The 8th European Congress on Computational Methods in Applied Sciences and Engineering
ECCOMAS Congress 2022
5--9 June 2022, Oslo, Norway

IMMERSED BOUNDARIES IN HYPERSONIC FLOWS WITH CONSIDERATIONS ABOUT HIGH-FIDELITY AND MASSIVE PARALLELISM

F. NAULEAU[1], T. BRIDEL-BERTOMEU[2], H. BEAUGENDRE[3] AND F. VIVODTZEV[4]

[1] CEA-CESTA, Le Barp, France, florent.nauleau@cea.fr

[2]CEA-CESTA, Le Barp, France, thibault.bridelbertomeu@cea.fr

[3] Univ. Bordeaux, INRIA, CNRS, Bordeaux INP, IMB, UMR 5251, F-33400, Talence, France, heloise.beaugendre@math.u-bordeaux.fr

[4] CEA-CESTA, Le Barp, France, fabien.vivodtzev@cea.fr

Key words: Hypersonics, Immersed Boundaries, Rasterization, Migratable Tasks, LES

Abstract. This paper inscribes itself in the ongoing doctoral work of the first author which aims at adapting the immersed boundary conditions (IBC) technique to three-dimensional (3D) large eddy simulations (LES) of viscous hypersonic flows around complex vehicles. The work relies on a pre-existing in-house IBC code, HYPERION (HYPERsonic vehicle design using Immersed bOuNdaries), originally developed in two dimensions (2D) as a proof of concept that it is possible to use IBC in the presence of strongly shocked flows [4].

As a first step towards the optimization of the 3D HYPERION, we discuss in this paper a novel MPI/Open MP hybrid rasterization algorithm allowing for the detection of immersed cells in record time even for very large problems.

We then consider the least-square-based reconstruction algorithm from HYPERION [4]. It was shown in the original paper that the number of neighbors used in the reconstruction is directly related to the condition number of the least-square matrix and an optimum can be found when the condition number reaches an asymptote. In 3D configurations it is found that the number of neighbors has to be very high to ensure the proper conditioning of the least-square matrix. If the computation is distributed on several MPI processes (as is always the case in 3D for realistic return times), gathering the information from that many neighbors can cause obvious communication issues - it amounts to covering large stencils with unrealistically large MPI halos. We therefore introduce an algorithm designed for a hybrid MPI/OpenMP environment based on migratable tasks and the consensus algorithm developed by [9] to remedy the former shortcoming.

Finally, we discuss the premise of the implementation of LES capabilities in HYPERION. The last milestone of the main author's doctoral work is indeed to study the feasibility of embedding wall laws in the IBC modeling and reconstruction algorithm to try and counteract the low accuracy of the near-wall phenomena caused by the lack of body-fitted mesh.

1 INTRODUCTION

Numerical simulations of hypersonic flows have been supplementing experimental wind-tunnel testing for many years. Today flow simulations in the high-supersonic and/or hypersonic regimes are becoming a common sight both in the academic and in the industrial worlds and in many domains [24, 38, 5, 15, 3]. One of the most important ingredients to obtain accurate results from such simulations is commonly found to be the mesh. However, the cost of generating a body-fitted mesh around complex geometries such as those used for hypersonic planes, blast-resistant building, etc. . . can be high, and it is still today a trending topic to accelerate and improve the robustness of the process. In the context of computer-aided end-to-end vehicle development, numerous computations have to be made, iterating on successive designs, and the days taken to make the mesh for every iteration can amount to a non-negligible part of the allotted project time. The present paper inscribes itself in the doctoral work of the first author and follows the work of the second author [4] that aims at showing that it is possible to use an immersed boundary technique to cut the mesh generation time, and produce a piece of software that is robust, efficient and accurate enough to handle viscous hypersonic flows.

In this short study, we investigate how to enable massively parallel simulations with an immersed boundary-based CFD code. Based on preliminary results from the code used in Bridel-Bertomeu study [4], we propose to overcome two major obstacles on the path towards massive parallelism. The first obstacle concerns the manipulation of the objects themselves, when immersed in a Cartesian mesh. For extremely large Cartesian meshes and finely tesselated objects, the different ray-casting algorithms mentioned in [4, 33, 23] prove to have too high an algorithmic complexity to be reasonably use in three-dimensional simulations. The second obstacle we propose a solution to, in the present study, occurs during the interpolation stage of the immersed boundary workflow. At each iteration of the fluid solver, when the values to be set in the immersed ghost cells are to be computed, the multidimensional interpolation process requires data from, potentially, an elevated number of neighbors. In the context of a partitioned mesh and distributed parallelism, accessing neighbors that are a few cell layers away can be challenging and can cause an unacceptable overhead in computational time due to the increased number of communications between neighboring processes.

With the two algorithms we will introduce, we will show that we finally make accessible massively parallel simulations with HYPERION on thousands of distributed processes. This will lead to a discussion and an investigation of the predictive capabilities of HYPERION in three dimensions. The investigation is ongoing but some elements for implementing a large eddy simulation (LES) framework in HYPERION will be discussed, and some preliminary results will be shown.

2 HYPERION IN A FEW WORDS

The code used in this study, *i.e.* HYPERION, solves the three-dimensional compressible Navier-Stokes equations with source term. A dimensionless formulation is used to minimize the stiffness of the system in high-Mach & high-Reynolds numbers regimes. Mathematically, the system can be written as follows [32]:

$$\mathbf{U}_t + \mathbf{F}_x + \mathbf{G}_y + \mathbf{H}_z = \mathbf{E}_x^{v,x} + \mathbf{E}_y^{v,y} + \mathbf{E}_z^{v,z} + \mathbf{S}, \tag{1}$$

where the subscripts indicate differentiation, $\mathbf{U}$ is the vector of conservative dimensionless variables and $\mathbf{F}$ ($\mathbf{E}^{v,x}$), $\mathbf{G}$ ($\mathbf{E}^{v,y}$) and $\mathbf{H}$ ($\mathbf{E}^{v,z}$) represent the inviscid (viscous) fluxes in $x-$, $y-$ and $z-$direction respectively and $\mathbf{S}$ represents a volumic source term. Those vectors are defined as such:

$$\mathbf{U} = \begin{bmatrix} \rho \\ \rho u \\ \rho v \\ \rho w \\ \rho E \end{bmatrix}, \ \mathbf{F} = \begin{bmatrix} \rho u \\ \rho u^2 + p \\ \rho u v \\ \rho u w \\ \rho u E + p u \end{bmatrix}, \ \mathbf{G} = \begin{bmatrix} \rho v \\ \rho v u \\ \rho v^2 + p \\ \rho v w \\ \rho v E + p v \end{bmatrix}, \ \mathbf{H} = \begin{bmatrix} \rho w \\ \rho w u \\ \rho w v \\ \rho w^2 + p \\ \rho w E + p w \end{bmatrix},$$

$$\mathbf{E}^{v,x} = \begin{bmatrix} 0 \\ \tau_{xx} \\ \tau_{yx} \\ \tau_{zx} \\ \tau_{xx}u + \tau_{xy}v + \tau_{xz}w - q_x \end{bmatrix}, \ \mathbf{E}^{v,y} = \begin{bmatrix} 0 \\ \tau_{xy} \\ \tau_{yy} \\ \tau_{zy} \\ \tau_{xy}u + \tau_{yy}v + \tau_{yz}w - q_y \end{bmatrix}, \tag{2}$$

$$\mathbf{E}^{v,z} = \begin{bmatrix} 0 \\ \tau_{xz} \\ \tau_{yz} \\ \tau_{zz} \\ \tau_{zx}u + \tau_{zy}v + \tau_{zz}w - q_z \end{bmatrix}.$$

In the above dimensionless expressions, t denotes the time and x, y and z are the Cartesian coordinates. ρ denotes density, u, v and w denote the $x-$, $y-$ and $z-$direction velocity components respectively, E denotes the specific total energy and p denotes the static pressure. τ is the viscous stress tensor and $\mathbf{q}$ the heat flux vector. Here a simple perfect gas is considered and therefore the specific total energy can be related to the other variables using:

$$E = \frac{1}{\gamma - 1}\frac{p}{\rho} + \frac{1}{2}\left(u^2 + v^2\right), \tag{3}$$

where γ is the ratio of specific heats and if not specified otherwise we will take $\gamma = 1.4$ in the rest of this study. As is often done for high-Mach number compressible flow solvers, we here use the freestream speed of sound a^*_∞ to nondimensionalize the velocity. The relations between the dimensionless and the dimensional variables are therefore:

$$\rho = \frac{\rho^*}{\rho^*_\infty}, \ p = \frac{p^*}{\rho^*_\infty a^{*,2}_\infty}, \ [u,v,w] = \left[\frac{u^*}{a^*_\infty}, \frac{v^*}{a^*_\infty}, \frac{w^*}{a^*_\infty}\right], \ [x,y,z] = \left[\frac{x^*}{L}, \frac{y^*}{L}, \frac{z^*}{L}\right], \ t = \frac{t^*}{L/a^*_\infty}, \tag{4}$$

where the starred variables denote dimensional variables and L is a reference length. With this formulation, and assuming the fluid is Newtonian, has zero bulk velocity (Stokes' hypothesis) and is only subject to heat diffusion, the viscous stress tensor and the heat flux vector can be written as:

$$\begin{aligned} \tau &= \hat{\mu}\left[-\frac{2}{3}(\nabla\cdot\mathbf{v})\mathbb{I} + \nabla\mathbf{v} + (\nabla\mathbf{v})^{\mathrm{t}}\right], \\ \mathbf{q} &= -\frac{\gamma\hat{\mu}}{(\gamma-1)Pr}\nabla\left(\frac{p}{\rho}\right), \end{aligned} \tag{5}$$

where the scaled dimensionless dynamic viscosity $\hat{\mu}$ is defined by:

$$\hat{\mu} = \frac{\mathrm{Ma}\,\mu}{\mathrm{Re}}, \tag{6}$$

with Ma the Mach number of the freestream and Re its Reynolds number. In this work, the dimensionless dynamic viscosity μ is assumed to depend on the dimensionless temperature $T = \gamma p/\rho$ by Sutherland's law (see *e.g.* [39]):

$$\mu = \frac{1+\dfrac{C}{T_{\mathrm{ref}}}}{T+\dfrac{C}{T_{\mathrm{ref}}}}T^{\frac{3}{2}}, \tag{7}$$

where $C = 110.4$ K and T_{ref} is the freestream temperature (in K). A detailed account of common nondimensionalization for the Euler and Navier-Stokes equations can be found, among others, in [32].

2.1 Cartesian fluid domain discretization

In this particular study, we use only Cartesian grids with constant grid spacings Δx, Δy and Δz. Based on such a mesh, the finite-volume method [26] is employed for space discretization of the compressible Navier-Stokes equations (1). An exhaustive description of the technique adapted to Cartesian grids is given in [4]: we shall only present here a few select details to expose the core of HYPERION.

2.1.1 Mixed finite-volume/finite-difference numerical scheme

In HYPERION, when solving numerically the Navier-Stokes equations (1), we mix a finite-volume flux-balance formulation for the hyperbolic terms ($\mathbf{F}_x$, $\mathbf{G}_y$ and $\mathbf{H}_z$) and a finite-difference formulation of the gradients involved in the parabolic terms ($\mathbf{E}^v_{x,y,z}$) whereas the (optional) source terms are computed directly at the cell centers. The flux-balance is obtained with numerical fluxes computed at each face of each cell using an approximate Riemann solver [37] relying on left- and right-interpolated values from the neighboring cells. The mathematical expression of this mix formulation, as well as a selection of reconstructions and Riemann solvers present in HYPERION will be discussed in the following paragraphs.

Considering that in the present case all cells are quadrangles, the flux-balance can be split by direction, and the final mathematical expression of the space discretization used in HYPERION can be written in cell c as:

$$\begin{aligned} \frac{d\overline{\mathbf{U}}_c}{dt} = &-\frac{\hat{\mathbf{F}}_{i+1/2,j,k}-\hat{\mathbf{F}}_{i-1/2,j,k}}{\Delta x} - \frac{\hat{\mathbf{G}}_{i,j+1/2,k}-\hat{\mathbf{G}}_{i,j-1/2,k}}{\Delta y} - \frac{\hat{\mathbf{H}}_{i,j,k+1/2}-\hat{\mathbf{H}}_{i,j,k-1/2}}{\Delta z} \\ &+ \mathcal{D}_x\left(\mathbf{E}^{v,x}_c\right) + \mathcal{D}_y\left(\mathbf{E}^{v,y}_c\right) + \mathcal{D}_z\left(\mathbf{E}^{v,z}_c\right) + \mathbf{S}_c \end{aligned} \tag{8}$$

where i, j and k are the indices of cell c along each direction, and $i \pm 1/2$, $j \pm 1/2$ and $k \pm 1/2$ represent the faces of the cell in each direction as well.

The vector $\hat{\mathbf{F}}$ (resp. $\hat{\mathbf{G}}$, $\hat{\mathbf{H}}$) is a numerical approximation of the inviscid physical flux $\mathbf{F}$ (resp. $\mathbf{G}$, $\mathbf{H}$) at the faces of the cell of interest and can be expressed in a most general manner as:

$$\hat{\mathbf{F}}_{i\pm1/2,j,k} = \mathcal{U}\left(\mathbf{F}^L_{i\pm1/2,j,k}, \mathbf{F}^R_{i\pm1/2,j,k}, \overline{\mathbf{U}}^L_{i\pm1/2,j,k}, \overline{\mathbf{U}}^R_{i\pm1/2,j,k}\right), \tag{9}$$

where $\mathcal{U}$ stands for a generic approximate Riemann solver while the superscripts L and R designate respectively the left- and right-biased reconstruction of either $\mathbf{F}$ or $\overline{\mathbf{U}}$ at the faces of the cell of interest. A similar expression stands for $\hat{\mathbf{G}}_{i,j\pm1/2,k}$ and $\hat{\mathbf{H}}_{i,j,k\pm1/2}$.

The derivatives involved in $\boldsymbol{\tau}$ and $\mathbf{q}$ are computed as follows for any variable φ in cell (i,j,k) and along any dimension $d \in \{x,y,z\}$:

$$\frac{\partial \varphi_{i,j,k}}{\partial d} = \mathcal{D}_d\left(\varphi_{i,j,k}\right), \tag{10}$$

where $\mathcal{D}_d$ represents a centered finite-difference-like differentiation operator along dimension d. In this study we use either a second- or a fourth-order operator, respectively denoted $\mathcal{D}_d^{(2)}$ and $\mathcal{D}_d^{(4)}$. They are defined as:

$$\mathcal{D}_d^{(2)}(\varphi_{i,j,k}) = \frac{\varphi_{i+1,j,k} - \varphi_{i-1,j,k}}{2\Delta d}, \tag{11}$$

and

$$\mathcal{D}_d^{(4)}(\varphi_{i,j,k}) = \frac{\varphi_{i-2,j,k} - 8\varphi_{i-1,j,k} + 8\varphi_{i+1,j,k} - \varphi_{i+2,j,k}}{12\Delta d}. \tag{12}$$

2.1.2 Reconstruction schemes

As mentioned in introduction to section 2, HYPERION is designed to provide numerical predictions of viscous compressible flows. In particular, we are interested in hypersonic regimes where the Mach number and the Reynolds number are very high. In layman terms, the flow will therefore exhibit very strong large scale discontinuities (shock & contact waves) as well as small scale viscous eddies - HYPERION has to capture both. To do so, we turned to high-order reconstructions (to capture the smallest scales) with good properties in the presence of discontinuities, especially that have a non-oscillatory property preventing too strong over- and under-shoots in the vicinity of discontinuities.

The well-known family of WENO-Z schemes[1] was added to HYPERION [29, 22, 21, 19] for their robustness and their theoretical high-order. Satisfactory results were obtained in the hypersonic regime, although the viscous small scales tended to be too dissipated.

This last comment pushed us to implement and use the newer generation of TENO [12, 20] schemes[2] that allow for better discrimination between the large discontinuous events and the small eddies produced by

[1]WENO stands for Weighted Essentially Non-Oscillatory.
[2]TENO stands for Targeted Essentially Non-Oscillatory.

viscous turbulence. The interested reader is referred to the family of papers by Fu *et al.* for more details [12, 13, 14, 10, 11], while we only recall below the 5th-order formulation that we make most use of.

$$\begin{aligned}
& \omega_1 \times \left[\overline{\mathbf{U}}^{L,1}_{i+1/2,j,k} = \frac{1}{6}\left(-\overline{\mathbf{U}}_{i-1,j,k} + 5\overline{\mathbf{U}}_{i,j,k} + 2\overline{\mathbf{U}}_{i+1,j,k}\right)\right] \\
\overline{\mathbf{U}}^{L}_{i+1/2,j,k} = & + \omega_2 \times \left[\overline{\mathbf{U}}^{L,2}_{i+1/2,j,k} = \frac{1}{6}\left(2\overline{\mathbf{U}}_{i,j,k} + 5\overline{\mathbf{U}}_{i+1,j,k} - \overline{\mathbf{U}}_{i+2,j,k}\right)\right] \\
& + \omega_3 \times \left[\overline{\mathbf{U}}^{L,3}_{i+1/2,j,k} = \frac{1}{6}\left(2\overline{\mathbf{U}}_{i-2,j,k} - 7\overline{\mathbf{U}}_{i-1,j,k} + 11\overline{\mathbf{U}}_{i,j,k}\right)\right]
\end{aligned} \tag{13}$$

where $\omega_{1,2,3}$ are nonlinear weights. To define ω_k we need to calculate the γ_k, *a.k.a.* the smoothness indicators:

$$\gamma_k = \left(C + \frac{\tau_k}{\beta_k + \varepsilon}\right)^q \tag{14}$$

where C is set to 1, q is hardcoded to 6, and ε is an input parameter typically kept very small ; in this study we use 10^{-6}. The per-stencil finite-difference operators β_k are given by:

$$\begin{aligned}
\beta_1 = & \quad \frac{1}{4}(\mathbf{f}_{j-1} - \mathbf{f}_{j+1})^2 + \frac{13}{12}(\mathbf{f}_{j-1} - 2\mathbf{f}_j + \mathbf{f}_{j+1})^2 \\
\beta_2 = & \quad \frac{1}{4}(3\mathbf{f}_j - 4\mathbf{f}_{j+1} + \mathbf{f}_{j+2})^2 + \frac{13}{12}(\mathbf{f}_j - 2\mathbf{f}_{j+1} + \mathbf{f}_{j+2})^2 \\
\beta_3 = & \quad \frac{1}{4}(\mathbf{f}_{j-2} - 4\mathbf{f}_{j-1} + 3\mathbf{f}_j)^2 + \frac{13}{12}(\mathbf{f}_{j-2} - 2\mathbf{f}_{j-1} + \mathbf{f}_j)^2
\end{aligned} \tag{15}$$

and

$$\tau_k = \left|\beta_k - \frac{1}{6}(\beta_2 + \beta_3 + 4\beta_1)\right|, \; k = 1,2,3. \tag{16}$$

Now we need to also define the scale-separator parameter χ_k (on which we rely most to discriminate between the large-scale discontinuities and the small-scale eddies) and the sharp cut-off function δ_k (on which we rely to handle large-scale discontinuities):

$$\chi_k = \frac{\gamma_k}{\sum_{j=1}^{3} \gamma_j}, k = 1,2,3 \tag{17}$$

$$\delta_k = 0 \quad \text{if} \quad \chi_k < C_T \quad \text{otherwise} \quad \delta_k = 1, \; k = 1,2,3, \tag{18}$$

where C_T is a very small value - here, 10^{-5}. With all this, the nonlinear weights ω_k can be defined as:

$$\omega_k = \frac{a_k}{\sum_{j=1}^{3} a_j}, \; k = 1,2,3 \tag{19}$$

where

$$a_1 = \frac{6}{10}\delta_1, \quad a_2 = \frac{3}{10}\delta_2, \quad a_3 = \frac{1}{10}\delta_3. \tag{20}$$

Note that to deal with the large stencil close to the edges of the Cartesian domain, HYPERION uses ghost cells - an illustration in two dimensions is provided on figure 1.

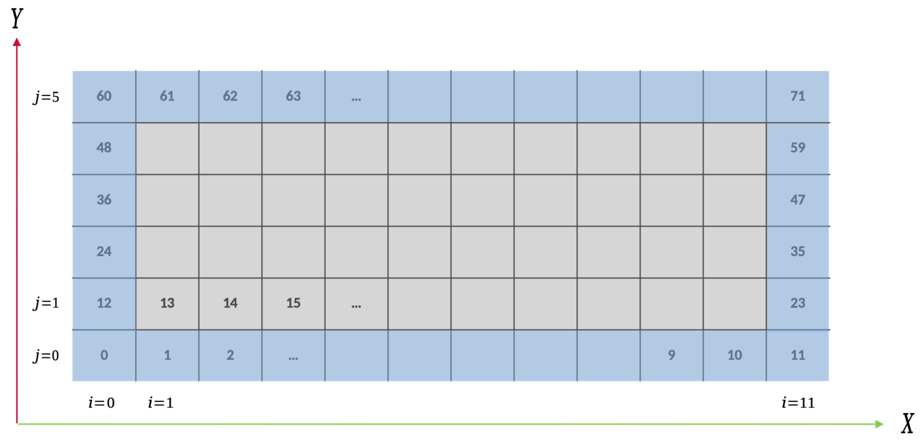


Figure 1: Illustration of the ghost cells layer (blue cells) around the Cartesian grid in HYPERION. The numbering of the cells is also shown. Note that the three-dimensional Cartesian meshes are handled in the same fashion, with index i in the X direction evolving the fastest, followed by j in the Y direction, then k in the Z direction.

2.1.3 Approximate Riemann solvers

Once the fields have been reconstructed at each face of each cell (see paragraph 2.1.2 above), the left/right discontinuous problem (see equation (9)) is solved approximately using a Riemann solver. The most used approximate solvers can be grouped in three large families: Flux Difference Splitting (FDS), Flux Vector Splitting (FVS) and Flux Type Splitting (FTS) [37, 34]. In HYPERION we focus on two types of solvers in particular, FDS solvers that work as a finite volume method to solve the Riemann problem and FVS Riemann solvers that combine the qualities of the other two families by separating kinematic and acoustic scales.

Several approximate Riemann solvers have therefore been implemented in HYPERION to serve different purpose. In the present study, similarly to the need for a reconstruction able to handle large and small scales simultaneously, the Riemann solver we were seeking had to have the ability to behave well in both high and low-Mach regimes (see *e.g.* [31]) and to not trigger the well-known carbuncle instability[25]. A series of tests and a bibliographic study lead us to opt in for a FVS solver constructed by Liou *et al.*, the AUSM^{+}-up [27, 28] because of its robustness and its increased level of accuracy at all speeds. Presenting the details of the mathematical implementations is out of the scope of this paper but the interested reader is referred to the aforementioned papers.

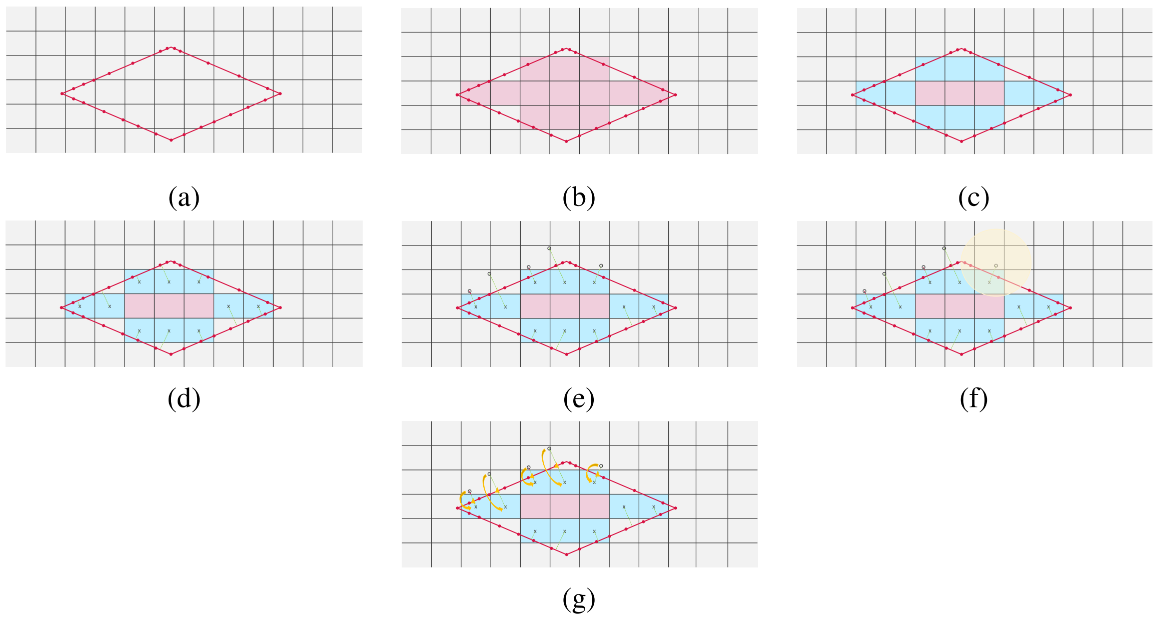


Figure 2: Immersed boundary method workflow - diagrams are shown in two dimensions but the extension to three dimensions in straightforward. (a) Introduction of a tesselated object in the Cartesian mesh. (b) Detection of the immersed cells (red), *i.e.* solid cells, at initialization. (c) Detection of the immersed boundary cells (blue), *i.e.* solid cells with at least one fluid neighbor within the extent of the stencil, at initialization. (d) Detection of the nearest facet to each immersed boundary cell, at initialization (e) Creation of the image points in the direction normal to the nearest facet for each immersed boundary cell, at initialization. (f) Illustration of the neighborhood (yellow) where fluid cells are queried for information to interpolate the values of the fields at the image points, at each iteration of the fluid solver. (g) Illustration of the linear extrapolation from the image points to the immersed boundary cells to fill the values allowing for the enforcement of the boundary condition at the wall of the immersed object, at each iteration of the fluid solver.

2.2 Sharp immersed boundary conditions

To handle the presence of obstacles in the compressible flows, HYPERION uses a sharp immersed boundary method (IBM) [33]. As mentioned in many papers, *e.g.* [23, 35], the sharp interface method proves to be well suited for compressible flows because the boundary conditions at the immersed boundary are taken into account directly rather than being computed indirectly *via* a forcing term or smoothed with a distribution function. Originally, to handle boundary conditions at the edges of the domain, HYPERION uses ghost cells so there is no need to degenerate the reconstruction stencils at the edges. To make use of the original data structures and logic of implementation as much as possible, we implemented a ghost-cell based immersed boundary method as well. In the present study, we assume that the immersed objects cannot move.

We will briefly introduce in this section the workflow for the initialization of the immersed boundary method as it will serve as reference for future sections 3 and 4. Overall the workflow is fairly classical [33, 6, 23, 35, 40, 41] and relies on six main steps:

- detection of the immersed cells, *i.e.* all the cells of the Cartesian mesh that find themselves *inside* the immersed object - figure 2(b),

- detection of the immersed boundary cells, that is immersed cells with at least one fluid neighbor within the extent of the reconstruction stencil, and wherein we shall impose the right values to enforce the boundary condition at the wall of the immersed object - figure 2(c),
- detection of the nearest facet to each immersed boundary cell in the sense of normal projection distance - figure 2(d),
- computation of the images of each immersed boundary cell center with respect to their nearest facet - figure 2(e),
- interpolation of the fluid variables at the image points - figure 2(f),
- linear extrapolation from the image points to the immersed boundary cells to enforce the boundary condition at the wall of the immersed object - figure 2(g).

In HYPERION, since we assume that all immersed objects are immovable, the four first steps can be done once during the initialization of the computation, whereas the last two steps, depending on the instantaneous state of the fluid, are repeated at each fluid iteration.

Section 3 will present the novel algorithm used in HYPERION to make the detection of the immersed cells (step 1) particularly inexpensive, even for three-dimensional objects and Cartesian meshes. Step 2 and 4 are quite straightforward and rely on simple geometrical notions ; to make step 3 efficient however for large Cartesian meshes and finely discretized immersed objects, HYPERION builds a spatial-median Bounding Volume Hierarchy (BVH) of the immersed objects to accelerate the ray-tracing-like queries for nearest facets[3]. Step 5 relies on the ENO-like least-square interpolation algorithm developed by one of the authors [4], and section 4 shall detail the strategy implemented in HYPERION to handle the large interpolation neighborhood in a massive parallelism context. Finally, step 6 is straightforward as well and the interested reader can find all the mathematical details in Bridel-Bertomeu [4].

3 A FAST RASTERIZATION ALGORITHM TO DETECT IMMERSED CELLS

To identify the solid cells, *i.e.* the cells of the Cartesian mesh that are found *inside* the immersed object, the most common algorithm is a somewhat brute-force ray-casting algorithm (see *e.g* [18, 33]) that can be described as follows.

From all cells in the Cartesian mesh a random ray is cast (see for example cells A and B in figure 3, but the ray could go any direction) and the intersections between this ray and the facets of the body are counted. If the number of intersections is odd, then the cell wherefrom the ray is cast is inside the immersed body (for example cell B) whereas if the number of intersections is even, the originating cell is outside the immersed body (for example cell A). The complexity of this algorithm is $O(N_c N_f)$, where N_c is the total number of cells in the Cartesian mesh and N_f is the total number of facets of the tesselated immersed body.
For two-dimensional problems, such a complexity hardly becomes an issue. As an illustration, let us nonetheless consider a three-dimensional worst case with a Cartesian mesh of $900 \times 900 \times 1300$ cells and an object with approximately 5×10^5 facets, then the algorithm runs for more than 48 hours on a mid-2017 Intel i7 processor with 4 OpenMP threads. Naturally if the Cartesian mesh is partitioned, then

[3]"Find Closest Point on (Tesselated) Surface" library developed by *ingowald* as a spin-off of the OSPRay project - see `github.com/ingowald/closestSurfacePointQueries`

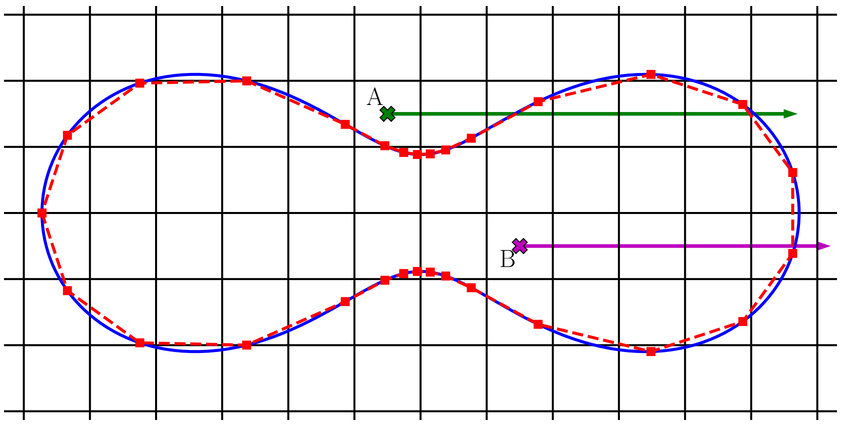


Figure 3: Illustration of the tesselation of a two-dimensional curve and of the simple ray-casting algorithm for the identification of immersed cells.

each partition handles its own set of rays and the time-to-completion drops to the order of magnitude of tens of minutes, but it still is not a reasonable performance - especially if one thinks of the possibility of having a moving object, for which the solid cells might change at each iteration of the fluid solver.

The technical name of the ”problem that consists in finding the solid cells” is *point classification*: in our case, we want to classify the cell centers according to whether they are inside or outside some shell. *Point classification* problems fall within the theory of computational geometry, which pushed the second author to associate himself with David Eberly [36] from Geometric Tools[4] to develop a better algorithm than the aforementioned brute-force one - this section is dedicated to the presentation of this new algorithm.

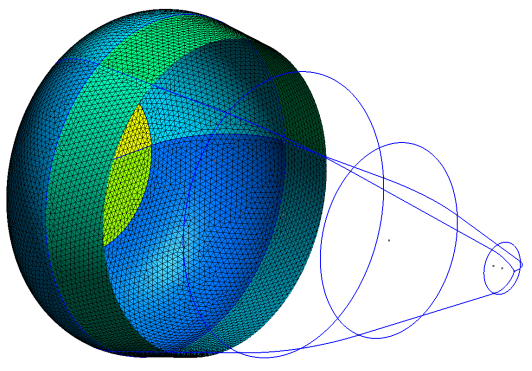
The ray-casting algorithm simply needs the immersed object to be defined by the discretization of its outermost shell - a segment-based tesselation in two dimensions or a triangle-based tesselation in three dimensions. The new algorithm relies on a tesselation of the outermost shell *and* on the existence of a tetrahedralization (triangulation) of the inside of the object in three (two) dimensions - this shift in paradigm is illustrated on figure 4.

Once a proper tetrahedralization (triangulation) has been obtained, the algorithm has the following steps - note that a description is provided in the three-dimensional case but the algorithm applies similarly in two dimensions.

- Step 1 - for each tetrahedron, create its smallest axis-aligned bounding box (AABBs).

```
function compute tetrahedra AABB
    for each tetrahedron t
        v1, v2, v3, v4 := the four vertices of the t
        t_min := v1
        t_max := t_max
        for each vertex v
            for each dimension d
                if coordinate d of v < coordinate d of t_min
```

[4]See https://www.geometrictools.com for more information about the company


(a)

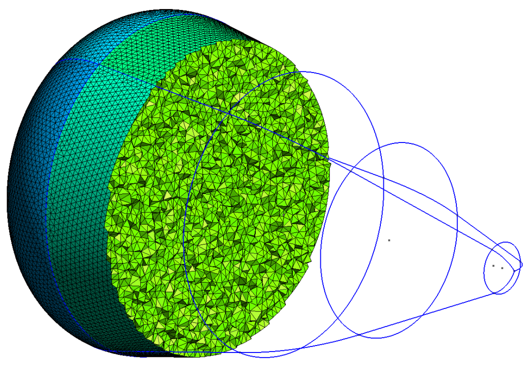

(b)

Figure 4: Discretization of the immersed objects necessary (a) for the simple ray-casting algorithm and (b) for the new *point classification* algorithm. The three-dimensional tetrahedralization required by the new algorithm has to be generated beforehand and is not handled by HYPERION.

```
                    then
                        coordinate d of t_min := coordinate d of v
                    else if coordinate d of v > coordinate d of t_max
                    then
                        coordinate d of t_max := coordinate d of v
                    end if
                end for each
            end for each
        end for each
    end function
```

- Step 2 - each AABB is clipped/culled against the underlying Cartesian mesh to ensure that when searching within the bounding boxes, the grid points are within range of the Cartesian mesh bounds. Note that the ”underlying Cartesian mesh” can very well be a local piece of mesh in the context of MPI partitioning, making this *point classification* inherently adapted to distributed parallelism.

```
function cull tetrahedra AABB
    m_min, m_max := lower, upper vertex of Cartesian box
    for each tetrahedron t
        t_min, t_max := lower, upper vertex of t bounding box
        for each dimension d
            coordinate d of t_min :=
                max between
                    | coordinate d of t_min,
                    | coordinate d of m_min
            coordinate d of t_max :=
                min between
                    | coordinate d of t_max,
                    | coordinate d of m_max
            if coordinate d of t_min > coordinate d of t_max
            then
                declare t invalid
            end if
        end for each
    end for each
end function
```

- Step 3 - the tetrahedra vertices and bounding boxes are then transformed to the (i, j, k) grid coordinate system.

```
    function transform to grid ijk
        m_min, m_max := lower, upper vertex
            of Cartesian box
        n.0, n.1, n.2 := number of cells
            of Cartesian box in each dimension
        for each dimension d
            factor.d := (n.d - 1) / (
                coordinate d of m_max -
                coordinate d of m_min
            )
        end for each
        for each tetrahedron t
            t_min, t_max := lower, upper vertex
                of t bounding box
            for each dimension d
                g_t_min := ceiling of factor.d * (
                    coordinate d of t_min -
                    coordinate d of m_min
                )
                g_t_max := floor of factor.d * (
                    coordinate d of t_max -
                    coordinate d of m_min
                )
            end for each
        end for each
    end function
```

- Step 4 - each grid-coordinate bounding box contains a relatively small number of the Cartesian grid cell centers, hence we can iterate over those with a triple loop in k, then j, then i as per HYPERION grid ordering. For a constant (j,k) in the grid box, a line segment containing cell centers with varying i is obtained. Note that one possible implementation of the function determining whether a point is in a tetrahedron can be found in [36] and shall not be repeated here.

```
function rasterize
```

```
        n.0, n.1, n.2 := number of cells
            of Cartesian box in each dimension
        for each tetrahedron t
            g_t_min, g_t_max := lower, upper vertex
                of t bounding box in ijk coordinates
            for k from g_t_min.2 to g_t_max.2
                for j from g_t_min.1 to g_t_max.1
                    for i0 from g_t_min.0 to g_t_max.0
                        if (i0,j,k) in tetrahedron t
                        then
                            break for i0
                        end if
                    end for
                    if i0 > g_t_max.0
                    then
                        cycle for j
                    end if
                    for i from g_t_max.0 down to i0
                        if (i,j,k) in tetrahedron t
                        then
                            break for i
                        end if
                    end for
                    base := n.0 * (j + n.1 * k)
                    for l from i0 to i
                        n := l + base
                        flag cell center of index n
                            as inside tetrahedron t
                    end for
                end for
            end for
        end for each
    end function
```

- Step 5 - the i-values are searched for the first and last cell centers that are inside each tetrahedron, if any, which yields a set of flagged (i,j,k) that can be said to be inside the immersed object.

As mentioned, the algorithm can be easily adapted to a code working in parallel. For a multithreaded code, then the tetrahedra can be split among the different threads to share the workload. There is a chance that two threads might try to flag the same (i,j,k) to the set of solid cells. In particular this can happen if a cell center is on the shared face of a pair of tetrahedra (or close to that face because of numerical rounding errors). However, there is no need for a critical section or an atomic write: first, from the perspective of either thread the cell center will be found to be inside the immersed object, and anyway the (i,j,k) grid coordinates are 32-bit integers that are written atomically by nature.
If the number of tetrahedra in the discretization of the object is noted N_t, then the complexity of this

algorithm is, mostly, $O(N_t)$. For the same worst-case problem as before, the time-to-completion falls to 8.5 seconds on a single OpenMP thread and the timings for 2, 4, 8 and 16 OpenMP threads are 4.3 seconds, 2.6 seconds, 2.1 seconds and 2.0 seconds respectively (versus $\gtrsim$ 48 hours for the brute-force algorithm for 4 OpenMP threads).

4 MIGRATABLE TASKS FOR THE MASSIVE PARALLELIZATION OF THE RECONSTRUCTION ALGORITHM

As mentioned in section 2.2, the interpolation of the fluid values at each image point (step 5) relies on the least-square ENO-like interpolation discussed in details in the paper by Bridel-Bertomeu [4]. A short study in the latter paper mentions that for the algorithm to be robust (*i.e* for the least-square matrix to be well enough conditioned), at least 25 neighbors are necessary in two dimensions for a third order interpolation. The same kind of study conducted in three dimensions shows that at least 85 neighbors are required for a third order interpolation so that the least-square matrix reaches its asymptotic minimum conditioning.

Furthermore, to balance the number of neighbors picked in each direction around the cell of interest, HYPERION implements a spiral walk (see [4], figure 4(a)): with that strategy, getting to 25 (85) neighbors in two (three) dimensions means having access to at least three layers of cells around the cell of interest. However, considering that such cells of interest are by nature in proximity to the immersed objects and because we cannot consider the immersed cells as valid neighbors in the interpolation, acquiring the information from enough valid neighbors can quickly lead to accessing the fifth or sixth layer of cells away from the cell of interest (*e.g.* for fourth or fifth order interpolation matching the order of the TENO reconstruction described in paragraph 2.1.2).

In a sequential, or even in a shared-memory parallelism context, accessing cells that are far away from the cell of interest is not a challenge - the algorithm presented in [4] could be implemented directly without any modifications in a purely sequential or only OpenMP version of HYPERION. In a distributed parallelism context however, things become rather tricky if one aims at minimizing the computational overhead related to communications between processes and/or the overall memory use of the application, as is the goal for HYPERION.

If only in terms of memory use, note that out of consistency, in HYPERION the size of the MPI halo equals the number of ghost cells used at the edges of the domain - see figure 5. Therefore, if we consider a rather common mesh size of 512^3 cells, and if we attempt to gain access to cells six layers away with a halo of size 6, the actual mesh handled by HYPERION has size $(512+2\times 6)^3$, which, if we only account for the five conservative variables, means an overhead of about 3 Gi in memory. HYPERION efficiency is around 1 to 1.5 μs_{CPU}/it/cell, hence using a size 6 halo tends to introduce an overhead of about 10 s_{CPU} at each iteration.

With these considerations in mind, along with the fact that having *e.g.* size 6 halos is highly detrimental to the performance of the MPI communications (although it has not been quantitatively evaluated in the present study), we worked on an algorithm that would not rely on the MPI halos at all in order to perform well in conjunction with any kind of reconstruction stencil. Inspired by the spiral "walk" used in a sequential context to gather information from enough valid neighbors for the least-square interpolation, we turned ourselves towards a migratable task paradigm.
In this paradigm, the first ingredient is a *task*. In our context, let us remember that for each image point

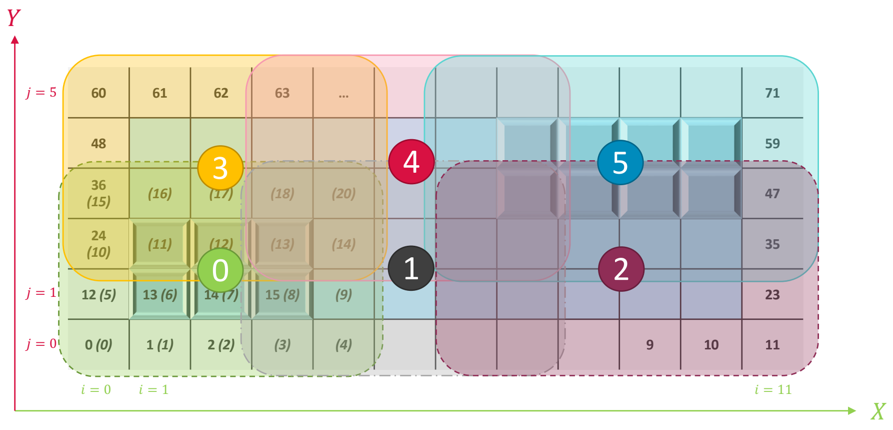
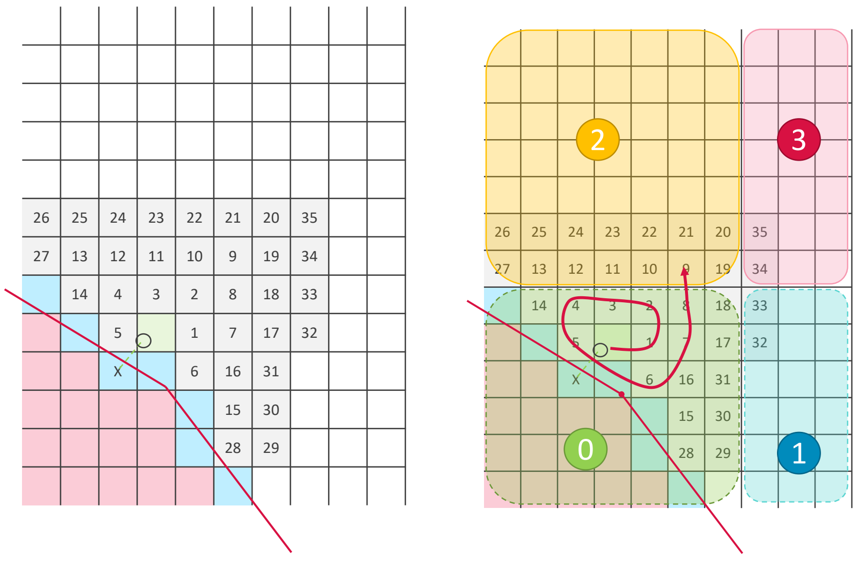


Figure 5: Illustration of the Cartesian partitioning of the Cartesian mesh in HYPERION. The overlapping halos (of size 1 in this case) ensuring communications between neighboring processes are also represented. Note that the numbering of the MPI processes is done the same way as the numbering of the fluid cells.

we want to gather information from enough "valid" neighbors to have a well-conditioned least-square matrix (see figures 2(f) and 6(a)).



(a) (b)

Figure 6: (a) Zoom on the neighborhood of an immersed boundary cell image and the corresponding surrounding cells numbering for the spiral walk. (b) Illustration of the spiral walk task in the migratable task algorithm and of the influence of partitioning thereupon - the halos are omitted as the algorithm does not make use of them.

To do so, we emit a probe that will march the surrounding cells in a spiral motion: for each "valid" neighbor (that is not an immersed cell), the index of the cell and the values of the fluid variables are stored in the probe structure before it goes on. This marching is the *task* in the migratable task paradigm

- see figure 6(b) - which is supported by a simple data structure:

```
struct _probe_task_state
    status  // status of the marching
    (x,y,z) // current coordinate of the head of the probe

    coords[] // coordinates of valid neighbors, accumulated
    values[] // fluid variables in the valid neighbors, accumulated

    origin_rank // rank of the processes wherefrom the probe originates
    rank // rank of the process the probe needs to be sent to, if any

    count // current number of neighbors
end struct
```

Because of the partitioning of the Cartesian mesh, several, if not all, processes own immersed boundary cells and must therefore emit probes. In terms of implementation, each process has a stack of such tasks that it works through either sequentially or in parallel with multiple shared-memory OpenMP threads. During its march, if a probe next step has to be taken on a different process (illustrated on figure 6(b)), then a message is sent from its current owner to the neighboring process containing the entire probe structure ; in so doing, the probe is popped from the previous owner stack of tasks and pushed to the neighbor process stack that can resume the walk of the probe. This march of each probe continues until it has found enough valid neighbors, *per* the user request, at which point the whole structure of the probe is sent back to the process it originated from so the least-square interpolation can take place and the immersed boundary cell can be populated with values enforcing the immersed boundary condition [4].

Note that during the course of the algorithm, no process can predict with simple logic whether it is going to receive a task from a neighbor or whether it will have to send one of its tasks to a neighbor - since no process knows whether to expect a message, they all wait for messages asynchronously while handling the tasks on their stacks. This poses a well-known problem in parallel computations, namely a termination issue. Since no process knows to expect a message, if nothing is done all the processes will eventually finish working on their own tasks and wait indefinitely for messages from their neighbors. To prevent such an infinite loop from occurring in HYPERION, we implemented the algorithm of Francez *et al.* [9] to achieve distributed termination without introducing any new communication.

The entire pseudocode algorithm is described below.

```
while not terminated
    %MASTER THREAD
    {
        while to_send_stack not empty
            e := pop to_send_stack
            async send e to e.rank
            sent_stack push e
        end while
        while sent_stack not empty
            check sent request has gone through
        end while
        probe all sources for any message
        if message
            new_e := deserialize message
            stack push new_e
        end if
        terminated ?= francez termination algorithm
    }
    %END MASTER THREAD
    while stack not empty
        e := pop stack
        until enough neighbors
            go to the next cell in the spiral
            if next cell is solid or ghost
            then
                skip
            else
                if next cell on current process
                    e.coords append cell center coordinates
                    e.values append cell values
                    increment e.count
                    if enough neighbors
                        e.status := SUCCESS
                        break until
                    end if
                else
                    e.rank := neighbor rank
                    break until
                end if
            end if
        end until
        if e.status == SUCCESS
            e.rank := e.origin_rank
        end if
        if e.status == SUCCESS and current process rank == e.origin_rank
            solve least square problem
            compute immersed ghost cell value
        else
            to_send_stack push e
        end if
    end while
end do
```

This migratable task strategy allowed us to use the ENO-like interpolation algorithm with large neigh-

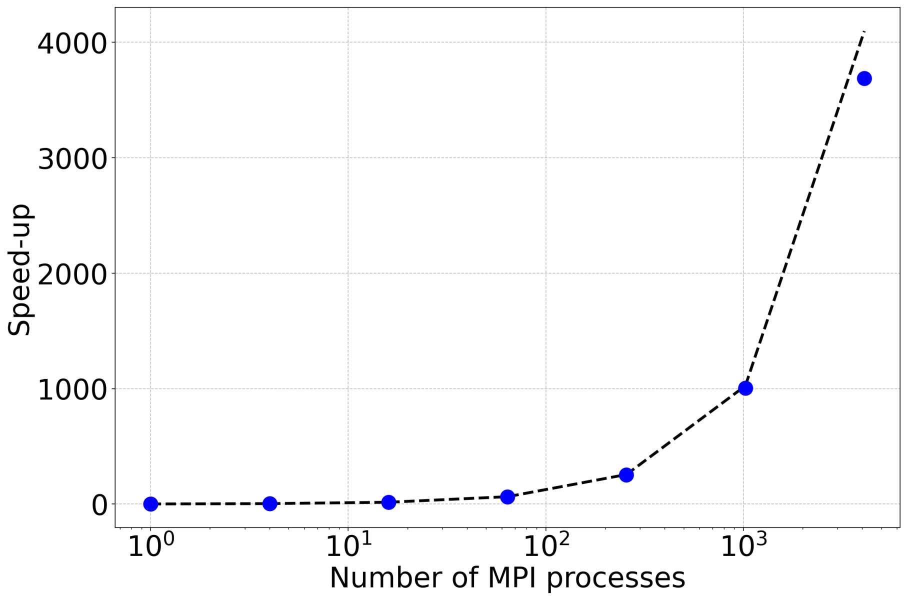


Figure 7: Speed up achieved for distributed parallelism with HYPERION and the migratable task algorithm - ideal speed-up (dashed line) versus actual speed-up (blue dots). Data collected for a single immersed body using AMD Rome processors.

borhoods in massively parallel computations with guaranteed stability, yielding a satisfactory speed-up up to 4096 processes as shown on figure 7.

5 TOWARDS HIGH-FIDELITY SIMULATIONS

The two algorithms/strategies presented in sections 3 and 4 made possible running massively parallel simulations in three dimensions with HYPERION: before the introduction of the former, detecting solid cells on realistic meshes of the order of 10^8 cells would take several days, and before the introduction of the latter, three-dimensional computations with immersed objects would be too expensive and, most often, would fail altogether during the spiral walks. In the high Mach and Reynolds numbers regimes where this study places itself, direct numerical simulations (DNS) are out of reach because of their prohibitive computational cost. However, we aim at running large eddy simulations (LES) with an *ad hoc* subgrid-scale model to improve the predictive capabilities of HYPERION in the compressible regime even on coarse meshes - gaining the ability to run three-dimensional simulations was a first step in that direction.

An in-depth presentation of the LES equations for compressible flows is out of the scope of this paper (the interested reader is referred *e.g.* to Garnier *et al.* [16]), but recall that solving the LES equations implies using a filtered version of the compressible Navier-Stokes equations (1)-(2), among which the momentum equations now read:

$$\frac{\partial \bar{\rho}\tilde{u}_i}{\partial t} + \frac{\partial \bar{\rho}\tilde{u}_i\tilde{u}_j}{\partial x_j} + \frac{\partial \bar{p}}{\partial x_i} - \frac{\partial \breve{\sigma}_{ij}}{\partial x_j} = -\frac{\partial \tau_{ij}}{\partial x_j} + \frac{\partial}{\partial x_j}\left(\overline{\sigma_{ij}} - \breve{\sigma}_{ij}\right), \tag{21}$$

where the $\tilde{\cdot}$ overset denotes Favre averaging, the $\bar{\cdot}$ overset denotes Reynolds averaging, and $\overset{\smile}{\sigma}_{ij}$ depends on the computable rate-of-strain tensor:

$$\overset{\smile}{\sigma}_{ij} = \mu(\tilde{T})\left(2\tilde{S}_{ij} - \frac{2}{3}\delta_{ij}\tilde{S}_{kk}\right), \;\; \tilde{S}_{ij} = \frac{1}{2}\left(\frac{\partial \tilde{u}_i}{\partial x_j} + \frac{\partial \tilde{u}_j}{\partial x_i}\right). \tag{22}$$

All overset variables are computable whereas we need a closure for the subgrid-scale stress tensor τ_{ij}. In HYPERION, we follow a Boussinesq type hypothesis [2] that leads to the subgrid-scale stress tensor having the following mathematical form:

$$\tau_{ij} - \frac{1}{3}\delta_{ij}\tau_{kk} = -2\bar{\rho}\nu_{\text{sgs}}\left(\tilde{S}_{ij} - \frac{1}{3}\delta_{ij}\tilde{S}_{kk}\right), \tag{23}$$

where ν_{sgs} represents a scalar subgrid viscosity that is left to be modelled. In the present study we rely on the Wall-Adapting Local Eddy-viscosity model (WALE) by Ducros *et al.* [8] to run preliminary tests with the LES version of HYPERION.

5.1 Taylor-Green Vortex

Before trying to evaluate the quality of the LES computations in the presence of immersed obstacles, let us run a very classical viscous test case that will allow us to check the implementation of the WALE model.

We are therefore considering the case of a Taylor-Green vortex which set-up and reference solution can be obtained from the 3rd international workshop on higher-order CFD methods [7]. The domain is a periodic cube of dimensions $[-\pi,\pi]^3$, and the initial solution is given by the following expressions:

$$\left\{\begin{array}{lcl} u & = & \sin(x)\cos(y)\cos(z), \\ v & = & -\cos(x)\sin(y)\cos(z), \\ w & = & 0, \\ p & = & \dfrac{1}{\gamma M_\infty^2} + \dfrac{1}{16}\left(\cos(2x)+\cos(2y)\right)\left(\cos(2z)+2\right), \\ \rho & = & \gamma M_\infty^2 p. \end{array}\right. \tag{24}$$

In this problem, we do not intend to validate in any way the behavior of the immersed boundaries algorithms. Rather, we shall inspect the temporal evolution of the kinetic energy E_k and of the enstrophy ε integrated over the domain and compare it to the reference solution [7]. Note that the mathematical expression used in this study for the kinetic energy is:

$$E_k = \frac{1}{N^3}\sum_{i,j,k}\frac{1}{2}\rho_{i,j,k}\left(u_{i,j,k}^2 + v_{i,j,k}^2 + w_{i,j,k}^2\right), \tag{25}$$

and that of the enstrophy is:

$$\varepsilon = \frac{1}{N^3} \sum_{i,j,k} \frac{1}{2} \rho_{i,j,k} \left(\omega_{x;i,j,k}^2 + \omega_{y;i,j,k}^2 + \omega_{z;i,j,k}^2 \right), \tag{26}$$

where N stands for the number of cells along one side of the domain (assuming the Cartesian mesh is homogeneous), and $\omega = \nabla \times \mathbf{u}$ is the vorticity. To observe the convergence of the solution towards the reference solution, computations results on meshes of sizes 64^3 and 512^3 are shown here. The computations were furthermore run using the AUSM$^+$-up Riemann solver in conjunction with the fifth-order TENO reconstruction scheme.

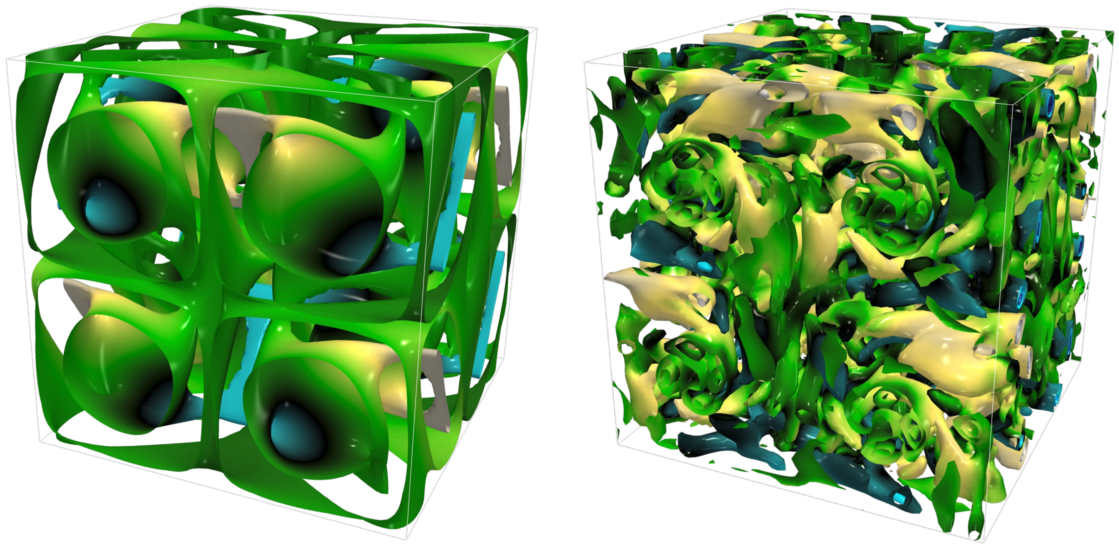
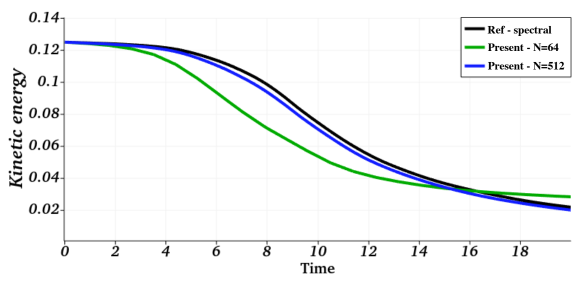

(a) (b)

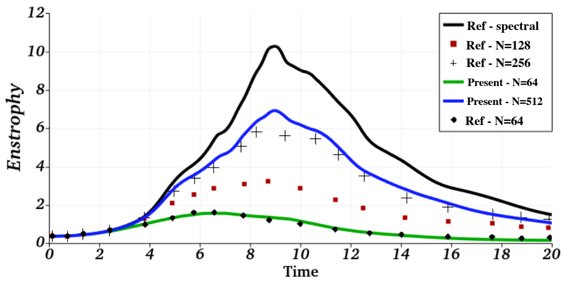
Figure 8: Snapshots of the state of the Taylor-Green vortex after (a) 1 characteristic time and (b) 20 characteristic times on the 64^3 mesh. Isocontours of enstrophy are represented, colored by the magnitude of velocity.

Figure 8 presents the three-dimensional structures present in both the coarse and the fine flow as isocontours of the Q-criterion colored by the magnitude of the vorticity. As expected, the turbulence is allowed to develop itself down to much smaller scales on the fine mesh.

Figure 9 introduces a more quantitative comparison between the present results and both the reference spectral computation conducted on a 512^3 mesh in [7] and some high-order reference computations from [17]. The results are satisfactory and show the convergence towards the reference for both metrics - the kinetic energy, figure 9(a) and the enstrophy, figure 9(b). The results in terms of enstrophy seem further away but in another exhaustive study (unpublished yet) the authors show the prime importance of the approximate Riemann solver on the quality of the enstrophy production in a finite-volume code, and although the AUSM$^+$-up solver is particularly adapted at handling all-speed flows with hypersonic regions, it might not be the ideal solver for fine turbulence behavior - such a discussion is however out of the scope of this paper.



(a)



(b)

Figure 9: Comparison of the temporal evolution of (a) the kinetic energy (see equation (25)) and (b) the enstrophy (see equation (26)) against the 512^3 spectral computation from [7] (black line) and the high-order computations from [17] (symbols).

5.2 Double cone

With the implementation of the subgrid-scale model validated by the turbulence testcase discussed in paragraph 5.1, we move on to the study of the flow around two documented configurations in supersonic or hypersonic regimes.

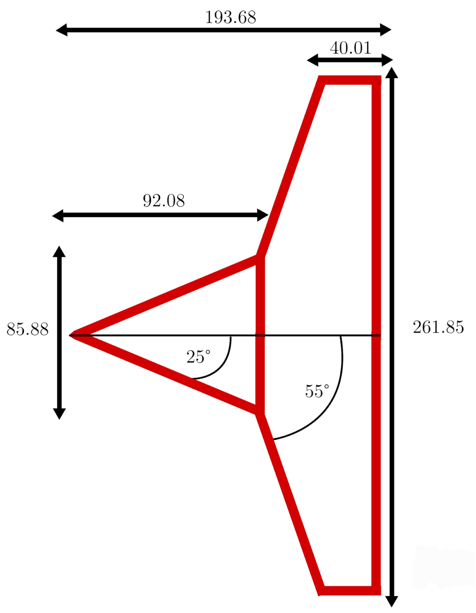


Figure 10: Geometrical details of the double-cone configuration.

For the first immersed boundary condition testcase, the hypersonic flow over an axisymmetric double cone configuration (see figure 10) is studied at Mach number 12.2 and Reynolds number 14×10^6. The wall temperature is fixed at $1.75T_\infty$. The results presented below correspond to meshes made of $1,200,000$ and $4,000,000$ cells, qualified of coarse and fine, respectively (as it is an axisymmetric obstacle, the simulation itself has been treated as a two-dimensional, axisymmetric simulation).

A qualitative visualization of the resulting flow is provided on figure 11 in the form of a numerical Schlieren image. It mostly shows the regions where strong changes in density occur, allowing the user to visualize the shock and contact waves, and, like in the present case, the boundary layer developing along the wall. An inset has been added to the figure to put the light on the lambda-shock pattern found after the boundary layer passes the recirculation region and reattaches itself to the wall of the second cone.

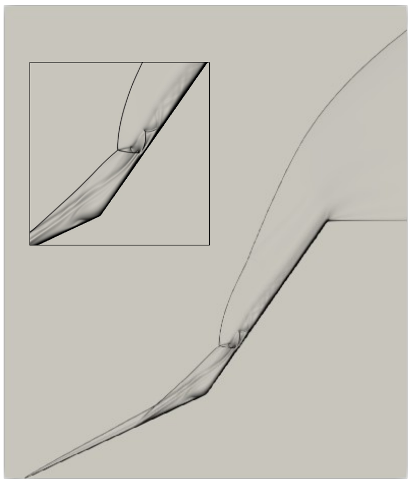
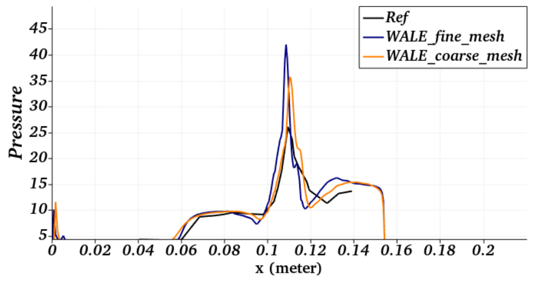


Figure 11: Numerical Schlieren imaging of the flow around the double cone configuration. The inset zooms in on the lambda-shock pattern found after the recirculation region at the change of wall angle.

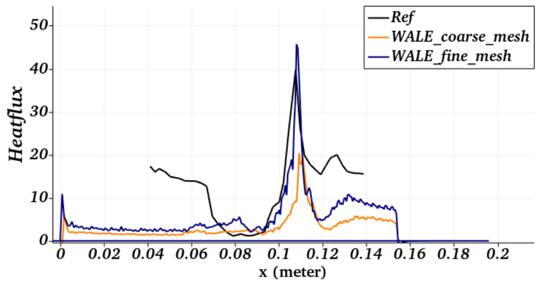
In a more quantitative manner, the wall pressure and the wall heat flux are compared against the experimental results obtained by MacLean *et al.* [30] - see figure 12(a) and (b). Without much effort, the wall pressure is correctly predicted - figure 12(a) - because it is mostly driven by the inviscid behavior of the simulation and does not depend too much on the resolution of the near-wall flow. In contrast, the viscous heat flux is not as well predicted because boundary layers are not resolved properly with the immersed boundary method implemented in HYPERION. Even with a fine mesh, although the peak of heat flux is correctly captured both in terms of coordinates and magnitude, the heat flux levels after the reattachment of the boundary layer are markedly underestimated. This is a well known shortcoming of any immersed boundary method, and a future study will be dedicated to correcting it with the help of *ad hoc* wall laws.

5.3 Flow study around a double ellipsoid configuration

The second immersed boundary condition testcase is the fully three-dimensional flow around a double ellipsoid shape [1] described mathematically by the following equations (see figure 13):



(a)



(b)

Figure 12: (a) Pressure coefficient and (b) heat flux obtained at the wall of the double cone. Comparison between the results obtained presently on the coarse and fine meshes (orange and blue line, respectively) and the reference solution from MacLean *et al.* [30] (black line).

$$\begin{cases} x \leq 0, & \left(\frac{x}{0.06}\right)^2 + \left(\frac{y}{0.025}\right)^2 + \left(\frac{z}{0.015}\right)^2 = 1 \\ x \leq 0,\ z \geq 0, & \left(\frac{x}{0.035}\right)^2 + \left(\frac{y}{0.0175}\right)^2 + \left(\frac{z}{0.025}\right)^2 = 1 \\ 0 \leq x \leq 0.016, & \left(\frac{y}{0.025}\right)^2 + \left(\frac{z}{0.015}\right)^2 = 1 \\ z \geq 0, & \left(\frac{y}{0.0175}\right)^2 + \left(\frac{z}{0.025}\right)^2 = 1 \end{cases} \tag{27}$$

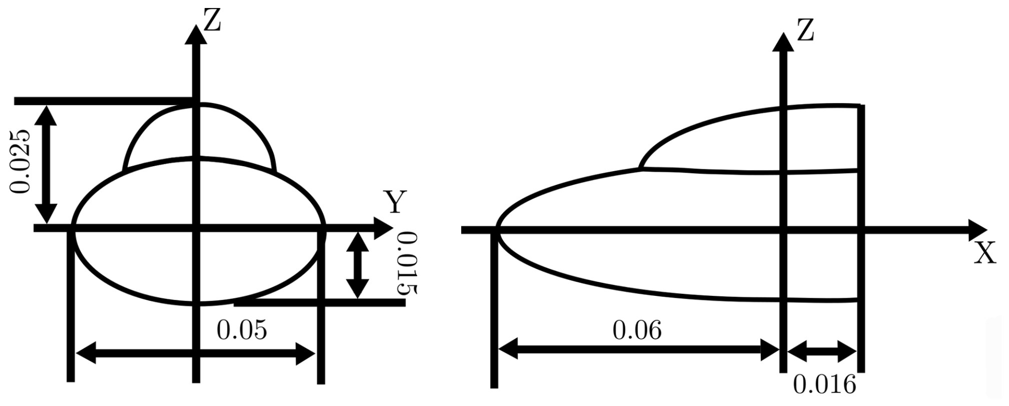


Figure 13: Geometrical details of the double ellipsoid configuration.

For that flow, the Mach number is set at 8.15, the Reynolds number is set at 16.7×10^6 and the freestream angle of attack is set at 30°. The Cartesian mesh contains approximately 7×10^7 cells, and the double ellipsoid object is tesselated using approximately 2×10^5 triangles; the computation was conducted on 4096 AMD Rome processors, using 2048 MPI processes and 2 OpenMP threads per process, and lasted approximately 48 hours before the wall quantities could be said to be converged (see below). Note that this computation, both in terms of manipulation of the immersed object and in terms of computational performance, is made possible only thanks to the two algorithms presented above in sections 3 and 4 respectively - without massive parallelism, the time-to-results for this computation would have been unrealistic.

A first quantitative visualization of the flow is given in figure 14, where both numerical Schlieren imaging and three-dimensional isocontours of the Q-criterion are presented. The strong shock waves can be clearly seen around the object whereas the wake seems characterized by multiple contact waves and large-scale turbulent structures that the LES simulation is able to capture, as expected.

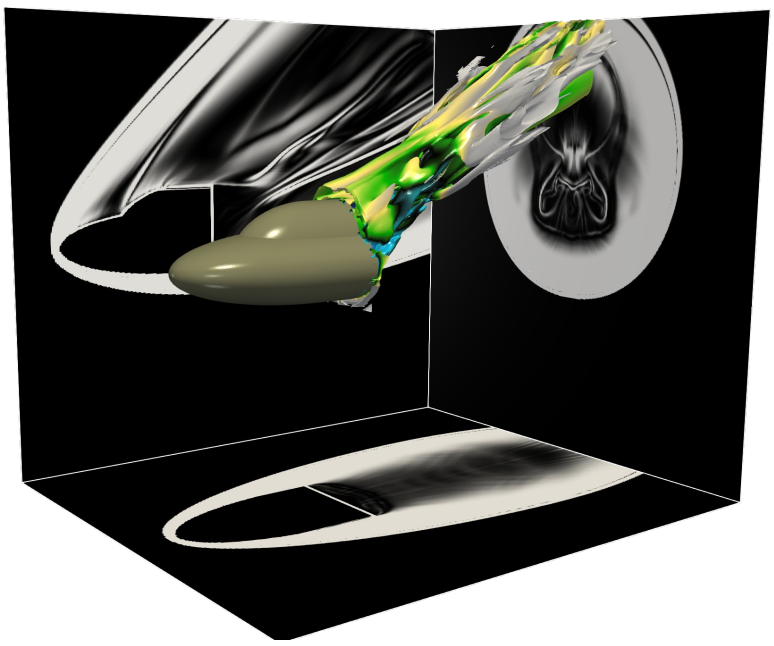
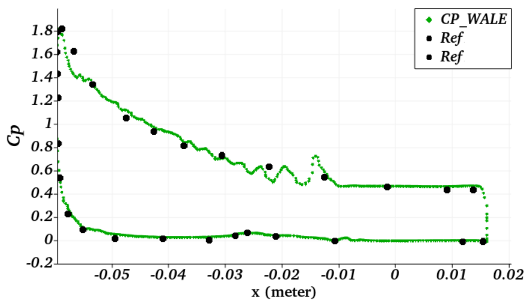


Figure 14: Illustration of the flow around the double ellipsoid immersed object with a 30° angle of attack. The walls of the visualization domain are colored using the numerical Schlieren imaging technique, and the wake of the double ellipsoid is shown using isocontours of Q-criterion, colored by the magnitude of the vorticity.

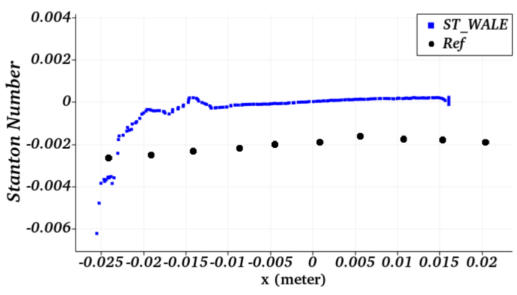
Similarly to the study conducted for the previous test case (see paragraph 5.2), figure 15 presents the pressure coefficient and the Stanton number at the wall of the double ellipsoid. The conclusions are similar to the ones for the double cone flow. The pressure coefficient is satisfactorily predicted - the experimental data does not allow to conclude on the nature of the oscillations observed in the vicinity of the recirculation region (when the second ellipsoid starts, around $x \simeq 0.0$). The heat flux accuracy is however fairly low again, mostly because of the lack of alignment between the Cartesian mesh and the object wall causing the boundary layers to be poorly described. This phenomenon and possible fixes will be, as aforementioned, explored in a future study.

6 CONCLUSIONS

In this paper, we present some work related to the immersed boundary method and in particular to the adaption of the method for massively parallel computations. A novel algorithm based on the theory of



(a)



(b)

Figure 15: (a) Pressure coefficient on both the pressure and the suction sides and (b) Stanton number on the pressure side of the double ellipsoid. Comparison between the present computations (green and blue lines) and the experimental reference data (black dots) extracted from [1].

computational geometry is introduced to drastically decrease the time necessary to identify a tesselated immersed body in a Cartesian box - from beyond 48 hours to a few seconds. The steps of this algorithm are detailed exhaustively and an example implementation of its non-trivial functions is provided. Still in order to lift major limitations to conducting three-dimensional simulations involving immersed objects on massive clusters, a second algorithm is presented that make the interpolation step of the immersed boundary workflow possible for any number of MPI processes. This algorithm represents the parallel extension of the works presented in [4] about an ENO-like least-square reconstruction mandatory to handle strong shocks in the vicinity of immersed bodies subject to hypersonic flows.

As those two new algorithms finally make massively parallel three-dimensional computations accessible with HYPERION, the improvement of the predictive capabilities of the code are then discussed. Rather than conducting direct numerical simulations that stay, even today, unreachable because of their prohibitive computational cost at high Reynolds numbers, HYPERION is turned towards large eddy simulations (LES). Preliminary axisymmetric and three-dimensional simulations conducted show that even if adequate subgrid-scale modeling allow for the prediction of large-scale turbulent structures on relatively coarse mesh, it is not enough to capture accurately the near-wall viscous phenomena. The lack of alignment between the Cartesian grid (fluid grid) and the wall of the immersed body makes for poorly resolved boundary layers which, in turn, yield poor performance when it comes to predicting the viscous-driven wall quantities such as heat flux and shear stress. A future study shall be dedicated to investigating these shortcomings and proposing solutions, perhaps axed towards the use of wall laws.

7 ACKNOWLEDGEMENTS

The authors want to address special thanks to David H. Eberly for his support in designing the rasterization algorithm and to Cédric Augonnet for his help in designing the migratable task algorithm.

References

[1] D. Aymer, T. Alziary, L. D. Luca, and C. Carlomagno. Experimental study of the flow around a double ellipsoid configuration. In *Hypersonic Flows for Reentry Problems*, pages 335–357. Springer, Berlin, Heidelberg, 1991.

[2] Joseph Boussinesq. *Essai sur la théorie des eaux courantes*. Impr. nationale, 1877.

[3] Pierre Brenner. Three-dimensional aerodynamics with moving bodies applied to solid propellant. In *27th Joint Propulsion Conference*, page 2304, 1991.

[4] Thibault Bridel-Bertomeu. Immersed boundary conditions for hypersonic flows using eno-like least-square reconstruction. *Computers & Fluids*, 215:104794, 2021.

[5] B Capra, J Moran, M Brown, R Boyce, E Trifoni, A Schettino, C Purpura, A Martucci, and R Gollan. Aerothermal analysis of 3d concave cone in hypersonic flow in scirocco.

[6] Cheng Chi, Bok Jik Lee, and Hong G Im. An improved ghost-cell immersed boundary method for compressible flow simulations. *International Journal for Numerical Methods in Fluids*, 83(2):132–148, 2017.

[7] L Diosady and S Murman. Case 3.3: Taylor green vortex evolution. In *Case Summary for 3rd International Workshop on Higher-Order CFD Methods*, 2015.

[8] F Ducros, F Nicoud, and Thierry Poinsot. Wall-adapting local eddy-viscosity models for simulations in complex geometries. *Numerical Methods for Fluid Dynamics VI*, pages 293–299, 1998.

[9] Nissim Francez and Michael Rodeh. Achieving distributed termination without freezing. *IEEE Transactions on Software Engineering*, SE-8:287–292, 1982.

[10] Lin Fu. A low-dissipation finite-volume method based on a new teno shock-capturing scheme. *Computer Physics Communications*, 235:25–39, 2019.

[11] Lin Fu. A very-high-order teno scheme for all-speed gas dynamics and turbulence. *Computer Physics Communications*, 244:117–131, 2019.

[12] Lin Fu, Xiangyu Y Hu, and Nikolaus A Adams. A family of high-order targeted eno schemes for compressible-fluid simulations. *Journal of Computational Physics*, 305:333–359, 2016.

[13] Lin Fu, Xiangyu Y Hu, and Nikolaus A Adams. Targeted eno schemes with tailored resolution property for hyperbolic conservation laws. *Journal of Computational Physics*, 349:97–121, 2017.

[14] Lin Fu, Xiangyu Y Hu, and Nikolaus A Adams. A new class of adaptive high-order targeted eno schemes for hyperbolic conservation laws. *Journal of Computational Physics*, 374:724–751, 2018.

[15] Mario De Stefano Fumo, Roberto Scigliano, Marika Belardo, Giuseppe Rufolo, Angelo Esposito, and Mauro Linari. Aero-thermal post flight analysis of ixv control surfaces. 2015.

[16] E. Garnier, N. Adams, and P. Sagaut. *Large Eddy Simulation for Compressible Flows*. Springer, 2009.

[17] Giorgio Giangaspero, Edwin van der Weide, MH Carpenter, and K Mattsson. Case c3. 3: Taylor-green vortex. In *Case Summary for 3rd Int. Workshop on Higher-Order CFD Methods*. NASA, 2015.

[18] Eric Haines. Point in polygon strategies. *Graphics Gems*, 4:24–46, 1994.

[19] Andrew K Henrick, Tariq D Aslam, and Joseph M Powers. Mapped weighted essentially non-oscillatory schemes: achieving optimal order near critical points. *Journal of Computational Physics*, 207(2):542–567, 2005.

[20] XY Hu and Nikolaus A Adams. Scale separation for implicit large eddy simulation. *Journal of Computational Physics*, 230(19):7240–7249, 2011.

[21] XY Hu, Q Wang, and Nikolaus A Adams. An adaptive central-upwind weighted essentially non-oscillatory scheme. *Journal of Computational Physics*, 229(23):8952–8965, 2010.

[22] Guang-Shan Jiang and Chi-Wang Shu. Efficient implementation of weighted eno schemes. *Journal of computational physics*, 126(1):202–228, 1996.

[23] M Ehsan Khalili, Martin Larsson, and Bernhard Müller. High-order ghost-point immersed boundary method for viscous compressible flows based on summation-by-parts operators. *International Journal for Numerical Methods in Fluids*, 89(7):256–282, 2019.

[24] Gilbert F Kinney and Kenneth J Graham. *Explosive shocks in air*. Springer Science & Business Media, 2013.

[25] Keiichi Kitamura, Eiji Shima, and Philip L Roe. Carbuncle phenomena and other shock anomalies in three dimensions. *AIAA journal*, 50(12):2655–2669, 2012.

[26] Randall J LeVeque et al. *Finite volume methods for hyperbolic problems*, volume 31. Cambridge university press, 2002.

[27] Meng-Sing Liou. A sequel to ausm: Ausm+. *Journal of computational Physics*, 129(2):364–382, 1996.

[28] Meng-Sing Liou. A sequel to ausm, part ii: Ausm+-up for all speeds. *Journal of computational physics*, 214(1):137–170, 2006.

[29] Xu-g Liu, Stanley Osher, and Tony Chan. Weighted essentially non-oscillatory schemes. *Journal of computational physics*, 115(1):200–212, 1994.

[30] M. MacLean, M. Holden, and A. Dufrene. Comparison between cfd and measurements for real-gas effects on laminar shock wave boundary layer interaction. In *AIAA Aviation, Atlanta GA*, 2014.

[31] Walter T Maier, Jacob T Needels, Catarina Garbacz, Fábio Morgado, Juan J Alonso, and Marco Fossati. Su2-nemo: An open-source framework for high-mach nonequilibrium multi-species flows. *Aerospace*, 8(7):193, 2021.

[32] Katate Masatsuka. *I do Like CFD, vol. 1*, volume 1. Lulu. com, 2013.

[33] Rajat Mittal and Gianluca Iaccarino. Immersed boundary methods. *Annu. Rev. Fluid Mech.*, 37:239–261, 2005.

[34] Feng Qu, Di Sun, Qingsong Liu, and Junqiang Bai. A review of riemann solvers for hypersonic flows. *Archives of Computational Methods in Engineering*, pages 1–30, 2021.

[35] Yegao Qu, Ruchao Shi, and Romesh C Batra. An immersed boundary formulation for simulating high-speed compressible viscous flows with moving solids. *Journal of Computational Physics*, 354:672–691, 2018.

[36] Philip Schneider and David H Eberly. *Geometric tools for computer graphics*. Elsevier, 2002.

[37] Eleuterio F Toro. *Riemann solvers and numerical methods for fluid dynamics: a practical introduction*. Springer Science & Business Media, 2013.

[38] Li Wang, Gaetano MD Currao, Feng Han, Andrew J Neely, John Young, and Fang-Bao Tian. An immersed boundary method for fluid–structure interaction with compressible multiphase flows. *Journal of Computational Physics*, 346:131–151, 2017.

[39] Frank M White and Isla Corfield. *Viscous fluid flow*, volume 3. McGraw-Hill New York, 2006.

[40] Mehrdad Yousefzadeh and Ilenia Battiato. High order ghost-cell immersed boundary method for generalized boundary conditions. *International Journal of Heat and Mass Transfer*, 137:585–598, 2019.

[41] Yang Zhang, Xinglong Fang, Jianfeng Zou, Xing Shi, Zhenhai Ma, and Yao Zheng. Numerical simulations of shock/obstacle interactions using an improved ghost-cell immersed boundary method. *Computers & Fluids*, 182:128–143, 2019.