Handling Parallelisation in OpenFOAM

Hrvoje Jasak

hrvoje.jasak@fsb.hr

Faculty of Mechanical Engineering and Naval Architecture

University of Zagreb, Croatia

Parallelisation in OpenFOAM

Objective

- Present the approach and implementation of massively parallel discretisation and solution method support in the OpenFOAM

Topics

- Background on architecture and massive parallelism
- Components
- Parallel communications library: `Pstream`
- Domain decomposition and reconstruction
- Zero halo layer discretisation support: processor boundary type
- Discretisation and Algorithmic issues
- Cell- and point-based algorithms: FVM and FEM
- Linear solvers and preconditioners
- Parallelisation of Lagrangian particle tracking
- Running OpenFOAM in parallel
- Summary

Background

Massively Parallel Computing

- Today, most large-scale CFD solvers rely on distributed-memory parallel computer architectures for all medium-sized and large simulations
- Parallelisation of commercial solvers is completed: if the algorithm does not parallelise well, it is not used calculation
- Current development work aimed at bottle-necks: **parallel mesh generation**

Parallel Computer Architecture

- Parallel CCM software operates almost exclusively in **domain decomposition mode**: a large loop (*e.g.* cell-face loop in the FVM solver) is split into bits and given to a separate CPU. Data dependency is handled explicitly by the software
- Similar to high-performance architecture, parallel computers differ in how each node (CPU) can see and access data (memory) on other nodes:
  - **Shared memory machines**: single node sees complete memory
  - **Distributed memory machines**: each node is a self-contained unit. Node-to-node communication involves considerable overhead
- For distributed memory machines, a node can be an off-the-shelf PC or a server node: cheap, but limited by speed of (network) communication

Components

Parallel Components and Functionality

1. Parallel communication wrapper
   - Basic information about the run-time environment: serial or parallel execution, number of processors, process IDs etc.
   - Passing information in transparent and protocol-independent manner
   - Global gather-scatter communication operations
2. Mesh-related operations
   - Mesh and data decomposition and reconstruction
   - Global mesh information, e.g. global mesh size
   - Handling patched pairwise communications
   - Processor topology communication scheduling data
3. Discretisation support
   - Processor data updates across processors: data consistency
   - Matrix assembly: executing discretisation operations in parallel
4. Linear equation solver support
5. Auxiliary operations, e.g. messaging or algorithmic communications, non-field algorithms (e.g. particle tracking), data input-output

Parallel Communications

Stream-Based Parallel Communication Library

- In C++, functional communication is replaced by a stream-based system

```
int a = 7;
// Some C programming, using ellipsis arguments
printf("Hello World, for the %ith time", a);

// Strongly checked, object-oriented C++ way
cout << "Hello, World " << a << endl;
```

- Stream-based system separates class-dependent part from communications
- A stream is responsible for handling the data in a stream-dependent way. Example: graph in a file or on a screen
- In OpenFOAM, all relevant classes contain `Istream` constructor and `operator<<()` (write operation)
- Parallel communication protocol is-a stream: `Pstream`

```
OPstream toNeighb(procPatch.neighbProcNo());
toNeighb << patchInfo;

IPstream fromNeighb(procPatch.neighbProcNo());
fromNeighb >> nbrPatchInfo;
```

Parallel Communications

Stream-Based Parallel Communication Library

- Parallel communications library holds basic run-time information
- Since `Pstream` is derived from IOstream, no object-level changes are required: sending a floating point array and a mesh object is of same complexity
- Buffered and compressed transfer are specified as `Pstream` settings

Implementation

- Most `Pstream` data and behaviour is generic (does not depend on the underlying communications algorithm): create and destroy, manage buffers, record processor IDs etc.
- The part which is communication-dependent is limited to a few functions: initialise, exit, send data and receive data
- OpenFOAM implements a single `Pstream` class, but protocol-dependent library is implemented separately: run-time linkage changes the communication protocol!
- Easy porting to new communication platform: re-implementing calls in a library
- . . . and no changes are required anywhere else in the code
- Communication protocol is chosen by picking the shared library: `libPstream`
- Optional use of compression on send/receive to speed up communication

Domain Decomposition Approach

Parallel FVM Simulation

- Computational domain is split up into meshes, each associated with a single processor:
  - Allocation of cells to processors
  - Physical decomposition of the mesh in the native solver format

  This step is termed **domain decomposition**
- A mechanism for data transfer between processors needs to be devised: a standard interface to a wrapped communications package
- Solution algorithm is analysed to establish the inter-dependence and points of synchronisation
- Mesh partitioning constraints
  - Load balance: all processing units should have approximately the same amount of work between communication and synchronisation points
  - Minimum communication, relative to local work. Performing local computations is orders of magnitude faster than communicating the data

Data Reconstruction

- Operate in the opposite direction: reconstruct decomposed mesh and data
- Longer term, this should be avoided: parallelisation of the complete simulation

Parallel Algorithms

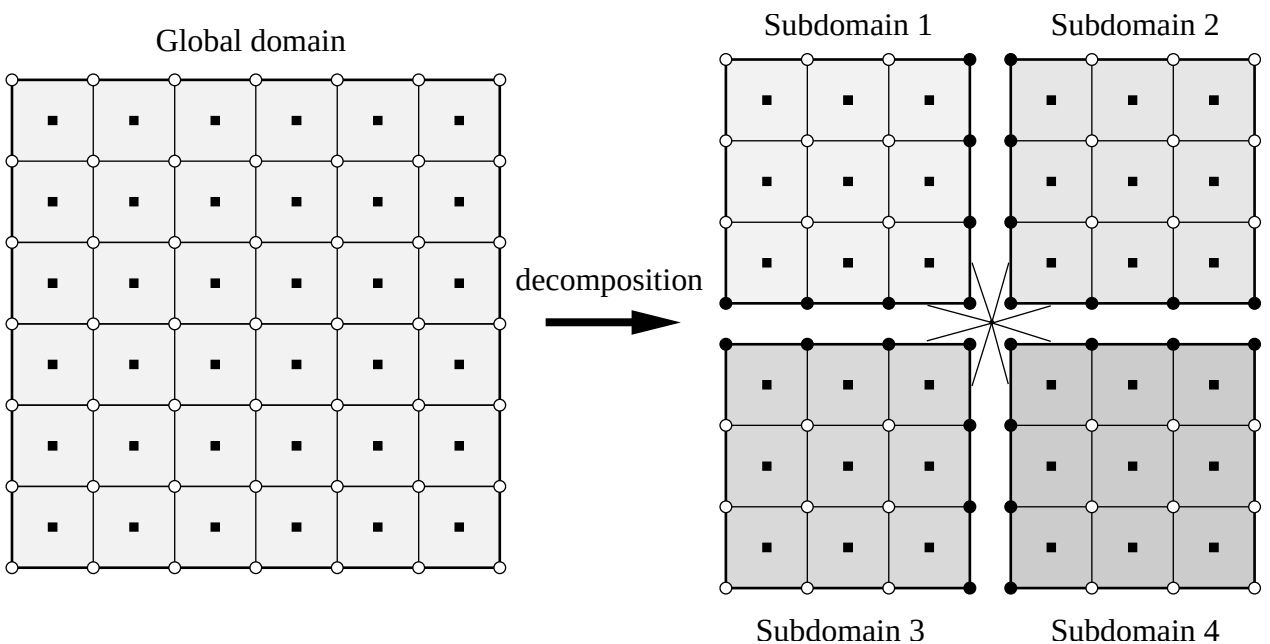
Mesh Support

- For purposes of algorithmic analysis, we shall recognise that each cell belongs to one and only one processor: no inherent overlap for computational points
- In FVM, mesh faces can be grouped as follows
  - Internal faces, within a single processor mesh
  - Boundary faces
  - Inter-processor boundary faces: faces used to be internal but are now separate and represented on 2 CPUs. No face may belong to more than 2 sub-domains
- FEM (and cell-to-point interpolation) operates on vertices in a similar manner but a vertex may be multiply shared between processors: overlap is unavoidable
- Algorithmically, there is no change for objects internal to the mesh and on the processor boundary: this is the source of parallel speed-up
- The challenge is to repeat the operations for objects on inter-processor boundaries

Parallel Algorithms

Halo Layer Approach

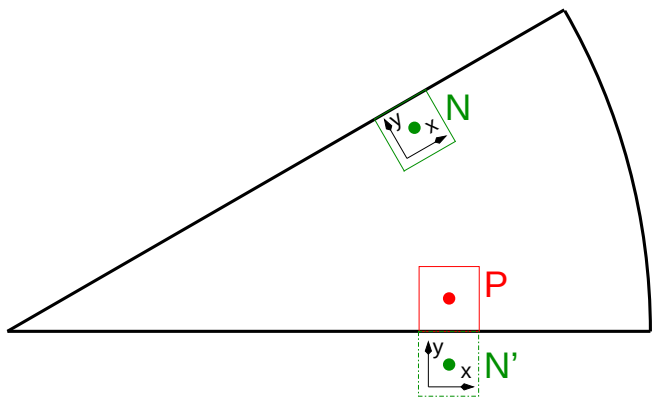
- Traditionally, FVM parallelisation uses **halo layer** approach: data for cells next to a processor boundary is duplicated
- Halo layer covers all processor boundaries and is explicitly updated through parallel communications calls
- Major impact on code design: all cell and face loops need to recognise and handle the presence of halo layer
- Communications pattern is prescribed: only halo information is exchanged



Parallel Algorithms

Zero Halo Layer Approach

- Algorithmically, halo is an inconsistent approach: coupled boundary with out-of-core addressing exists in other places in the code
- Example: cyclic and periodic “boundary conditions” pose the same problem



- Implementation principle: introduce implicit updated boundaries with “out-of core” addressing
- Specific types: cyclic, periodic, processor (= cyclic with communication), GGI
- Processor boundary update encapsulates communication to do evaluation. Virtual function mechanism does the rest: no impact in the rest of the code!
- Difference between parallel and serial execution is the presence of processor boundary type and serial or parallel deployment (`mpirun`)

FVM Discretisation

Explicit FVM Operators: Gradient Calculation

- Using Gauss' theorem, we need to evaluate face values of the variable. For internal faces, this is done trough **interpolation**:

$$\phi_f = f_x \, \phi_P + (1 - f_x) \, \phi_N$$

  Once calculated, face value may be re-used until cell-centred ϕ changes

- In parallel, ϕ_P and ϕ_N live on different processors. Assuming ϕ_P is local, ϕ_N can be fetched through communication: this is once-per-solution cost and obtained by pairwise communication
- Note that all processors perform identical duties: thus, for a processor boundary between domain A and B, evaluation of face values can be done in 3 steps:
  1. Collect a subset internal cell values from local domain and send the values to the neighbouring processor
  2. Receive neighbour values from neighbouring processor
  3. Evaluate local processor face value using interpolation

FVM Discretisation

Finite Volume Matrix Assembly

- Similar to gradient calculation above, assembly of matrix coefficients on processor boundaries can be done using simple pairwise communication
- In order to assemble the coefficient, we need geometrical information and some interpolated data: all readily available, maybe with some communication
- Example: off-diagonal coefficient of a Laplace operator

$$a_N = |\mathbf{s}_f| \frac{\gamma_f}{|\mathbf{d}_f|}$$

  where γ_f is the interpolated diffusion coefficient and the rest are geometry-related properties. In actual implementation, geometry is calculated locally and interpolation factors are cached to minimise communication
- Discretisation of a convection term is similarly simple
- Sources, sinks and temporal schemes all remain unchanged: each cell belongs to only one processor

FEM Discretisation

Supporting FEM Discretisation

- Unlike the FVM, FEM computational points are present on multiple processors
- Since discretisation is element-based, matrix assembly is completed by combining contribution from various processors
- Parallel communication pattern may now be more complex: n processors sharing the same point
- . . . but for consistency FEM must operate on the same domain decomposition
- In order to avoid communications overheads of many small messages in a complex connectivity graph, a **2-tier update** is used
  - Pairwise patch-based communications: long messages, similar to FVM
  - Collapse all globally shared communication into a single gather-scatter operation: avoid small messages and complex comms pattern

Linear Solver Algorithms

Linear Equation Solvers

- Major impact of parallelism in linear equation solvers is in choice of algorithm. Only algorithms that can operate on a fixed local matrix slice created by local discretisation will give acceptable performance
- In terms of code organisation, each sub-domain creates its own **numbering space**: locally, equation numbering always starts with zero and one cannot rely on global numbering: it breaks parallel efficiency
- Coefficients related to processor interfaces are kept separate and multiplied through in a separate matrix update
- Impact of processor boundaries will be seen in:
  - Every matrix-vector multiplication operation
  - Every Gauss-Seidel or similar smoothing sweep

  . . . but nowhere else!

Linear Solver Algorithms

Processor Interface Updates

- Data dependency in out-of-core vector- matrix multiplication is identical to explicit evaluation of shared data during discretisation
- This appears for all implicitly coupled boundary conditions: virtual base class interface needed
- `lduCoupledInterface` handles all out-of-core updates. It is updated after every vector-matrix operation or smoothing sweep `processorLduCoupledInterface` is a derived class, using `Pstream` for communications and `processorFvPatch` for addressing

Parallel Algebraic Multigrid (AMG)

- As a rule, **Krylov space solvers** parallelise naturally: global updates on scaling and residual combined with local vector-matrix operations
- In **Algebraic Multigrid** care needs to be given to coarsening algorithms
  - Aggregative AMG (AAMG) work naturally on matrices without overlap (FVM)
  - For cases with overlap (FEM), Selective AMG works better
- Currently, all algorithms assume uniform communications performance across the machine. For very large clusters, AMG suffers due to lack of scaling: coarse level serialises the work and boosts communications latency issues

Parallel Algorithms

Synchronisation

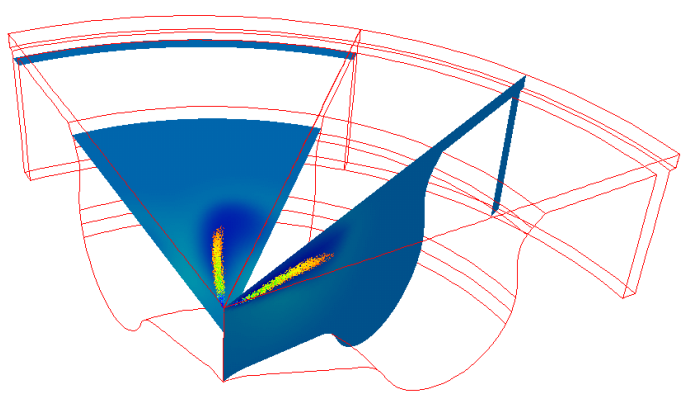
- Parallel domain decomposition solvers operate such that **all processors follow identical execution path in the code**. In order to achieve this, some decisions and control parameters need to be synchronised across all processor
- Example: convergence tolerance. If one of the processors decides convergence is reached and others do not, they will attempt to continue with iterations and simulation will lock up waiting for communication
- **Global reduce operations** synchronise decision-making and appear throughout high-level code. This is built into reduction operators: `gSum, gMax, gMin` etc.
- Communications in global reduce is of gather-scatter type: all CPUs send their data to CPU 0, which combines the data and broadcasts it back
- Actual implementation is more clever: using native gather-scatter functionality optimised for type of inter-connect, or hierarchical communications

```
vector force = sum
(
    mesh.Sf().boundaryField()[patchI]*p.boundaryField()[patchI]
);
reduce(force, sumOp<vector>());
```

Lagrangian Particle Tracking

Parallelisation of Particle Tracking

- Particle tracking algorithm operates on each segment of decomposed mesh by tracking local particles
- Tracking class implements boundary interaction: what happens when a particle hits a boundary face is handled through virtual functions
- **Processor boundary interaction**: answering the virtual function interface, a particle is migrated to connecting processor
- Issues with load balancing: loss of performance for cases where particles are not uniformly distributed in the domain



Running OpenFOAM in Parallel

Case Preparation

- Typically, the case will be prepared in one piece: **serial mesh generation**
- `decomposePar`: parallel decomposition tool, controlled by `decomposeParDict`
- Options in the dictionary allow choice of decomposition and auxiliary data
- Upon decomposition, `processorNN` directories are created with decomposed mesh and fields; solution controls, model choice and discretisation parameters are shared. Each CPU may use local disk space
- `decomposePar` may output cell-to-processor decomposition
- Manual decomposition (debugging): provide cell-to-processor file

Parallel Execution

- Top-level code does not change between serial and parallel execution: operations related to parallel support are embedded in the library
- Launch executable using `mpirun` (or equivalent) with `-parallel` option
- Data in time directories is created on a per-processor basis
- It is possible to visualise a single CPU data (but we do not do it often): there may be problems with processor boundaries
- Field initialisation may also be run in parallel: trivial parallelisation

Running OpenFOAM in Parallel

On-the-Fly Data Analysis

- Sampling, graphing and post-processing tools will execute correctly in parallel: location of probes is reduced over CPUs
- In parallel decomposition, all patches are present on all CPUs (even with zero size). This allows global reduction of patch-based data

Graphical Post-Processing

- Used most often: reconstruct the data to a single CPU: `reconstructPar`
- Reconstructed data can be re-decomposed to a different number of CPUs
- Dynamic load balancing and some mesh handling features are work-in-progress
- For large data sets, sampling tools may be executed in parallel: extract and merge iso-surfaces for visualisation

Summary

Parallelisation in OpenFOAM

- Parallel communications are wrapped in `Pstream` library to isolate communication details from library use
- Discretisation uses the domain decomposition with zero halo layer approach
- Parallel updates are a special case of coupled discretisation and linear algebra functionality: the code transparently implements parallel coupling
  - `processorFvPatch` and equivalent FEM/FAM class for coupled discretisation updates
  - `processorLduInterface` and field for linear algebra updates

Conclusion

- OpenFOAM is a mature and validated object-oriented implementation of domain decomposition parallelism
- There are no specific top-level parallelisation requirements: identical code operates in serial and parallel execution
- Porting to new communication protocols is easy
- Future work involves chasing performance with non-trivial communication setup and dynamic load balancing