Instructional workshop on OpenFOAM programming
LECTURE # 8

Pavanakumar Mohanamuraly

April 30, 2014

Outline

The Euler's Equation

$$\frac{\partial}{\partial t}\int_V \mathbf{Q}dV + \int_S \mathbf{F}\cdot\mathbf{n}\ dS = 0 \tag{1}$$

The vectors $\mathbf{Q}$ and $\mathbf{F}\cdot\mathbf{n}$ are defined as,

$$\mathbf{Q} = \begin{bmatrix} \rho \\ \rho v_i \\ \rho e \end{bmatrix} \qquad \mathbf{F}\cdot\mathbf{n} = \begin{bmatrix} \rho v_n \\ \rho v_i v_n + p n_i \\ (\rho e + p) v_n \end{bmatrix} \tag{2}$$

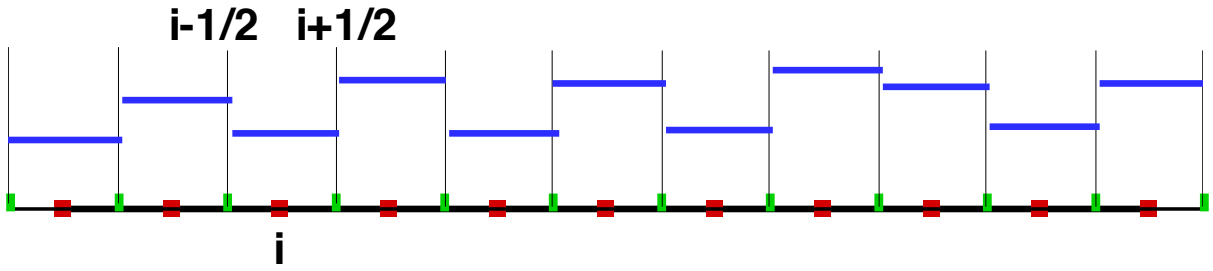
where, ρ, p and v are the fluid density, pressure and velocity respectively. The internal energy e per unit mass of the fluid is defines as,

$$\rho e = \frac{p}{\gamma - 1} + \frac{1}{2}\rho v_i v_i \tag{3}$$

The fluid pressure p, density ρ and temperature T are related by the prefect gas relation,

$$p = \rho R T \tag{4}$$

Spatial and Temporal discretization



For $3d$ meshes we get,

$$Q_i^{n+1} = Q_i^n + \frac{\Delta t}{V_i} \sum_j \left(F_j^n \; S_j \right) \tag{5}$$

- Q_i is the cell average value $Q_i = \frac{1}{V_i} \int\limits_{V_i} Q dV$
- F_j^n is approximation to average flux along face j

$$F_j^n = \mathcal{F} \left(Q_{L_j}^n, Q_{R_j}^n \right) \tag{6}$$

- Q_{L_j/R_j} = left/right states of face j

Flux function $\mathcal{F}$

- Run-time selectable using function pointers
- Selected using dictionary keyword *fluxScheme* in *system/fvSchemes*
- Implemented Roe approximate flux function

$$\hat{\mathcal{F}}_{roe}(Q_L, Q_R, S) = \frac{1}{2}\left[\hat{F}_n(Q_L, S) + \hat{F}_n(Q_R, S) - \|\tilde{A}(Q_L, Q_R, S)\|(Q_R - Q_L)\right] \tag{7}$$

  where, $\tilde{A}$ is the Roe average matrix
- Also has implementation of Van Leer flux splitting

Local time stepping

$\frac{\Delta t}{v_{cell}}$ term is replaced by the CFL number (N_c) and local maximum eigenvalue

$$\frac{\Delta t}{V_i} = \frac{N_c}{\sum_{j=1}^{N_f} \left(|U_n| + a\right)_j S_j} \tag{8}$$

where, N_f is the total number of faces in a cell and S_j is the face area of the j^{th} face.

Boundary Conditions - Slip wall

- Physically imposes a zero mass flux crossing the rigid wall
- Written mathematically as $u_n = 0$
- Flux formulation for the wall boundary face becomes

$$F_n^{wall} = \begin{bmatrix} \rho u_n \\ \rho u u_n + p n_x \\ \rho v u_n + p n_y \\ \rho w u_n + p n_z \\ (e + p) u_n \end{bmatrix} = \begin{bmatrix} 0 \\ p n_x \\ p n_y \\ p n_z \\ 0 \end{bmatrix} \tag{9}$$

- Pressure at wall face computed by extrapolation from interior
- Use face-owner cell centroid value extrapolation

Boundary Conditions - Supersonic Inflow/Outflow

- No influence of the downwind disturbances to upwind
- Safe to fix inlet face values to inlet conditions
- For outflow use extrapolated interior solution (owner cell value) to boundary face

Implementation details - Flux divergence

```
/// Internal faces get face flux from L and R cells
forAll( mesh.owner() , iface ) {
 /// Get the left and right cell index
 const label& leftCell = mesh.owner()[iface];
 const label& rightCell = mesh.neighbour()[iface];
 /// Approximate Riemann solver at interface
 scalar lambda = (*fluxSolver)
 (
  &rho[leftCell], &U[leftCell], &p[leftCell], // L
  &rho[rightCell], &U[rightCell], &p[rightCell], // R
  &massFlux[iface], &momFlux[iface], &energyFlux[iface],
  &nf[iface] // Unit normal vector
 );
 /// Multiply with face area to get face flux
 massFlux[iface] *= mesh.magSf()[iface];
 momFlux[iface] *= mesh.magSf()[iface];
 energyFlux[iface] *= mesh.magSf()[iface];
 localDt[ leftCell ]  += lambda * mesh.magSf()[iface];
 localDt[ rightCell ] += lambda * mesh.magSf()[iface];
}
```

Implement Boundary Conditions

- Loop over all boundary patches

```
forAll ( mesh.boundaryMesh() , ipatch ) {
```

- Access the physicalType of the patch

```
word BCTypePhysical =
  mesh.boundaryMesh().physicalTypes()[ipatch];
```

- Access the geometric type of the patch

```
word BCType = mesh.boundaryMesh().types()[ipatch];
```

- Access the patch names

```
word BCName = mesh.boundaryMesh().names()[ipatch];
```

Implement Boundary Conditions

- Access the owner-cell ids of patch

```
const UList<label> &bfaceCells =
  mesh.boundaryMesh()[ipatch].faceCells();
```

- Note that alternatively you can use *patchInternalField* to access owner-cell values of patch faces
- Try it yourself !

Implement Boundary Conditions - Slip wall

```
if( BCTypePhysical == "slip" ||
    BCTypePhysical == "symmetry" ) {
  forAll( bfaceCells  , iface ) {
  /// Extrapolate wall pressure
  scalar p_e = p[ bfaceCells[iface] ];
  vector normal = nf.boundaryField()[ipatch][iface];
  scalar face_area =
      mesh.magSf().boundaryField()[ipatch][iface];
  scalar lambda = std::fabs
  (
    U[ bfaceCells[iface] ] & normal ) +
    std::sqrt ( gama * p_e / rho[ bfaceCells[iface] ]
  );
  momResidue[ bfaceCells[iface] ] += p_e * normal *
      face_area;
  localDt[ bfaceCells[iface] ] += lambda * face_area;
 }
}
```

Implement Boundary Conditions - Extrapolated outflow

```
if( BCTypePhysical == "extrapolatedOutflow" ) {
 forAll( bfaceCells  , iface ) {
  label myCell = bfaceCells[iface];
  vector normal = nf.boundaryField()[ipatch][iface];
  scalar face_area
  (
   mesh.magSf().boundaryField()[ipatch][iface]
  );
  /// Use adjacent cell center value of face
  //  to get flux (zeroth order)
  scalar lambda = normalFlux( ... ); // Normal face flux
  massResidue[ myCell ]    += bflux[0] * face_area;
  momResidue[ myCell ][0] += bflux[1] * face_area;
  momResidue[ myCell ][1] += bflux[2] * face_area;
  momResidue[ myCell ][2] += bflux[3] * face_area;
  energyResidue[ myCell ] += bflux[4] * face_area;
  localDt[ myCell ]        += lambda * face_area;
 }
}
```

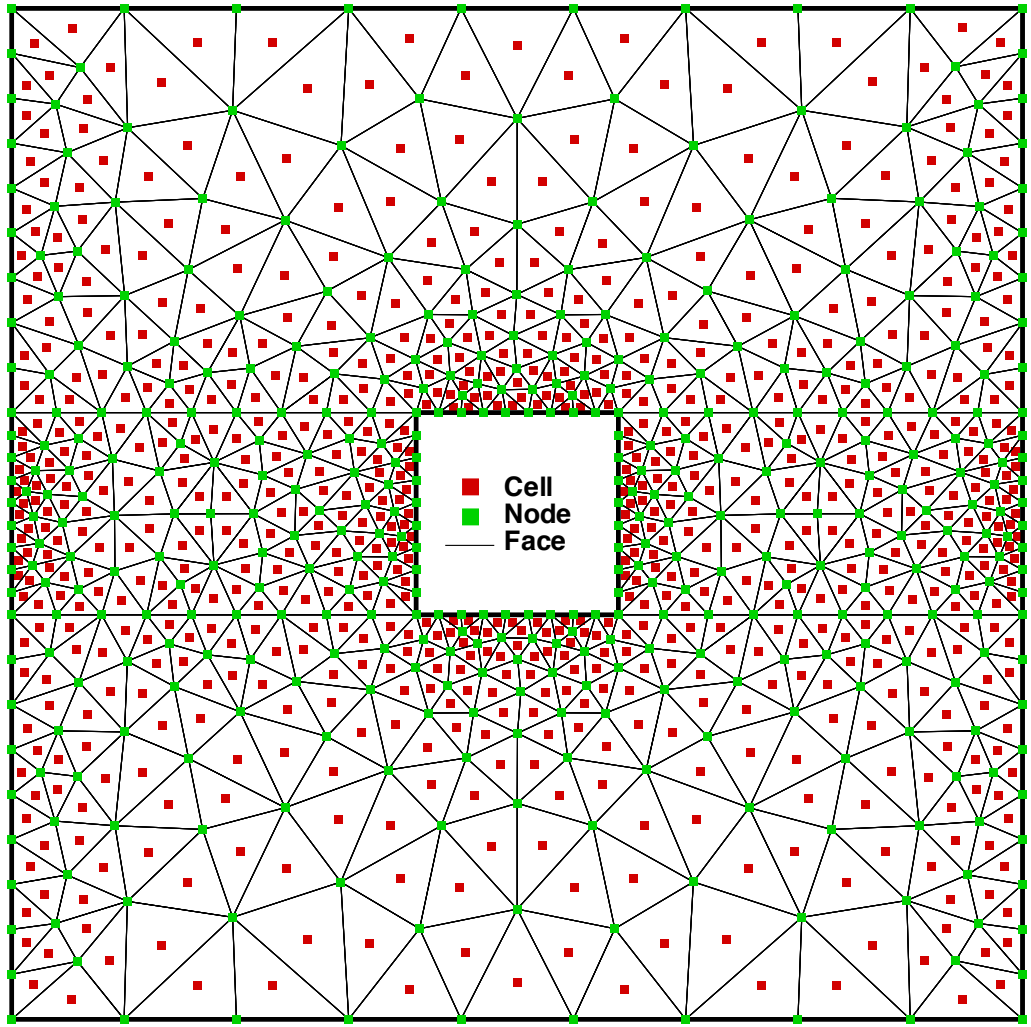
Implement Boundary Conditions - Supersonic inlet

```
if( BCTypePhysical == "supersonicInlet" ) {
 forAll( bfaceCells  , iface ) {
  label myCell = bfaceCells[iface];
  vector normal = nf.boundaryField()[ipatch][iface];
  scalar face_area
  (
    mesh.magSf().boundaryField()[ipatch][iface];
  );
  /// Use free-stream values to calculate face flux
  scalar lambda = normalFlux( &rho_inf , &u_inf , &p_inf
       , &normal , bflux );
  massResidue[ myCell ]   += bflux[0] * face_area;
  momResidue[ myCell ][0] += bflux[1] * face_area;
  momResidue[ myCell ][1] += bflux[2] * face_area;
  momResidue[ myCell ][2] += bflux[3] * face_area;
  energyResidue[ myCell ] += bflux[4] * face_area;
  localDt[ myCell ]       += lambda * face_area;
 }
}
```

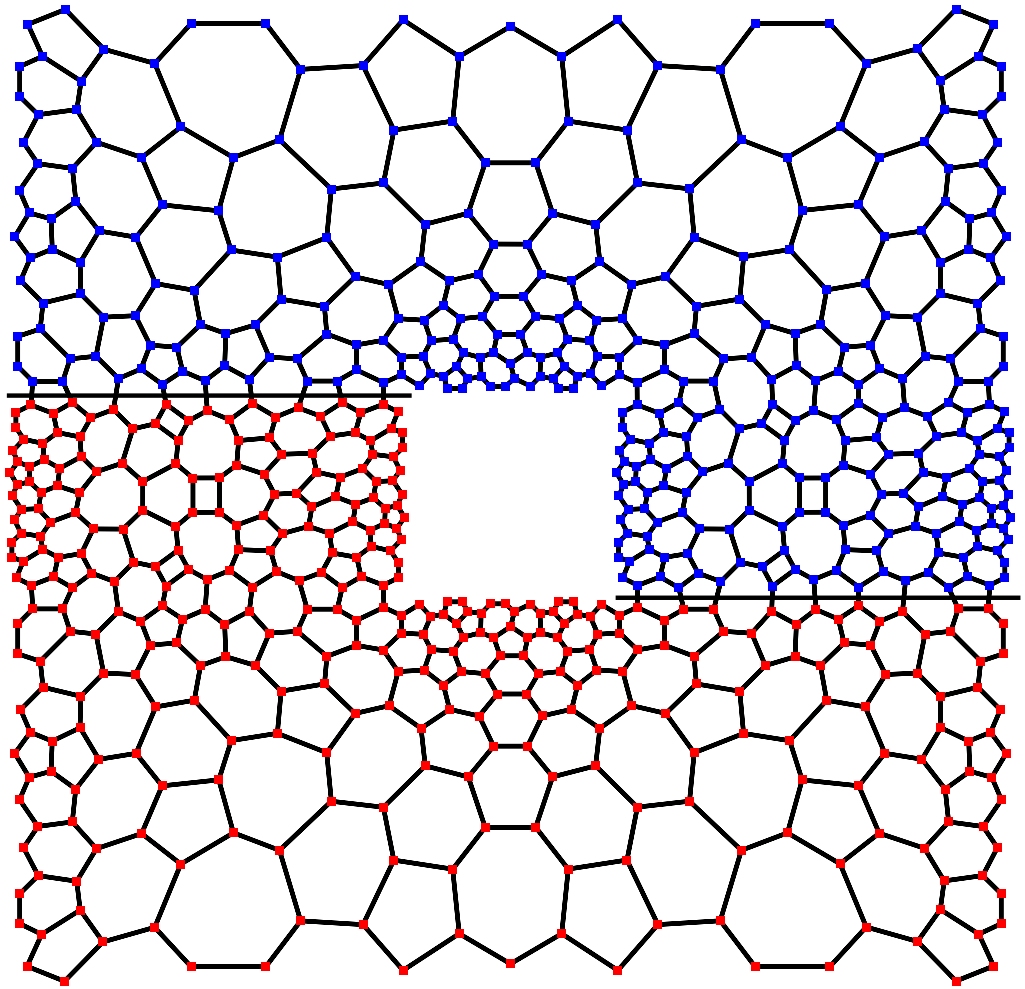
Hands on - Supersonic flow over wedge

- Compile the solver
- Setup inputs for the wedge case
- Run the example wedge case
- Plot the results

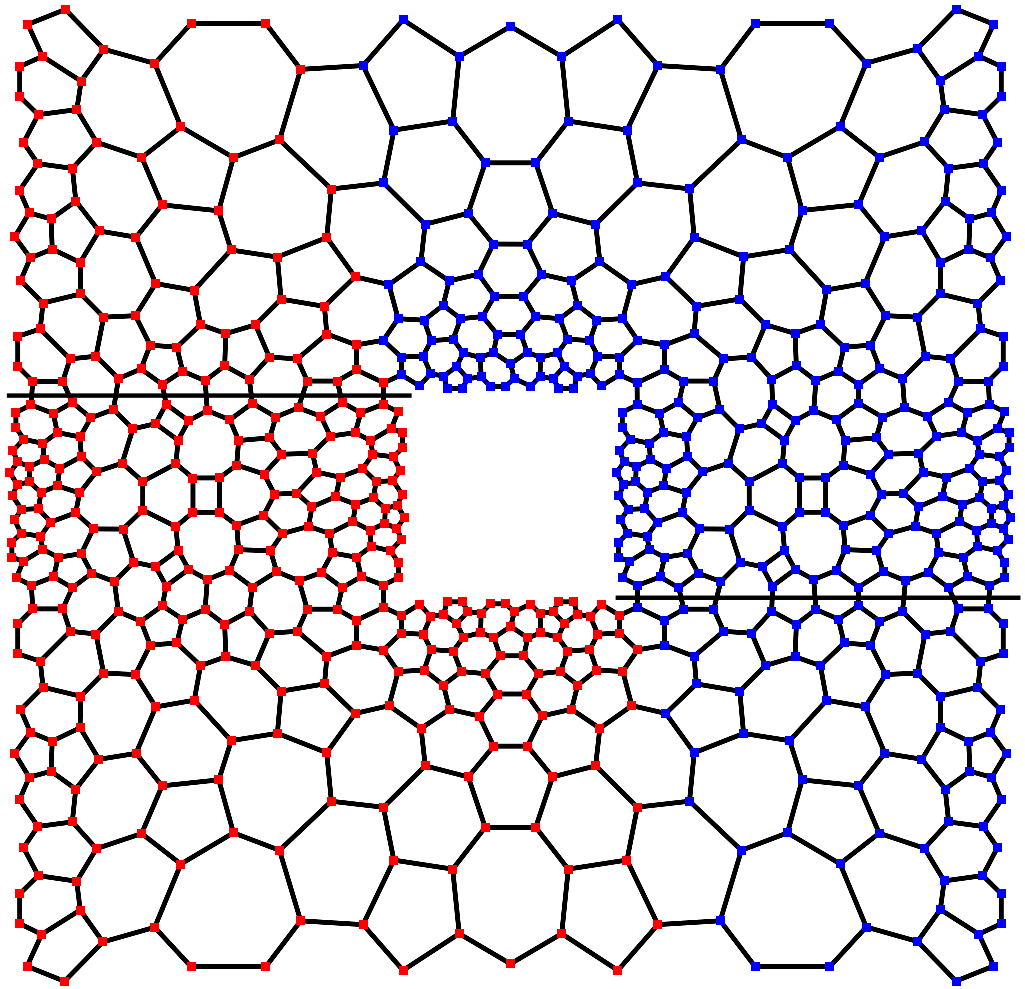
The unstructured mesh



The dual graph

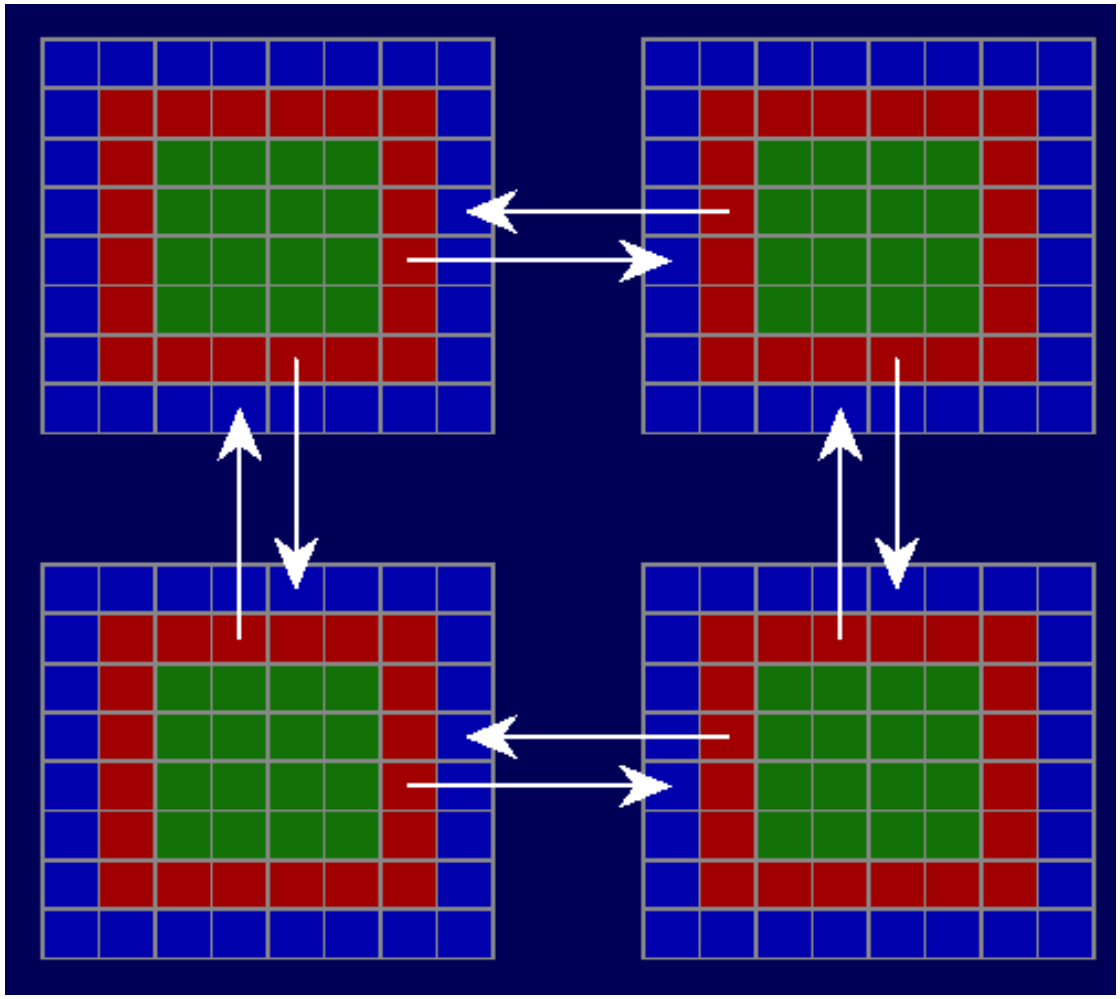


After graph partitioning



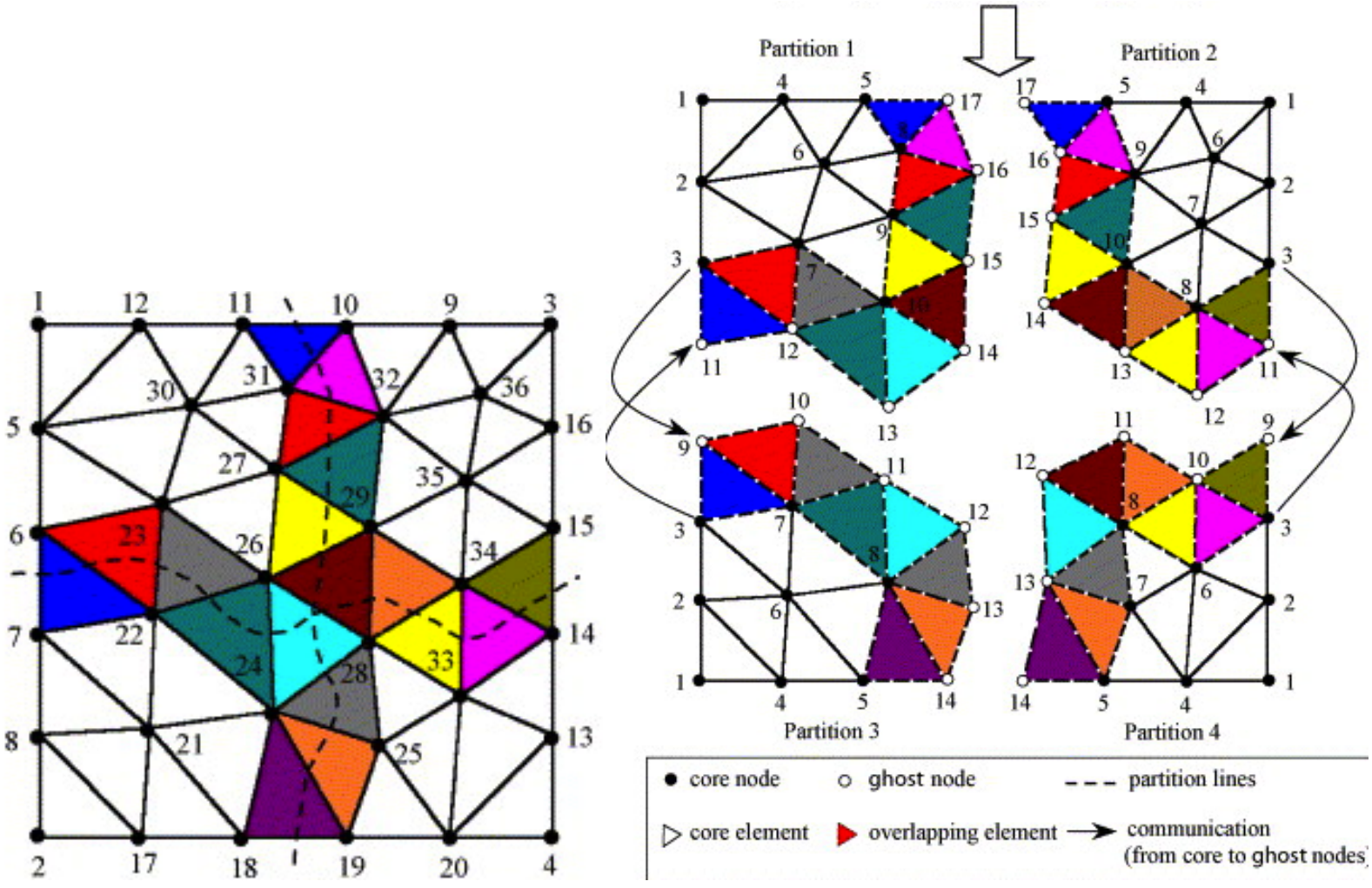
Structured Grid Decomposition

Structured Grid Decomposition with One Ghost Cell Padding (CC)



Graph based Unstructured Grid Decomposition

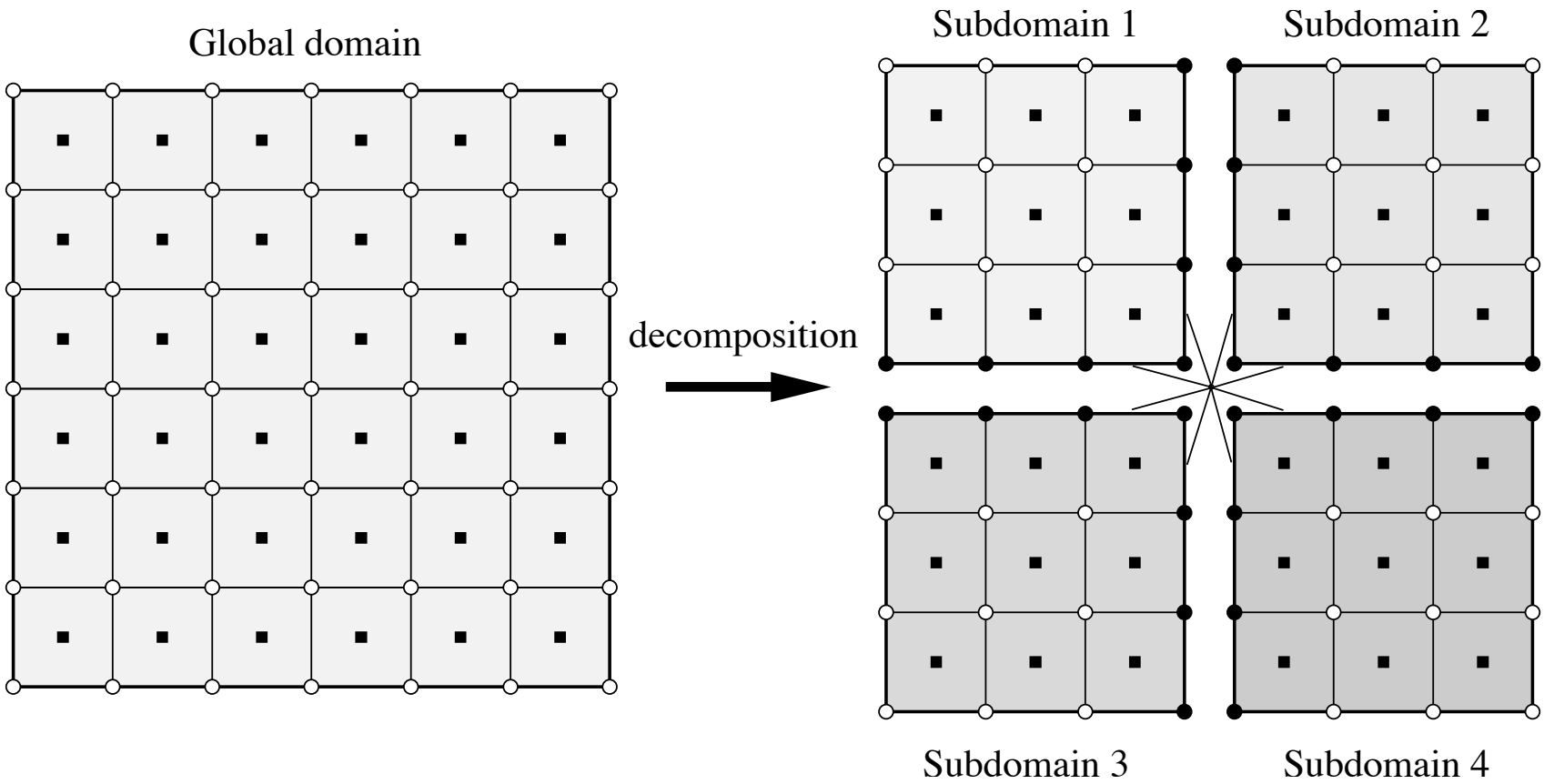
Node Based Unstructured Grid Decomposition



Parallelization in OpenFOAM (from Jasak's slides)

- Parallel communications are wrapped in *Pstream* library to isolate communication details from library use
- Discretization uses the domain decomposition with zero halo layer approach
- Parallel updates are a special case of coupled discretization and linear algebra functionality
  - *processorFvPatch* class for coupled discretization updates
  - *processorLduInterface* and *field* for linear algebra updates

Parallelization in OpenFOAM (from Jasak's slides)



Some Useful API in Pstream library

Pstream::parRun()

Check if *-parallel* is defined in command line

```
if( !Pstream::parRun() ) {
  Info << "Error: The program should be run under mpirun
      \n";
}
```

Pstream::myProcNo() and Pstream::masterNo()

Processor (and Master) Rank in MPI Communicator

```
Info << "My rank in MPI Communicator is "
     << Pstream::myProcNo() << " and master rank "
     << Pstream::masterNo() << "\n";
```

Parallel Streams - C++ i/ostreams

Perr

Error stream

```
Perr << "Some error occurred in parallel job \n";
```

Parallel Streams - C++ i/ostreams

OPstream/IPstream

Used to send/receive data to adjacent processor

```
vector data(0, 1, 2);
OPstream toMaster(Pstream::scheduled, Pstream::masterNo
    ());
toMaster << data;
...
IPstream fromMaster(Pstream::scheduled, Pstream::
    masterNo());
fromMaster >> data;
/* Types of schedules       **
**  Pstream::scheduled       **
**  Pstream::blocking        **
**  Pstream::nonBlocking     */
```

Hands on - First parallel code

- Check if parallel environment is defined using Pstream
- If it is indeed parallel run print the processor rank

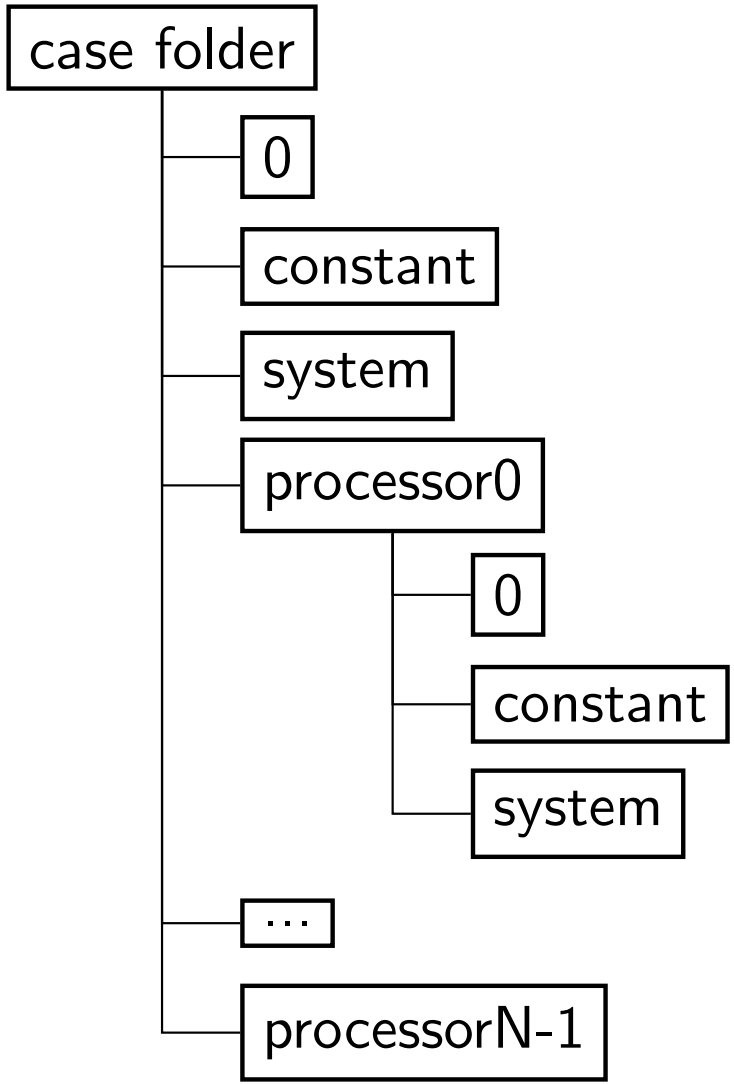
Creating the decomposition in OF

decomposePar

- Tool decomposes mesh into **N** partitions.
- Create the file *"decomposeParDict"* in the case *"system"* folder.

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "system";
    object      decomposeParDict;
}
numberOfSubdomains N;
method          metis;
```

"processor" directories



- *"processorX/constant"* folder has the *"boundary"* file
- Additional boundary patches are found with the type *"processor"*
- *"processor"* patch faces abut with neighbor decomposition

Hands on - Create Decompositons

- Create the decomposeParDict in case folder of wedge
- Set the number of partitions to 2
- Plot the decomposed mesh

Field Data - Parallel Exchange

- Field variables see the adjacent processor information via coupled boundary condition
- Coupled boundaries naturally allow non-blocking parallelization
- Volume field are the logical candidates for parallel exchange of faces fluxes

Hands on - Simple Parallel Exchange Example

Make rho values in each processor equal to its rank

```
    forAll( rho , icell  )
      rho[icell] = Pstream::myProcNo();
```

Loop over all boundary patches and get the type and alias of boundary cells attached to boundary face

```
forAll ( mesh.boundaryMesh() , ipatch ) {
  word BCtype = mesh.boundaryMesh().types()[ipatch];
  const UList<label> &bfaceCells =
    mesh.boundaryMesh()[ipatch].faceCells();
```

Hands on - Simple Parallel Exchange Example

Now check if boundary patch type is processor

```
if( BCtype == "processor" ) {
```

Verify if the non-blocking receive has been made successfully

```
rho.correctBoundaryConditions();
```

Hands on - Simple Parallel Exchange Example

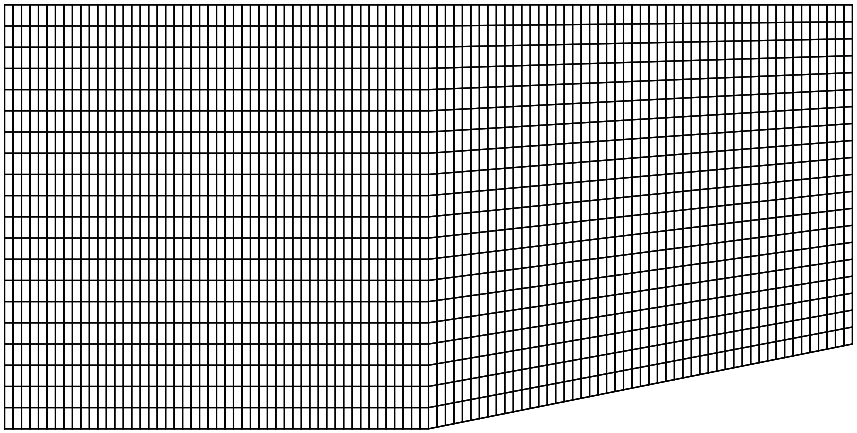
Now get the received field values to variable *exchange*

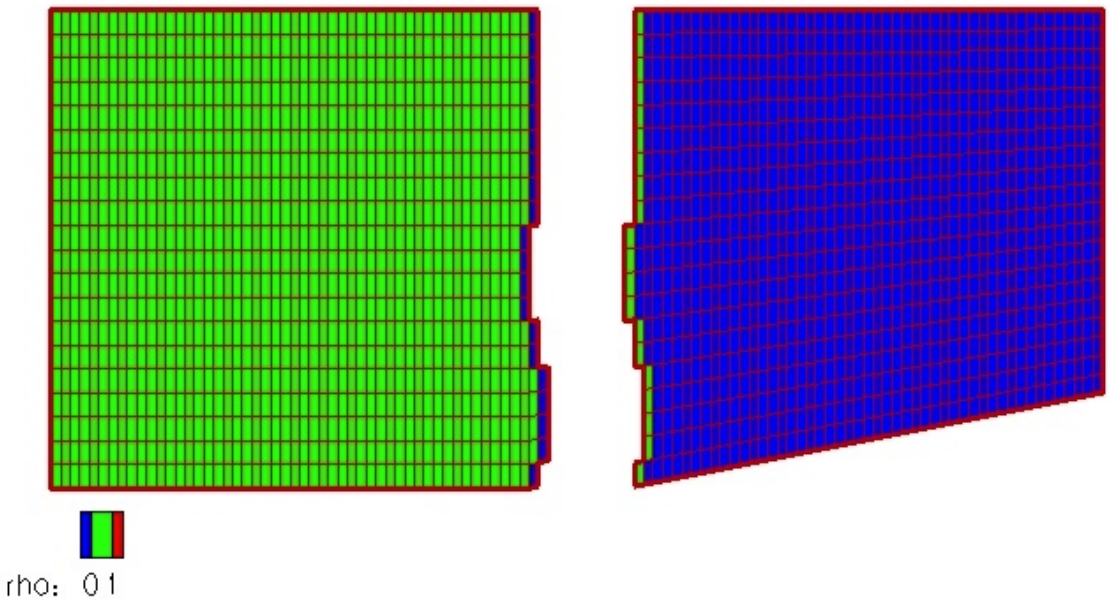
```
scalarField exchange =
            rho.boundaryField()[ipatch].
                patchNeighbourField();
```

Assign the boundary cell value to the received value

```
      forAll( bfaceCells , icell ) {
         rho[ bfaceCells[icell] ] = exchange[icell];
         std::cout << exchange[icell] << "\n";
       }
```

Output





"processor" BC and Halo

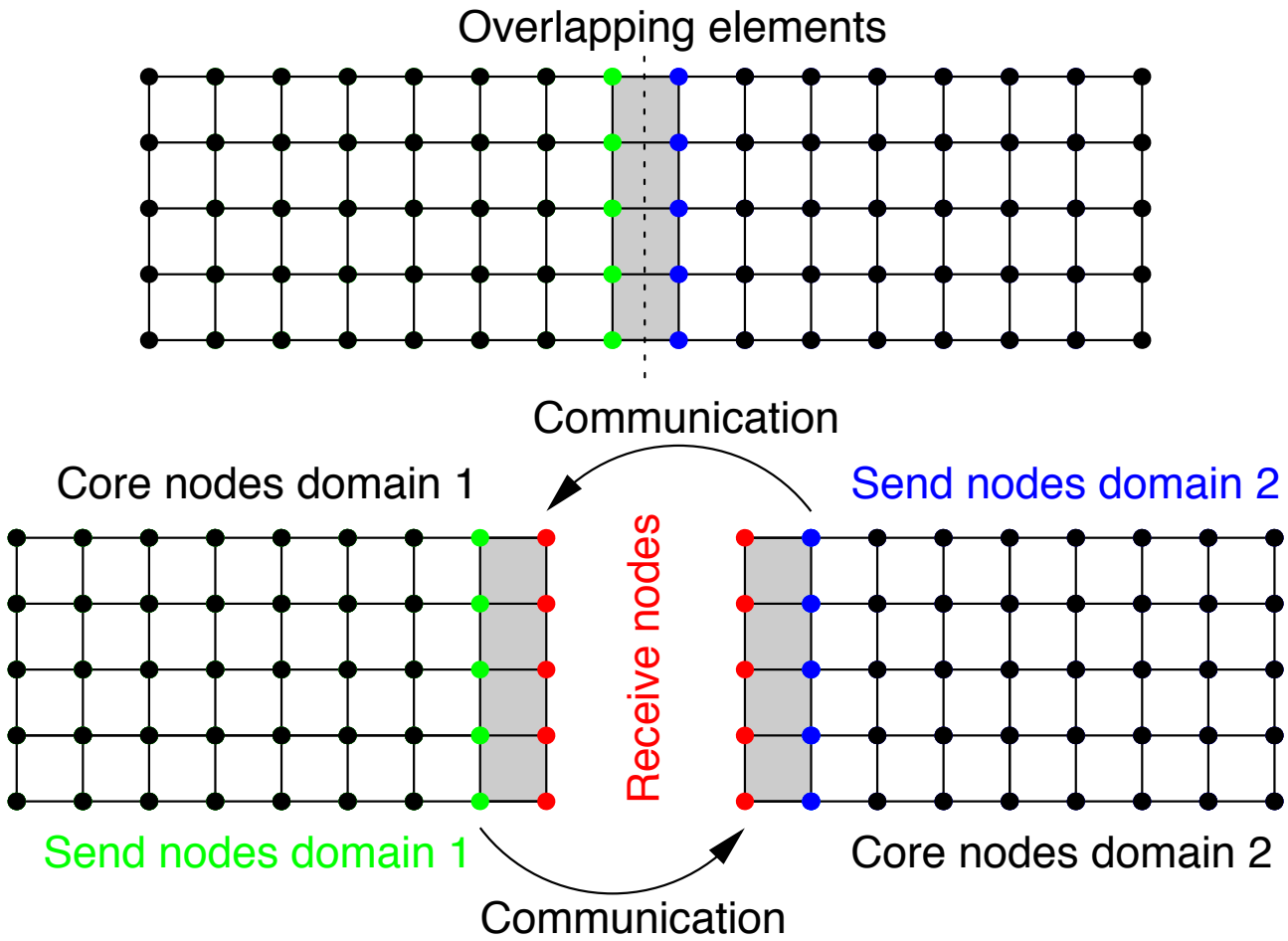


Figure: Halo/Ghost nodes in domain decomposition

"processor" BC and Halo

- OpenFOAM does not create the extra cell padding (Halo/Ghost)
- Reference to immediate cell quantities to *"processor"* boundary faces stored
- Quantities/Fields are imported during computation at *"processor"* boundary
- But import can happen only if export is posted ??

OF *"processor"* boundary - *type* and *physicalType*

```
...
7
(
    inlet
    {
        type            patch;
        physicalType    supersonicInlet;
        nFaces          0;
        startFace       1889;
    }
    ...
    ...
    procBoundary0to1
    {
        type            processor;
        nFaces          24;
        startFace       3967;
        myProcNo        0;
        neighbProcNo    1;
    }
```

Non-blocking communication and *"runTime"* object

```
/// Time step loop
while( runTime.loop() ) {
    ...
    ...
    runTime.write();
}
```

- For every iteration of the *runTime.loop*() non-blocking export/import is initiated
- During application of *"processor"* BC the export/import is confirmed using blocking call
- This essentially forms the Zero-Halo implementation in OF
- Computation and communication happen simultaneously - *latency hiding*

The key data-structure and functions

Communication - blocking wait on the send/recv call

```
    rho.correctBoundaryConditions();
    U.correctBoundaryConditions();
    p.correctBoundaryConditions();
```

Access Ghost/Halo data

```
scalarField rhoGhost = rho.boundaryField()[ipatch].
    patchNeighbourField();
vectorField UGhost   = U.boundaryField()[ipatch].
    patchNeighbourField();
scalarField pGhost   = p.boundaryField()[ipatch].
    patchNeighbourField();
```

Putting it all together

```
forAll( bfaceCells , iface ) {
 label leftCell   = bfaceCells[iface];
 label rightCell  = iface;
 vector normal    = nf.boundaryField()[ipatch][iface];
 scalar face_area =
          mesh.magSf().boundaryField()[ipatch][iface];
 // Approximate Riemann solver at interface
 (*fluxSolver)( &rho[leftCell]  , &U[leftCell]  ,
   &p[leftCell] , &rhoGhost[rightCell] ,
   &UGhost[rightCell] , &pGhost[rightCell] ,
   &tempRhoRes , &tempURes , &tempPRes , &normal );
 // Multiply with face area to get face flux
 massResidue[leftCell]    += tempRhoRes * face_area;
 momResidue[leftCell]     += tempURes   * face_area;
 energyResidue[leftCell]  += tempPRes   * face_area;
}
```


That's all Folks!