  value  uniform 0.0;
}
```

Boundary conditions (12/22)

Inlet group (2/2):

epsilon:

```
inlet
{
  type              turbulentMixingLengthDissipationRateInlet;
  mixingLength      0.2;
  value             uniform 1.665138;
}
```

k:

```
inlet
{
  type              turbulentIntensityKineticEnergyInlet;
  intensity         0.5;
  value             uniform 3.778352;
}
```

Boundary conditions (13/22)

Outlet group:

U:

```
outlet
{
  type  pressureInletOutletVelocity;
  value uniform (0.0 0.0 0.0);
}
```

alpha.water:

```
outlet
{
  type       inletOutlet;
  inletValue uniform 0;
  value      uniform 0;
}
```

nut:

```
outlet
{
  type  calculated;
  value uniform 0.0;
}
```

p_rgh:

```
outlet
{
  type   fixedValue;
  value  uniform 0.0;
}
```

k, epsilon:

```
outlet
{
  type   zeroGradient;
}
```

Boundary conditions (14/22)

Wall group (1/2):

U:
```
"(backWall|bottomWall|sideWall)"
{
  type    fixedValue;
  value   uniform (0.0 0.0 0.0);
}
```

alpha.water:
```
"(backWall|bottomWall|sideWall)"
{
  type   zeroGradient;
}
```

p_rgh:
```
"(backWall|bottomWall|sideWall)"
{
  type   fixedFluxPressure;
  value uniform 0.0;
}
```

nut:
```
"(backWall|bottomWall|sideWall)"
{
  type    nutkWallFunction;
  value   uniform 0.0;
}
```

Boundary conditions (15/22)

Wall group (2/2):

epsilon:

```
"(backWall|bottomWall|sideWall)"
{
  type   epsilonWallFunction;
  value  uniform 1.665138;
}
```

k:

```
"(backWall|bottomWall|sideWall)"
{
  type   kqRWallFunction;
  value  uniform 3.778352;
}
```

Boundary conditions (16/22)

Symmetry group, for all 6 fields:

```
symmetry
{
  type symmetry;
}
```

Special group, interfaces between domains, in all 6 fields:

```
"procBoundary.*"
{
  type                processor;
  value               uniform init_value_or_vector;
}
```

Boundary conditions (17/22)

Manipulation of fields and domains (1/3):

We haven’t used this in our example case, but the idea is simple:

> What if we need to initialize a part of the internal field and/or fixed value patches with a value specific only to a group of cells or faces?

This is where **setFields** comes into play. This utility will use the settings given in the dictionary file “system/setFieldsDict”, for assigning values to each desired field.

Example in the next couple of slides.

Boundary conditions (18/22)

Manipulation of fields and domains (2/3):

```
defaultFieldValues
(
  volScalarFieldValue alpha.water 0
);

regions
(
  // Set cell values
  // (does zerogradient on boundaries)
  boxToCell
  {
    box (-2.0 -2.0 -1) (11.0 1.0 0.2);
    fieldValues
    (
      volScalarFieldValue alpha.water 1
    );
  }

…
```

Boundary conditions (19/22)

Manipulation of fields and domains (3/3):

```
…

  // Set patch values (using ==)
  boxToFace
  {
    box (-2.0 -2.0 -1) (11.0 1.0 0.2);
    fieldValues
    (
      volScalarFieldValue alpha.water 1
    );
  }
);
```

Note: The selection box is meant to include the cell centres and/or face centres, for selecting the respective cells and faces.

Boundary conditions (20/22)

Turbulence Models (1/3):

Three categories of files have to be taken into account:

- “constant/turbulenceProperties” – for defining which major group of turbulence modelling to be used. Not all solvers need this file, since those will only use one group.
- “constant/RASProperties” – this file has the options and settings for the RAS group.
- “constant/LESProperties” – this file has the options and settings for the LES group.
- In the time folders, we then have the fields associated to the turbulence model we want to use.
    - Example in our “0.org” folder are: *k*, *epsilon* and *nut*

Boundary conditions (21/22)

Turbulence Models (2/3):
Content of “constant/turbulenceProperties”:

```
simulationType  RASModel;
```

Content of “constant/RASProperties”:

```
RASModel                kEpsilon;
turbulence              on;
printCoeffs             on;
```

Want laminar flow modelling?

```
RASModel                laminar;
turbulence              off;
printCoeffs             off;
```

Boundary conditions (22/22)

Turbulence Models (3/3):
The last critical detail for turbulence models is:

> What initial values should we use and what values at the inlets?

This depends on your simulation, but a few guidelines exist online, e.g.:

- http://www.cfd-online.com/Wiki/Turbulence_free-stream_boundary_conditions
- http://support.esi-cfd.com/esi-users/turb_parameters/

But the bottom line is that you will have to test which values seem more well suited to your simulation.

Convective term discretization (1/4)

Convective term refers to the divergence operator $\nabla\cdot$

The OpenFOAM Programmer's Guide goes into more details about this in the subsection “2.4.2 The convection term”, in which the following expression can be found:

$$\int_V \nabla \bullet (\rho \mathbf{U} \phi)\, dV = \int_S d\mathbf{S} \bullet (\rho \mathbf{U} \phi) = \sum_f \mathbf{S}_f \bullet (\rho \mathbf{U})_f \phi_f = \sum_f F \phi_f$$

which shows how the convection term is integrated and linearized.

We will be addressing on this topic how we can control this term.

Convective term discretization (2/4)

The settings for the divergence schemes in the file “system/fvSchemes”, namely in the block “divSchemes”.

From our example case:

```
divSchemes
{
    default               none;
    div(rhoPhi,U)         Gauss upwind;
    div(phi,alpha)        Gauss upwind;
    div(phirb,alpha)      Gauss upwind;
    div(phi,k)            Gauss upwind;
    div(phi,epsilon)      Gauss upwind;
    div((muEff*dev(T(grad(U))))) Gauss linear;
}
```

This is currently defined to be mostly of first order discretization, i.e. *upwind*.

Convective term discretization (3/4)

How do we know which terms we need?

We either:

1. Let the solver complain when they are not present.
2. Or we look at the source code. For example:

```
cat $FOAM_SOLVERS/incompressible/simpleFoam/UEqn.H
```

we can see this:

```
// Momentum predictor
tmp<fvVectorMatrix> UEqn
(
    fvm::div(phi, U)
  + turbulence->divDevReff(U)
  ==
    fvOptions(U)
);
```

Refers to this entry:

`div(phi,U)`

Convective term discretization (4/4)

How can we increase accuracy in our example case?

- Use an intermediate scheme between *linear* and *upwind*:

```
div(rhoPhi,U)   Gauss linearUpwind grad(U);
```

- Special limiter, great for VOF-related fields:

```
div(phi,alpha)  Gauss vanLeer;
```

- Second order scheme:

```
div(phirb,alpha) Gauss linear;
```

- Problems with running in parallel with *linearUpwind*?

```
div(phi,epsilon)  Gauss linearUpwind limitedGrad;
```

  along with the gradSchemes block having this entry:

```
limitedGrad     cellLimited Gauss linear 1;
```

Diffusive term discretization (1/4)

Diffusive term refers to the Laplace operator ∇^2

The OpenFOAM Programmer’s Guide goes into more details about this in the subsection “2.4.1 The Laplacian term”, in which the following expression can be found:

$$\int_V \nabla \bullet (\Gamma \nabla \phi)\, dV = \int_S d\mathbf{S} \bullet (\Gamma \nabla \phi) = \sum_f \Gamma_f \mathbf{S}_f \bullet (\nabla \phi)_f$$

which shows how the diffusion term is integrated and linearized.

We will be addressing on this topic how we can control this term.

Diffusive term discretization (2/4)

The settings for the Laplacian schemes in the file “system/fvSchemes”, in the block “laplacianSchemes”.

From our example case:

```
laplacianSchemes
{
    default             Gauss linear corrected;
}
```

Meaning:

- The same setting is used for all Laplacian terms.
- Second order or above should always be used.
- The *corrected* option is actually the surface normal gradient scheme to be used.

Diffusive term discretization (3/4)

Other examples from OpenFOAM’s tutorials:

```
Gauss linear limited corrected 0.5;
Gauss linear limited corrected 0.333;
```

These usually depend on how orthogonal or non-orthogonal the mesh cells are.

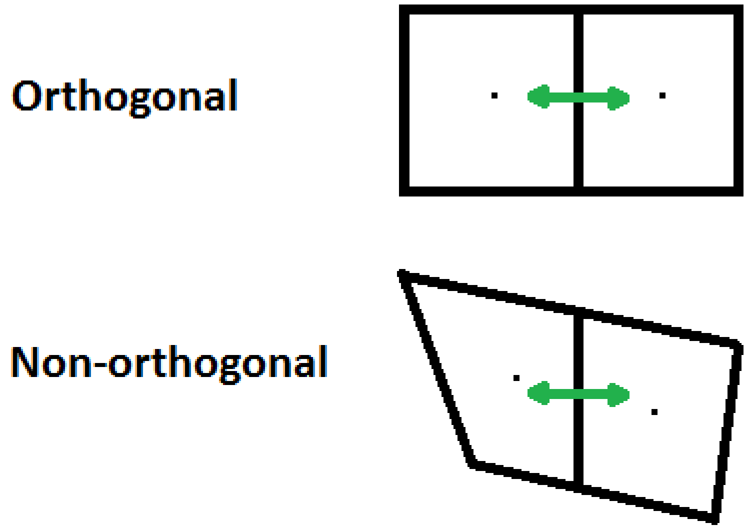
```
Gauss linear orthogonal;    ≡   limited corrected 0.5
Gauss linear corrected;     ≡   limited corrected 1
Gauss linear uncorrected;   ≡   limited corrected 0
```

Diffusive term discretization (4/4)

Since the Laplacian schemes depend on the surface normal gradient discretization (block “snGradSchemes”).

Therefore we have to take into account the mesh we are using, i.e. how the normal of the face is related to the centres of the cells that share this face.

Specifically, an orthogonal mesh has cell centres aligned with the centres and normals of the faces shared by those cells.



Linear system solvers (1/35)

The end result of the discretization process are linear systems of equations:

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$$

Where:

- the bold forms designate tensor quantities,
- uppercase letters stand for matrices,
- lowercase for vectors.

These equations will appear for every conservation law and at every outer iteration. Depending on the cases, their approximate solution can easily reach 75% of the overall CPU time.

Linear system solvers (2/35)

Therefore, let us start with a few basic concepts.

For example, in order to configure the solver entry for each named block, we need to assess the type of equation that will be solved, from the point of view of solving the equation when it is reduced to the matrix form $\boldsymbol{Ax} = \boldsymbol{b}$, where:

- $\boldsymbol{A}$ is the coefficient matrix that correlates the values between the centres, i.e., our unknowns;
- $\boldsymbol{x}$ is the vector that represents the values at the cell centres for which we are solving the linear system;
- $\boldsymbol{b}$ keeps the source terms for each respective cell.

Linear system solvers (3/35)

As a result of this construct, the following types of equations will exist and the respective matrix solvers should be used:

- The equation for the pressure field, i.e., continuity equation, gives rise to a symmetric matrix, hence it should use solvers devised for this type of matrices.
- All other equations give rise to a usually asymmetric matrix due to convection, which is why we can not use solvers that are meant for symmetric matrix equations. Otherwise their solution will not converge.

Linear system solvers (4/35)

Symmetric $\boldsymbol{A}$ matrix

$$\begin{bmatrix} i & kk & 0 & \cdots & 0 & 0 & 0 \\ kk & j & hh & \cdots & 0 & 0 & 0 \\ 0 & hh & k & \ddots & & 0 & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots & \vdots \\ 0 & 0 & & \ddots & \ddots & gg & 0 \\ 0 & 0 & 0 & \cdots & gg & r & ll \\ 0 & 0 & 0 & \cdots & 0 & ll & s \end{bmatrix}$$

Asymmetric $\boldsymbol{A}$ matrix

$$\begin{bmatrix} i & hj & qw & \cdots & 0 & 0 & 0 \\ kz & j & kq & \cdots & 0 & 0 & 0 \\ we & le & k & \ddots & & 0 & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots & \vdots \\ 0 & 0 & & \ddots & \ddots & ae & 0 \\ 0 & 0 & 0 & \cdots & ax & r & qu \\ 0 & 0 & 0 & \cdots & 0 & lw & s \end{bmatrix}$$

Linear system solvers (5/35)

The file “system/fvSolution” must be used to select whatever method is found appropriate to solve.

Here are a few expressions that we will be using:

- **outer iteration** – this usually refers to one step in time, if transient, or to a sweep of all transport equations which are being solved.
- **matrix solver iteration** – In the next slide you will see at the end of each line the information “No Iterations”, which refers to the number of iterations it took to solve the respective linear system of equations.

Note: More details are available in section "4.5 Solution and algorithm control" in the OpenFOAM User Guide

Linear system solvers (6/35)

This is an example output for an **outer iteration**:

```
Time = 2

DILUPBiCG:  Solving for Ux, Initial residual = 0.33783062, Final residual = 0.022319355, No Iterations 2
DILUPBiCG:  Solving for Uy, Initial residual = 0.16839243, Final residual = 0.0037979544, No Iterations 2
DILUPBiCG:  Solving for Uz, Initial residual = 0.17283387, Final residual = 0.016074394, No Iterations 1
DILUPBiCG:  Solving for h, Initial residual = 0.97900298, Final residual = 0.020167345, No Iterations 1
GAMG:  Solving for p, Initial residual = 0.90628728, Final residual = 0.0064221651, No Iterations 12
time step continuity errors : sum local = 0.00046531931, global = 1.4428116e-006, cumulative = -3.528428e-006
rho max/min : 1.1274839 0.72937754
DILUPBiCG:  Solving for epsilon, Initial residual = 0.99931859, Final residual = 4.869335e-005, No Iterations 1
DILUPBiCG:  Solving for k, Initial residual = 0.82482828, Final residual = 9.5040752e-006, No Iterations 1
ExecutionTime = 2.509 s  ClockTime = 2 s
```

Zooming in on one equation:

```
DILUPBiCG:  Solving for Ux, Initial residual = 0.33783062,
Final residual = 0.022319355, No Iterations 2
```

Linear system solvers (7/35)

system/fvSolution:
This dictionary file was designed to handle the settings for the linear equation solvers and the algorithms to be used by a solver application, e.g. **interFoam**.

Starting with the linear equation solvers, these are configured inside this block list:

```
solvers
{
   …
}
```

It’s in here that we will be configuring the matrix solvers.

Linear system solvers (8/35)

Starting with our example case, for configuring the linear equation solvers for the fields named “alpha.water”, we are using the following settings:

```
"alpha.water.*"
{
  nAlphaCorr      2;
  nAlphaSubCycles 1;
  cAlpha          1;

  MULESCorr       yes;
  nLimiterIter    3;

  solver          smoothSolver;
  smoother        symGaussSeidel;
  tolerance       1e-8;
  relTol          0;
}
```

$\}$ Specific for the linear equations related to the phase fraction equations

Linear system solvers (9/35)

Keep in mind that each solver has its own settings, where the first tier of possible options for a matrix solver refers to one of two major possible types of settings:

- **preconditioner** is needed for solvers that rely on a preconditioning strategy to speed up their iterative process.
  - For more details on what preconditioning is, see http://en.wikipedia.org/wiki/Preconditioner.
- **smoother** is designed to smooth-out numerical issues that usually arise from ill-formed matrices and strongly uneven intermediate solutions for the matrix equation.

More details available at:
http://www.tfd.chalmers.se/~hani/kurser/OS_CFD_2008/TimBehrens/tibeh-report-fin.pdf

Linear system solvers (10/35)

There are 3 other parameters that are common to most of the matrix solvers. Let us look at an example output from an outer iteration:

```
DILUPBiCG:  Solving for Ux, Initial residual = 0.33783062,
Final residual = 0.022319355, No Iterations 2
```

The residual is essentially the result from this equation:

$$residual = sum(abs(\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}))$$

The reported residual values are normalized values from this equation, in order to keep values between 0.0 and 1.0 for an easier interpretation of how *good or bad* the residuals are.

Linear system solvers (11/35)

The 3 major parameters we need to control:

- **tolerance** – this is the minimum residual value we want to achieve at the end of the iterations of the matrix solver. I.e. if the *Final Residual* falls below this value, the matrix solver stops iterating. Default value is 1e-6.
- **relTol** – this relative tolerance refers to whether the residual for the current iteration is lesser than **relTol** times the *Initial Residual*. Default value is 0.0 (= off).
- **maxIter** –maximum number of iterations for the matrix solver to perform, regardless of the convergence status. Anti-infinite loop counter-measure. Default value is 1000.

Linear system solvers (12/35)

One tolerance will suffice to stop the matrix solver:

- In order to only define the relative tolerance:

```
    tolerance         0.0;
    relTol            0.01;
```

- In order to only define the (absolute) tolerance:

```
    tolerance         1e-06;
    relTol            0.0;
```

- In order to allow the maximum number of iterations to be reached:

```
    tolerance         0.0;
    relTol            0.0;
```

Linear system solvers (13/35)

Which values should you use?

It all depends on:

- the problem you are solving;
- how accurate you want it to be, while weighing:
    - it is usually never possible to reach the exact solution for a matrix equation…
    - and even if it is, the solution might be useless if an outer iteration is still needed for balancing the results over all equations.

Therefore, this is usually something that can be adjusted after reaching good solutions for your cases.

Linear system solvers (14/35)

A few good reference values are as follows:

- For the pressure field, make sure you have a tighter control, such as:

```
tolerance          1e-06;
relTol             0.001;
maxIter            250;
```

- For all other equations, you can loosen up a bit the control, since most other equations will be affected by the pressure field in the next major iteration:

```
tolerance          1e-05;
relTol             0.01;
maxIter            100;
```

Linear system solvers (15/35)

Which matrix solvers should we use and in which situations? The answer strongly depends:

- on the simulation being performed;
- on whether the run in serial or in parallel.

Therefore, don't assume that there is a fool proof way of selecting the solvers and respective preconditioners or smoothers.

Nonetheless, it is possible to present some guidelines:

1. When in doubt, use the values in the tutorials which are the most similar to your problem.

(continues…)

Linear system solvers (16/35)

2. If you need the matrix solving steps to be as fast/efficient as possible, a good combination is to use:
   - For the pressure equations, use “GAMG”. Example:

```
p
{
        solver                GAMG;
        smoother              GaussSeidel;
        cacheAgglomeration true;
        nCellsInCoarsestLevel 10;
        agglomerator        faceAreaPair;
        mergeLevels           1;
        tolerance             1e-06;
        relTol                0.001;
        maxIter               250;
};
```

(continues…)

Linear system solvers (17/35)

- GAMG can be somewhere 2 and 5 times faster than using PCG+DIC.
- There is a downside to using GAMG: it is only efficient enough if you properly calibrate its parameters. For example:
  - "nCellsInCoarsestLevel" can depend on the number of cells your case has, but make sure to test the values for a few major iterations first, before using the value in a real simulation scenario.
- The PBiCG solver is usually more efficient for running in parallel for all other equations (non-pressure). Although in some cases, GAMG is faster than PBiCG.

Linear system solvers (18/35)

3. The PCG solver can in many cases be better than GAMG for the pressure equations, therefore, you should always double-check which one is best for your simulation.
4. In some cases, the FDIC preconditioner may prove to be more efficient than DIC and give results faster (symmetric matrices only).
5. GAMG can also be used as a preconditioner for working cooperatively with a matrix solver, but isn't very common. But again, it can be and should be tested for your own cases.

Linear system solvers (19/35)

6. A choice of smoother (specific matrix solvers only), may depend strongly on your case. A few examples:
   - GaussSeidel is commonly used in conjunction with GAMG, since it can offer a direct resolution for each major block.
   - DIC and DILU can also be used as smoothers, but they can prove to be more efficient if used in conjunction with GaussSeidel, namely by using the variants DICGaussSeidel and DILUGaussSeidel.

Linear system solvers (20/35)

7. “smoothSolver” as a matrix solver can prove to be more efficient, if a preconditioner has issues due to numerical spikes. A smoother will instead try to solve the equation directly, while *sort-of not fretting over imperfections* in the achieved solutions. Configuration example:

```
U
{
    solver              smoothSolver;
    smoother            GaussSeidel;
    tolerance           1e-8;
    relTol              0.1;
    nSweeps             1;
}
```

Linear system solvers (21/35)

Continuing with the settings we have on our example case:

```
pcorr
{
  solver              GAMG;
  tolerance           1e-5;
  relTol              0.001;
  smoother            GaussSeidel;
  nPreSweeps          0;
  nPostSweeps         2;
  cacheAgglomeration on;
  agglomerator        faceAreaPair;
  nCellsInCoarsestLevel 10;
  mergeLevels         1;
}
```

```
p_rgh
{
  $pcorr;
  tolerance         1e-07;
  relTol            0.05;
}

p_rghFinal
{
  $p_rgh;
  relTol            0;
}
```

Linear system solvers (22/35)

And for all of the other fields:

```
"(U|k|epsilon).*"
{
  solver          smoothSolver;
  smoother        symGaussSeidel;
  tolerance       1e-06;
  relTol          0;
  minIter         1;
}
```

Linear system solvers (23/35)

Algorithm configurations

The most common algorithms implemented in OpenFOAM:

- PISO – Pressure-Implicit Split-Operator
- SIMPLE – Semi-Implicit Method for Pressure-Linked Equations
- PIMPLE – it’s a PISO-SIMPLE hybrid

In our case example, we use with **interFoam** these settings:

```
PIMPLE
{
  momentumPredictor   no;
  nOuterCorrectors    1;
  nCorrectors         3;
  nNonOrthogonalCorrectors 1;
}
```

Linear system solvers (24/35)

In general, the algorithms have one or more of the following parameters:

```
nOuterCorrectors    0;
nCorrectors         0;
nNonOrthogonalCorrectors 0;
turbOnFinalIterOnly   no;
momentumPredictor yes;
transonic          no;
residualControl
{
    p                1e-2;
    U                1e-3;
    "(k|epsilon|omega)" 1e-3;
}
pRefCell        0;
pRefPoint       (0  0  0);
pRefValue       0;
```

Linear system solvers (25/35)

Description for each parameter:

- **nOuterCorrectors** – number of iterations that should be used for the external loop of the algorithm (not to be confused with "outer loop iteration" we've been referring to for the time step).
- **nCorrectors** – number of iterations that should be used for the internal loop of the algorithm.
- **nNonOrthogonalCorrectors** – number of iterations for attempting to correct the effect that non-orthogonal cells have on the solution of the problem.
  - 0 for a fully orthogonal mesh;
  - 20 iterations for the most non-orthogonal meshes;
  - 1-3 iterations if there are a few non-orthogonal cells.

Linear system solvers (26/35)

- **turbOnFinalIterOnly** – this flag allows us to postpone the calculation of the turbulence fields to the last iteration.
- **momentumPredictor** – Not all solvers use this parameter. Those that do support this parameter, will not solve the momentum equation (*U*) if this parameter is set to “no”.
- **transonic** – Only solvers that have implementations for sonic flow will support this flag.
- **residualControl** - This is a named block gives the ability to add an additional stopping criteria for the *Initial Residual* of each one of the listed equations.

Linear system solvers (27/35)

Side note

The following three parameters fall in a new topic: having a location in the domain with a fixed reference pressure value.

This is necessary whenever all of pressure boundary conditions do not have a fixed value, i.e. if they are all defined as zero gradient or a similar boundary condition.

For such a situation, we must avoid having an incomplete definition of the pressure equation, therefore we can rely on the following parameters for defining a specific location in the mesh that has a fixed and pre-defined value of pressure.

Linear system solvers (28/35)

- **pRefCell** or **pRefPoint** - For selecting a location in the mesh where the cell centre is used for pressure reference. Keep in mind that:
    - **pRefCell** refers to the cell ID on the mesh.
    - **pRefPoint** ensures us that the provided position is used for selecting the cell.
    - Note: **pRefCell** takes precedence over **pRefPoint**.
- **pRefValue** - This is the pressure value, which should be defined with the same units as the pressure fields.
- Last but not least, these 3 could have the prefix **p_rghRef** instead of **pRef**, if the solver accounts for gravity.

Linear system solvers (29/35)

Almost complete example for SIMPLE:

```
SIMPLE
{
    nNonOrthogonalCorrectors 0;

    momentumPredictor yes;
    transonic         no;

    residualControl
    {
        p               1e-2;
        U               1e-3;
        "(k|epsilon|omega)" 1e-3;
    }

    pRefCell        0;
    pRefPoint       (0 0 0);
    pRefValue       0;
}
```

$\left.\begin{array}{l}\\ \\ \end{array}\right\}$ Depends on the solver

Linear system solvers (30/35)

Almost complete example for PISO:

```
PISO
{
    nCorrectors              2;
    nNonOrthogonalCorrectors 0;

    momentumPredictor yes;

    pRefCell          0;
    pRefPoint         (0 0 0);
    pRefValue         0;
}
```

$\}$ Depends on the solver

Linear system solvers (31/35)

Almost complete example for PIMPLE:

```
PIMPLE
{
    nOuterCorrectors    1;
    nCorrectors         2;
    nNonOrthogonalCorrectors 0;
    turbOnFinalIterOnly   no;
    momentumPredictor yes;
    transonic           no;
    residualControl
    {
        p               1e-2;
        U               1e-3;
        "(k|epsilon|omega)" 1e-3;
    }

    pRefCell        0;
    pRefPoint       (0 0 0);
    pRefValue       0;
}
```

$\Big\}$ Depends on the solver (turbOnFinalIterOnly, momentumPredictor, transonic)

Linear system solvers (32/35)

Relaxation factors

Technically, OpenFOAM uses under-relaxation factors, because the values are between 0.0 and 1.0.

These help the outer iterative convergence process, making it more robust while simultaneously increasing the likelihood that we can reach a numerical solution for our problem. A poor choice of values can delay and even prevent convergence.

This is further explained in the OpenFOAM User Guide, subsection “4.5.2 Solution under-relaxation.

Linear system solvers (33/35)

1. If the value 0.0 is given, then the solution is kept unchanged between each outer iteration.
2. If 1.0 is given, then the under-relaxation does not take place at all.
3. As we reduce the factor from 1.0 towards 0.0, we increase the impact of the under-relaxation. Examples:
   a) 0.9 – 90% of current solution of outer iteration is preserved and 10% previous outer iteration.
   b) 0.1 – the current solution has a very small impact in the final solution after this relaxation step, making the flow results to evolve slower with every outer iteration.

Linear system solvers (34/35)

Relaxation factors: transient vs steady-state

- In most cases, the relaxation factors for transient simulations are either simply set to 1.0 or not at all.
- This is because PISO and PIMPLE algorithms perform time accurate simulations, where the time step essentially does what the relaxation factor is used for in steady-state simulations.
- Nonetheless, PIMPLE is a hybrid algorithm, therefore, it can also rely on the relaxation factors for the SIMPLE loop within the PIMPLE algorithm. Therefore, those relaxation factors are still applicable.

Linear system solvers (35/35)

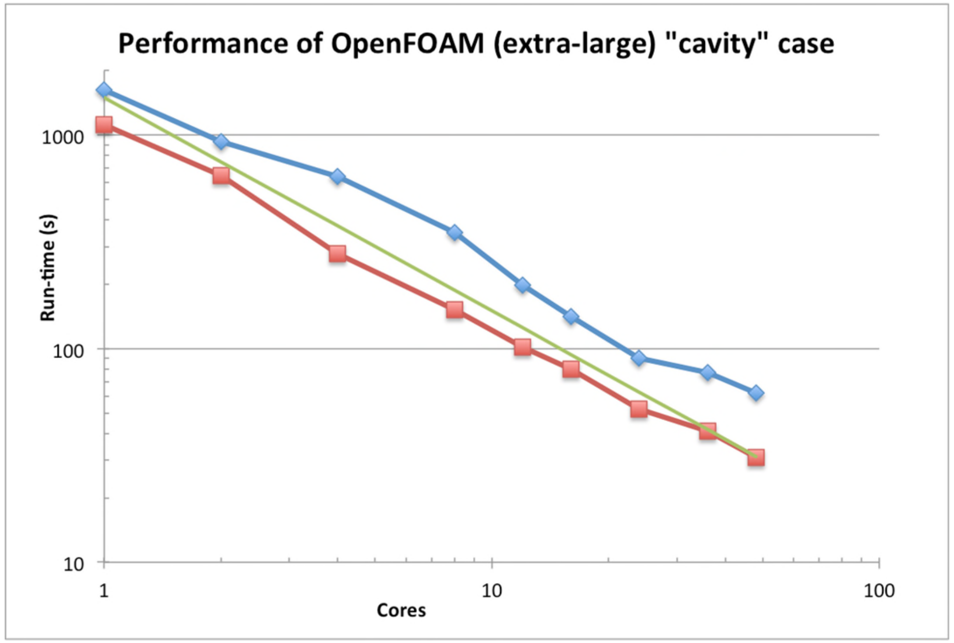
From our example case it, doesn’t need much for **interFoam** (PIMPLE):

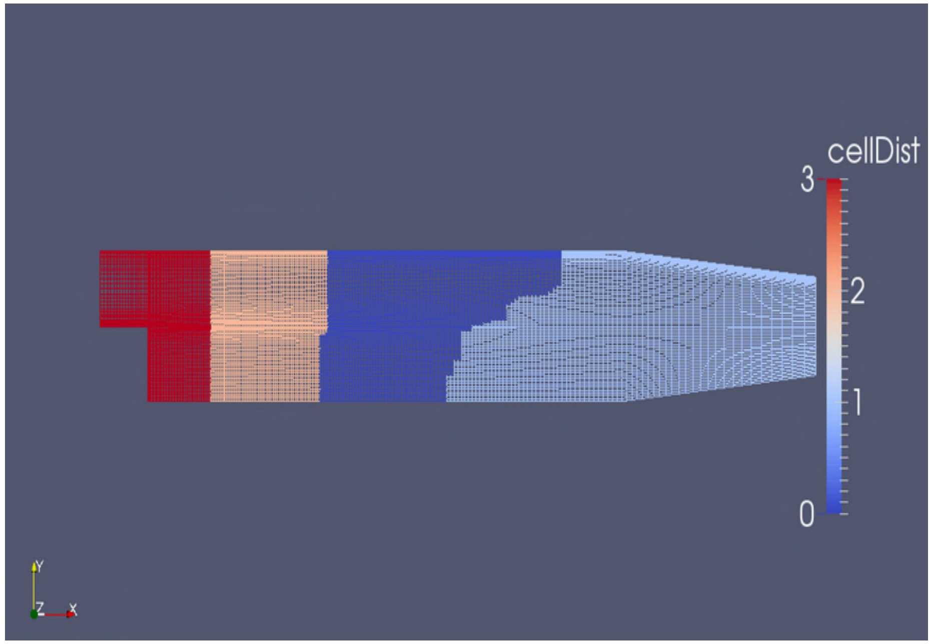
```
relaxationFactors
{
  fields
  {
  }
  equations
  {
    ".*" 1;
  }
}
```

When compared to the tutorial “incompressible/ simpleFoam/motorBike”:

```
relaxationFactors
{
  fields
  {
    p                0.3;
  }
  equations
  {
    U                0.7;
    k                0.7;
    omega            0.7;
  }
}
```

Parallel runs (1/9)





We will address the following topics:

1. Domain decomposition
2. Domain balancing
3. Running in parallel
4. Domain reconstruction

Parallel runs (2/9)

Domain decomposition (1/2)

Application: **decomposePar**

Dictionary: **system/decomposeParDict**

Relevant parameters:

- **numberOfSubdomains** is the number for sub-domains, i.e. how many processors for running in parallel.
- **method** is for choosing the algorithm for decomposing the domain. The easiest to use is the **scotch** option.
- **scotchCoeffs** is the block relative to the method **scotch**, which doesn't even need be present, since its internal parameters are for advanced users.

Parallel runs (3/9)

Domain decomposition (2/2)

Example of “system/decomposeParDict”:

```
numberOfSubdomains 4;
```

```
method               hierarchical;

hierarchicalCoeffs
{
    n                (1 2 1);
    delta            0.001;
    order            xyz;
}
```

```
method                   scotch;

scotchCoeffs
{
    //writeGraph  true;
    //strategy "b";
}
```

Example uses:

```
    decomposePar
    decomposePar -help
    decomposePar -force
    decomposePar -cellDist
    decomposePar -noZero -fields -time 10
```

Parallel runs (4/9)

Domain balancing

There are essentially two types of domain balancing:

1. Complete sub-domain redistribution, by using **redistributePar**, which can help balance the number of cells per processor.
   - Requires “system/decomposeParDict”.
   - Can run in parallel.
2. Reordering the connections between cells, by using **renumberMesh**, which will improve the configuration of the equations in matrix form, for an optimum memory access.
   - Can run in parallel.

Parallel runs (5/9)

Running in parallel (1/4)

The common denominator is that the “-parallel” option must be used. For example, if we run this command:

```
simpleFoam -help
```

We will see this line:

```
-parallel         run in parallel
```

Therefore, for running in parallel, the simplest command would be:

```
mpirun -np 2 simpleFoam -parallel
```

where the “-np” means that the number on the right is the number of processors to be used, i.e. 2.

Parallel runs (6/9)

Running in parallel (2/4)

The use of *mpirun* is not a standard on all platforms, e.g. clusters can use dedicated job schedulers and use dedicated scripts.

OpenFOAM has another two ways for running in parallel:

- **foamJob** is a script that comes in handy for running any utility and it has the ability to either run in serial or in parallel.
- **runParallel** is a function-script that is accessible only when we source the script **RunFunctions**. This is why this function is only seen inside the **Allrun** scripts that are present in OpenFOAM's tutorials.

Parallel runs (7/9)

Running in parallel (3/4) – Examples for **foamJob**:

Run in parallel as a background job:

```
foamJob -p simpleFoam
```

Run in parallel and show on-screen the output:

```
foamJob -p -s simpleFoam
```

For more details:

```
foamJob -help
```

Note: Application output is saved into the file named “log”.

Parallel runs (8/9)

Running in parallel (4/4) – Details for **runParallel**:

Will only work one this command is used (or similar) :

```
source $WM_PROJECT_DIR/bin/tools/RunFunctions
```

commonly found in the **Allrun** scripts.

Usage structure:

```
runParallel app_name number_of_cores app_arguments
```

Example:

```
runParallel snappyHexMesh 4 -overwrite
```

Parallel runs (9/9)

Domain reconstruction

As mentioned before, there are two types of domain reconstruction:

1. When the mesh was generated in parallel, we need to reconstruct it with **reconstructParMesh**.
2. When the mesh is the same before and after decomposing/reconstructing, then **reconstructPar** should be used for reconstructing the time snapshots.

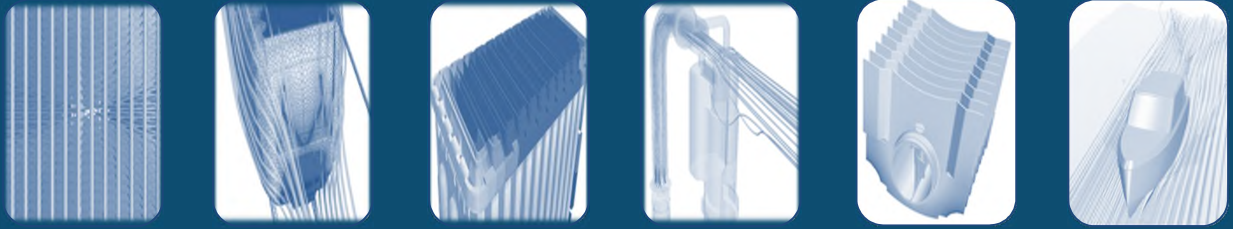
Both can only be executed in serial mode (not in parallel).

More details with the “-help” option, e.g.:

```
reconstructPar -help
```

Thank you for your time.

Next:
5 – Hands-On: Clarifier



bluecaPe
Computer Applications
in Science & Engineering