Contents lists available at ScienceDirect

Journal of Computational Science

journal homepage: www.elsevier.com/locate/jocs

Highly parallel computing of a multigrid solver for 3D Navier–Stokes equations

Charles-Henri Bruneau[a,b,*], Khodor Khadra[a]

[a] University of Bordeaux IMB, CNRS UMR 5251, 351, cours de la Libération, F-33405 Talence, France
[b] INRIA Bordeaux-Sud-Ouest Team MEMPHIS, France

ARTICLE INFO

Article history:
Received 17 May 2016
Received in revised form
15 September 2016
Accepted 17 September 2016
Available online 26 September 2016

Keywords:
Navier–Stokes equations
Multigrid solver parallelism
Gauss–Seidel smoother parallelism
Strong and weak scalability
MPI and hybrid MPI/OpenMP parallelism

ABSTRACT

In order to efficiently obtain all 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 straightforwardly parallelizable. Moreover 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 us to get a high strong and weak scalability of the whole parallel software. Comparison is done between MPI and hybrid MPI/OpenMP parallelism.

1. Introduction

It is nowadays impossible to perform high performance computing without strong parallelism which yields high efficiency on as many cores or threads as possible. The direct numerical simulation (DNS) of complex flows requires a large number of degrees of freedom. This 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) [16]. Because of the increase of computer resources today, it is possible to use DNS in two-dimensions and even in three-dimensions when the simulations are performed with an efficient parallel software.

Parallelism has quite a long history starting with some pioneering clusters in the sixties [26] and transputers in the late seventies [4,19,23]. Nearly fifty years ago, some researchers already thought that a way to improve computational power was to put some microprocessors in a row and to teach them how to communicate. At the same period other researchers had another idea that consisted in transforming sequential operations into vectorized operations. This idea gave rise to the famous CRAY computers able to perform for instance 64 operations at the same time inside a loop. Since then there were many attempts to develop various parallel computers and more recently the spreading of computer clusters all over the world [12]. The two main ideas above are linked respectively to distributed and shared memory and gave rise to the Message Passing Interface (MPI) in 1993 and to the Open Multi-Processing (OpenMP) in 1997 that are nowadays commonly used. On new computer architectures, it is possible to couple the two libraries in order to perform hybrid parallelism MPI/OpenMP, MPI on different nodes or cores and OpenMP on different cores or threads as the nodes of the computer clusters have generally several cores with several threads each.

The parallelization of CFD softwares started with the development of parallel computers, specially to solve linear systems using for instance gradient type methods that are very easy to parallelize. For Navier–Stokes equations, a well-known method is to use a projection method involving a Poisson solver to get the pressure. Thus the parallelism is achieved with MPI directives at different steps of the algorithm and for solving a linear system at each time step. A lot of papers have been written on this subject: In [3] a good efficiency is achieved up to 60 processors, in [32,31] on big meshes of size 240^3 an efficiency of up to 0.75 for MPI on 48 cores or MPI/OpenMP on 32 cores is obtained. In [28] a preconditioned GMRES method is used to solve Navier–Stokes equations with 3.3 billion unknowns

* Corresponding author at: University of Bordeaux IMB, CNRS UMR 5251, 351, cours de la Libération, F-33405 Talence, France.
E-mail addresses: Charles-Henri.Bruneau@math.u-bordeaux.fr (C.-H. Bruneau), Khodor.Khadra@math.u-bordeaux.fr (K. Khadra).

http://dx.doi.org/10.1016/j.jocs.2016.09.005
1877-7503/© 2016 Elsevier B.V. All rights reserved.

on 65536 cores and a good relative efficiency is reached on fine grids.

Another interesting aspect is the use of the multigrid method that works for all but the coarsest grids. These cannot be parallelized as they do not contain enough unknowns. In the literature we find a survey in 2006 exploring the parallelism techniques for multigrid solvers [18]. The multigrid method can be applied in two ways, either by inverting a linear system using the multigrid as a preconditioner or by fully resolving a given problem on a series of grids. Many authors use the first method: In [13] they first solve a Laplace problem with very good efficiency on 32,768 cores, then they apply the multigrid solver to the Poisson equation. In [24] a multigrid preconditioner is used to solve Stokes problem with finite elements approximation. They use the PETSc library [25] and obtain a good relative speedup using up to 4,096 cores and fine meshes of size 384^3. In [14] a multigrid solver is applied to solve the Stokes problem. The method is run on an Intel Xeon cluster up to 30,720 cores and on IBM Blue Gene up to 122880 cores with 1 to 4 threads (these computers are the ones we have access to). A relative efficiency is provided that decreases to 0.55 when multiplying the number of cores by 16. In [17] the tests range up to 29 billion points involving 11 grid levels in the multigrid preconditioner for the Poisson problem on 13,824 cores. The weak scalability is given for 2.1 million unknowns per core and is very good.

These last few years a very good scalability was achieved solving linear systems with the help of PETSc library or not. In our opinion the full multigrid method is much more efficient to get the Navier–Stokes solution. However it is not so easy to get good parallel performances as on the one hand the parallelism is not efficient on the coarsest grids and on the other hand the Gauss–Seidel smoother cannot be parallelized straightforwardly. There are very few results available. For instance in [20] the author uses a 2D sub-domain decomposition into 32 rectangular domains and gets a relative efficiency of up to 0.86 and a global efficiency that decreases from 0.76 to 0.56 when increasing the number of cores. In [21] the author uses a multigrid solver with Gauss–Seidel smoother and gets better results with hybrid MPI/OpenMP parallelism than with MPI parallelism on up to 24,576 cores that compute 500,000 unknowns each. Good results are obtained for the weak scalability. A fully MPI or hybrid MPI/OpenMP parallelism are also studied on unstructured meshes using up to 32 processors in [29]. The results show that the best score is obtained using a maximum of MPI processes, for instance 32 MPI processes are faster than 16 MPI ones and 2 OpenMP threads. It was shown in [30] that a cell-by-cell Gauss–Seidel smoother is much more efficient to get convergence in the full multigrid method. This smoother is studied in [11] on up to 256 processors in two-dimensions. They show for instance that they can get a weak efficiency as high as 0.93 on 4 cores on fine grids containing about one million finite elements but this weak efficiency decreases to 0.78 on 64 cores.

This literature review is far from being exhaustive, indeed we can find in the literature other ideas to get parallelism. For instance in [1] the authors study a parallelization of Navier–Stokes equations in space and time on up to 8 processors and get an efficiency between 0.87 and 0.61 when increasing the number of cores from 2 to 8 on a 129^3 mesh. A recent study using multi-color ordering gives results with OpenMP [27] that are very good on 4 threads and quite good until 16 threads. In [10] the domain decomposition is studied in details and its application to parallel implementation with numerical experiments up to 8192 cores show good relative efficiency.

In this work we use a method based on a full 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

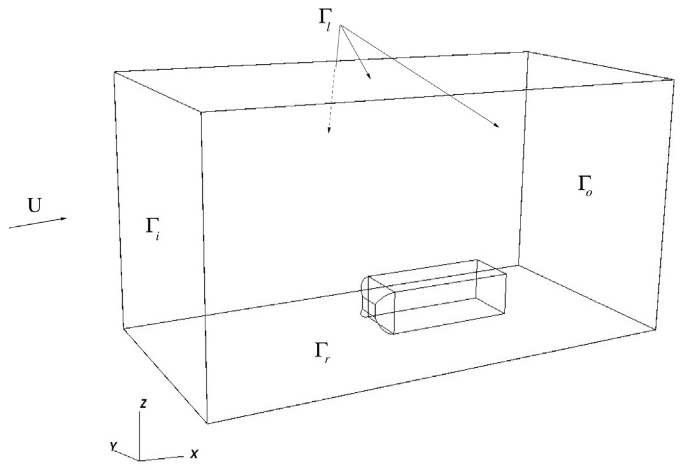

Fig. 1. Computational domain around a simplified vehicle on top of a road.

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 possible efficiency. 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. As we use the volume penalization method to handle obstacles immersed in a fluid, the computational domain is a 3D box on which a domain decomposition can be easily applied.

In the following the modelling and the numerical method to approximate Navier–Stokes equations are given, then the multigrid solver. Next, the paper focuses 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 (Fig. 1). To ease the 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 [2,22]:

$$\begin{array}{ll}
\partial_t U + (U\cdot\nabla)U - \dfrac{1}{Re}\Delta U + \dfrac{U}{K_P} + \nabla p = 0 & \text{in } \ \Omega \times I \\
\operatorname{div} U = 0 & \text{in } \ \Omega \times I \\
U(0,.) = U_0 & \text{in } \ \Omega \\
U = U_\infty & \text{on } \ \Gamma_i \cup \Gamma_r \times I \\
\sigma(U,p)\ \ n + \dfrac{1}{2}(U\cdot n)^-(U - U_{ref}) = \sigma(U_{ref}, p_{ref})\ \ n & \text{on } \ \Gamma_l \cup \Gamma_o \times I
\end{array} \qquad (1)$$

where U is the velocity field, p the pressure, $I = (0, T)$ with T the simulation time, $Re = \rho\bar{U}H/\mu$ is the Reynolds number, $K_P = \rho k \bar{U}/\mu\Phi H$ is the non-dimensional coefficient of permeability of the medium without gravity. In these two numbers without dimension, ρ is the

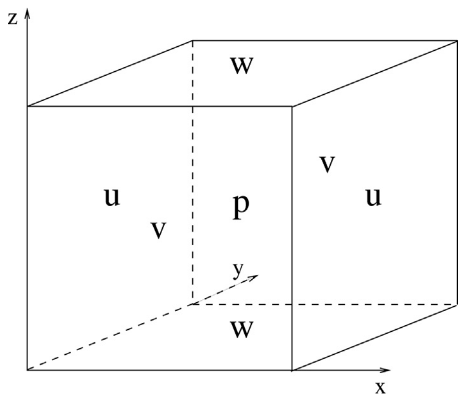

Fig. 2. The staggered unknowns in a cell.

density of the fluid, $\bar{U}$ 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 conditions require to know the flow at infinity U_∞, the tensor σ and the reference flow (U_{ref}, p_{ref}) used to write the traction [7], in practice this reference flow is taken as the flow computed just before the exit section [5]. This last boundary condition conveys properly the vortices downstream without any reflection on the artificial frontier than can be located close to the body.

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 all 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 Fig. 2. Indeed, the divergence operator at the pressure point in the center of the cells is directly obtained at second order with the values of the velocity at the center of the faces in 3D. The time discretization is achieved using a second-order Gear scheme 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. Thus at each time step the system to solve reads:

$$\begin{cases} \dfrac{3U^m}{2\delta t} - \dfrac{1}{Re}\Delta U^m + \dfrac{U^m}{K_P} + \nabla p^m = \dfrac{2U^{m-1}}{\delta t} - 2(U^{m-1}\cdot\nabla)U^{m-1} - \dfrac{U^{m-2}}{2\delta t} + (U^{m-2}\cdot\nabla)U^{m-2} & \text{in } \Omega \\ \nabla\cdot U^m = 0 & \text{in } \Omega \\ U^m = U_\infty & \text{on } \Gamma_i \cup \Gamma_r \\ \sigma(U^m, p^m)\ \ n + \dfrac{1}{2}(U^{m-1}\cdot n)^-(U^{m-1} - U_{ref}) = \sigma(U_{ref}, p_{ref})\ \ n & \text{on } \ \Gamma_l \cup \Gamma_o \end{cases} \tag{2}$$

to get the solution at time $t^m = m\delta t$ where t^{m-2} and t^{m-1} are the previous times. The CFL condition related to the convection term requires a time step δt of the order of magnitude of the space step h as $\bar{U} = (1, 0, 0)$ [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 $40 \times 12 \times 8 = 480 \times 8 = 3840$ cells and so on, each grid having 8 times the number of cells of the previous one. If $L = 8$, the finest grid has $2560 \times 768 \times 512 = 1,006,632,960$ cells and 4, 030, 201, 856 unknowns because of the staggered cells. This is the least number of unknowns required to perform a direct numerical simulation at Reynolds numbers up to 30, 000.

3. Full multigrid solver of Navier–Stokes equations

The discretization of the coupled velocity–pressure system of equations above yields to solve a discrete linear system $A_h V_h^m = B_h^{m-1}$ where A_h represents the linear part of the discrete operator, B_h^{m-1} is the discrete right hand side (previous times and convection terms) and $V_h^m = (U_h^m, p_h^m)$ is the approximate solution to compute at time $t = m\delta t$. Despite the discrete system is linear, we solve it by the FAS (Full Approximation Storage or Scheme) nonlinear multigrid algorithm [19] as the artificial boundary condition is nonlinear. Indeed this boundary condition uses the flow computed just before the exit section as the reference flow. Which implies also to have a progression of the smoother to the x direction. In addition to impose this boundary condition on Γ_l, the domain is split into two parts in the y direction, left and right starting from the middle and the smoother moves from the road to the top in the z direction. Let us note that the fact that the domain is split into two parts in the y direction increases the efficiency of the initialization process as two rows are computed in parallel. The sequence of grids is denoted G_l with $1 \le l \le L$. Thus, the multigrid algorithm using a V-cycle procedure is illustrated below for the computation of the fine grid solution $V_h^m = V_L^m$ on grid G_L.

```
For $q = 1$ to number_of$_V$-cycles do
$\tilde{V}_L^q = S_L^{(\nu_1)}(A_L, V_L^{q-1}, B_L^{m-1})$
Correction on coarse grids
    For l = L-1 to 1 by −1 do
    $\bar{V}_l^q = R_l^{l+1}\tilde{V}_{l+1}^q$
    $B_l^q = R_l^{l+1}(B_{l+1}^q - A_{l+1}\tilde{V}_{l+1}^q) + A_l\bar{V}_l^q$
    $\tilde{V}_l^q = S_l^{(\nu_1)}(A_l, \bar{V}_l^q, B_l^q)$
Updating of the fine grids
    For l = 2 to L do
    $\hat{V}_l^q = \tilde{V}_l^q + P_{l-1}^l(V_{l-1}^q - \bar{V}_{l-1}^q)$
    $V_l^q = S_l^{(\nu_2)}(A_l, \hat{V}_l^q, B_l^q)$
Convergence test
$if\ \ ||B_L^{m-1} - A_L V_L^q|| \le \epsilon$ stop iterations
```

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} is the linear interpolation operator from the finer grid G_{l+1} and the prolongation P_{l-1}^l is the linear interpolation operator from the coarser grid G_{l-1}. The smoothing operator S performs ν iterations of an iterative cell-by-cell method that leads to solving a 7×7 linear system corresponding to the 7 unknowns of a cell. Let us point out to the reader that this FAS multigrid algorithm provides the solution of Navier–Stokes system on each grid, even the coarsest. Indeed, as our coarsest grid is very coarse (for instance $20 \times 4 \times 4$ cells), the results with the smoother on that grid is very close to the one obtained by a direct method.

$$\begin{pmatrix} \alpha & 0 & 0 & 0 & 0 & 0 & 1/h \\ 0 & \alpha & 0 & 0 & 0 & 0 & -1/h \\ 0 & 0 & \alpha & 0 & 0 & 0 & 1/h \\ 0 & 0 & 0 & \alpha & 0 & 0 & -1/h \\ 0 & 0 & 0 & 0 & \alpha & 0 & 1/h \\ 0 & 0 & 0 & 0 & 0 & \alpha & -1/h \\ -1/h & 1/h & -1/h & 1/h & -1/h & 1/h & 0 \end{pmatrix} \begin{pmatrix} u^m_{i,j,k} \\ u^m_{i+1,j,k} \\ v^m_{i,j,k} \\ v^m_{i,j+1,k} \\ w^m_{i,j,k} \\ w^m_{i,j,k+1} \\ p^m_{i,j,k} \end{pmatrix} = \begin{pmatrix} Du_{i,j,k} \\ Du_{i+1,j,k} \\ Dv_{i,j,k} \\ Dv_{i,j+1,k} \\ Dw_{i,j,k} \\ Dw_{i,j,k+1} \\ 0 \end{pmatrix}$$

In this system $\alpha = (3/2\delta t) + (6/Reh^2) + (1/K_p)$ 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 $(B_l^{m-1})_{i,j,k}$. We solve this 7×7 coupled system directly. Indeed, eliminating the first six unknowns in the first six equations we get $p^m_{i,j,k}$, then the other unknowns follow. Let us point out that the seventh equation ensures the divergence-free constraint in each cell.

4. Parallel algorithm of the Gauss–Seidel smoother

The iterative method used as smoother is a cell-by-cell Gauss–Seidel method. Indeed the Jacobi method, despite it is easily parallelized, does not converge as the reference flow in the traction boundary condition must be taken at the same iteration. The smoother is performed on each grid including the coarsest one. Some authors use a direct solver on the coarsest one but this does not change the convergence in our application. Due to the staggered grids, 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 dependency as for computing the solution in the next cell it is necessary to have the new solution on the previous one. Taking into account the backward dependency, it is not possible to write a parallel algorithm in one of the three dimensions. Here the larger length corresponds to the flow direction that is chosen in the x direction. Thus the domain is cut into uniform sub-domains in both y and z directions. This is a strong limitation as the parallelization is achieved on two dimensions instead of three. The result will be a good efficiency on a few thousands of cores on fine grids instead of tens or hundreds thousands of cores if it was possible to parallelize the three dimensions. To illustrate this we show in Fig. 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 the 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 Fig. 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 traction 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

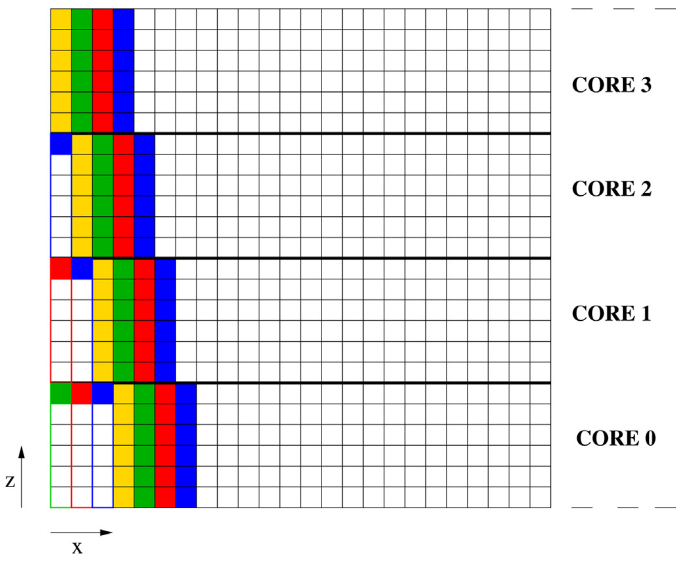

Fig. 3. Propagation of parallel algorithm of cell-by-cell Gauss–Seidel smoother in two dimensions. Core 1 can start to work when core 0 has sent its upper cell of the first row coloured in green. The yellow color represents the beginning of the full parallel computing of the four cores.

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 send their first cell to their neighbour for the next iteration (see the right part of Fig. 4). 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 Fig. 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:

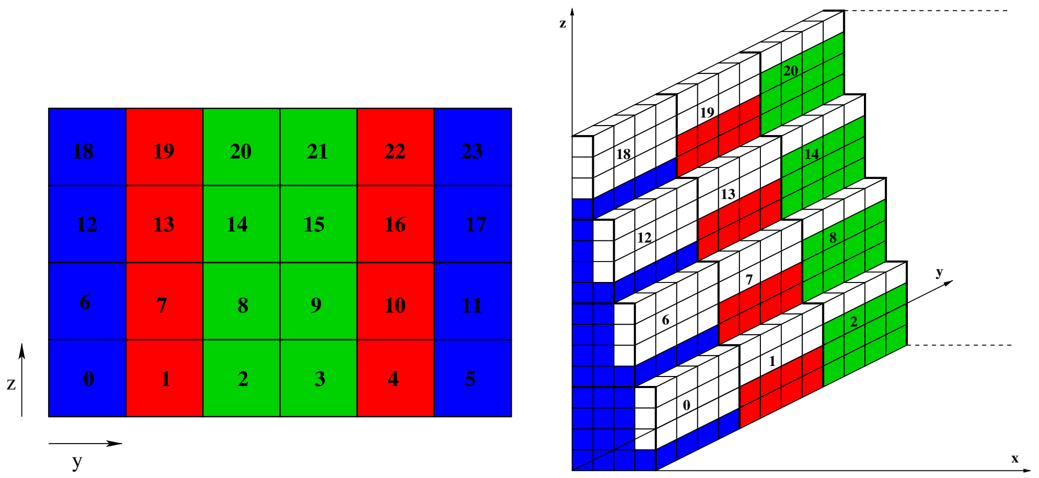

Fig. 4. Example of cores numbering in three dimensions in a cross section (y, z) with 24 cores numbered from left to right and bottom to top. Core 3 starts to the right. Core 2 starts to the left as soon as core 3 sends it its first cell. Cores 4 and 1 can start when they have received the last cell 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 as shown on the right for the left hand part of the domain. 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.

```
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* operates 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

In our methodology each core computes on its corresponding domain on each grid level from L to 1. All the cores compute on their given sub-domain on the finest grids and generally compute on the full domain on grid G_1 (see next section). In between they can compute on intermediate sub-domains larger than their given sub-domain and smaller than the full domain. In addition the restriction operator is not straightforward. Indeed, because of the staggered grids, the restriction operator is not a projection. 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 an *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 the 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 coarse-grids 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 coarse-grids 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 computes the full domain from grid G_1 to grid G_{L-3}. Indeed, instead of having one core doing the job on the coarse grids and then sending the solution to the other cores, we have chosen to compute the solution on the coarse grids on all the cores as the storage of the coarse grids is negligible compared to the finest grids. Consequently no communications are required and all the cores are perfectly synchronized.

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 coarse-grids 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 a computer like IBM Blue Gene has nodes with 16 cores and each core can use 4 threads. In that case it can be judicious to perform MPI parallelism on the cores (one MPI process per core) 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 elapsed real 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 MPI communications cannot be handled efficiently by OpenMP directives that always damage the efficiency. So the hybrid parallelism is not used explicitly for the smoother. Nevertheless the compilation option *qsmp = omp* saves 20% of the elapsed real time. At the end, as all the other parts of the program benefit of the OpenMP directives, the smoother requires a larger part of the whole elapsed real time.

8. Applications to unsteady turbulent flows

The simulations concern either the flow over a simplified ground vehicle on top of a road with active or passive control procedures to reduce the drag coefficient [6] or 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 [16,8].

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 $d = 3.625H$ 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].

The solution is computed on a finest grid involving 4 billions unknowns and the simulation time is chosen large enough to get realistic mean flows. So a simulation requires 40, 000 time steps. To improve the efficiency we initialize the solution on the finest grid with the solution obtained on the previous grid from a much

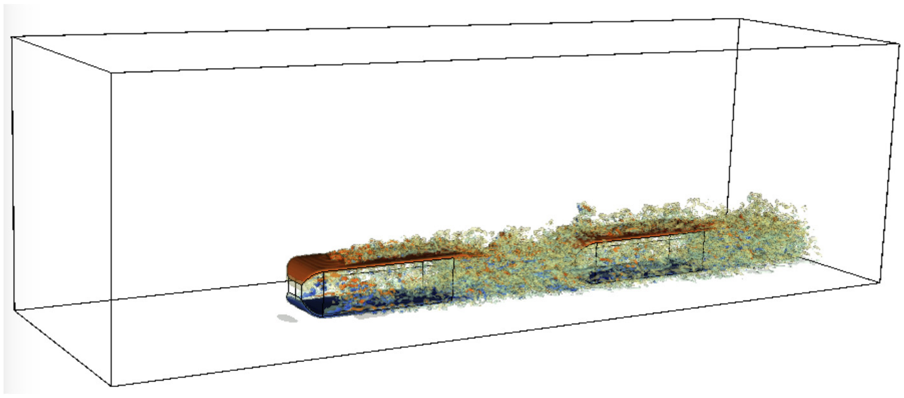

Fig. 5. Instantaneous z-vorticity field around the two bodies in three dimensions.

cheaper simulation, mimicking the full multigrid method. The precision required on the residual is $\epsilon = 10^{-4}$ that is reached after a few V-cycles as the solution is well initialized. During the computation one to four V-cycles are necessary to get the convergence, these V-cycles are performed with $\nu_1 = 2$ iterations down to the coarsest grid and $\nu_2 = 1$ iteration the other way to minimize the number of iterations on the finest grid. The global cost of the simulation is proportional to the total number of V-cycles necessary to reach the final time. This number is about 100, 000 and requires about eight days of elapsed real time on 384 cores for instance.

Fig. 5 shows the instantaneous z-vorticity field around the two bodies computed on eight consecutive grids.

9. Efficiency and scalability

In the application above there is a set of grids starting from $G1$ with $20 \times 6 \times 4 = 480$ cells. As said in Section 2 the second grid $G2$ has $40 \times 12 \times 8 = 480 \times 8 = 3840$ 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 $2560 \times 768 \times 512 = 1{,}006{,}632{,}960$ cells and 4,030,201,856 unknowns for the velocity and the pressure on staggered grids.

So (see Section 6) the maximum number of cores that can be used on that grid $G8$ is $(2^5 \times 6) \times (2^5 \times 4) = 192 \times 128 = 24{,}576$ 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 computer 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 1 GB 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 to compute 6×4 cells, on $G7$ each core has to compute 12×8 cells and on $G8$ each core has to compute 24×16 cells in a cross section (y, z).

Let us give the useful definitions required to appreciate the performances of the parallelization. The (absolute) speedup is the ratio of the sequential time (on one core) to the parallel time on nc cores $S = T_{seq}/T_{par}(nc)$ on a given grid. The ideal ratio must be equal to the number of cores. The (absolute) efficiency or strong scalability is the speedup on the number of cores $E_f = S/nc$. Thus the ideal number is one. Sometimes the sequential time cannot be obtained due to the size of the problem. Therefore we use the relative speedup or relative strong scalability evaluating S or E_f with respect to the time obtained on a smaller number of cores nc_{ref} different from one setting $S = nc_{ref} * T_{par}(nc_{ref})/T_{par}(nc)$.

The weak scalability consists in increasing the number of cores together with the mesh grids in order to keep the same number of cells on each sub-domain computed by one core. The (absolute) weak scalability is the scalability between consecutive grids when the number of unknowns computed on one core does not change starting from one single core. For instance one core computes $48 \times 32 = 1536$ cells (G4-1c) on the whole domain, four cores compute the same amount of cells on a $1536 \times 4 = 6144$ grid (G5-4c) on one fourth of the domain, sixteen cores compute again the same amount of cells on a $1536 \times 16 = 24{,}576$ grid (G6-16c) and so on. Here again the ideal value is one. It is also possible to consider a relative weak scalability which is the ratio of the parallel time using nc_{ref} cores different from one over the parallel time using nc cores, but the results can be quite far from the absolute values. In addition the weak scalability can be close to one and not the strong scalability.

9.1. Efficiency or strong scalability

The first results concern only MPI parallelism with IBM MPI library and the highest level of optimization proposed by the compiler (O5 giving the same results as 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 by forcing buffered communications is up to 40% of the global elapsed real 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 $G4$, $G5$, $G6$, $G7$. 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 in 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 $G7$ is 1004 times greater than the sequential time on $G4$ instead of 512 for the solver!

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

	*G*4	*G*5	*G*6	*G*7	*G*4	*G*5	*G*6	*G*7
Time in seconds	16	108	1141	10,320	7	54	726	7031
Ratio		6.8	10.6	9		7.7	13.4	9.7

Table 2
Elapsed real time (*T*) and ratio of time (*R*) between two consecutive number of cores on five levels (*G*4 to *G*8) of grids for MPI parallelism.

	*T G*4	*T G*5	*T G*6	*T G*7	*T G*8	*R G*4	*R G*5	*R G*6	*R G*7	*R G*8
1 core	16	108	1141	10,320						
4 cores	8	33	319	2574	21,628	0.5	0.31	0.28	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

Once we have got the elapsed real 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 coarse-grids parallelism is used, that means for instance that when $16 \times 16 = 256$ cores are used on *G*6, the 256 cores are used on grids *G*5 and *G*6, 64 cores are used on *G*4, 16 cores are used on *G*3, 4 cores are used on *G*2 and one single core on *G*1. In practice sets of 4 cores are doing the same job on *G*4, sets of 16 cores are doing the same job on *G*3, sets of 64 cores are doing the same job on *G*2 and the 256 cores are doing the same job on *G*1. 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 ratios of time can be closer to 1 when the number of cores is too high. On *G*4 the results are already bad on 4 cores whereas on *G*8 the results are still good on 256 cores.

Using these times it is easy to compute the strong scalability (in Fig. 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 slower on fine grids. For instance the efficiency is still greater than 0.8 for 64 cores on *G*8 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 elapsed real times even on a single thread! That means that with this compiler option the level of optimization is higher. For the smoother there is a real gain on four threads although there is no OpenMP directives. In addition the OMP directives for the DO loop parallelism on the larger loops in the *x* direction reduce again the elapsed real times.

These elapsed real 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. We can even notice that for *G*7 and *G*8 the time is slightly increased for a low number of cores, this is probably due to cache effects for large OpenMP loops. On the other hand the elapsed real times are really decreased for large grids on a high number of cores, for instance on *G*8 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.

Fig. 6 shows again that on the one hand the elapsed real 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 *G*8, 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

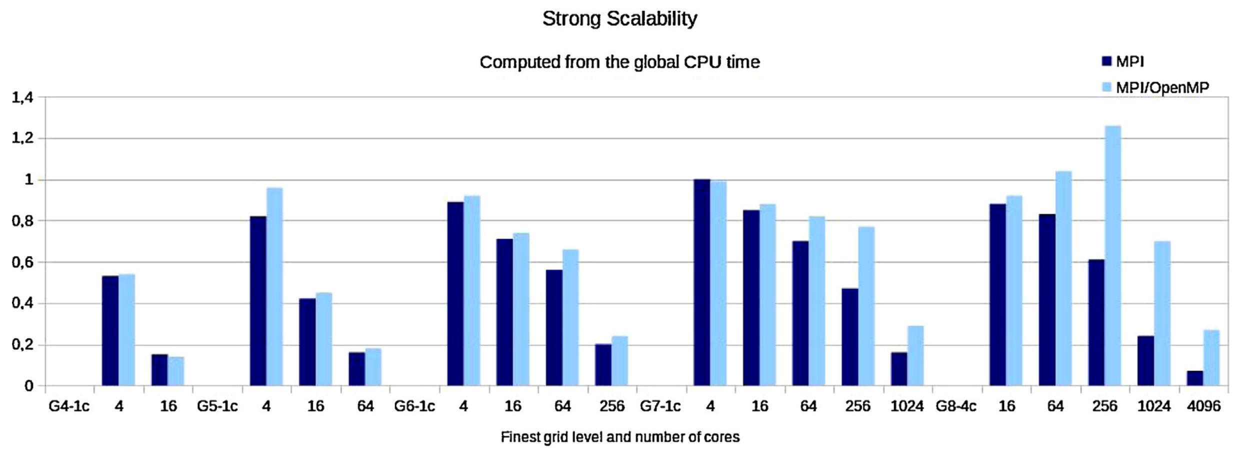

Fig. 6. Comparison of the strong scalability of the MPI (dark) and the MPI/OpenMP (light) parallelism on five levels of grids (*G*4 to *G*8) for the global program. The values refer to the sequential time obtained on each grid except for the grid *G*8 that cannot be solved on one single core.

Table 3
Elapsed real time (T) and ratio of time (R) between two consecutive number of cores on five levels ($G4$ to $G8$) of grids for hybrid parallelism.

	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	10,868						
4 cores	7	28	311	2615	22,006	0.5	0.34	0.28	0.24	
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

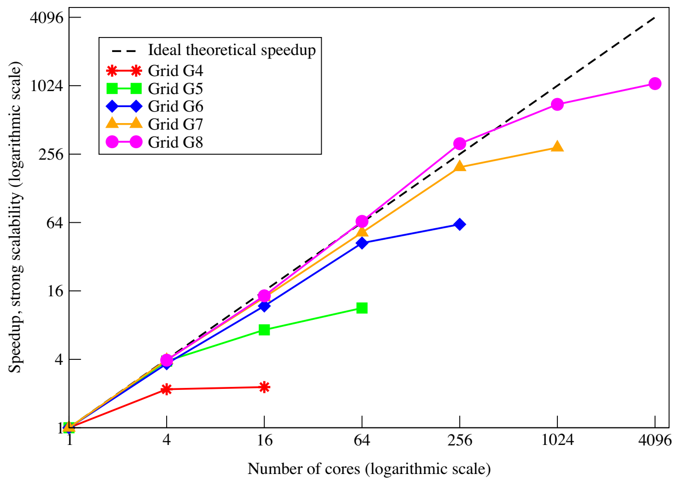

Fig. 7. Comparison of the speedup of the MPI/OpenMP parallelism on five levels ($G4$ to $G8$) of grids for the global program. The ideal speedup is achieved for a larger number of cores as the mesh increases.

good efficiency as can be seen in Fig. 7 that gives the speedup on five different grids.

A crucial point is to analyze the elapsed real 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 Fig. 8).

An interesting remark is the percentage of the elapsed real time spent for the smoother with respect to the global elapsed real 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 Fig. 9(left). The same percentage is shown for the MPI/OpenMP case (Fig. 9(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 OpenMP directives in the smoother subroutine.

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 in the cross section (y, z). Consequently the weak salability is computed by dividing the time by a factor 2 as there is no parallelism in the x direction. This factor 2 is an ideal factor that is correct for the computation because we have noticed that doubling the number of cells doubles effectively the job for a cell-by-cell solver. In addition we have used very large buffers for the communications in order to have the same efficiency when the size of the communications is increased avoiding buffer restrictions. Then Fig. 10 shows that the weak scalability decreases from 0.96 for $G5$ to 0.46 for $G8$ in the MPI case. But 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. Nevertheless, as we have seen above, it does not mean that we have a good efficiency. Indeed on the right hand side of this figure the reference elapsed time is the one obtained with 16 cores on $G4$ that has a very bad efficiency as shown in Fig. 7. That is why the relative weak scalability is very

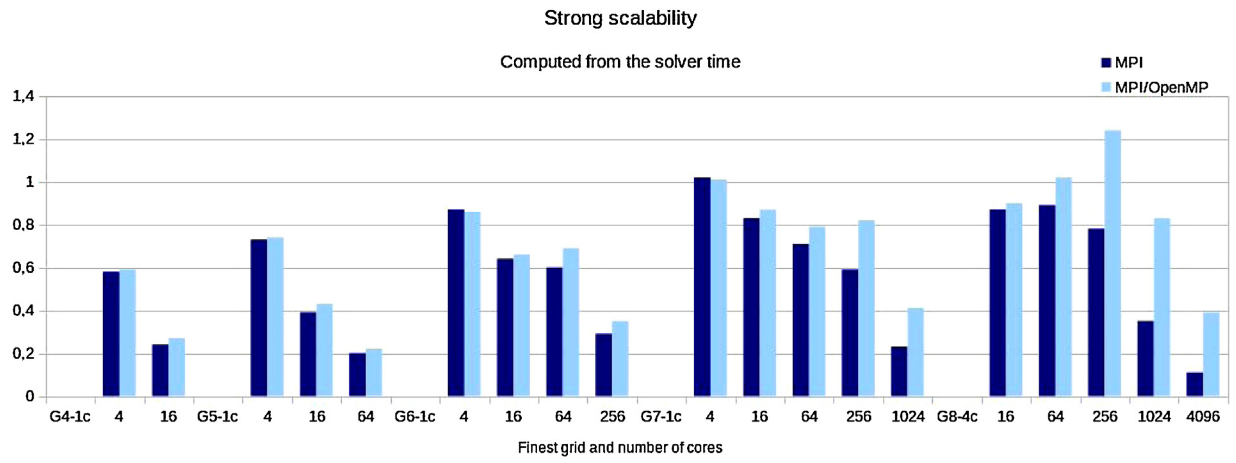

Fig. 8. Comparison of the strong scalability of the MPI (dark) and the MPI/OpenMP (light) parallelism on five levels ($G4$ to $G8$) of grids for the cell-by-cell Gauss–Seidel smoother. The values refer to the sequential time obtained on each grid except for the grid $G8$ that cannot be solved on one single core.

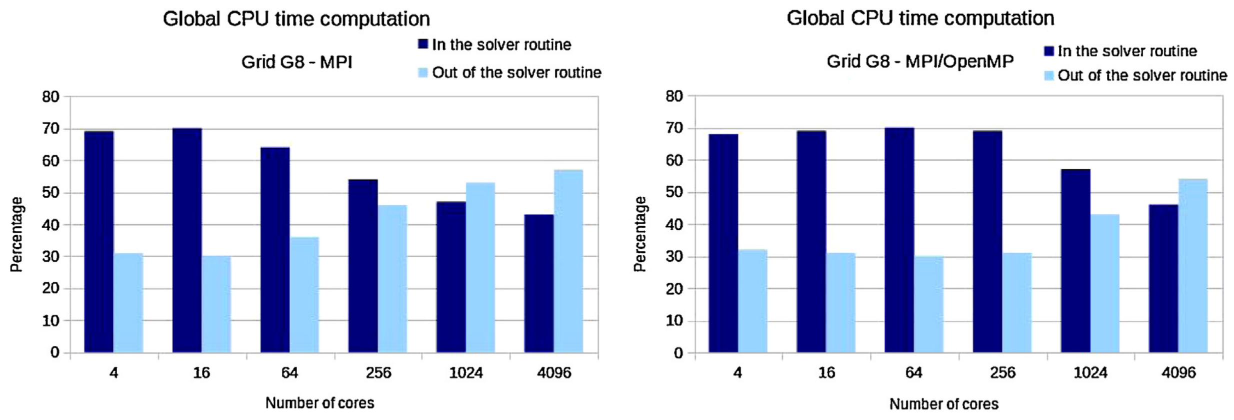

Fig. 9. Percentage of the elapsed real time of the smoother (dark) and of the elapsed real time of the rest of the program (light) for MPI (left) and hybrid MPI/OpenMP (right) parallelism. The part of the smoother represents 70% of the elapsed real time when the parallelism is efficient and then decreases because of the communications.

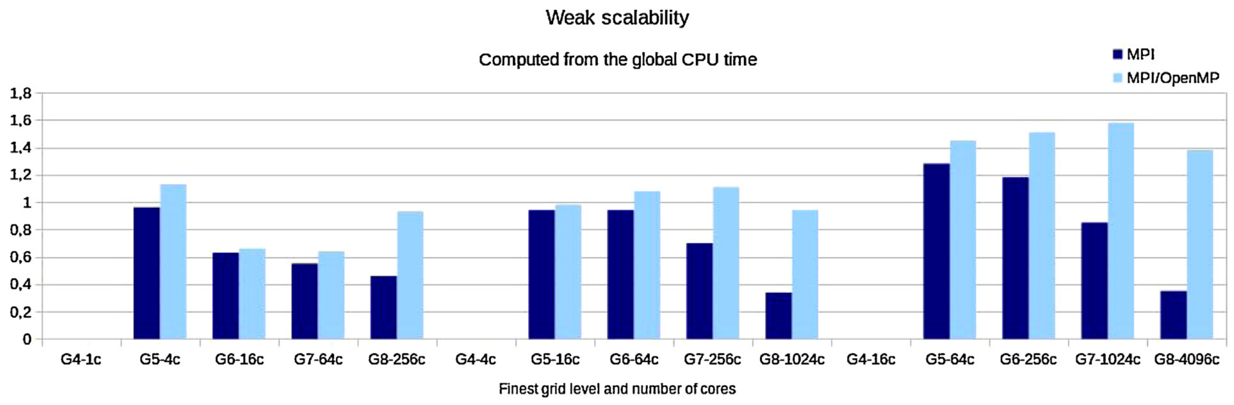

Fig. 10. Comparison of the weak scalability of the MPI (dark) and the MPI/OpenMP (light) parallelism on five levels ($G4$ to $G8$) of grids for the global program. The cores compute 1536 cells on the left, 384 cells in the middle and 96 cells on the right in the cross section (y, z).

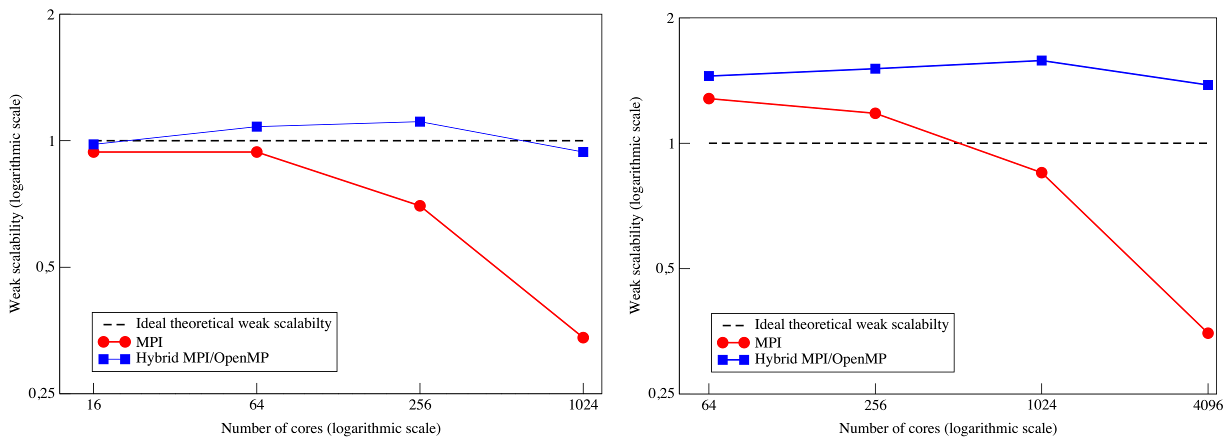

Fig. 11. Comparison of the weak scalability of the MPI (red) and the MPI/OpenMP (blue) parallelism on five levels ($G4$ to $G8$) of grids for the global program. On the left for 384 cells by core and on the right for 96 cells by core. (For interpretation of reference to color in this figure legend, the reader is referred to the web version of this article.)

good! 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 $G8$ as shown in Fig. 11. However we have seen that the speed up is lower than one.

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 Fig. 12 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. This shows

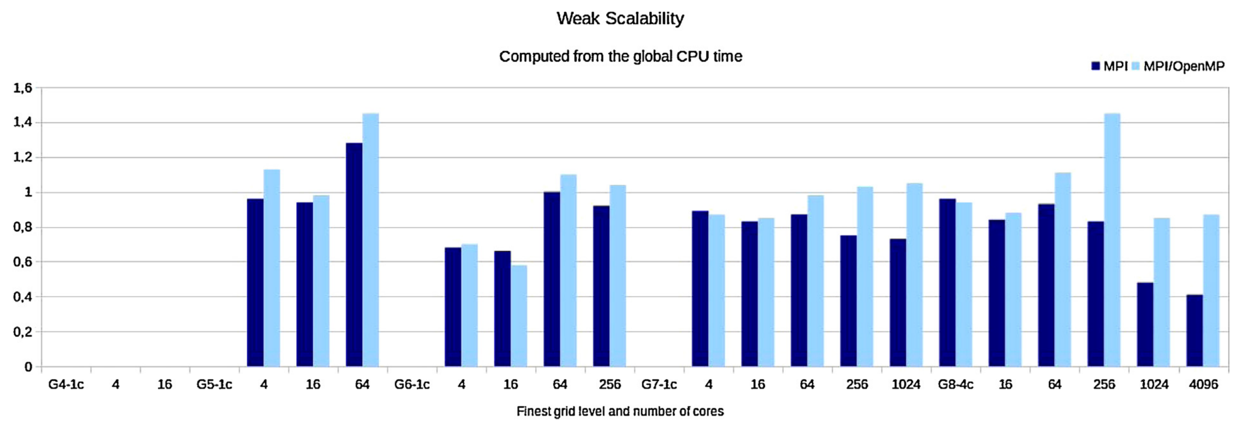

Fig. 12. Comparison of the relative weak scalability of the MPI (dark) and the MPI/OpenMP (light) parallelism on two consecutive grids for the global program.

the drawback of using relative quantities as they give generally better results than the absolute quantities.

10. Conclusions

The goal of this paper is to show that it is possible to reach high scalability results for solving Navier–Stokes equations 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 architecture of the platform, 16 cores per node and 4 threads per core, the efficiency with the hybrid parallelism is close to one for a medium number of cores in relation to the number of unknowns of the finest grid. The computation is quite efficient on one billion mesh cells with 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 on which all the cores are active 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 dimensional 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] I.I. Albarreal Nunez, M.C. Calzada Canalejo, J.L. Cruz Soto, H. Fernandez Cara, J.R. Galo Sanchez, M. Marin Beltran, Time and space parallelization of the Navier–Stokes equations, Comput. Appl. Math. 24 (3) (2005).
[2] Ph. Angot, C.-H. Bruneau, P. Fabrie, A penalization method to take into account obstacles in incompressible viscous flows, Numer. Math. 81 (1999) 497–520.
[3] A. Averbuch, L. Ioffe, M. Israeli, L. Vozovoi, Highly scalable two- and three-dimensional Navier–Stokes parallel solvers on MIMD multiprocessors, J. Supercomput. 11 (1997) 7–39.
[4] I.M. Barron, The transputer, in: Microprocessor and its Applications, Cambridge University Press, 1978.
[5] C.-H. Bruneau, Boundary conditions on artificial frontiers for incompressible and compressible Navier–Stokes equations, Math. Model. Numer. Anal. 34 (2) (2000).
[6] C.-H. Bruneau, E. Creusé, D. Depeyras, P. Gilliéron, I. Mortazavi, Coupling passive and active techniques to control the flow past the square back Ahmed body, Comp. Fluids 38 (10) (2010).
[7] C.-H. Bruneau, P. Fabrie, Effective downstream boundary conditions for incompressible Navier–Stokes equations, Int. J. Numer. Meth. Fluids 19 (1994) 693–705.
[8] C.-H. Bruneau, K. Khadra, I. Mortazavi, Analysis and active control of the flow around two following Ahmed bodies, in: Proceedings FEDSM13, 2013.
[9] C.-H. Bruneau, M. Saad, The 2D lid-driven cavity problem revisited, Comp. Fluids 35 (2006) 326–348.
[10] V. Dolean, P. Jolivet, F. Nataf, An Introduction to Domain Decomposition Methods: Algorithms, Theory, and Parallel implementation, SIAM, 2015.
[11] O. Dorok, V. John, U. Risch, F. Schieweck, L. S Tobiska, L. S Tobiska, Parallel finite element methods for the incompressible Navier–Stokes equations Flow Simulation with High-Performance Computers II, Notes Numer. Fluid Mech. 52 (1996) 20–33.
[12] D. Padua (Ed.), The Encyclopedia of Parallel Computing, Springer, 2011.
[13] J. Frisch, R.P. Mundani, P. Rank, Adaptive multi-grid methods for parallel CFD applications, Scalable Comput.: Pract. Exp. 15 (2014) 33–48.
[14] B. Gmeiner, U. Rude, H. Stengel, C. Waluga, B. Wohlmuth, Performance and scalability of hierarchical hybrid multigrid solvers for Stokes systems, SIAM J. Sci. Comput. 37 (2015) C143–C168.
[16] X. Han, S. Krajnović, C.-H. Bruneau, I. Mortazavi, Comparison of URANS, PANS, LES and DNS of flows around simplified ground vehicles with passive flow manipulation, in: Proceedings DLES9, 2013.
[17] R. Henniger, D. Obrist, L. Kleiser, High-order accurate solution of the incompressible Navier–Stokes equations on massively parallel computers, J. Computat. Phys. 229 (2010) 3543–3572.
[18] M.A. Heroux, P. Raghavan, H.D. Simon, Parallel Processing for Scientific Computing, Series: Software, Environments and Tools, SIAM, 2006.
[19] C.A.R. Hoare, Communicating sequential processes, Commun. Assoc. Comput. Mach. 21 (1978) 666–677.
[20] M. Jung, On the parallelization of multi-grid methods using a non-overlapping domain decomposition data structure, Appl. Numer. Math. 23 (1997) 119–137.
[21] K.S. Kang, Scalable implementation of the parallel multigrid method on massively parallel computers, Comput. Math. Appl. 70 (2015) 2701–2708.
[22] K. Khadra, Ph. Angot, S. Parneix, J.P. Caltagirone, Fictitious domain approach for numerical modelling of Navier–Stokes equations, Int. J. Numer. Methods Fluids 34 (2000) 651–684.
[23] D. May, R. Taylor, OCCAM – An overview, Micro-process. Microsyst. 2 (2) (1984).
[24] D.A. May, J. Brown, L. Le Pourhiet, A scalable, matrix-free multigrid preconditioner for finite element discretizations of heterogeneous Stokes flow, Comput. Methods Appl. Mech. Eng. 290 (2015) 496–523.
[25] PETSc: S. Abhyankar, M. Adams, S. Balay, J. Brown, L. Dalcin, T. Isaac, D. Karpeev, M. Knepley, L.C. Mc Ines, K. Rupp, J. Sarich, B. Smith, S. Zampini, H. Zhang, Portable, Extensible Toolkit for Scientific Computation, https://www.mcs.anl.gov/petsc/.
[26] G. Pfister, In Search of Clusters, Prentice Hall, 1998.
[27] Y. Satoa, T. Hinob, K. Ohashic, Parallelization of an unstructured Navier–Stokes solver using a multi-color ordering method for OpenMP, Comp. Fluids 88 (2013) 496–509.
[28] J. Šistek, F. Cirak, Parallel iterative solution of the incompressible Navier–Stokes equations with application to rotating wings, Comp. Fluids 122 (2015) 165–183.
[29] C.H. Tai, Y. Zhao, K.M. Liew, Parallel computation of unsteady three-dimensional incompressible viscous flow using an unstructured multigrid method, Comp. Struct. 82 (2004) 2425–2436.

[30] S. Vanka, Block-implicit multigrid calculation of two-dimensional recirculating flows, Comput. Methods Appl. Mech. Eng. 59 (1986) 29–48.
[31] Y. Wang, Solving incompressible Navier–Stokes equations on heterogeneous parallel architectures, Paris South University, Orsay, 2015 (PhD thesis).
[32] Y. Wang, M. Baboulin, J. Dongarra, J. Falcou, Y. Fraigneau, O. Le Maitre, A parallel solver for incompressible fluid flows, Procedia Comput. Sci. 18 (2013) 439–448.

Charles-Henri Bruneau received his PhD degree from Paris XI University in 1980. He is now full professor in applied mathematics at the University of Bordeaux. His research interests include numerical analysis, computational fluid dynamics, flow control and scientific computing.

Khodor Khadra has got his PhD degree in applied mathematics in 1994 at the University of Bordeaux. He is a scientific computing research engineer at the Institute of Mathematics of Bordeaux. He helps researchers in the use of scientific libraries and in the development of scientific parallel computing softwares. He also participates in research projects on modeling problems in fluid mechanics.