Parallel

Table of Contents

Introduction

When running a simulation in parallel, the geometry must first be decomposed (segmented) into individual geometries for each MPI process. These separate geometries are connected together with special processor boundary patches. The processor-specific `constant/polyMesh/boundary` files will contain this type of entry:

```
procBoundary0to14
{
    type            processor;
    inGroups        1(processor);
    nFaces          131;
    startFace       34983;
    matchTolerance  0.0001;
    transform       unknown;
    myProcNo        0;
    neighbProcNo    14;
}
```

The decomposePar utility is a commonly used method to decompose domains and subsequently distribute the fields. The reconstructPar and reconstructParMesh utilities can be used to reconstruct a single domain from the processor sub-domains. In addition to the above non-parallel decomposition and reconstruction tools there is a parallel all-in-one tool `redistributePar`:

- decompose:
  ```
  mpirun -np XXX redistributePar -decompose -parallel
  ```
- reconstruct
  ```
  mpirun -np XXX redistributePar -reconstruct -parallel
  ```

The decomposeParDict

The `decomposeParDict` is required by decompose utilities and for any solvers or utilities running in parallel. It is normally located in the simulation `system` directory. The `-decomposeParDict name` command-line option can be used to specify an alternate file.

The `numberOfSubdomains` entry is mandatory:

```
numberOfSubdomains  <int>;
```

The `method` entry is required for the decomposePar utility and specifies the decomposition method type:

```
method  <word>;
```

The `method` entry is generally not required when running a simulation.

OpenFOAM offers a variety of decomposition methods and interfaces to external, third-party decomposition routines. The types of decomposition methods available will thus depend on your particular installation.

Name	Class
none	Foam::noDecomp
manual	Foam::manualDecomp
simple	Foam::simpleGeomDecomp
hierarchical	Foam::hierarchGeomDecomp
kahip	Foam::kahipDecomp
metis	Foam::metisDecomp
scotch	Foam::scotchDecomp
structured	Foam::structuredDecomp
multiLevel	Foam::multiLevelDecomp

If a decomposition method requires any additional configuration controls, these are specified either within in a generic `coeffs` dictionary that or a method-specific version. For example,

```
method  hierarchical;

coeffs
{
    n   (4 2 3);
}

// -----

method  metis;

metisCoeffs
{
    method  k-way;
}
```

For simplicity, the generic `coeffs` dictionary is generally preferable. However, for some specific decomposition methods, e.g., **multiLevel**) only the method-specific coefficients dictionary is permitted.

Multi-region

When running multi-region simulations, it may be desirable to use different decomposition methods for one or more regions, or even to have fewer processors allocated to a particular region. If, for example, the multi-region simulation contains a large fluid region and a very small solid region, it can be advantageous to decompose the solid onto fewer processors.

The region-wise specification is contained in a regions sub-dictionary with `decomposeParDict`. For example,

```
numberOfSubdomains  2048;
method  metis;

regions
{
    heater
    {
        numberOfSubdomains  2;
        method  hierarchical;
        coeffs
        {
            n   (2 1 1);
        }
    }

    "*.solid"
    {
        numberOfSubdomains  16;
        method  scotch;
    }
}
```

> **Note**
>
> The top-level `numberOfSubdomains` remains mandatory, since this specifies the number of domains for the entire simulation. The individual regions may use the same number or fewer domains. The `numberOfSubdomains` entry within a region specification is only needed if the value differs.

Multi-level decomposition

The Foam::multiLevelDecomp decomposition provides a general means of successively decomposing with different methods. Each application of the decomposition is termed a level. For example,

```
numberOfSubdomains  2048;
method  multiLevel;

multiLevelCoeffs
{
    nodes
    {
        numberOfSubdomains 128;
        method  hierarchical;
        coeffs
        {
            n   (16 4 2);
        }
    }
    cpus
```

```
    {
        numberOfSubdomains 2;
        method  scotch;
    }
    cores
    {
        numberOfSubdomains 8;
        method  metis;
    }
}
```

For cases where the same method is applied at each level, this can also be conveniently written in a much shorter form:

```
numberOfSubdomains  2048;
method  multiLevel;

multiLevelCoeffs
{
    method  scotch
    domains (128 2 8);
}
```

When the specified `domains` is smaller than `numberOfSubdomains` but can be resolved as an integral multiple, this integral multiple is used as the first level. This can make it easier to manage when changing the number of domains for the simulation. For example,

```
numberOfSubdomains  1024;
method  multiLevel;

multiLevelCoeffs
{
    method  scotch
    domains (2 8);    //< inferred as   domains (64 2 8);
}
```

Constraints

These are constraints applied to the decomposition. Typical uses might be e.g. to keep a **cyclicAMI** patch on a single processor (might speed up simulation) or have maximum local donors in an **overset** case.

```
constraints
{
    // Keep owner and neighbour of baffles on same processor
    // (ie, keep it detectable as a baffle).
    // Baffles are two boundary face sharing the same points
    baffles
    {
        type    preserveBaffles;
        enabled true;
    }
```

```
// Keep owner and neighbour on same processor for faces in zones
faces
{
    type    preserveFaceZones;
    zones   (".*");
    enabled true;
}

// Keep owner and neighbour on same processor for faces in patches
// (only makes sense for cyclic patches. Not suitable for e.g.
// cyclicAMI since these are not coupled on the patch level.
// Use singleProcessorFaceSets for those.
patches
{
    type    preservePatches;
    patches (".*");
    enabled true;
}

// Keep all of faceSet on a single processor. This puts all cells
// connected with a point, edge or face on the same processor.
// (just having face connected cells might not guarantee a balanced
// decomposition)
// The processor can be -1 (the decompositionMethod chooses the
// processor for a good load balance) or explicitly provided (upsets
// balance)
processors
{
    type    singleProcessorFaceSets;
    sets    ((f1 -1));
    enabled true;
}

// Decompose cells such that all cell originating from single cell
// end up on same processor
refinement
{
    type    refinementHistory;
    enabled true;
}

// Prevent decomposition splitting of the geometric regions
// Uses any topoSetFaceSource for selecting the constrained faces
geometric
{
    type    geometric;

    grow    false;

    selection
    {
```

```
            box1
            {
                source  box;
                min     (-10 -10 -10);
                max     (1 1 1);
            }

            ball1
            {
                source  sphere;
                origin  (-2 -2 1);
                radius  1;
            }

            arbitrary
            {
                source  surface;
                surfaceType  triSurfaceMesh;
                surfaceName  blob.obj;
            }
        }
    }
}
```

Would you like to suggest an improvement to this page? Create an issue