Highly Parallel Computing of a Multigrid Solver for 3D Navier-Stokes equations

Charles-Henri Bruneau, Khodor Khadra

High Parallel Computing of a Multigrid Solver for 3D Navier-Stokes equations

Charles-Henri Bruneau[1,2], Khodor Khadra[1]

1 Univ. Bordeaux IMB,CNRS UMR 5251
351, cours de la Libération, F-33405 Talence
Charles-Henri.Bruneau@math.u-bordeaux.fr, Khodor.Khadra@math.u-bordeaux.fr
2 INRIA Bordeaux-Sud-Ouest Team MEMPHIS

Abstract

To get efficiently the whole frequencies of the solution, a multigrid solver is used to solve the Navier-Stokes equations for incompressible flows. The method uses a cell-by-cell Gauss-Seidel smoother that is not straightforward parallelizable. Besides the coarsest grids are very coarse and cannot be solved in parallel. The proposed method splits the 3D Cartesian computational domain into well balanced sub-domains with respect to two dimensions. Efficient parallel procedures using MPI libraries permit to get a high strong and weak scalability of the whole parallel software. Comparison is done between MPI and hybrid MPI/OpenMP parallelism.

Keywords: Navier-Stokes equations; Multigrid solver parallelism; Gauss-Seidel smoother parallelism; Strong and weak scalability; MPI and hybrid MPI/Open MP parallelism.

1 Introduction

It is nowadays impossible to perform high performance computing without taking a special care on the parallelism and to get high efficiency on a number of cores or threads as large as possible. The direct numerical simulation (DNS) of complex flows require a large number of degrees of freedom. It is the main reason for introducing intermediate methods like Reynolds-averaged Navier-Stokes equations (RANS), unsteady Reynolds-averaged Navier-Stokes equations (URANS), partially-averaged Navier-Stokes equations (PANS), detached eddy simulation (DES) or large eddy simulation (LES) [11]. With the increase of the computer resources it is possible to use DNS in two-dimensions if the simulation is performed with an efficient parallel software. However it is not easy to get high efficiency on a large number of cores except may be solving large linear systems, which is not our case.
In this work we use a method based on a multigrid algorithm to capture the modes of the flow solving Navier-Stokes equations by DNS in three-dimensions. The smoother used in the multigrid algorithm is a cell-by-cell Gauss-Seidel method that is unfortunately based on a backward dependency. However, it is not possible to replace it by a cell-by-cell Jacobi method (which could be easily parallelized) as the multigrid algorithm would not converge any more. So we have to focus on this part of the program in order to get the best efficiency possible. In addition the multigrid algorithm involves very coarse grids on which no parallelism can be applied and intermediate grids on which parallelism can be performed only on a restricted number of cores. Despite these difficulties, the challenge is to get the best efficiency on as many cores or threads as possible.
In the following the modelling and the numerical method to approximate Navier-Stokes equations are given, then the multigrid solver. Forward, the paper focus on MPI parallelism of the tough subroutines of the program, namely the Gauss-Seidel relaxation procedure, the multigrid

algorithm and the prolongation and restriction operators. At the end an hybrid MPI/OpenMP parallelism on nodes with 16 cores and 4 threads (IBM Blue Gene) is performed. A comparison of the full MPI and hybrid parallelisms is given as well as the efficiency and the scalability obtained in each case.

2 Modelling and numerical simulation

The aim is to simulate the flow governed by the dimensionless Navier-Stokes equations in a computational domain Ω that is a box eventually around obstacles (Figure 1). To ease the

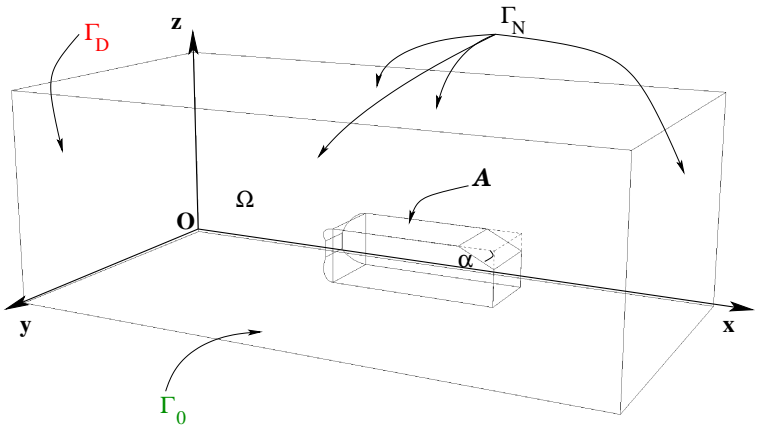

Figure 1: Computational domain around a simplified vehicle on top of a road.

parallelism, the first idea is to use an uniform Cartesian mesh on a regular three-dimensional domain, let us say $\Omega = (0, X) \times (0, Y) \times (0, Z)$ where X, Y and Z are integer numbers in practice. In case of a flow around obstacles, an immersed boundary method is used to take them into account. Namely the volume penalization method that consists in adding a term U/K_P in the momentum equation is used, the obstacles being considered as porous bodies with a very low permeability coefficient [2]:

$$\begin{array}{ll}
\partial_t U + (U \cdot \nabla)U - \dfrac{1}{Re}\Delta U + \dfrac{U}{K_P} + \nabla p = 0 & \text{in } \Omega \times I \\
\text{div } U = 0 & \text{in } \Omega \times I \\
U(0,.) = U_0 & \text{in } \Omega \\
U = U_\infty & \text{on } \Gamma_D \times I \\
U = U_\infty & \text{on } \Gamma_0 \times I \\
\sigma(U,p)\, n + \frac{1}{2}(U \cdot n)^-(U - U_{ref}) = \sigma(U_{ref}, p_{ref})\, n & \text{on } \Gamma_N \times I
\end{array} \tag{1}$$

where U is the velocity field, p the pressure, $I = (0, T)$ with T the simulation time, $Re = \frac{\rho \overline{U} H}{\mu}$ is the Reynolds number, $K_P = \frac{\rho k \overline{U}}{\mu \Phi H}$ is the non dimensional coefficient of permeability of the medium without gravity. In these two numbers without dimension, ρ is the density of the fluid, $\overline{U}$ is the velocity of the vehicle, H is the height of the vehicle, μ is the viscosity of the fluid, k and Φ are respectively the permeability and the porosity of the bodies. The boundary equations require to know the flow at infinity U_∞, the tensor σ and the reference flow (U_{ref}, p_{ref}) used to write the traction [5], in practice this reference flow is taken as the flow computed just before the exit section [3].
The system of equations (1) is solved by a strongly coupled approach for the physical unknowns $(U = (u, v, w), p)$. To get efficiency a multigrid solver using V-cycles is used to capture easily

the whole frequencies approached by the mesh. This solver is coupled to a cell-by-cell Gauss-Seidel relaxation smoother and to reinforce the coupling, the unknowns are set on staggered cells as shown in Figure 2. The time discretization is achieved using a second-order Gear scheme

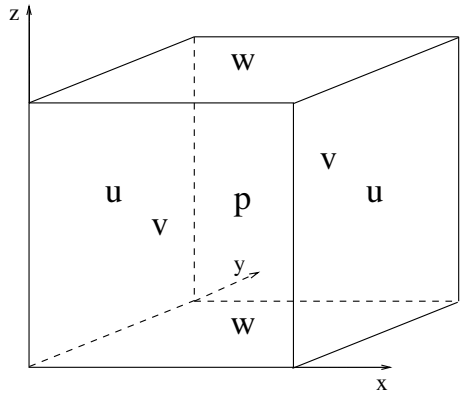

Figure 2: The staggered unknowns in a cell.

with an explicit treatment of the convection term that is approached by a third-order finite difference upwind scheme. All the linear terms are treated implicitly and discretized through a second-order centred finite difference scheme. The CFL condition related to the convection term requires a time step δt of the order of magnitude of the space step h as $\overline{U} = (1, 0, 0)$ [9]. A set of grids is defined starting from the coarsest mesh with $X \times Y \times Z$ cells to the finest mesh defined by dyadic refinement on L levels. For instance in the domain $\Omega = (0, 20) \times (0, 6) \times (0, 4)$, the coarsest uniform grid numbered 1 has $20 \times 6 \times 4 = 480$ cells, the second uniform grid has $40 \times 12 \times 8 = 480 \times 8 = 3,840$ cells and so on, each grid having 8 times the number of cells of the previous one. If $L = 8$, the finest grid has $2,560 \times 768 \times 512 = 1,006,632,960$ cells and $4,030,201,856$ unknowns. This is at least the number of unknowns required to perform a direct numerical simulation at Reynolds numbers up to 10^5.

3 Multigrid solver of Navier-Stokes equations

The discretization of the coupled velocity-pressure system of equations yields to solve a discrete linear system $A_h V_h^n = B_h^{n-1}$ where A_h represents the linear part of the discrete operator, B_h^{n-1} is the discrete right hand side (previous times and convection terms) and $V_h^n = (U_h^n, p_h^n)$ is the approximate solution to compute at time $t = n\delta t$. Despite the discrete system is linear, we solve it by the FAS nonlinear multigrid algorithm [10] as the genuine system is nonlinear as well as the artificial boundary conditions. The sequence of grids is denoted G_l with $1 \leq l \leq L$. Thus, the multigrid algorithm using a V-cycle procedure is illustrated below for the computation of

the fine grid solution $V_h^n = V_L^n$ on grid G_L.

$$
\left\{
\begin{array}{l}
\text{For q = 1 to number\_of\_V-cycles do} \\
\tilde{V}_L^q = S_L^{(\nu_1)}(A_L, V_L^{q-1}, B_L^{n-1}) \\
\text{Correction on coarse grids} \\
\left\{
\begin{array}{l}
\text{For l = L-1 to 1 by -1 do} \\
\overline{V}_l^q = R_l^{l+1} \tilde{V}_{l+1}^q \\
B_l^q = R_l^{l+1}(B_{l+1}^q - A_{l+1}\tilde{V}_{l+1}^q) + A_l \overline{V}_l^q \\
\tilde{V}_l^q = S_l^{(\nu_1)}(A_l, \overline{V}_l^q, B_l^q)
\end{array}
\right. \\
\text{Updating of the fine grids} \\
\left\{
\begin{array}{l}
\text{For l = 2 to L do} \\
\hat{V}_l^q = \tilde{V}_l^q + P_{l-1}^l(V_{l-1}^q - \overline{V}_{l-1}^q) \\
V_l^q = S_l^{(\nu_2)}(A_l, \hat{V}_l^q, B_l^q)
\end{array}
\right. \\
\text{Convergence test} \\
if\ ||B_L^{n-1} - A_L V_L^q|| \leq \epsilon \ \text{ stop iterations}
\end{array}
\right.
$$

In this algorithm $S_l^{(\nu)}$ denotes the smoother used on grid G_l to compute an approximate solution of the linear system doing ν iterations. The restriction R_l^{l-1} and prolongation P_{l-1}^l operators describe the linear interpolation operators from fine-to-coarse and from coarse-to-fine grids respectively. The smoothing operator S performs ν iterations of a cell-by-cell Gauss-Seidel procedure that leads to solve a 7×7 linear system corresponding to the 7 unknowns of a cell.

$$
\begin{pmatrix}
\alpha & 0 & 0 & 0 & 0 & 0 & 1/h \\
0 & \alpha & 0 & 0 & 0 & 0 & -1/h \\
0 & 0 & \alpha & 0 & 0 & 0 & 1/h \\
0 & 0 & 0 & \alpha & 0 & 0 & -1/h \\
0 & 0 & 0 & 0 & \alpha & 0 & 1/h \\
0 & 0 & 0 & 0 & 0 & \alpha & -1/h \\
-1/h & 1/h & -1/h & 1/h & -1/h & 1/h & 0
\end{pmatrix}
\begin{pmatrix}
u_{i,j,k}^n \\ u_{i+1,j,k}^n \\ v_{i,j,k}^n \\ v_{i,j+1,k}^n \\ w_{i,j,k}^n \\ w_{i,j,k+1}^n \\ p_{i,j,k}^n
\end{pmatrix}
=
\begin{pmatrix}
Du_{i,j,k} \\ Du_{i+1,j,k} \\ Dv_{i,j,k} \\ Dv_{i,j+1,k} \\ Dw_{i,j,k} \\ Dw_{i,j,k+1} \\ 0
\end{pmatrix}
$$

In this system $\alpha = \frac{1}{\delta t} + \frac{6}{Reh^2}$ and the other quantities of the linear operator are relaxed in the second member. So the $Du_{i,j,k}$ term represents the sum of these relaxed terms and $(B_l^{n-1})_{i,j,k}$. We solve this 7×7 coupled system directly. Indeed, eliminating the first six unknowns in the first six equations we get $p_{i,j,k}^n$, then the other unknowns follow. Let us point out that the seventh equation ensures the free divergence constraint in each cell.

4 Parallel algorithm of the Gauss-Seidel smoother

With the smoother presented in the previous section, the velocity components located on the sides are updated twice whereas the pressure in the centre of the cell is updated once. Besides there is a backward dependence as for computing the solution in the next cell it is necessary to have the new solution on the previous one. This is due to the fact that we are using a Gauss-Seidel smoother and not a Jacobi one. However, the multigrid method does not converge with a

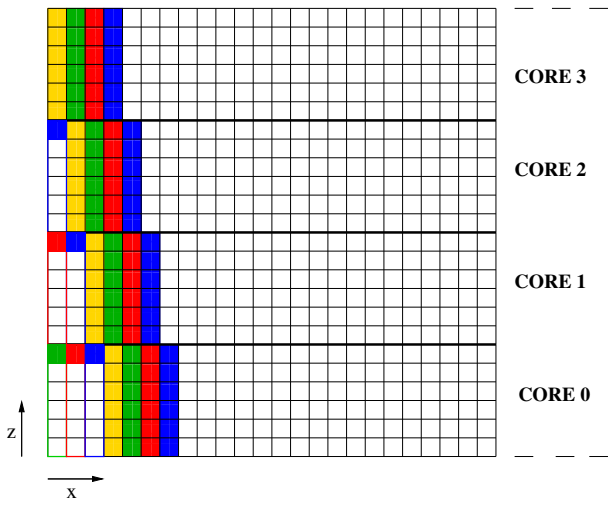

Figure 3: Propagation of parallel algorithm of cell-by-cell Gauss-Seidel smoother in two dimensions. Core 1 can start to work when core 0 has sent its upper cell of the first row coloured in green. The yellow color represents the beginning of the full parallel computing of the four cores.

Jacobi smoother which would be much easier to parallelize. Taking into account the backward dependence, it is not possible to write a parallel algorithm in one of the three dimensions. To improve the efficiency of the parallelization we have decided to put the larger length of the domain in x direction and to cut into uniform sub-domains in both y and z directions. Generally this larger length in x corresponds to the main direction of the flow. To illustrate this we show in the Figure 3 what is done in two dimensions taking for x the main direction and cutting the domain into four sub-domains in z direction. Each core computes one sub-domain. At the beginning the first core computes the first row and the other cores do not work, the first core (0) sends the solution of the last cell in the first row to the second core (1) and then the two first cores compute a row and the other cores do not work and so on until the last core is reached. A core c cannot work until it receives the last cell of core $c-1$. When the last core is reached, all the cores can compute in parallel until the first core reaches the end of the domain in x direction. That is why it is important to take the larger length in x direction. At this point, the first core waits until the last one has finished.
In three dimensions the domain is split into uniform sub-domains in both y and z directions as shown in the Figure 4. The core numbers go from left to right and bottom to top. So one core computes the solution in an elongated sub-domain containing all the cells in x direction but only a small piece in (y, z) planes. In the application presented in section 2, both computational domain limits in the y direction are artificial and to specify the non reflecting boundary condition, it is necessary to know the solution on the second or the penultimate cell. So this suggests to start in the middle of the domain and to propagate the computation both sides. This way the parallelism is improved but an even number of cores in the y direction is compulsory. So the Gauss-Seidel smoother is propagated both from left to right on the right part and from right to left on the left part of the domain and from bottom to top inside a sub-domain. That means that when the first core of the right part (3) has computed all the unknowns of the first cell it can send the values to the first core of the left part (2) that starts to compute immediately. Then the first core (3) goes on computing to the right and when it has computed the last cell of the first line it can send it to the next core on its right (4) that can start in its turn. Symmetrically when the first core of the left part (2) has computed the last cell of the first line to the left, it can send it to the next core on its left (1) that can start in its turn. In addition, the cores must

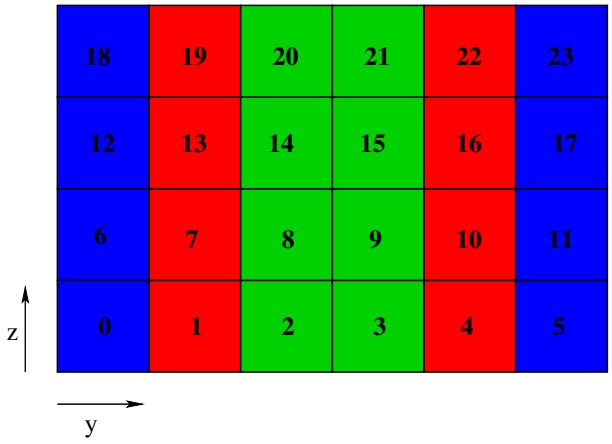

Figure 4: Example of cores numbering in three dimensions in a cross section (y, z) with 24 cores numbered from left to right and top to bottom. Core 3 starts to the right. Core 2 starts to the left as soon as core 3 sends it its first cell. Cores 4 and 1 can start when they have received the last cell of the first line from cores 3 and 2 respectively and so on. Core 9 can start when it has received the last line of core 3 and the first cell of the last line of core 2. Core 8 can start when it has received the last line of core 2, the first cell of the last line of core 3 and the first cell of the first line of core 9. In addition, core 9 will send its first line, core 8 will send its first left cell of the first line and core 10 will send its first right cell of the first line to core 3 for the next x plane. The cores of the same colour work in parallel but not in the same plane in x direction, when cores 21 and 20 start on the first plane in that direction cores 3 and 2 start the fourth plane. Cores 3 and 2 will finish the first ones and will wait until cores 23 and 18 finish.

send their first cell to their neighbour for the next iteration. At the end, when one core c has computed its whole sub-domain, it sends the whole last line up to the core above. In addition it has to send the right cell of the last line in diagonal to the core on its right-above side and the left cell of the last line in diagonal to the core on its left-above side. Reciprocally this core c will receive the neighbouring cells from its neighbours for the next iteration. All the receive and send directives are blocking communications.
Let us suppose there are $N1$, $N2$ and $N3$ cells in x, y and z directions respectively in the global domain, and each sub-domain goes from the cell 1 to the cell $N1$ in x direction and goes from the cell $D2$ to the cell $F2$ and from the cell $D3$ to the cell $F3$ in the parallel y and z directions. Let us point out to the reader that for the discretization of the Laplace operator we use a centered second order finite difference scheme that requires one fictitious row on each side of the sub-domain in the y and z directions. In the algorithm below are given the communications inter cores for the smoother using the four directions to indicate the core. For instance $Receive-left$ means from the core on the left (from core 9 for core 10 in the Figure 4) and $Receive-left-down$ means from the core on the left and below (from core 3 for core 10). So, receiving one cell from the left means receiving the 7 values of the cell $D2-1$ of the fictitious zone corresponding to a $Send-right$ of the cell $F2$ by the left core. In addition the cores send or receive either a cell or a full line of length $6(F2-D2+1)$ in z direction. The sketch of the solver is the following:

```
Do I = 1 to N1
  If ((D2 >= N2/2+1) and (D3 > 1)) Receive-line-down (I, D3-1)
                                   Receive-cell-left-down (I, D2-1, D3-1)
    If (F2 < N2) Receive-cell-right-down (I, F2+1, D3-1)
  End If
```

```
  If ((F2 <= N2/2  ) and (D3 > 1)) Receive-line-down (I, D3-1)
                                   Receive-cell-right-down (I, F2+1, D3-1)
    If (D2 > 1 ) Receive-cell-left-down (I, D2-1, D3-1)
  End If

  Do K = D3 to F3
    If ((D2 >= N2/2+1) and (F3 < N3) and (K == F3))
                                   Receive-line-up (I-1, F3+1)
                                   Receive-cell-left-up (I-1, D2-1, F3+1)
      If (F2 < N2) Receive-cell-right-up (I-1, F2+1, F3+1)
    End If
    If ((F2 <=  N2/2  ) and (F3 < N3) and (K == F3))
                                   Receive-line-up (I-1, F3+1)
                                   Receive-cell-right-up (I-1, F2+1, F3+1)
      If (D2 > 1 ) Receive-cell-left-up (I-1, D2-1, F3+1)
    End If
    If ((D2 >= N2/2+1) and (K > D3)) Receive-cell-left  (I, D2-1, K-1)
    If  (D2 >  N2/2+1)               Receive-cell-left  (I, D2-1, K)
    If ((F2 <= N2/2  ) and (K > D3)) Receive-cell-right (I, F2+1, K-1)
    If  (F2 <= N2/2  )               Receive-cell-right (I, F2+1, K)

    Do J = D2 to F2
      If ((D2 >= N2/2+1) and (F2 < N2) and (K > D3))
                                   Receive-cell-right (I, F2+1, K-1)
      If ((F2 <= N2/2  ) and (D2 > 1 ) and (K > D3))
                                   Receive-cell-left  (I, D2-1, K-1)

      COMPUTE the solution in the cell (I,J,K)

      If ((D2 >= N2/2+1) and (J = D2+1)) Send-cell-left  (I, D2, K)
      If ((F2 <= N2/2  ) and (J = F2-1)) Send-cell-right (I, F2, K)
    End Do

    If ((D2 >= N2/2+1) and (F2 < N2)) Send-cell-right (I, F2, K)
    If ((F2 <= N2/2  ) and (D2 > 1 )) Send-cell-left  (I, D2, K)
    If ((D3 > 1) and (K == D3+1)) Send-line-down (I, D3)
      If (D2 > 1 ) Send-cell-left-down  (I, D2, D3)
      If (F2 < N2) Send-cell-right-down (I, F2, D3)
    End If
  End Do

  If (F3 < N3) Send-line-up (I, F3)
    If (D2 > 1 ) Send-cell-left-up  (I, D2, F3)
    If (F2 < N2) Send-cell-right-up (I, F2, F3)
  End If
End Do
```

Let us point out to the reader that the number of *send* and *receive* is not the same as the *receive* opers in the left and the right parts of the domain.

5 Parallel algorithm of the prolongation and restriction operators

Let us point out to the reader that on the full computational domain as well as on the sub-domains it is necessary to add fictitious cells in all directions in order to perform easily the interpolations.

5.1 Prolongation operator

The operator P_{l-1}^{l} consists in interpolating the solution from a coarse grid $l-1$ to a finer one l. The important point in the algorithm is the size of the finer grid on the current core. When the dimensions $D2(l)$, $F2(l)$, $D3(l)$ and $F3(l)$ are equal to 1, $N2(l)$, 1 and $N3(l)$ respectively the two consecutive grids are not treated in parallel and there is nothing to do. As soon as the dimensions are different the grid l is computed in parallel and thus the prolongation is necessary only on the real size corresponding to $(D2(l), F2(l)) \times (D3(l), F3(l))$ of the sub-domain computed by the core. So it is only necessary to test if the indices $(I, J, K)_l$ of the unknowns of the fine grid are included in the sub-domain with extended fictitious cells to perform the interpolation.

5.2 Restriction operator

The operator R_{l}^{l-1} consists in interpolating the solution from a fine grid l to a coarser one $l-1$. Whatever is the sub-domain at level $l-1$, the restriction is performed only on the size of the sub-domain at level l. Then there are two cases: either the size of the sub-domain at level $l-1$ is identical to the size of the sub-domain at level l or not. In that last case the sub-domain at level $l-1$ is always larger than the sub-domain at level l and can even be the full domain. In the first case it is necessary to communicate the values of the unknowns in the fictitious cells between the cores. In the second case it is necessary to gather the restricted values of the smaller sub-domains corresponding to level l in order to obtain the restricted values on the larger sub-domains or domain at level $l-1$. This operation is performed using a $MPI\_ALLREDUCE$ routine with MPI_SUM. Using this routine it is required to give the dimensions of the full domain at level $l-1$ to perform the sum that is stored in a full array corresponding to the full domain. If several cores do the same calculation in a same sub-domain at level $l-1$, it is necessary to divide the values in the array by this number of cores as the sum performed by the $MPI\_ALLREDUCE$ routine adds the same value in the array several times. The values on a sub-domain different from the full domain at level $l-1$ can be obtained by picking the necessary values in the full array. In that case it is also necessary to communicate the values of the unknowns in the fictitious cells between the cores.

6 Full multigrid parallelism

Due to the finite difference approximation that uses a five-cell stencil in each direction for the convection terms, the fictitious zones contain two row cells and so a sub-domain must contain at least four row cells in each direction to avoid overlapping. The full multigrid parallelism consists in computing the maximum number of grids in parallel. On the computational domain Ω we choose to use the coarsest grid $G_1 : N1(1) \times N2(1) \times N3(1)$ cells of a uniform mesh with the three dimensions as small as possible but greater than or equal to four; and then to compute on L consecutive grids up to G_L obtained from dyadic refinement as $G_l : N1(l) \times N2(l) \times N3(l)$ cells where $N1(l) = 2^{l-1}N1(1)$. So, with $2^{L-3}N2(1) \times 2^{L-3}N3(1)$ cores it is possible to compute in parallel only the finest grid. This number can be seen as an upper limit of cores than can be used but in practice it is necessary to compute in parallel several grids, at least three to get

a good efficiency. So the limit is in fact $2^{L-5}N2(1) \times 2^{L-5}N3(1)$ cores to compute in parallel the three finest grids and to compute sequentially the coarsest ones. That means that each core compute the full domain from grid G_1 to grid G_{L-3}.
To increase the efficiency and the upper limit of the number of cores, it is possible for instance to use $2^{L-3}N2(1) \times 2^{L-3}N3(1)$ cores on grid G_L, $2^{L-4}N2(1) \times 2^{L-4}N3(1)$ cores on grid G_{L-1} and so on while the number of cells in each direction is greater than four. Which means that four cores will do the same job on grid G_{L-1}, sixteen cores will do the same job on grid G_{L-2} and so on. This is what we call the full multigrid parallelism, going from the finest grid to the coarsest one, a core will work on larger and larger sub-domains until it reaches the full domain.

7 Hybrid parallelism

In the sections above is presented MPI parallelism of the whole method but to take into account the specificity of a new generation of computers it is also interesting to consider hybrid MPI/OpenMP parallelism. Indeed computers like Intel Xeon Phi or IBM Blue Gene have nodes with a various number of cores and each core has 4 threads. In that case it can be judicious to perform MPI parallelism on the cores and to use the 4 threads for OpenMP procedures to increase the efficiency.
Some operations all along the code are suitable for OpenMP directives as they involve several loops without dependency. This is the case for the computation of the linear operator, the computation of the residuals or the interpolations in the prolongation or restriction operators as they require three do loops in I, J and K that can be written in any way. As the product $X \times Y \times Z$ is large the gain will compensate the opening of OpenMP to give a higher efficiency. In some cases it is useful to change the order of the DO loops putting the longer one in I in the outer place to increase the gain on a large number of cores.
However the main part of the CPU time is devoted to solve the linear system by means of the Gauss-Seidel smoother. Unfortunately, we have seen in section 4 that to have a good MPI parallelism it is necessary to put the larger loop in I as the outer loop but the complexity of its contents with many communications cannot be handled efficiently by OpenMP directives that always damage the efficiency. At the end, as all other parts of the program benefit of the OpenMP directives, the smoother requires a large part of the whole CPU time.

8 Applications to unsteady turbulent flows

The aim is to compute the flow over a simplified ground vehicle, namely the Ahmed body [1], on top of a road to compute the drag coefficient and then to apply active or passive control procedures to reduce this drag [12, 8, 6, 4]. Another application is to compute the flow over two following ground vehicles to study the effect of the distance between the two vehicles on the drag coefficient of both vehicles [11, 7].
So we consider a computational domain $\Omega = (0, 20) \times (0, 6) \times (0, 4)$ in three dimensions where the first square-back Ahmed body is set at 5.3 in x direction from the entrance section. The distance between the two bodies is set $D = 5H$ where H is the height of the body located between 0.17 and 1.17 from the road. The Reynolds number is $Re = 15,000$ and the non dimensional permeability coefficient is $K_P = 10^{16}$ in the fluid so that the term U/K_P vanishes and $K_P = 10^{-7}$ inside the bodies to get a coupling with the pressure term and recover Darcy equation [2]. Figure 5 shows the instantaneous z-vorticity field around the two bodies computed on seven consecutive grids.

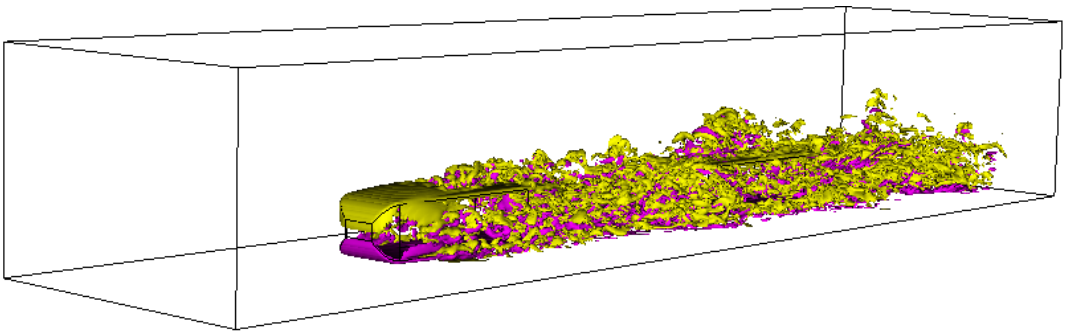

Figure 5: Instantaneous z-vorticity field around the two bodies in three dimensions.

9 Efficiency and scalability

In the application above there is a set of grid starting from $G1$ with $20 \times 6 \times 4 = 480$ cells. As said in section 2 the second grid $G2$ has $40 \times 12 \times 8 = 480 \times 8 = 3,840$ cells and so on with a dyadic refinement, each grid having 8 times the number of cells of the previous one. If $L = 8$, the finest grid $G8$ has $2,560 \times 768 \times 512 = 1,006,632,960$ cells and $4,030,201,856$ unknowns for the velocity and the pressure on staggered grids.
As said in section 6 the maximum number of cores that can be used on that grid $G8$ is $(2^5 \times 6) \times (2^5 \times 4) = 192 \times 128 = 24576$ and if we want to compute the three last grids with the same number of cores the maximum is then $(2^3 \times 6) \times (2^3 \times 4) = 192 \times 128 = 1536$. The question is to know if this number of cores will give the best efficiency. In addition to the numeric, it is necessary to take into account the architecture of the computers platform and the policy of the computer centre. Indeed we had the opportunity to work on the IBM Blue Gene computer of IDRIS in France. This platform has many nodes with 16 cores each and each core has 4 threads and $1GO$ of memory. In addition there is a dyadic distribution of the nodes and an invoice based on this distribution. So it is much more advantageous to reserve a number of nodes equal to 2^p. Thus, instead of choosing 1536, we have to take 1024 cores. Therefore on $G6$ each core has 24×16 cells, on $G7$ each core has 12×8 cells and on $G8$ each core has 6×4 cells in a cross section (y, z).

9.1 Efficiency or strong scalability

The first results concern only MPI parallelism with the high level of optimization proposed by the constructor (O5 giving the same results than the default O3). Let us point out that for our applications and specially for the communications inside the Gauss-Seidel smoother program it is strongly recommended to use buffered communications instead of synchronized communications (the default is buffered until 2048 bytes and synchronized beyond). The gain forcing buffered communications is up to 40% of the global CPU time.
According to the required memory the number of cores used on one node is 16 if possible. Of course to get the sequential time only one core on one node is used. Then, as the domain is split in the (y, z) section, the tests are performed on 2×2 cores, 4×4 cores and so on until 32×32 cores. On $G8$ for instance, because of the required memory, it is not possible to run on 64 cores using only 4 nodes. Indeed 16 nodes are required and thus the run is performed using 4 cores only on each node.
The times provided concern the solution obtained after three time steps on $G4$, $G5$, $G6$, $G7$ and $G8$. In the sequential case the ratio between two consecutive grids should be equal to 8. However this is not the case due to the memory access and the length of the cache as can be

seen on table 1. On the coarsest grids the ratio is lower at 6.8 but this ratio increases suddenly on $G6$ because of this. Of course this remark has a great impact on the weak scalability as the sequential time on $G8$ is 801 times greater than the sequential time on $G5$ instead of 512! Let us point out to the reader that the sequential times on grids $G7$ and $G8$ are extrapolated due to the lack of memory on one node. For instance on grid $G7$ it is possible to run on a single core only in a domain $\Omega = (0,5) \times (0,6) \times (0,4)$ instead of $\Omega = (0,20) \times (0,6) \times (0,4)$, so the time is multiplied by 4 to get the time on the large domain.

	G4	G5	G6	G7	G8	G4	G5	G6	G7	G8
Time in seconds	16	108	1141	10320	86512	7	54	726	7031	59553
Ratio		6.8	10.6	9	8.4		7.7	13.4	9.7	8.5

Table 1: Sequential time for three time steps on various grids, global time on the left and solver time on the right.

Once we have got the CPU times on five consecutive grids it is possible to get the efficiency or strong scalability of the MPI program when increasing the number of cores. The reader has to remind that a full multigrid parallelism is used, that means for instance that when $16 \times 16 = 256$ cores are used on $G6$, the 256 cores are used on grids $G5$ and $G6$, 64 cores are used on $G4$, 16 cores are used on $G3$, 4 cores are used on $G2$ and one single core on $G1$. In practice sets of 4 cores are doing the same job on $G4$, sets of 16 cores are doing the same job on $G3$, sets of 64 cores are doing the same job on $G2$ and the 256 cores are doing the same job on $G1$. The times are summarized in table 2, the number of cores on a grid level is limited by the number of cells in the z direction divided by four and generally the two or three finest grids benefit of the higher number of cores. We see that the ratio of times between the sequential case and the 4 cores case converges to $1/4$ when the grid or the size of the problem increases. On the contrary, with respect to a grid level, the two consecutive number of cores ratio of time can be closer to 1 when the number of cores is too high. On $G4$ the results are already bad on 4 cores whereas on $G8$ the results are still good on 256 cores.

	T G4	T G5	T G6	T G7	T G8	R G4	R G5	R G6	R G7	R G8
1 core	16	108	1141	10320	86512					
4 cores	8	33	319	2574	21628	0.5	0.31	0.28	0.25	0.25
16 cores	7	16	100	763	6128	0.88	0.48	0.31	0.3	0.28
64 cores		10	32	230	1632		0.63	0.32	0.3	0.27
256 cores			23	86	555			0.72	0.37	0.34
1024 cores				62	355				0.72	0.64
4096 cores					304					0.86

Table 2: CPU time (T) and two consecutive number of cores ratio of time (R) with respect to the number of cores on five levels of grids for MPI parallelism.

Using these times it is easy to compute the strong scalability (in figure 6). It appears that with MPI the best efficiency is always obtained for 4 cores, then the efficiency decreases quite fast on coarse grids and much slowly on fine grids. For instance the efficiency is still greater than 0.8 for 64 cores on $G8$ but decreases to 0.61 for 256 cores and to 0.24 for 1024 cores although on this last case the three finest grids use the 1024 cores. Therefore the results are a little bit disappointing using the full MPI parallelism.
Now the results with the hybrid MPI/OpenMP parallelism are presented. Let us recall that the

first step is to add OMP directives to benefit from the 4 threads available by core. However it appears that only the compiler option OMP adds a lot of optimization and change the CPU times even on a single thread! Then the OMP directives for the DO loop parallelism on the larger loops in the x direction change again the CPU times.
These CPU times are given in table 3. They are close to those obtained with MPI parallelism on coarse grids or for a low number of cores. But they are very different for large grids on a high number of cores, for instance on $G8$ with 4096 cores the time is almost divided by four. Consequently the efficiency changes drastically. Moreover on the finest grid, the two consecutive number of cores ratio of time is more than optimal for 256 cores as it is less than one fourth.

	T G4	T G5	T G6	T G7	T G8	R G4	R G5	R G6	R G7	R G8
1 core	14	82	1123	10868	88024					
4 cores	7	28	311	2615	22006	0.5	0.34	0.28	0.24	0.25
16 cores	7	15	97	736	5972	1	0.54	0.31	0.28	0.27
64 cores		10	27	197	1321		0.67	0.28	0.27	0.22
256 cores			18	53	274			0.67	0.27	0.21
1024 cores				35	123				0.66	0.45
4096 cores					81					0.66

Table 3: CPU time (T) and two consecutive number of cores ratio of time (R) with respect to the number of cores on five levels of grids for hybrid parallelism.

The figure 6 shows again that on the one hand the CPU times are close for coarse grids and a small number of cores and on the other hand there are large discrepancies for the fine grids and a large number of cores. Indeed now the efficiency is not always decreasing when the number of cores increases on a given grid. For instance on $G8$, the best efficiency is obtained for 256 cores and is even greater than one. Which is very good news for the numerical simulations as we can use a large number of cores on the fine grids with a very good efficiency.

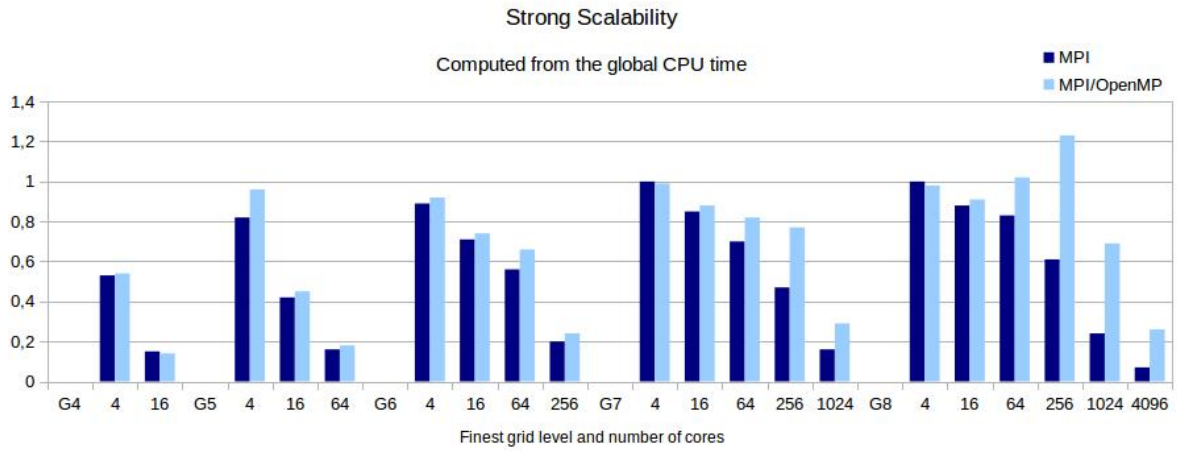

Figure 6: Comparison of the strong scalability of the MPI (dark) and the MPI/OpenMP (light) parallelism on five levels of grids for the global program.

A crucial point is to analyze the CPU times of the smoother to see if the MPI parallelism described in section 4 is efficient enough. We know that this smoother requires a lot of time and is difficult to parallelize. Surprisingly it appears that the efficiency for this solver is close and

quite often higher than the efficiency obtained for the whole program (see figure 7).

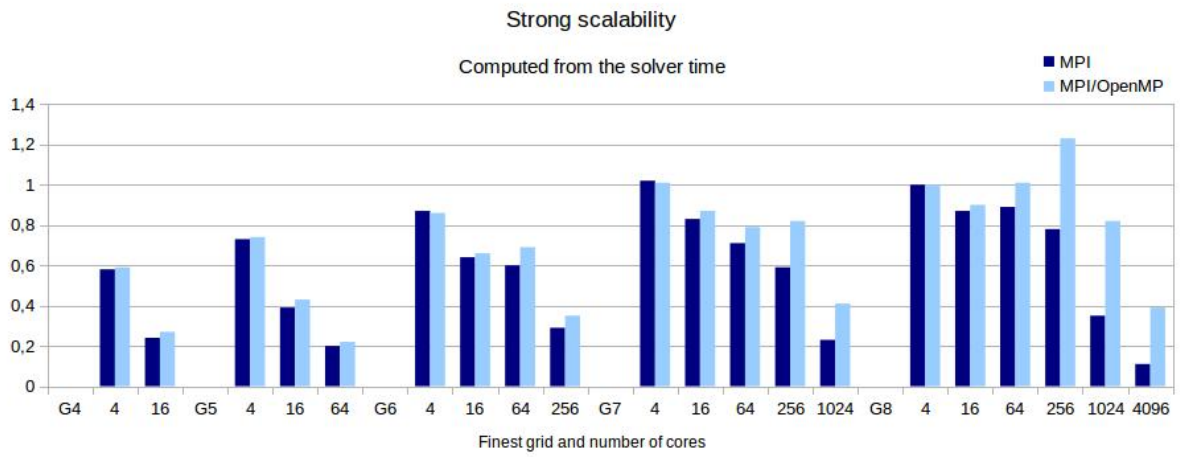

Figure 7: Comparison of the strong scalability of the MPI (dark) and the MPI/OpenMP (light) parallelism on five levels of grids for the smoother.

An interesting remark is the percentage of the CPU time spent for the smoother with respect to the global CPU time varies a lot when the number of cores increases with MPI parallelism. For instance on $G8$, it goes from almost 70% for a low number of cores to less than 50% for 1024 cores as shown in figure 8 (left). The same percentage is shown for the MPI/OpenMP case (figure 8 right). It stays close to 70% for every number of cores except for 1024 cores that give 58%. This is due to the fact that there are no OMP directives in the smoother subroutine.

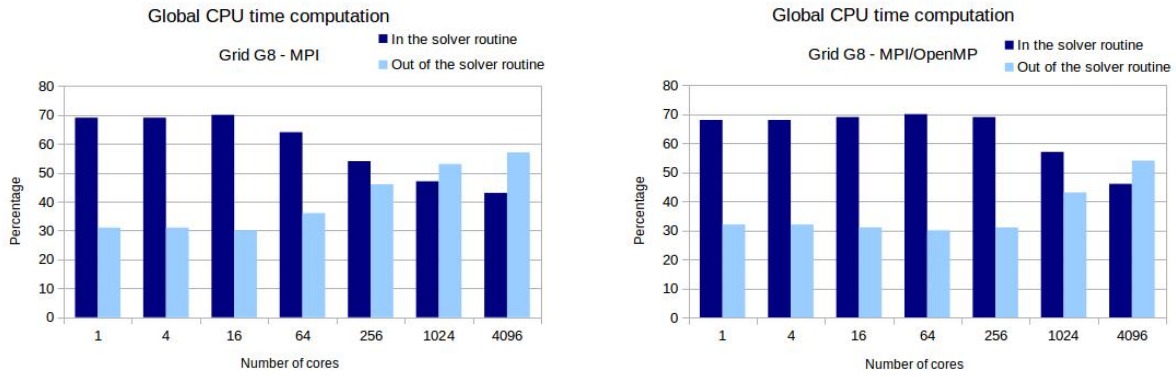

Figure 8: Percentage of the CPU time of the smoother (dark) and of the CPU time of the rest of the program (light).

9.2 Weak scalability

Now we would like to observe the weak scalability with respect to the cross section (y, z), that means that when the level of grid increases, so does the number of cores in such a way that the number of cells per core in a cross section (y, z) is the same. For instance there are $48 \times 32 = 1536$ cells on $G4$ in this cross section, so if we take 1 core on $G4$, 4 cores on $G5$, 16 cores on $G6$, 64 cores on $G7$ and 256 cores on $G8$ the cores have to solve 1536 cells in each case. Then the figure

9 shows that the weak scalability decreases from 0.96 for $G5$ to 0.46 for $G8$ in the MPI case. However this weak scalability increases to 0.93 for $G8$ in the MPI/OpenMP case. Once again we see that high scalability can be reached with the hybrid parallelism. Further if we refer to 4 cores on $G4$ having $24 \times 16 = 384$ cells each in the cross section (y, z), the weak scalability still decreases from 0.94 for $G5$ to 0.34 for $G8$ in the MPI case but is close to one in every case for the hybrid parallelism even for the 1024 cores on $G8$.

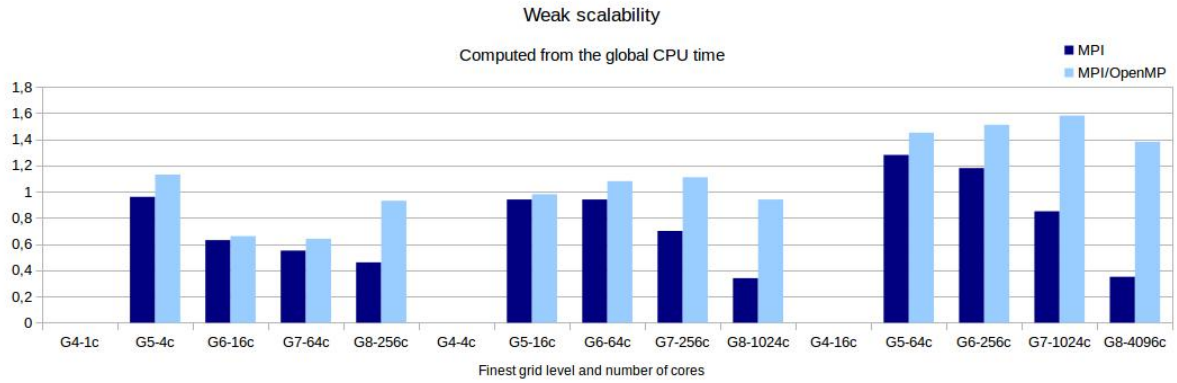

Figure 9: Comparison of the weak scalability of the MPI (dark) and the MPI/OpenMP (light) parallelism on five levels of grids for the global program.

Another possibility is to look at the relative weak scalability. That means to compute the weak scalability between two consecutive grids, for instance considering $G5$ with 16 cores and $G6$ with 64 cores. Then the figure 10 shows that this relative weak scalability is mostly greater than 0.8 even for MPI parallelism and that peaks as high as 1.4 can be reached with the hybrid parallelism.

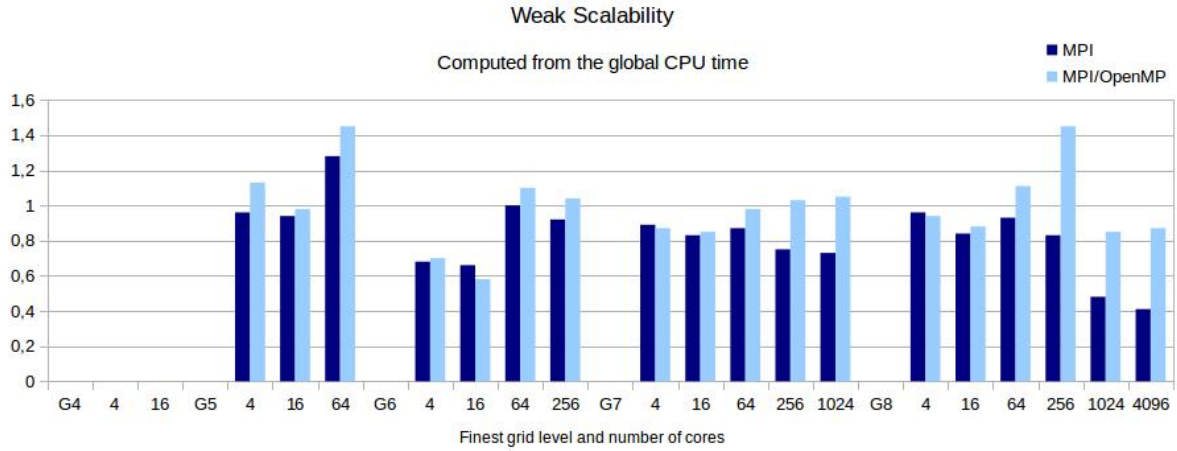

Figure 10: Comparison of the relative weak scalability of the MPI (dark) and the MPI/OpenMP (light) parallelism on two consecutive grids for the global program.

10 Conclusions

The goal of this paper is to show that it is possible to reach high scalability results for solving Navier-Sokes system with a multigrid solver using a cell by cell Gauss-Seidel smoother. Despite the difficulties due to the smoother and the multigrid solver the program is fully parallel with MPI or MPI/OpenMP directives.
Taking benefit of the platform made of nodes with 16 cores, each core having 4 threads, the efficiency with the hybrid parallelism is close to one for a medium number of cores linked to the finest grid. A run can be quite efficient on the one billion cells mesh on 1024 cores. Unfortunately the number of cores can not be increased much more than that as four cells are needed in the y and z directions in each sub-domain. So, when the number of cores is increased, the number of grids using all the cores is reduced due to the multigrid algorithm. In addition it is not possible to extend the parallelism to the x direction because of the smoother.
In conclusion it is possible to solve complex flows by DNS involving billions of unknowns using a few thousands of cores with an efficiency close to one.

Acknowledgements

The authors would like to thank Pierre Gay from the computer centre who set up the first MPI instructions in the two dimensions version of our software.
The numerical simulation presented in section 8 on seven grids has been run on PLAFRIM platform supported by IMB University of Bordeaux and INRIA Bordeaux - Sud Ouest.
The other simulations presented in section 9 were granted access to the HPC resources of IDRIS under the allocation 2015-6651 made by GENCI (Grand Equipement National de Calcul Intensif).

References

[1] Ahmed, S. R., Ramm, G. & Faltin, G., Some Salient Features of the Time -Averaged Ground Vehicle Wake, *SAE-Paper* **840300**, 1984.

[2] Angot, Ph., Bruneau, C.-H. & Fabrie, P., A penalization method to take into account obstacles in incompressible viscous flows, *Numerische Mathematik* **81**, 497-520, 1999.

[3] Bruneau, C.-H., Boundary conditions on artificial frontiers for incompressible and compressible Navier-Stokes equations, *Math. Model. Num. Anal.* **34**, n° 2, 2000.

[4] Bruneau, C.-H., Creusé, E., Depeyras, D., Gilliéron, P. & Mortazavi, I., Coupling passive and active techniques to control the flow past the square back Ahmed body, *Comp. & Fluids*, **38**, n° 10, 2010.

[5] Bruneau, C.-H. & Fabrie, P., Effective downstream boundary conditions for incompressible Navier-Stokes equations, *Int. J. Num. Meth. Fluids* **19**, 693-705, 1994.

[6] Bruneau, C.-H., Gilliéron, P. & Mortazavi, I. , Passive control around the two-dimensional square back Ahmed body using porous devices, *J. Fluids Eng.* **130**, 1-33, 2008.

[7] Bruneau, C.-H., Khadra, K. & Mortazavi, I., Analysis and active control of the flow around two following Ahmed bodies, *Proceedings FEDSM13*, 2013.

[8] Bruneau, C.-H. & Mortazavi, I., Passive control of the flow around a square cylinder using porous media, *Int. J. Num. Meth. Fluids* **46**, 415-433, 2004.

[9] Bruneau, C.-H. & Saad, M., The 2D lid-driven cavity problem revisited, *Comp. & Fluids* **35**, 326-348, 2006.

[10] Hackbush W., Multigrid methods and applications, *Springer-Verlag*, 1985.

[11] Han, X. , Krajnović, S., Bruneau, C.-H. & Mortazavi, I., Comparison of URANS, PANS, LES and DNS of flows around simplified ground vehicles with passive flow manipulation, *Proceedings DLES9*, 2013.

[12] Krajnović, S. & Davidson, L., Numerical study of the flow around the bus-shaped body, *AS-ME J. Fluids Eng.* **125**, 2003.