Development of Fully-Automatic Parallel Algorithms for Mesh Handling in the OpenFOAM®-2.2.x technology

F. Piscaglia, A. Montorfano, A. Onorati
Politecnico di Milano, Italy

The current development to set up an automatic procedure for mesh generation and handling for internal combustion engine simulation in OpenFOAM®-2.2.x is described. In order to automatically generate high-quality meshes of cylinder geometries, some technical issues need to be addressed: 1) automatic mesh generation should be able to control anisotropy and directionality of the grid; 2) during point motion, cells and faces must be introduced and removed without varying the overall area and volume of the cells, to avoid conservation errors. This paper outlines the recent development of the sliding interface, an algorithm to handle motion and topological changes in a moving-mesh Finite Volume Method (FVM) framework, that has been implemented to work with the newest mesh handling strategy of OpenFOAM®-2.2.x in a massively parallel environment. The algorithm allows for the use of grids having non-conformal interfaces, opening the use of the code to a wide range of applications: the simulation of the piston motion trough scavenging ports in a two-stroke engine, the generation of cylinder grids characterized by a high quality mesh near the valve region, the reduction of the mesh size in external flow calculations. Finally, in order to automate the mesh generation process of complex engine geometries, a fully parallelised algorithm with automatic domain decomposition for run-time generation of hexahedra and split-hexahedra grids of engines, without the loss of any geometric feature, is presented.

INTRODUCTION

Dynamic mesh handling in CFD codes is a suitable technique to simulate in-cylinder flows in internal combustion engines (moving piston and valves), rotor-stator interaction in turbomachinery simulation, floating bodies in naval hydrodynamics, deforming fluid mesh when simulations including fluid-structure interaction are carried out. OpenFOAM® represents a suitable platform for complex physical modeling, since it includes a common interface to all dynamically changing meshes. To preserve the mesh quality during extreme boundary deformation due to piston and valve motion, the number of the cells in the mesh needs to be changed. For this reason a set of “topological changes” need to be defined, allowing the possibility of attaching or detaching boundaries, adding or removing cell layers and using sliding mesh interfaces. In topological changes the idea is to use self-contained objects, where a topology change is executed on demand, rather than at prescribed moments. Mesh motion can be performed in two ways, namely layering and deformation. In the first case, the mesh topology is changed by adding or removing cells or layers of cells, with the position of most mesh points remaining unchanged. In the deformation mode, the majority of points in the cylinder is moved, where the motion of internal points is obtained by solving a mesh motion equation. During mesh motion with sliding interface, the computational mesh is decomposed into a series of different regions, depending on geometry or mode of motion. Boundary zones can include non-conformal interfaces, where the boundaries between cell zones present mesh node locations that are not identical. Non-conformal interfaces may overlap either partially or fully; cell zones in the non-conformal interfaces are connected to each other by passing fluxes. Regions are then disconnected when mesh points are moved.

A novel implementation of the sliding interface algorithm in OpenFOAM, to handle motion and topological changes in a moving-mesh Finite Volume Method (FVM), is described. With respect to former implementations [1], the work presented in this paper includes some significant changes:

- the algorithm for moving mesh is now based on a FV mesh motion technique rather than on the vertex-based motion;
- the implementation is fully parallel: an extensive low-level work has been carried out to parallelise topological changes, that can happen only on some processors but not to other. Handling local and global mesh data involves communication, which need to be synchronized when a topological change is performed locally: this has been achieved by data mapping on neighboring processor, that is may require local update for parallel synchronization;
- the implementation is compatible with the mesh definition employed in the latest official release of OpenFOAM®-2.2.x [2, 3], that strongly differs by other distributions [4].

The approach has been tested for the simulation of two-stroke engines, to manage the overlap between cylinder and port meshes. Also, sliding interface has been used to reduce the mesh size in the generation of cylinder grids, characterized by a high quality mesh near the valve region, and for the simulation of external flows.
Finally, a novel strategy for automatic mesh generation and case setup of complex engine geometries, valid both for steady-flow and full-cycle engine simulation, is presented. The method consists of apply the prescribed motion of moving boundaries (valves and pistons) to an initial engine geometry imported by a triangulated surface representation, that automatically generates a multiple number of cylinder STL geometries to be used by an open-source fully parallelised algorithm with automatic domain decomposition for run-time mesh generation. Different meshes containing hexahedra (hex) and split-hexahedra (split-hex) are therefore generated for different crank angles without involving significant manual work that strongly increases the time typically required for the simulation setup.

SLIDING INTERFACE

The sliding interface topology modifier allows for the creation of a reversible, non-conformal grid interface between two mesh regions with different mesh structure. The coupling is achieved by changing the local mesh topology in order to get strict one-to-one point correspondence. After the coupling operation, no “interface” still exist, and the former detached regions can be considered as one. Therefore, the sliding interface philosophy is different from other region coupling algorithms, that mostly rely on baffle faces to achieve a fluid-dynamic link between topologically separated zones.

If necessary, the topological change can be reversed and mesh can be brought back to the detached configuration. An example is when a relative tangential (“sliding”, hence the name) motion is prescribed between two different mesh regions, namely A and B. The algorithm employed to move points without degrading the cell quality consists of the following steps:

1. Mesh regions are initially detached
2. Region A is moved (slides) with respect to region B
3. Region A and B are coupled
4. Flow equations are solved
5. Region A and B are detached
6. Loop restarts from step #1

Figure 1: Sliding interface allows for the creation of a reversible, non-conformal grid interface between mesh regions with different mesh structure.

Mesh definition in OpenFOAM-2.2.x

Domain discretization within the OpenFOAM framework is based on an unstructured mesh of polyhedral cells with an arbitrary number of faces [3]. Mesh definition is enclosed in five different files:

- **points**: contains the coordinates of all mesh nodes (vertices) in form of a list. The position index of a point into the list represents its label, or addressing.

- **faces**: contains the definition of all (active) faces in form of a list of arrays. Each face is defined as a list of points, that are identified by their respective labels; the number of point constituting a face can be arbitrarily large. Their order inside the face definition identifies the direction of face normal vector, according to the right-hand rule. Again, position index of face into this file gives its addressing.

- **owner**: each face is *owned* by a cell, whose label is specified in the file *owner*. The *i*-th element of the list is the addressing of the cell that owns the face addressed by the position index *i*.

- **neighbour**: the cell adjacent to face *i* on the opposite side with respect to the owner cell is the face neighbour. The neighbor cells are specified in this file following the same logic used for owners. Not all faces have a neighbor: boundary faces have only an owner; in this case, the neighbor cell addressing is −1.

- **boundary**: contains specification of boundary “patches”, onto which the physical boundary conditions for each variable will be defined. Each patch is a list of consecutive face labels, and it is defined through the start face addressing and the number of faces the patch is composed of. Boundary faces always lie at the end of the face list, after internal faces.

Splitting the mesh definition in different files allows for reducing the storage memory overhead in case of simulation with dynamic meshes. If no topology change is involved, only the point coordinates have to be saved in the result folders; when mesh topology changes, the complete mesh definition must be stored for restart and post-processing. Note that “edges” are not considered as part of mesh definition; in fact, they can be easily inferred from faces. Also, cell topology is implicitly contained in the neighbour/owner addressing, hence reconstructing the list of faces constituting a single cell is not trivial. However, this allows for a great flexibility in cell shapes, that are not limited by a maximum number of faces.

A strict internal checking is performed every time the mesh is read or updated, to verify self-consistence of its definition. Any unused point or face, as well as open cells or connectivity inconsistencies would result in a fatal exception thrown by the code. Therefore, no ‘stand-alone’ entities can subsist.

In addition, it is possible to specify subsets of point, faces or cells called ‘zones’. This is achieved by simply storing the affected entities addresses in a file called, respectively, pointZones, faceZones or cellZones. Zones are automatically updated anytime a topological change occurs.
**Topology modifiers**

In OpenFOAM, all topological changes, including mesh layering and sliding mesh interfaces, are executed by self-contained objects (mesh modifiers) that are activated on demand whenever a topology change is requested, rather than at prescribed moments. A common interface and the use of virtual functions allows for a common top-level code and run-time selection of topology changers.

When the top-level application triggers a mesh update, a loop is performed over all modifiers; if a topological action is requested, the corresponding algorithm embedded in the specific modifier is invoked. As the loop proceeds, the modified mesh is stored in a temporary location of memory and an object called `mapPolyMesh` is created. Since points-, faces- and possibly cell-addressing change during a topological action, the `mapPolyMesh` object stores all correspondences between original and modified mesh. This is used not only to recalculate the actual FV mesh, but also to update all mesh-related information like zones, fields, other mesh modifiers, etc. Only after all mesh modifiers have been executed, the global FV mesh is changed and execution can continue.

The operation of the *sliding interface* topology changer depends on the current mesh state at the time it is triggered: if mesh regions are detached, the algorithm stitches them and vice-versa. Details of the coupling and decoupling operation, that have been implemented by the authors in the OpenFOAM®-2.2.x technology, will be given in the following paragraphs.

**Coupling algorithm**

The opposite facing surfaces that will be attached by the sliding interface are specified by the user at start time: they need to be valid patches belonging to the external surface of the mesh, they must lie on the same ideal surface (up to a tolerance) and they obviously must overlap. The user is also requested to specify which one of the surfaces is the “master” patch, while the other one will be implicitly defined as “slave”: the difference lies in the fact that the slave patch will possibly undergo point deletion to adapt to the other side, while the master patch will remain unchanged as long as possible. All the original points on the master patch are retained, some new points (coming from the slave side) might be added; for both sides the face definition changes.

When the topology modifier is triggered, the topological change proceeds automatically without further external intervention. Points belonging to the “slave” side are projected onto the “master” patch; in case of no valid projection (i.e. the two patches do not overlap), the algorithm terminates without errors. Projected point location is corrected for direct point-to-point, point-to-edge and edge-to-edge hit, eliminating all degenerate cases of intersection. If the projection is valid, i.e. there is an (even partial) overlap between slave and master patches, the remainder of the algorithm consists of enforcing the topological change on polyhedral cells and preserving full connectivity.

In case of a direct point-to-point hit (within a specified tolerance), the slave point is merged with the master and the master point is retained, while the slave point is removed. In case of a point-to-edge hit, the affected edge is split and the adjacent faces modified accordingly; the same rule applies for an edge-to-edge intersection: in this case a new point is added at the intersection. Finally, if projected point lies onto the a face, the face itself is split by adding new edges. The process is illustrated in Fig. 2. An enriched patch, containing all points and intersections of both sets of faces is assembled and used to create consistent mesh structure. Face sets are isolated using a right-hand-walk algorithm which identifies individual facets as closed loops. Afterward, original master and slave faces affected by the projection are removed and substituted by the corresponding ones coming from the enriched patch; also, each newly inserted face is linked to an owner cell (belonging to the region including the master patch) and a neighbor cell (belonging to the region including the slave patch). Faces affected by the projection but not master nor slave are called *stick-outs*: those are, for example, internal faces with an edge lying on the interface. They need to be modified too, since points might have been added to external edges.

After the coupling algorithm is completed, the resulting mesh will present a seamless junction between the former separated regions, as shown in Fig. 3. The case of many faces lying on the same cell side is allowed by the use of degenerate polyhedral cells, as shown in Fig. 3. In any case, no cells are added or removed by the algorithm, that preserves the global cell numbering.

As the different mesh regions are merged, topology modifications need to be stored, because this information is
Figure 4: Different decomposition strategies on 4 subdomains for the parallel computation of the 2-stroke engine described in Tab. 1; a) each processor computes at least one sliding interface; b) all the five sliding interfaces are computed in the same processor. Each subdomain is marked by a different color.

needed by the decoupling algorithm; hence, all point positions as well as face, edge and cell definitions as they were before the coupling must be saved before proceeding to actual topology change. This can be done by keeping all removed points and faces in the mesh definition and by marking them as “not active”, accordingly to the mesh management [5] historically adopted in the “extend” distribution of OpenFOAM [4]. In the mesh definition of the code released by OpenCFD®, stand-alone (unused) points and faces are no more included and mesh changes are stored in external files (meshModifiers). In authors’ development, modifications introduced by the coupling algorithm (removed point- and face- definitions) are then stored in the external file format required by the latest release of OpenFOAM.

The storage of removed entities for a later use has represented one of the improvement made to the original algorithm: at the moment this paper is written, the code available with the official distribution does not retain any information about removed entities, so interface coupling is always irreversible. To minimize memory consumption and storage overhead, only the changing quantities are stored. Faces and points belonging to the enriched patch created at projection time are saved in appropriate faceZones, while removed entities definitions are written to a separate meshModifiers file per each timestep. In the end, each mesh modifier contains:

- The name of the interface, with its current status (enabled/disabled, attached/detached)
- The names of master and slave patches, as well as the face- and point- zones containing the enriched patch definition
- The pairs of merged master and slave points (specified by their labels)
- The coordinates of removed slave points
- The definitions of removed master and slave faces
- The cell ownership of removed faces
- The definitions of stick-out faces before the coupling
- The list of cut edges
- The geometrical tolerances used in point projection and merging

Decoupling algorithm

Decoupling of the interface completely relies on the information saved during the previous stage. To begin with, the interface has to be cleared by removing all points and faces previously added, which have been saved as face- or point- zones. Subsequently, all slave points removed by the initial merge are brought back to life at their original coordinates. Afterward, the removed faces list is read and, on the basis of the information stored therein, master and slave original faces are reinstated; at the same time, their ownership is transferred back to original cells. In a similar manner stick-out faces are restored. After a successful decouple, no information needs to be saved but the master and slave patch labels, thus the mesh modifier is cleared out. At the moment this paper is written, a working decoupling algorithm is missing in the version of the code released by OpenCFD®.

Parallel operation of sliding interface

Parallelism is implemented in OpenFOAM by the so-called domain decomposition technique. The whole mesh is divided into several sub-domains, each assigned to a separate process. Communication between sub-domains is carried out by specific boundary conditions based on the MPI protocol: this ensures physical consistency disregarding the specific equations and models implemented in the solver.

Insertion of topology modifier in a decomposed mesh is straightforward, provided that sufficient care is taken about the decomposition strategy. In general, since the sliding interface changes the addressing and definition of faces on both its sides, it is mandatory that no inter-processor boundary interferes with it: in other words, the pair of patches defining the sliding interface must be contained in one single sub-domain, and no inter-processor boundaries can lie on the interface itself. However, there
are no limitations in the number of sliding interfaces for each sub-domain. Fig. 4-a and 4-b show an example of domain decompositions of an engine geometry. The algorithm for decomposition can be constrained to not split sliding interfaces over different sub-domains, as well as to assign each interface to a specified subdomain; on the other hand, some manual work is needed to achieve proper load balancing among processors.

Reconstruction of whole FV mesh and fields is usually requested at the end of simulation; however the point, face and cell processor addressing lists, generated at decomposition time, are no longer valid once topological changes have occurred. As a consequence, the only way to reconstruct a parallel case involving dynamic mesh with topological changes consist of using an algorithm based on geometry instead than topology. The OpenFOAM utility specifically designed for this task is `reconstructParMesh`, that reconstructs the whole mesh from the parallel decomposition and generates for each time step the addressing lists to link the meshes of the sub-domains to the global mesh. All the resolved fluid-dynamic fields (pressure, velocity, etc.) will be mapped afterward by the application `reconstructPar`. The dynamic mesh library was compiled and tested both on a PLX and on the IBM-BlueGene/Q architecture, available by CINECA (Bologna, Italy).

**VALIDATION**

The capabilities of the implemented algorithm have been verified for a number of test-cases. Here, two of them are reported, that might be seen as representative of more general classes of problems: scavenging in a two-stroke engine and non conformal meshes for in-cylinder flow calculation.

**Scavenging in a two-stroke engine**

The proposed sliding interface algorithm has been included in a specialized dynamic mesh class for the simulation of two-stroke engines. A single-cylinder SI engine, whose main characteristics are listed in Table 1, has been modeled using OpenFOAM-2.2.x.

![Simulation of a two-stroke engine with moving mesh.
When overlap between liner and ports exists and the sliding interface different fluid dynamic regions are merged. Patches keep their original definition of solid walls otherwise. Master and slave patches of the sliding interfaces are colored, with red and blue color respectively (right column).](image)

<table>
<thead>
<tr>
<th>Bore</th>
<th>100 mm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stroke</td>
<td>95 mm</td>
</tr>
<tr>
<td>Compression Ratio</td>
<td>20</td>
</tr>
<tr>
<td>Effective Comp. Ratio</td>
<td>13.2</td>
</tr>
<tr>
<td>Displacement</td>
<td>746 cm³</td>
</tr>
<tr>
<td>Squish distance</td>
<td>5 mm</td>
</tr>
<tr>
<td>Speed</td>
<td>2500 rpm</td>
</tr>
<tr>
<td>Boost pressure</td>
<td>1.05 bar</td>
</tr>
</tbody>
</table>

Table 1: Geometry of the two-stroke engine. [6]

The combustion chamber has one exhaust port with elliptical cross section, and four scavenging ports with approximately rectangular cross sections. No valves are present between ports and cylinder, so timing is determined by piston motion and port position. Dynamic cell layering is adopted near the piston surface, so that only cells lying on the piston surface are deformed by the moving mesh algorithm, thus preserving mesh quality in the rest of the domain. At the same time, five sliding interfaces are defined in correspondence of engine ports.

The dynamic mesh topology is represented in Fig. 5 for three crank angles (respectively 60°, 110° and 180° ATC). In the first image of each pair, the mesh connectivity is highlighted: separate fluid region are identified by different colors. As long as all scavenging port are closed (e.g. at 60° ATC, first row), six unconnected regions can be found in the mesh. At 110° ATC, the piston opens the exhaust port: combustion chamber and exhaust duct are now part of the same region. Finally, at BDC, all ports are opened and only one fluid region is present.

In the right column of Fig. 5, the patches defining the interfaces are highlighted, with red and blue color representing, respectively, master and slave sides. If no actual
overlap exist between liner and ports, the sliding interface is not activated and all patches keep their original definition as solid walls (this is the case, e.g., of 60° ATC). If projection leads to a partial superposition (like at 110° ATC), patches are partially modified and some original faces on both sides are removed and substituted by internal faces. At BDC, projection of slaves onto masters is full: slave patches are removed and substituted by internal faces; some master faces are still unaffected because they lie outside of the projected area.

Domain decomposition used for parallel runs can be seen in Fig. 4, each color representing a subdomain. Decomposition method used for this case is the constrained scotch algorithm. The first decomposition of Fig. 4 was obtained by specifying not only that sliding interfaces must not be decomposed, but also the specific processor they had to be assigned to. On the second image, the last constraint is removed, so a more balanced decomposition has been achieved.

**Non conformal meshes for in-cylinder flow calculation**

Block-structured hexahedral meshes are usually the preferred choice for FV grids, since they represent the best compromise between accuracy and simplicity [7]. However, a complete block-structured mesh is often nearly impossible to generate for very complex geometries. Moreover, local mesh refinement needed e.g. by solid walls or small geometrical details affects grid spacing even in distant regions, thus increasing significantly the total number of elements. A sliding interface can be used to create a static link between mesh regions with non-conformal interfaces, thus making grid spacing of one region independent from the others. In this particular case, the global mesh is generated with two or more fluid regions, reciprocally separated by master/slave patch pairs. Sliding interface mesh modifiers are then created, and they are activated only once during the preprocessing stage to perform the “couple” operation. After that, all mesh regions are physically connected and the simulation may start. Since in principle there is no need to bring back the mesh to the split configuration, all informations saved during the couple stage can be cleared.

This strategy has been applied to in-cylinder cold flow LES simulation of an engine-like geometry, that also has been investigated in [9]. In Fig. 6-b, the detail of discretization around the valve seat is shown. To reduce the computational cost, a relatively large cell size has been adopted for the cylinder region. At the same time, to correctly account for near-wall effects, much smaller cells are used in the valve seat and in proximity of the valve bottom plate. Fig. 6-a reports the isosurfaces of the velocity field at $\theta = 440^\circ$. The position of the sliding interface is showed in Fig. 6-b. A detailed description of the numerical setup and of the physical models used in the calculations is reported in [8, 10].

**AUTOMATIC MESH GENERATION OF ENGINE GEOMETRIES**

When multiple meshes are needed in engine simulation, a lot of manual work is required and the time needed for the setup significantly increases. In steady flow simulations, cylinder grids must be generated for each valve lift; in full-cycle engine simulation meshes are generated for different crank angles, where flow fields are remapped onto. In order to automate the meshing process, the prescribed motion of moving boundaries (valves and pistons) has been developed to work also on geometries imported by a triangulated surface representation. The proposed strategy is based on two different tools: the application `surfaceEngineCreate`, developed by the authors, and the `snappyHexMesh` utility of OpenFOAM, that automatically generates 3D meshes of hexahedra and split-hexahedra starting from an initial fully hexahedral block mesh and according to the provided STL geometry.

![Figure 6: Cold flow in-cylinder LES simulation by OpenFOAM [8]. a) Isosurfaces of velocity field; b) detail of the non-conformal interface between the valve seat and the cylinder region where the sliding interface is required.](Image)

The application `surfaceEngineCreate` is able to merge different triangulated surface geometries, if needed. The STL file describing the engine includes different patches, that are labeled to allow the subsequent set up of the boundary conditions during mesh generation. In the initial geometry, both valves are closed and the piston is moved to the top-dead center. Engine geometry for tests was kindly provided by Centro Ricerche Fiat (CRF), Turin, Italy.

The application `surfaceEngineCreate` is able to merge different triangulated surface geometries, if needed. The STL file describing the engine includes different patches, that are labeled to allow the subsequent set up of the boundary conditions during mesh generation. In the initial geometry, both valves are closed and the piston is moved to the top-dead center. Engine geometry for tests was kindly provided by Centro Ricerche Fiat (CRF), Turin, Italy.

Figure 7: In the STL cylinder geometry, created or provided to `surfaceEngineCreate`, patches are labeled, to favor the subsequent automatic set up of the boundary conditions during mesh generation. In the initial geometry, both valves are closed and the piston is moved to the top-dead center. Engine geometry for tests was kindly provided by Centro Ricerche Fiat (CRF), Turin, Italy.
- steady-state engine simulation: different lifts of the intake (or exhaust) valve are generated. In this case, an arbitrary cylinder length will be assumed and the piston surface will be replaced by a flat surface representing the outlet;
- full-cycle engine simulation: surfaces at different crank angle positions are generated (see Fig. 8); valves and piston position are defined accordingly to the valve timing defined by a dictionary.

Figure 8: Full-cycle engine simulation: mesh motion strategy applied on moving boundaries of a triangulated surface representation (STL). Piston surface is moved for each crank angle, while valves are automatically moved along the axial direction accordingly to the prescribed valve motion provided by a dictionary. Engine geometry for tests was kindly provided by Centro Ricerche Fiat.

Starting from the geometries imported in STL format, the OpenFOAM application \texttt{snappyHexMesh} makes use of a background mesh of hexahedral cells that fills the entire region within the external boundary; the initial hexahedral mesh is snapped to the STL surface by iteratively refining the starting block mesh and by morphing the resulting split-hex mesh onto the surface; different refinement level can be set on the patches of the STL surface. The specification of mesh refinement level is very flexible and the surface handling is robust with a fine final mesh quality, which allows a good control on cells parameters: orthogonality, skewness, concavity, min/max volume. Finally, an optional phase will shrink back the resulting mesh and insert a desired number of cell layers. Fig. 9 shows the detail of the mesh generated by \texttt{snappyHexMesh} near the cylinder valve of the engine used for testing, that was kindly provided by Centro Ricerche Fiat (CRF).

Figure 9: Detail of the mesh generated by \texttt{snappyHexMesh} near the cylinder valve of the engine used for testing. Engine geometry for tests was kindly provided by Centro Ricerche Fiat.

The described procedure is configured to work both off-line, as a preprocessing tool, but its fully object oriented implementation allows for an easy extension to be performed runtime, when full cycle simulation is performed.

CONCLUSIONS

A novel implementation of the sliding interface algorithm in OpenFOAM, to handle motion and topological changes in a moving-mesh FV Method, has been implemented and tested; the implementation is compatible with the mesh definition employed in the versions of OpenFOAM released by OpenCFD®. Code validation of the moving mesh algorithms has been performed on two stroke engine-simulation and on cold-flow LES simulation of IC engines, that is current main topic of the current authors' work.

ACKNOWLEDGMENTS

Authors would like to acknowledge Dr. Giorgio Carpegna from Centro Ricerche FIAT, Turin (Italy), for providing experimental data and Mr. Marco Fiocco, graduate research assistant at the ICE-PoliMi group, for his help during the development of the algorithm for automatic meshing. The geometry of the 2-stroke engine was provided by Dr. Tommaso Lucchini, that is kindly acknowledged. Algorithms ran on the computational resources made available by CINECA (Bologna, Italy).

CONTACT

Prof. Federico Piscaglia, Ph.D.
Politecnico di Milano
ph. (+39) 02 2399 8620
e-mail: federico.piscaglia@polimi.it

REFERENCES


