Computers & Fluids 88 (2013) 210–224

Contents lists available at ScienceDirect

Computers & Fluids

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

Parallel computing strategy for a flow solver based on immersed boundary method and discrete stream-function formulation

Shizhao Wang, Guowei He, Xing Zhang *

LNM, Institute of Mechanics, Chinese Academy of Sciences, Beijing 100190, China

ARTICLE INFO

Article history:
Received 3 February 2012
Received in revised form 26 August 2013
Accepted 2 September 2013
Available online 11 September 2013

Keywords:
Parallelization
Domain decomposition
Massage passing interface
Immersed boundary method
Discrete stream-function formulation

ABSTRACT

The development of a parallel immersed boundary solver for flows with complex geometries is presented. The numerical method for incompressible Navier–Stokes equations is based on the discrete stream-function formulation and unstructured Cartesian grid framework. The code parallelization is achieved by using the domain decomposition (DD) approach and Single Program Multiple Data (SPMD) programming paradigm, with the data communication among processes via the MPI protocol. A 'gathering and scattering' strategy is used to handle the force computing on the immersed boundaries. Three tests, 3D lid-driven cavity flow, sedimentation of spheres in a container and flow through and around a circular array of cylinders are performed to evaluate the parallel efficiency of the code. The speedups obtained in these tests on a workstation cluster are reasonably good for the problem size up to 10 million and the number of processes in the range of 16–2048.

1. Introduction

In recent years, there has been an increasing popularity of immersed boundary (IB) methods [1,2]. The reason behind this trend is that the meshes are not required to conform to the body surfaces. In the IB methods, appropriately defined forcing terms are added to the governing equations to mimic the effect of the immersed body on the motion of the fluid. The use of non-body-fitted meshes can significantly reduce the time and labor involved in meshing complex geometries. IB methods have now become a powerful tool for simulating flow involving complex, moving (or morphing) bodies. Comparing with the body-fitted-grid methods, IB methods usually require larger number of mesh points to achieve a proper resolution near the boundaries. This noticeable demerit is due to the use of simple Cartesian grid in the majority of IB methods. Some measures have already been taken to mitigate the situation, such as the use of stretched mesh, locally-refined mesh [3] or curvilinear mesh[4,5]. Although these strategies aforementioned can reduce the total mesh number to some extent, several million grid-points are still required in some high-fidelity 3D simulations (even for laminar flows at moderate Reynolds numbers).

With the continued rapid growth in computational power, larger and larger simulations are now conducted to study the phenomena in complex flow configurations. At the same time, massively parallel distributed-memory platforms have prompted new programming paradigms. To facilitate more efficient use of these computational resources in performing high-fidelity simulations, we need to investigate the parallelization strategy in addition to the discretization schemes and solution algorithms. Here, 'efficient' could refer to the ability to solve a problem of given size as fast as possible, but it could also mean that the overall time to solve the problem remains (nearly) constant when the problem size and the number of processors are increased at the same rate. The former definition relates to the strong scalability of a parallel implementation, whereas the latter requires weak scalability. In the present work, both types of scalability will be evaluated but the focus is put on the strong scalability property of the code.

The parallelization strategies under investigation in this paper include the following two aspects: (a) parallelization of the basic Navier–Stokes solver; (b) parallelization of the forcing computation near the immersed boundary. For the basic flow solver, most of the existing numerical models for the solution of the Navier–Stokes equations are based on Finite Difference Method (FDM), Finite Element Method (FEM) or Finite Volume Method (FVM). The space can be discretized with structured or unstructured grids, and the time with explicit or implicit techniques. The parallelization strategy differs greatly among different data structures and time advancing schemes. Numerical methods which depend on structured grids (such as FDM) and an explicit time discretization have certain advantages concerning the parallelization. The parallel implementation is easier and higher parallel efficiencies can be expected when compared with other candidates.

In the present work, we first describe the parallelization a sequential Navier–Stokes solver which is based on the discrete stream function formulation for incompressible flows. This

* Corresponding author. Tel.: +86 10 82543929; fax: +86 10 82543977.
E-mail address: zhangx@lnm.imech.ac.cn (X. Zhang).

0045-7930/$ - see front matter © 2013 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.compfluid.2013.09.001

algorithm is essentially implicit and is currently implemented on an unstructured Cartesian grid. Although structured grid possesses the advantage of coding simplicity, fully unstructured data structure allows an easy treatment of anisotropic local mesh refinement (esp. hanging-nodes) [6]. Of course, the unstructured data management unavoidably raises the complexity of parallelization. Many references on parallelization of implicit unstructured solvers can be found in the published literatures, such as [7–10] on FVM, [11] on FEM and [12–15] on Control Volume Finite Element Method (CVFEM) (which is a mixture of FVM and FEM), just to list a few. The implicit parallel solution algorithm for Navier–Stokes equations in this work is based on the Domain Decomposition (DD) approach, which is preferable on a computer system with distributed memory. When applying the DD strategy, the sequential algorithm of discretization is mainly kept and each processor computes part of the basic tasks such as matrix–vector multiplication with its assigned data and the data exchange only occurs at the boundaries among sub-domains.

The existence of immersed boundaries (esp. moving ones) further complicates the parallel implementation. Intuitively, it is most reasonable to have a given processor deal with the 'markers' (Lagrangian points) which are currently located within its local sub-domain. However there are some open questions related to this strategy of parallelization, such as the load-balancing issue due to the unequal distribution of 'marker' points across processors; communication overhead produced by shared processing and handing-over of points among processes. These issues are seldom addressed in the literatures. In a report by Uhlmann [16], a 'master and slave' strategy is proposed for the simulation of freely-moving particles in fluid using the IB method. In this strategy, one 'master' processor is assigned to each particle for the general handling of it. If necessary, there will be a number of additional processors ('slaves') assigned to the particle to help the 'master'. Recently, Wang et al. [17] proposed a different method to handle the situation when particle crosses the boundaries of sub-domains. Preliminary 2D computations have demonstrated that the parallel efficiency and speedup in these two studies above are acceptable. In the present work, we employ a 'gathering-and-scattering' strategy. At each time step, a master processor is exclusively responsible for computing the force and then the result is scattered the slave processors on which the Navier–Stokes equations are solved in parallel. This strategy circumvents some difficulties aforementioned and is very easy to program.

The organization of the paper is as follows. Numerical method and data structures are presented in Section 2. Basic strategies employed in the parallelization will be discussed in Section 3. Validations and parallel performance tests will be presented in Section 4. Finally, the discussion and conclusions are given in Section 5.

2. Algorithm description

A brief introduction to the numerical methodology and data structure is given here to make the paper as self-contained as possible. For a more complete description, please refer to [18].

2.1. Immersed boundary method in discrete stream-function formulation

The three-dimensional incompressible viscous flow is considered in the present work. The governing equations of the flow can be written as:

$$\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u}\cdot\nabla\mathbf{u} = -\nabla \mathrm{p} + \frac{1}{\mathbf{Re}}\nabla^2\mathbf{u} + \mathbf{f}, \tag{1}$$

$$\nabla\cdot\mathbf{u} = 0, \tag{2}$$

where $\mathbf{u}$ and p are the velocity vector and pressure respectively. $\mathbf{f}$ is the force representing the effect of the immersed body on the flow. The Reynolds number is defined as $Re = UL/\nu$, where U and L are the reference velocity and length respectively, and ν is the kinematic viscosity of the fluid. The discretized form of Eqs. (1), (2) can be expressed by a matrix form as

$$\begin{bmatrix}\mathbf{A} & \mathbf{G}\\ \mathbf{D} & \mathbf{0}\end{bmatrix}\begin{bmatrix}q^{n+1}\\ p\end{bmatrix} = \begin{bmatrix}r^n\\ 0\end{bmatrix} - \begin{bmatrix}bc_1\\ bc_2\end{bmatrix} + \begin{bmatrix}\tilde{f}\\ 0\end{bmatrix}, \tag{3}$$

where q, p, and $\tilde{f}$ are the discrete velocity flux, pressure, and body force, respectively. The discrete velocity u, is related to q by multiplying the cell face area. $\mathbf{A}$, $\mathbf{G}$ and $\mathbf{D}$ are the implicit operator, gradient operator and divergence operator respectively. In addition, the negative transpose of the divergence operator is the gradient operator, i.e. $\mathbf{G} = -\mathbf{D}^T$. r^n is the explicit right-hand-side term of the momentum equation. bc_1 and bc_2 are the boundary condition vectors for the momentum and continuity equation respectively.

The discrete stream function (null-space) approach is a numerical method for solving Eq. (3) proposed by Chang et al. [19]. Unlike the classic fractional step method, in this method the divergence-free condition is satisfied to machine precision and there are no splitting errors associated with it. In the discrete stream function approach, a discrete stream-function s is defined, such that $q = Cs$, where $\mathbf{C}$ is the *curl* operator (which is a non-square matrix). This matrix is constructed in such a way that $\mathbf{D}$ and $\mathbf{C}$ enjoy the relation $\boldsymbol{DC} = 0$, thus the incompressibility condition is automatically satisfied.

In this approach, another type of curl operator $\mathbf{R}$, which is called the *rotation* operator, is also defined. The matrix $\mathbf{R}$ and matrix $\mathbf{C}$ enjoy the relation $\mathbf{R} = \mathbf{C}^T$. By pre-multiplying the momentum equation with $\mathbf{R}$, the pressure can be eliminated and the system of Eq. (3) is reduced to a single equation for s at each time step

$$\mathbf{C}^T\mathbf{A}\mathbf{C}s^{n+1} = \mathbf{R}(r^n - bc_1) + \mathbf{R}\tilde{f}. \tag{4}$$

The matrix $\mathbf{C}^T\mathbf{A}\mathbf{C}$ is symmetric, positive-definite and thus can be solved using the Conjugate Gradient (CG) method.

The forcing term $\mathbf{f}$ on the right hand side of Eq. (1) is computed in an implicit way. Within one step of time advancing (from n to n + 1), this procedure can be summarized as the following four sub-steps.

(i) A 'predicted' stream function is computed with the forcing at time step n and the velocity vectors are reconstructed using the 'predicted' stream function.

(ii) A 'force correction' is applied to achieve the desired velocity on the boundary. In this paper, we follow a similar procedure as that in [21] in computing the force. Mathematically, the 'force corrections' at the Lagrangian points are computed using the formula

$$\sum_{j=1}^{M}\left(\sum_{x}\delta_h(\mathbf{x}-\mathbf{X}_j)\delta_h(\mathbf{x}-\mathbf{X}_k)\Delta s^2 h^3\right)\Delta\mathbf{F}(\mathbf{X}_j) = \frac{\vec{U}^{n+1}(\mathbf{X}_k) - \vec{U}^{*}(\mathbf{X}_k)}{\Delta t}, \tag{5}$$

where $\vec{U}^{n+1}$ and $\vec{U}^{*}$ are the desired and 'predicted' velocities at the Lagrangian points respectively; δ_h is the regularized Delta function; h and Δs are the grid sizes of the Euler and Lagrangian points respectively; Δt is the time step. $\vec{U}^{*}$ is evaluated by

$$\vec{U}^{*}(\mathbf{X}_k) = \sum_{x}\vec{u}^{*}(\mathbf{x})\delta_h(\mathbf{x}-\mathbf{X}_k)h^3, \tag{6}$$

where $\vec{u}^{*}$ is the 'predicted' velocity computed in step (i). By definition, the force correction $\Delta\mathbf{f}$ at the Eulerian grid points can be computed using the transformation

$$\Delta \mathbf{f}(\mathbf{x}) = \sum_{j=1}^{M} \Delta \mathbf{F}(\mathbf{X_j}) \delta_h(\mathbf{x} - \mathbf{X}_j) \Delta s^2. \tag{7}$$

(iii) The forcing term is 'corrected' by using

$$\mathbf{f}^{n+1} = \mathbf{f}^{n} + \Delta \mathbf{f}. \tag{8}$$

(iv) The stream function at time step n + 1 is computed with the 'corrected' forcing term. In the time advancement, the diffusion term is implicit and treated by using the trapezoidal method; the convection term is explicit and a three-step second-order, low storage, Runge – Kutta scheme is used [22]. The formulation of this numerical scheme is:

$$\begin{aligned} \phi^{n+1,(1)} &= \phi^n + \Delta t G(\phi^n), \\ \phi^{n+1,(2)} &= \phi^n + \frac{1}{2}\Delta t[G(\phi^n) + G(\phi^{n+1,(1)})], \\ \phi^{n+1} &= \phi^n + \frac{1}{2}\Delta t[G(\phi^n) + G(\phi^{n+1,(2)})]. \end{aligned} \tag{9}$$

This scheme is of second order accurate and very easy to implement. For wave equation, the stability criterion of this particular scheme is that the CFL number based on the wave speed remains less than 2. This restriction is equivalent to require a match between the temporal and spatial accuracy and not too tight in most cases. Please note that the forcing term remains fixed in the three sub-steps of the Runge–Kutta scheme. The procedure (i)–(iv) is repeated until the terminating time is reached.

It is noted that the number of unknowns of the linear system (5) is much smaller than that of the Navier–Stokes equations, thus the extra computational cost for solving Eq. (5) is negligible.

A 4-point regularized Delta function is used in Eq. (5) and Eq. (6) for all the simulations thereafter. The formulation of this regularized Delta function is

$$\delta_h(\mathbf{x} - \mathbf{X}) = \frac{1}{h^3}\phi\left(\frac{x - X}{h}\right)\phi\left(\frac{y - Y}{h}\right)\phi\left(\frac{z - Z}{h}\right). \tag{10}$$

Here ϕ is the following piecewise function which is proposed in [1],

$$\phi(r) = \begin{cases} \frac{1}{8}\left(3 - 2|r| + \sqrt{1 + 4|r| - 4r^2}\right), & |r| \leqslant 1, \\ \frac{1}{8}\left(5 - 2|r| - \sqrt{-7 + 12|r| - 4r^2}\right), & 1 \leqslant |r| \leqslant 2, \\ 0, & 2 \leqslant |r|. \end{cases} \tag{11}$$

2.2. Data structure outline

An unstructured Cartesian grid is used in the present implementation. The definitions of four topological entities (face, cell, edge and node) are shown in Fig. 1, for both the two- and three-dimensional situations.

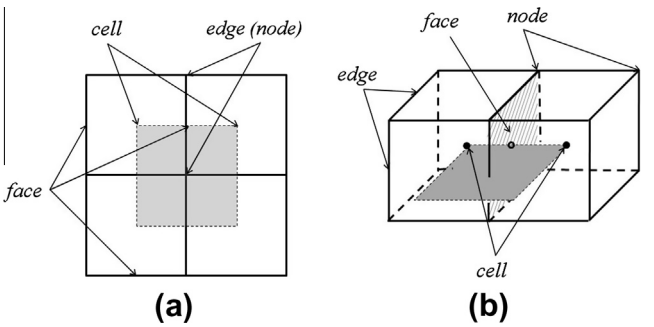

Fig. 1. The definition of topological entities on an unstructured Cartesian mesh in: (a) two; (b) three dimensions. The node is defined as the collapse of the edge in the two-dimensional mesh.

Four integer arrays, *F2C*, *E2N*, *LINK_FE* and *LINK_CN*, are used to store the connectivity among these topological entities. A schematic representation of the mesh connectivity contained in these arrays is shown in Fig. 2. These arrays provide sufficient information for both the solution and the post-processing procedure in the present work. More specifically, the connectivity information contained in these arrays is listed as follows.

(i) *F2C*(*i*,*j*):
This array contains the connectivity between face and cell. $i = 1,2$ and $j = 1\ldots,num\_face$, where *num_face* is the total number of faces. *F2C*(1,*j*) denotes the index of one cell connected to face *j*; similarly, *F2C*(2,*j*) denotes the index of the other cell connected to the same face.

(ii) *E2N*(*i*,*j*):
This array contains the connectivity between edge and node. $i = 1,2$ and $j = 1\ldots,num\_edge$, where *num_edge* is the total number of edges. *E2N*(1,*j*) denotes the index of one node connected to edge j; similarly, *E2N*(2,*j*) is the index of the other node connected to the same edge. This connectivity becomes non-trivial only in three-dimensional situations.

(iii) *LINK_FE*(*i*,*j*):
This array contains the connectivity between face and edge. $i = 1,2$ and $j = 1\ldots,num\_link\_fe$, where *num_link_fe* is the total number of links connecting face and edge. *LINK_FE*(1,*j*) denotes the index of the face connected to link j and *LINK_FE*(2,*j*) denotes the index of the edge connected to the same link.

(iv) *LINK_CN*(*i*,*j*):
This array contains the connectivity between cell and node. $i = 1,2$ and $j = 1\ldots num\_linke\_cn$, where *num_linke_cn* is the total number of links connecting cell and node. *LINK_CN*(1,*j*) is the index of the cell connected to link j and *LINK_CN*(2,*j*) is the index of the node connected to the same link.

All the discrete differential operators mentioned in Section 2.1 manipulate variables defined on one type of topological entity and store the result on another type of entity. These operators are equivalent to sparse matrices mathematically. However, in the code implementation, these matrices are never explicitly formed, but just programmed as functions that perform vector–matrix multiplications.

To make the expressions of the operators more clear and concise, the definitions of two unit vectors and a sign convention are first introduced (see Fig. 3)). $\vec{t}_e$ is the vector pointing from one node to the other sharing the same edge; similarly, $\vec{n}_f$ is the face normal vector pointing from one cell to the other sharing the same face. The sign convention at each edge-face link is defined as

$$sign(f, e) = sign[\vec{n}_f \cdot (\vec{t}_e \times \vec{r}_{fe})], \tag{12}$$

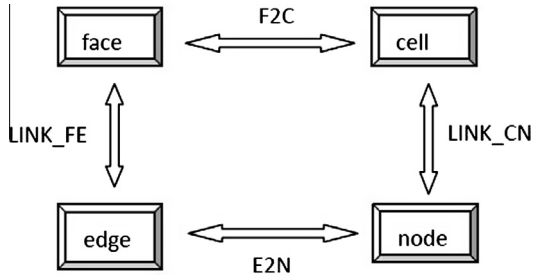

Fig. 2. Schematic representation of the mesh connectivity. Four integer arrays are used to store the connectivity among these topological entities: *F2C*, *E2N*, *LINK_FE* and *LINK_CN*.

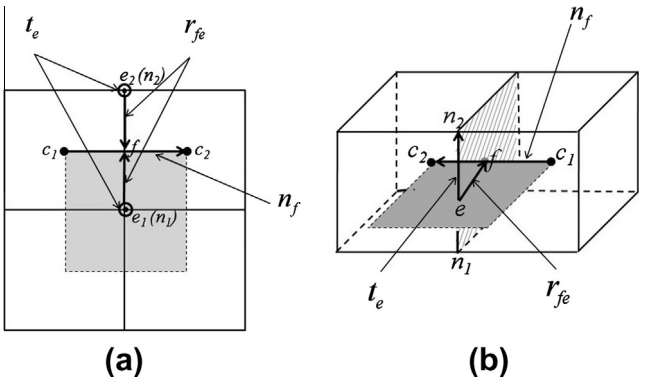

Fig. 3. The definitions two vectors and a sign convention at a face-edge link in: (a) two; (b) three dimensions. $\vec{t}_e$ is the vector pointing from one node to the other sharing the same edge; $\vec{n}_f$ is the face normal vector pointing from one cell to the other sharing the same face.

where $\vec{r}_{fe}$ is the vector pointing from the edge center to the face center.

A representation of variable locations in the staggered grid system is shown in Fig. 4. The normal velocity flux is defined at face center, while the velocity vector is defined at cell center. The stream function component s, which is defined at edge center, actually represents an integral $l_e(\vec{\psi}\cdot\vec{t}_e)$, where $\vec{\psi}$is the stream function vector and l_e is the edge length. In two-dimensional situation, the stream function is located at the edges (or nodes) and oriented perpendicular to our physical plane.

The programming of the discrete operators (gradient, divergence and curl) can be expressed by the following pseudo-codes.

(1) **G**(p):

This discrete gradient operator manipulates variable defined on cell centers and stores the result on face centers.

```
Function CGRAD (p)
Do i = 1, num_face
    c1 = F2C(1,i)
    c2 = F2C(2,i)
    CGRAD(i) = p(c2) − p(c1)
End do
End Function CGRAD
```

(2) $D(q)$:

This discrete divergence operator manipulates variable defined on face centers and stores the result on cell centers. It is noted that in the computation of discrete divergence, the face-based algorithm (instead of the cell-based algorithm) is adopted to avoid duplicated computation on each face.

```
Function CDIV (q)
CDIV = 0
Do i = 1, num_face
    c1 = F2C(1,i)
    c2 = F2C(2,i)
    CDIV(c1) = CDIV(c1) + q(i)
    CDIV(c2) = CDIV(c2) − q(i)
End do
End Function CDIV
```

(3) **C**(s):

This discrete curl operator manipulates variable defined on edge centers and stores the result on face centers.

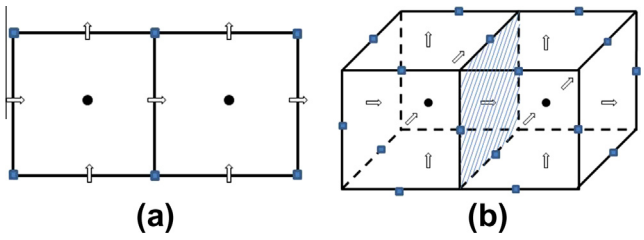

Fig. 4. The location of variables on a: (a) 2D mesh; (b) 3D mesh. The normal velocity flux q is defined at face centers (denoted as hollow arrows). The stream function component s is defined at edge centers (denoted as solid squares). The velocity vector **u**, pressure p and volume force are defined at cell centers (denoted as solid circles).

```
Function CURL (s)
CURL = 0
Do i = 1, num_link_fe
    f = link_fe(1,i)
    e = link_fe(2,i)
    CURL(f) = CURL(f) + sign(f,e)*s(e)
End do
End Function CURL
```

(4) $R(r)$:

This is another type of curl operator which manipulates variable defined on face centers and stores the result on edge centers.

```
Function ROT (r)
ROT = 0
Do i = 1, num_link_fe
    f = link_fe(1,i)
    e = link_fe(2,i)
  ROT(e) = ROT(e) + sign (f,e)*r(f)
End do
End Function ROT
```

The four operators aforementioned are the basic building blocks of the present numerical implementation. In addition to these four operators, in the discretization of the Navier–Stokes equations, other more complex operators are also needed, such as the Laplacian operator (for computing the implicit matrix **A**), convective operator (for computing the right-hand-side of Eq. (3)) and interpolation operators, etc. Most of these complex operators can be constructed using the four basic ones, or can be easily constructed in a similar fashion. For conciseness, these complex operators are not discussed here. Please refer to [18] for the programming of these operators.

2.3. Conjugate Gradient (CG) solver

In the linear system of Eq. (4), the coefficient matrix is symmetric and positive-definite. Additionally, in Eq. (5) for computing the force, the coefficient matrix is also symmetric and positive-definite, providing that the inter-distance among Lagrangian points is roughly the same as the size of the Eulerian grid used in the Navier–Stokes solver.

In this work, the Conjugate Gradient (CG) method with a Jacobi pre-conditioner is used as an iterative solver to solve the linear system. In the mathematical formulation, a linear system can be written as

$$\mathbf{A}\mathbf{x} = \mathbf{b}. \tag{13}$$

To improve the rate of convergence of the iterative method, the coefficient matrix can be preconditioned as

$$\mathbf{M}^{-1}\mathbf{A}\mathbf{x} = \mathbf{M}^{-1}\mathbf{b}, \tag{14}$$

where M^{-1} is the left pre-conditioner and in this work **M** is simply constructed by

$$\mathbf{M} = \mathrm{diag}(\mathbf{A}). \tag{15}$$

The pseudo-code for the pre-conditioned CG Method is given as follows.

Compute $r^{(0)} = b - Ax^{(0)}$ *for some initial guess* $x^{(0)}$
For i = 1,2...
  Solve $Mz^{(i-1)} = r^{(i-1)}$
  $\rho^{(i-1)} = r^{(i-1)\mathrm{T}} z^{(i-1)}$
  If i = 1
    $p^{(1)} = z^{(0)}$
Else
  $\beta^{(i-1)} = \rho^{(i-1)}/\rho^{(i-2)}$
  $p^{(i)} = z^{(i-1)} + \beta^{(i-1)} p^{(i-1)}$
Endif
  $q^{(i)} = Ap^{(i)}$
  $\alpha^{(i)} = \rho^{(i-1)}/p^{(i)^{\mathrm{T}}} q^{(i)}$
  $x^{(i)} = x^{(i-1)} + \alpha^{(i)} p^{(i)}$
  $r^{(i)} = r^{(i-1)} - \alpha^{(i)} q^{(i)}$
  Check if convergence, continue if necessary
End

We use the following stopping criterion to check if the iteration converges

$$\|r^{(i)}\|^2 \leqslant \varepsilon_1^2 \|r^{(0)}\|^2 + \varepsilon_2^2 \|x^{(0)}\|^2. \tag{16}$$

Double precision arithmetic implementation is used for all the simulations in this work. The values of the two controlling parameters in Eq. (4) for solving the equation of s are: $\varepsilon_1 = 10^{-9}$; $\varepsilon_2 = 10^{-10}$; while the values for solving Eq. (5) are: $\varepsilon_1 = 10^{-8}$; $\varepsilon_2 = 10^{-9}$.

3. Parallel implementation

3.1. Domain decomposition

In the code parallelization, the Single Program Multiple Data (SPMD) paradigm is employed. In this paradigm, each process executes basically the same piece of code but on a different part of the data. This type of parallelism is also referred to as domain decomposition (DD), in the sense that the computational grid has to be split (partitioned) among the available processors. In this work, the grid partitioning is done by using the free software – METIS, where the algorithm of multilevel graph partitioning is implemented [20,23]. It provides high quality partitions with balanced computational loads and minimized communication among the processors. A schematic representation of the mesh partitioning using METIS is shown in Fig. 5.

The parallel code implements a master–slave model and the program is written using the MPI (Message Passing Interface) protocol. The master program is responsible for the adjusting of time-step (based on a global maximum CFL number allowed) and the task of data output. The slave programs perform the conjugate gradient (CG) iterations to solve Eq. (4). In every iteration of the CG solver, the slave processes not only have to complete the 'local' work within its sub-domain, but also receive (send) data from (to) other slave processes and finish the work on its partition boundaries. To achieve this, the discrete operators described in Section 2.2 need to be parallelized. The parallelization of these operators will be discussed in the next subsection. In the parallelized CG solver, the matrix (or vector) operations are fully synchronized among slave processes by the placement of barriers. Thus an identical result from the parallelized code is obtained when compared with that from the sequential one. In the present work, the task to compute the forcing term at the immersed boundary is assigned to the master process. The parallelization issues related to the immersed boundary force will be addressed in Section 3.4.

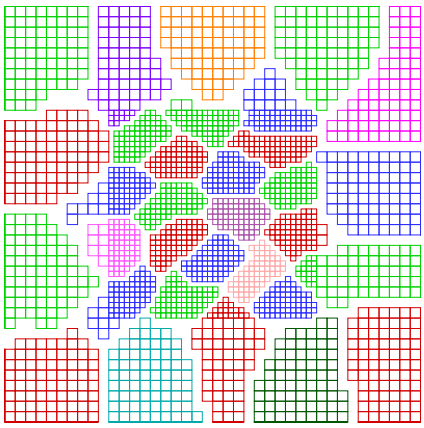

Fig. 5. A schematic representation of mesh partitioning. The 2D unstructured Cartesian mesh (with hanging-nodes) is partitioned into 32 parts.

3.2. Parallelization of discrete operators

The parallel execution of the operations in Section 2.2 in a distributed computing environment requires the communication among processes across the boundaries of partitions. Unlike the parallel implementations in [7,17], where one or more 'overlapping' layers of ghost nodes are added to each partition boundary for the communication. In the present work, we propose to use a non-overlapping or 'seamless connectivity' approach. Some auxiliary data structures for the mesh connectivity *across* the partition boundaries are designed for this purpose. After mesh partitioning, the output file from METIS contains the flags of partition attribution assigned to each cell and node. The mesh entities adjacent to the partition boundaries can be easily identified using these flags. The auxiliary arrays of connectivity are then built on these mesh entities in the pre-processing stage.

Fig. 6 is a schematic representation of the auxiliary data structures appended to *F2C* for the 'face-to-cell' communication between partition I (which is assigned to process 1) and partition II (which is assigned to process 2). On the 'boundary' face which has one of its neighboring cells lying in a different partition (such the faces denoted by thick solid lines in Fig. 6), the inter-process 'face-to-cell' connectivity is built for data *receiving*. This type of connectivity includes: (1) cf_buffer (array of cell variables to be received); (2) Bf2cF (array of local face indices for the faces associated with cf_buffer); (3) cf_pt (array of offset pointers for positioning the data received from one particular process in cf_buffer); (4) cf_CPU (array of process indices corresponding to the pointers in cf_pt). At the same time, the original connectivity on these faces is also modified such that F2C(1,$*$) points to the local cell while F2C(2,$*$) points to a 'null' position (where a value of zero is stored). This arrangement ensures that the local operations are handled properly without any contribution from the 'null' cells.

Similarly, if any cell in one partition is linked to one face lying in a different partition (the face denoted by dashed lines in Fig. 6), the inter-process 'face-to-cell' connectivity is then built for data *sending*. This type of connectivity includes: (1) fc_buffer (array of cell variables to be sent to other processes); (2) Bf2cC (array of local cell indices for the cells associated with fc_buffer); (3) fc_pt (array of offset pointers for positioning the data sent to one particular process in fc_buffer); (4) fc_CPU (array of process indices corresponding to the pointers in fc_pt).

In the parallelization of the discrete operators, in order to lower communication costs, the concept of overlapping communication

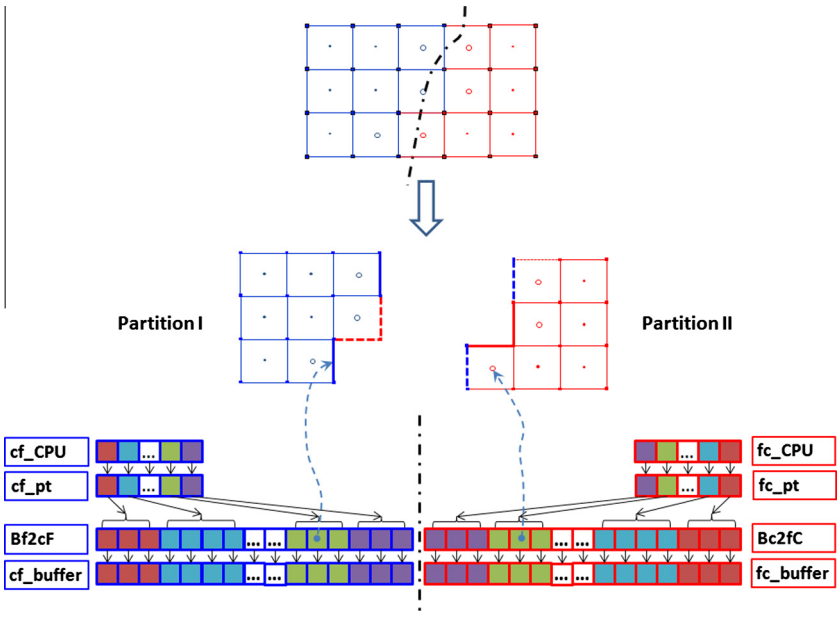

Fig. 6. A schematic representation of the face-cell connectivity between different partitions. The connectivity between two neighboring partitions is built without adding ghost nodes. The thick solid lines denote faces where the auxiliary data structures are built for data receiving. The thick dashed lines denote faces where the auxiliary data structures are built for data sending.

with computation is implemented and the non-blocking sends (receives) of MPI are used in the coding. The parallelized execution of these operations can be summarized into the following five steps: (a) filling the buffer array with data to be sent and start to send; (b) starting to receive data from other processes and store them in a buffer array; (c) finishing the local work on the local process; (d) waiting until the sends (receives) are accomplished; (e) completing the work on the partition boundary using the data received. The pseudo-code for the parallelized gradient operator *p_CGRAD* is presented as follows.

```
Function p_CGRAD (p)
$$ fill the buffer to be sent
do i = 1, num_send_buff
    fc_buffer(i) = cell(Bf2C(i))
end do
$$ start to send
do i = 1,num_fc_CPU
    call mpi_isend (fc_buffer,. . . . . handle_s (i),. . . . . )
end do
$$ start to receive
do i = 1,num_cf_CPU
    call mpi_irecv (cf_buffer,. . . . . handle_r (i),. . . . . )
end do
$$ interior work
Do i = 1, num_face
    c1 = F2C(1,i)
    c2 = F2C(2,i)
    p_CGRAD(i) = p(c2)− p(c1)
End do
$$ wait for the completion of sends and receives
    call mpi_waitall (num_fc_CPU, handle_s,. . . . . )
    call mpi_waitall (num_cf_CPU, handle_r,. . . . . )
$$ complete work on the partition boundary
do i = 1,num_cf_CPU
    fnum = Bf2F(i)
    p_CGRAD (fnum) = p_CGRAD (fnum) + cf_buffer(i)
end do
End Function p_CGRAD
```

Other operators (such as **D**, **C** and **R**) are parallelized in a similar fashion. For example, in the parallelization of **C** and **R**, auxiliary data structures appended to *linke_fe* are also built for the 'face-to-edge' communication across partitions. These parallelized operators are then used in place of the sequential ones described in Section 2.2 to perform matrix–vector multiplications in the CG iterations.

3.3. Parallelization of vector dot-production in CG solver

From the pseudo-code in Section 2.3, it is known that three types of vector operations are involved in the CG solver. They are: (1) matrix–vector multiplication; (2) vector addition (or subtraction) and scalar-vector multiplication; (3) vector dot-production. The parallelization of the first type of operation has been discussed in the previous subsection. The operation of the second type is inherently parallel. Here we focus on the parallelization of the third type of operation. The vector dot-product is calculated in parallel by computing a local summation on each process and then doing a summation reduction using *mpi_allreduce*. The pseudo-code for the parallelized vector dot-production is presented as follows.

```
Function p_dot_product(a,b)
$$ each process computes its local dot_product
    dot_L = dot_product(a,b)
$$ sum reduction and return p_dot_product
call mpi_allreduce (dot_L,p_dot_product,1,. . .,mpi_sum,. . .,. . . )
End Function p_dot_product
```

3.4. Parallelization of force-computing on immersed boundaries

The interpolate operations in Eqs. (5)–(7) between the data at Eulerian and Lagrangian points further complicate the parallelization. Unlike the strategy proposed in [16,17], where some slave-processes are assigned the job of interpolation using the discrete delta function, we propose to use a completely different approach.

The schematic representation of this 'gathering-and-scattering' strategy is shown in Fig. 7. The basic thoughts behind this approach are: (a) the master-process is solely responsible for the computation of Lagrangian force in Eq. (5) (including the assembling of the coefficient matrix and solving the linear system); (b) in the case of moving-boundary, the master-process is also responsible for updating the positions of the Lagrangian points (and solving extra equations if the problem is coupled with the dynamics of a rigid body); (c) only the Eulerian variables (such as the velocities and forces defined at cell centers) are exchanged between the master and slaves.

In the parallel implementation, after step (i) of Section 2.1, the predicted velocity $\vec{u}^*$ is sent to the master process from all slaves. The master process first updates the positions of the Lagrangian points and then computes the Lagrangian velocity $\vec{U}^*$ and assembles the coefficient matrix in Eq. (5). After obtaining the Lagrangian force correction $\Delta\mathbf{F}$ by solving Eq. (5), the interpolation in Eq. (7) is performed to compute $\Delta\mathbf{f}$. The Eulerian force correction $\Delta\mathbf{f}$ is then sent back to the slave processes and the Runge–Kutta steps in step (iii) are then executed on the slave processes in parallel.

To minimize memory occupation on the master process, the full mesh connectivity information for a 'global' unstructured mesh is not stored. Instead, we map the unstructured grids on the slave processors into a single *structured* grid on the master. More specifically speaking, we build and save a 'table of indices' on the master process. This table maps the cell indices on the slaves onto the global indices of (i,j,k) on the master. The assumption that we make here is that a uniform Eulerian gird is used (at least adjacent to the immersed boundary). Thus giving a reference position (x_0, y_0, z_0) and a grid size h, the positions of cell centers can be easily determined on the master. The number of unknowns for the flow field of the largest problem in this paper is approximately 10 million. For this problem size, the storage of velocity, force and 'table of indices' will not pose any problem on the memory space of modern computer system. At the same time, as that will be shown in the next section, the communication time for the data exchange amounts to only a small proportion of the computing time for solving the flow field. If the problem size becomes much larger, other measures have to be taken to reduce the memory usage on the master and the communication overhead between the master and slaves. For example, only the information for the Eulerian grid points adjacent to immersed boundary is included in the 'table of indices'.

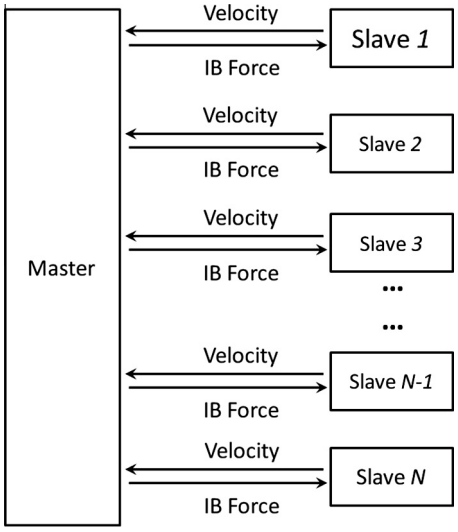

Fig. 7. A schematic representation of the 'gathering-and-scattering' strategy. The master-process is solely responsible for the computation of Lagrangian force and update of the positions of the Lagrangian points. The slave processes are responsible for solving the flow field.

Unlike the treatment of Eq. (4) for the stream function, the matrix in Eq. (5) is explicitly formed and a sequential CG solver is used to solve it on the master. For the problem size of this paper, the parallel solution of Eq. (5) is not performed although it can be done every easily. Such strategy is only necessary when the problem size becomes much larger.

In the parallelization of the immersed boundary code, the strategy proposed in [16] is cumbersome to implement due to frequent handing over of control of the moving Lagrangian points among processes. The performance of such parallelization can also deteriorate because of the load imbalance when the freely-moving points are not evenly distributed among processes. The present 'gathering-and-scattering' strategy greatly simplifies the parallelization and circumvents the problem of load imbalance. Of course, there are some drawbacks in the present strategy such as extra memory usage on the master and extra communication overhead between the master and the slaves. The test of the parallel performance of this strategy will be conducted in Section 4.2 using the problem of sphere sedimentation.

4. Code validation and parallel performance tests

In this section, three simulations are conducted to validate the code and to test its parallel performance. The three-dimensional lid-driven cavity problem is used to test the performance of the solver in the absence of immersed boundary. Sedimentation of spheres in a container is used to test its performance in the presence of complex and moving boundaries. In these two tests, only uniform hexahedral meshes are used. In the numerical simulation of flow through and around a circular array of cylinders, a locally-refined mesh with hanging-nodes is employed.

The three tests in this paper are carried out using the Tianhe-1A supercomputer at the National Supercomputing Center in Tianjin (NSCC-TJ). We use up to 171 Blade Server nodes (2048 processes or CPU cores) of the supercomputer. The basic configuration for each node is: CPU 2* Intel Six-Core Xeon E5670 (2.93 GHz) with 24 GB RAM. The interconnection in the supercomputer is the Chinese-designed NUDT custom designed proprietary high-speed interconnect called Arch that runs at 160 Gbps (bi-directional bandwidth) with a delay of 1.57 μs.

4.1. 3D Lid-driven cavity flow

This benchmark is a simple extension of the 2D lid-driven cavity problem. A sketch of the geometric configuration is shown in Fig. 8. As with the 2D case, the top wall moves in the x-direction at a prescribe velocity, while the left and the right walls remain stationary. The free-slip boundary conditions are applied at the back and front walls. In our simulation, a cubic cavity with the edge size of 1 is used. The Reynolds based on the cavity's edge size and the top lid velocity is 400. Uniform hexahedral elements are used in the computation and the mesh size is $128 \times 128 \times 128$. Time step is adjusted such that the maximum CFL number never exceeds 0.5. The solution is assumed to have reached the steady state if the following criterion is satisfied:

$$|u^{n+1}_{marker} - u^{n}_{marker}| \leqslant \varepsilon. \tag{17}$$

Here u_{marker} is the horizontal velocity component at a marker point, which is located at $(0.75, 0.5, 0.75)$ in this test. ε is a small number which is set to 10^{-4}. It should be noted that with the free-slip boundary conditions on the back and front and at relatively low Reynolds numbers, the solution is inherently two-dimensional

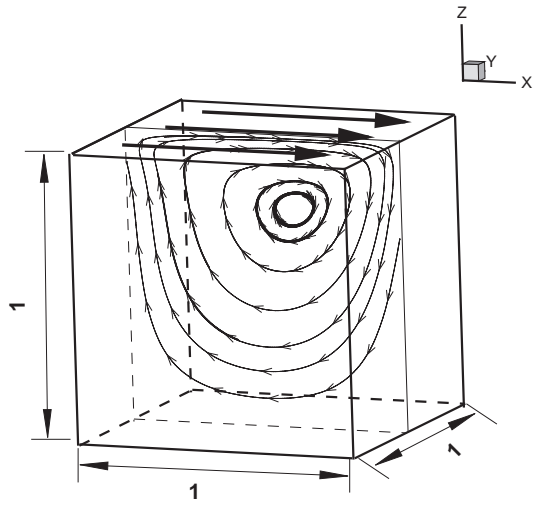

Fig. 8. A sketch of the domain configuration for the 3D cavity flow. The top wall moves in the x-direction at a prescribed velocity. The left and right walls are stationary and the front and back walls are free-slip.

although a three-dimensional computational domain is adopted in the simulation.

The velocity profiles for the steady solution at two centerlines, ($y = 0.5$; $z = 0.5$) and ($y = 0.5$; $x = 0.5$), are compared with the reference ones from Ghia et al. [24] in Fig. 9. It is seen that an excellent agreement has been achieved. A plot of streamlines on the slide of $y = 0.5$ is shown in Fig. 10. From the figure it is seen that the vortex-system (one large vortex at the center and two smaller ones at two bottom corners) which characterizes this type of flow has been successfully captured.

The strong scalability property of the code is evaluated by using a constant-size problem running on different number of processes. To quantify the parallel performance, the most commonly used measurement is the speed up, which is defined as

$$S_p = \frac{T_1}{T_p}; \tag{18}$$

or the parallel efficiency which is defined as

$$E_p = \frac{T_1}{pT_p}, \tag{19}$$

where p is the number of processes; T_1 is the execution time of the sequential algorithm; T_p is the execution time of the parallel algorithm with p processes. It is considered very good scalability if a linear (ideal) speedup is obtained, i.e., $S_p = p$. In practical tests, T_1 is sometimes replaced by $n_0 \cdot T_{n_0}$, where T_{n_0} is the execution time while the parallel code is running on a small number of processes (n_0) [25].

Two tests are conducted in this work: a problem with the mesh size of 32^3 running on 3–63 processes and a problem with the mesh size of 256^3 running on 127–2047 processes. The timings are performed as follows. In each case, we take the average execution time of the first 200 time steps starting from t = 0. For the purpose of comparison among all cases, the time step is fixed to 5×10^{-3}. For the small-size problem (on the mesh of 32^3), the speedup as a function of process number is plotted in Fig. 11. From this figure, an almost linear speedup is observed when the number of processes is less than 15. As the number of processes becomes larger than 15, the speedup curve diverges from the ideal and eventually flattens out when 63 processes are used.

As that stated in Section 3.2, a good (strong) scalability of the parallel algorithm is largely due to the successful overlapping of communication with computation. The overlapping can effectively

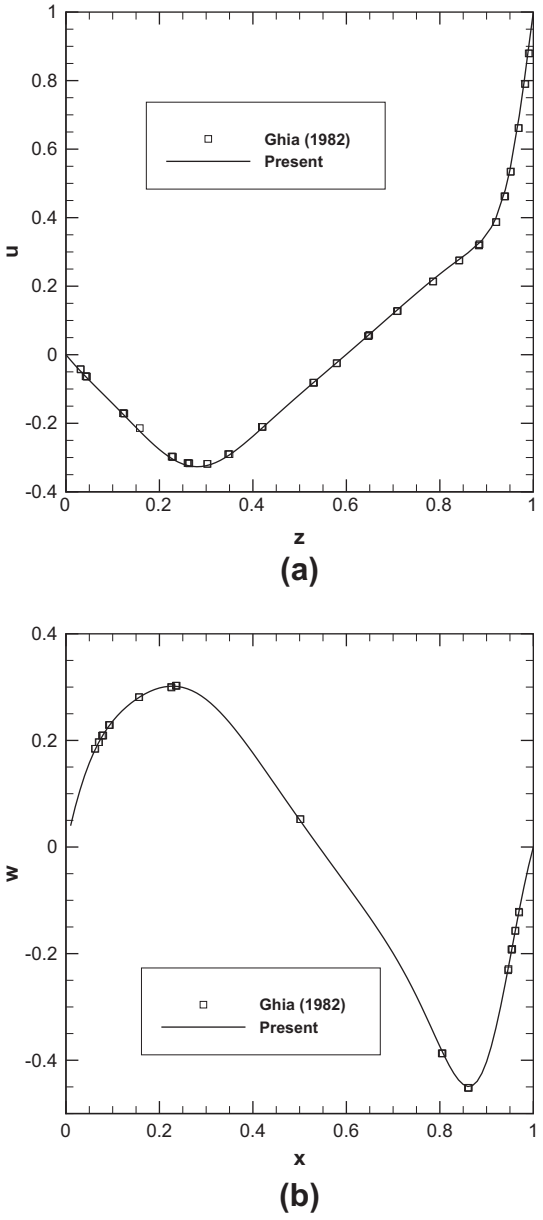

Fig. 9. Velocity profiles on the slice of $y = 0.5$: (a) horizontal speed vs. z at $x = 0.5$; (b) vertical speed vs. x at $z = 0.5$.

'hide' the cost of inter-process communication. The possible causes of the degradation in parallel performance on large number of processes are as follows. For a problem with constant and small size, if the number of processes becomes large, the computation time on each process becomes quite low. Thus the communication time cannot be effectively 'hidden' and the overheads in the latency of the MPI calls also become relatively significant. From the result of this test, it is concluded that good parallel efficiency (in terms of strong scalability) can only be achieved when the problem size is larger than 2000 cells per process.

For the large-size problem (on the mesh of 256^3), the speedup as a function of process number is plotted in Fig. 12. From this figure, a linear speedup is observed in the entire range of process numbers. The good parallel performance in this test can be anticipated since the minimum number of cells per process is around 8000, which is much larger than that of the previous test.

To investigate the weak scalability of the code, we also test a series of cases with increasing problem size by progressively

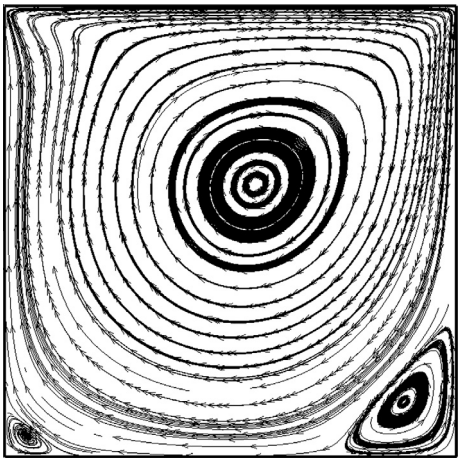

Fig. 10. Streamlines of the cavity flow on the slice of $y = 0.5$ for Re = 400.

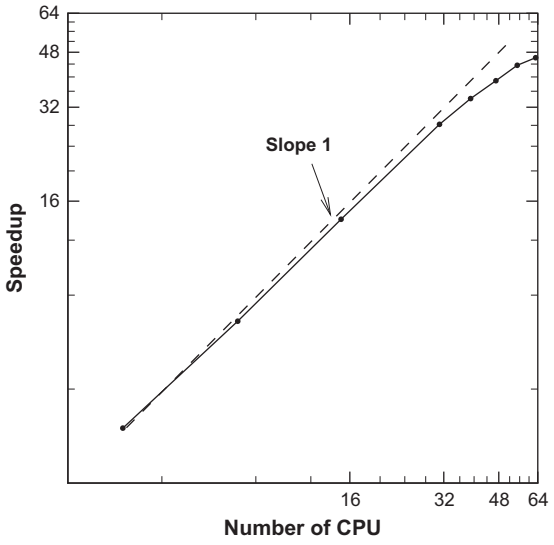

Fig. 11. The speedup vs. process number for the simulation of 3D cavity flow on a mesh of 32^3. An ideal (linear) speedup is obtained when the number of processes is less than 15. If the number of processes is larger than 15, the speedup curve diverges from the ideal and eventually flattens out when 63 processes are used.

refining the mesh. At the same time, we manage to keep the number of unknowns per process (almost) constant (16,500 ± 200) by proportionally increasing the number of processes used. The time step is proportional to the mesh size so that a constant nominal CFL number is kept among all cases. The computational settings for the series of cases studied are listed in Table 1. We study the variation of execution time with the number of processes for a fixed physical time duration of 0.5 starting from $t = 0$.

For this setting, the CPU time for a given physical time duration can simply be estimated by

$$T = N_1(n_p)N_{CG}(n_p)T_1(n_p), \tag{20}$$

where T is the total execution time and n_p is the number of processes used; N_1 is the number of time advancing steps; N_{CG} is the number of iterations performed in the CG solver for each time step and T_1 is the execution time per iteration.

The scaling of T with n_p can be estimated as follows. Since the physical time duration is fixed and the CFL number is kept as a constant, $N_1 \propto \Delta t^{-1} \propto h^{-1} \propto n_p^{1/3}$. For the CG solver (without a sophisticated pre-conditioner), the number of iterations required

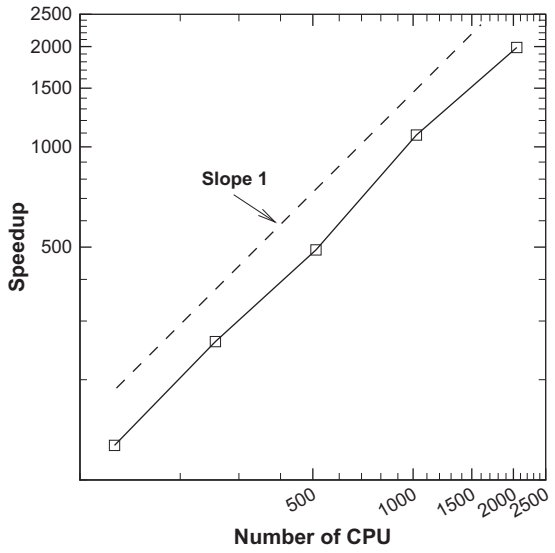

Fig. 12. The speedup vs. process number for the simulation of 3D cavity flow on a mesh of 256^3. A linear speedup is observed in the range of 127–2047 processes.

Table 1
Computational settings for the weak scalability test on the 3D cavity flow.

	n	n_p	h	Δt
1	128^3	127	0.0078	0.005
2	161^3	255	0.0062	0.004
3	204^3	511	0.0049	0.0031
4	256^3	1023	0.0039	0.0025

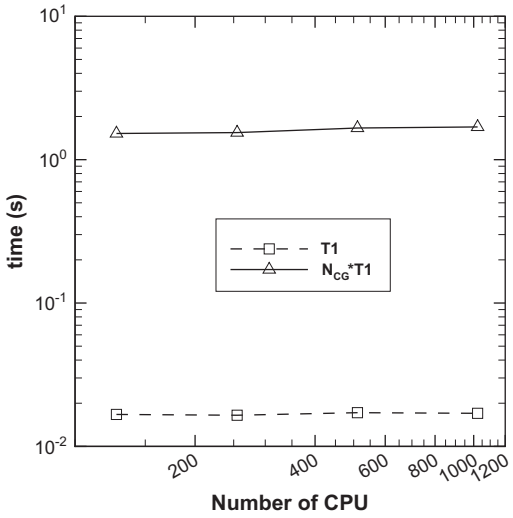

Fig. 13. The execution time for each iteration in the CG solver and for each time advancing step as a function of process number in the weak scalability test of 3D cavity flow.

to achieve convergence (close to machine zero) is proportional to the initial residue and the cubic root of the total number of unknowns [25]. Remarking that the initial residue is proportional to the time step size, thus $N_{CG} \propto \Delta t \cdot n^{1/3} \propto h \cdot n^{1/3} \propto n^{1/3} \cdot n^{1/3} = C_1$, which is a constant independent of n. If we assume that an ideal speed up is achieved in terms of strong scalability, thus T_1 is another constant C_2 independent of n (or n_p). As a result, $T \propto n_p^{1/3}$.

Fig. 13 shows the execution time per iteration and per time advancing step as a function of the process number. It can be seen

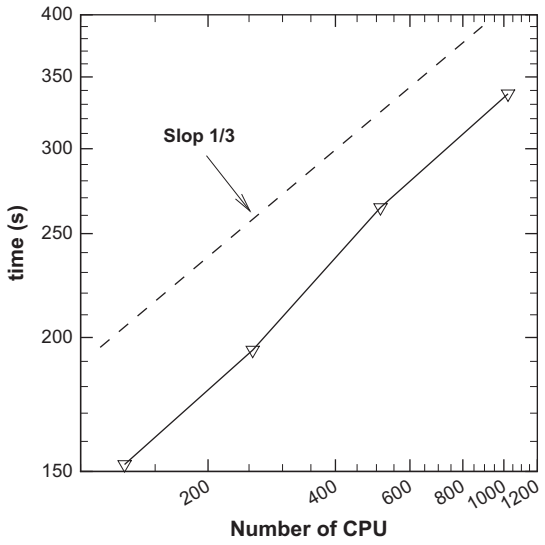

Fig. 14. The total execution time for a given physical time duration as a function of process number in the weak scalability test of 3D cavity flow.

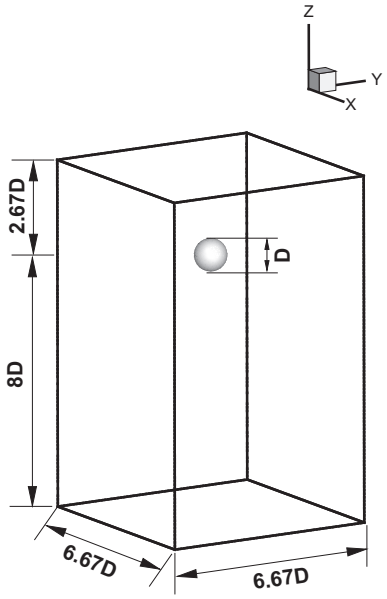

Fig. 15. A sketch of the domain configuration for the sphere sedimentation problem. The computational domain is $6.67D \times 6.67D \times 10.67D$. The distance between the center of the sphere and the bottom wall is $8D$.

that the execution time per iteration remains a constant with the increase of problem size. The execution time per time step slightly increases with the increase of process number due to the slight increase of N_{CG} (less than 5%). Fig. 14 shows the total execution time (for a given physical time duration) as a function of the process number. From this figure, it is clearly seen that the total execution time is proportional to $n_p^{1/3}$, which agrees well with the aforementioned statement.

From the analysis and test above, it is seen that the bottleneck to good weak scalability is the scaling of N_{CG} with n_p. In the present code, N_{CG} scales with the cubic root of n_p. If a grid-independent convergence rate can be achieved, then $N_{CG} \propto \Delta t \propto n^{-1/3}$, thus T can be independent of n(or n_p). Since it appears that a grid-independent convergence rate can be achieved by using the multigrid (MG) method for solving the linear systems, the implementation of MG to accelerate the code will be explored in the future.

Table 2
Physical parameters for the simulations of sphere sedimentation.

Re_p	ρ_f/(kg m^{-3})	ρ_p/(kg m^{-3})	U_∞/(m s^{-1})
1.5	970	1120	0.038
11.6	962	1120	0.091
31.9	960	1120	0.128

4.2. Sphere sedimentation in a container

In this test, the numerical simulation of spheres settling in a container is performed using the immersed boundary method. First, we perform the simulation of the sedimentation of one single sphere in a container. The dimensions of the computational domain are the same as those in the experiments by ten Cate et al. [26] for the purpose of comparison. A sketch of the computational domain configuration is shown in Fig. 15. The physical parameters used in the three testing cases (also the same as those in the experiments of [26]) are listed in Table 2. Here $Re_p = U_\infty d_p/\nu$ is the Reynolds number based on the terminal velocity U_∞, the diameter of the sphere d_p and the kinematic viscosity of the fluid ν.

For the purpose of avoiding ill-conditioned matrix in Eq. (5) and achieving accurate result in the interpolation, the Lagrangian points should be 'evenly' distributed on the surface of the sphere and the inter-point distance should approximate the size of the Eulerian grid. However, the 'even' distribution is not a trivial problem in geometry and actually the very definition of 'even' is not evident either. In this work, we define 'even' as the configuration of points which minimizes the total repulsive energy in a system of charged particles [27].

In the sedimentation, to ensure that the sphere undergoes rigid motions, the velocity $\mathbf{U}_d$ of the Lagrangian point on the spherical surface at position $\mathbf{X}$ can be decomposed into a translational part and a rotational part according to

$$\mathbf{U}_d(\mathbf{X}) = \mathbf{u}_s + \boldsymbol{\omega}_s \times (\mathbf{X} - \mathbf{X}_0), \tag{21}$$

where $\mathbf{X}_0$ is the position of the center of the sphere; $\mathbf{u}_s$ and $\boldsymbol{\omega}_s$ are the translational and angular velocities respectively.

The equations for the translational and rotational velocities of the sphere are:

$$\mathrm{V_s}\frac{\rho_s}{\rho_f}\cdot\frac{\mathrm{d}\mathbf{u}_s}{\mathrm{d}t} = -\int_s \mathbf{f}\mathrm{d}v + \frac{\mathrm{d}}{\mathrm{d}t}\int_s \mathbf{u}\mathrm{d}v + \left(\frac{\rho_s}{\rho_f} - 1\right)\mathrm{V_s}\cdot\mathbf{g}, \tag{22}$$

$$\frac{\mathrm{I_s}}{\rho_f}\cdot\frac{\mathrm{d}\boldsymbol{\omega}_s}{\mathrm{d}t} = \int_s \mathbf{f}\times\mathbf{r}\mathrm{d}v - \frac{\mathrm{d}}{\mathrm{d}t}\int_s \mathbf{u}\times\mathbf{r}\mathrm{d}v, \tag{23}$$

where ρ_s and ρ_f are the densities of the particle and the fluid respectively; $\mathrm{V_s}$ and $\mathrm{I_s}$ are the volume and the momentum of inertia of the sphere respectively; $\mathbf{g}$ is the gravitational acceleration.

An explicit Euler scheme is used to integrate Eqs. (22) and (23). The integrations in the second terms on the right-hand-side of Eqs. (22) and (23) are computed by

$$\int_s \mathbf{u}\mathrm{d}v \approx \sum_{i=1}^{N_x}\sum_{j=1}^{N_y}\sum_{k=1}^{N_z}\alpha_{i,j,k}\mathbf{u}_{i,j,k}V_{i,j,k}, \tag{24}$$

$$\int_s \mathbf{u}\times\mathbf{r}\mathrm{d}v \approx \sum_{i=1}^{N_x}\sum_{j=1}^{N_y}\sum_{k=1}^{N_z}\alpha_{i,j,k}(\mathbf{u}_{i,j,k}\times\mathbf{r}_{i,j,k})V_{i,j,k}, \tag{25}$$

where $V_{i,j,k}$ is the volume of each computational cell and $\alpha_{i,j,k}$ the volume fraction occupied by the solid sphere in each cell.

After obtaining the velocities at the Lagrangian points, the positions of these points are updated by

$$\mathbf{X}^{n+1} = \mathbf{X}^n + \mathbf{U}_d(\mathbf{X}^n)\cdot\Delta t. \tag{26}$$

As that described in Section 3.4, the update of velocities and positions at the Lagrangian points (Eqs. (21)–(26)) is only executed on the master process.

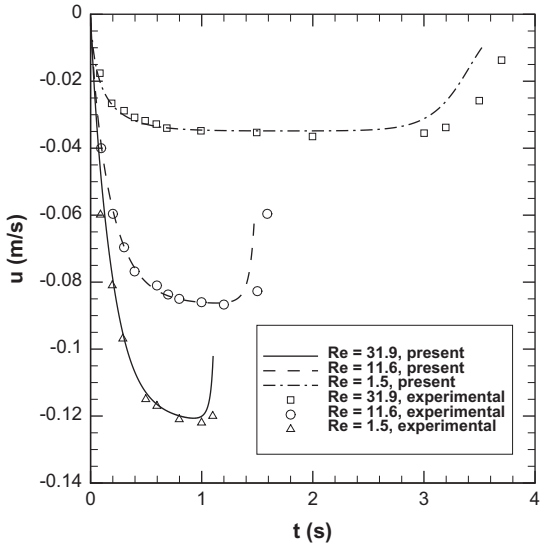

Fig. 16. The time history of settling velocity at three different Reynolds numbers. The settling velocity agrees reasonably well with the experimental measurement in almost the entire sedimentation process. Larger discrepancy is only observed when the sphere approaches the bottom-wall.

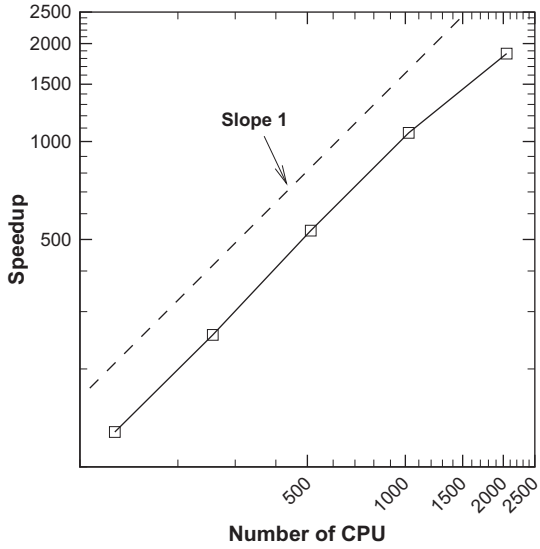

Fig. 17. The strong scalability test for the sphere sedimentation problem running on 128–2048 processes. The computational mesh used in this test is $200 \times 200 \times 320$. A linear speedup is observed in the entire range of process numbers.

To validate the code, simulation is first performed on a mesh of $100 \times 100 \times 160$. The time step is selected to ensure that the maximum CFL number is 0.5. The number of Lagrangian points representing the particle surface is 575. Fig. 16 shows the numerical results for the time history of settling velocity at three different Reynolds numbers. The experimental results from [26] are also plotted for the purpose of comparison. It is seen that the simulation results for the settling velocity agree well with the experimental measurements in all three cases. Larger discrepancy between the computational and experimental results is observed when the sphere approaches the bottom-wall. This discrepancy can be reduced by using a locally-refined mesh together with the lubrication correction and the soft-sphere collision schemes [28,29]. Since the focus of this study is the parallel performance of the code, those strategies are not implemented in the present work.

To evaluate the strong scalability of the code, a constant-size problem (for $Re_p = 31.9$) with the mesh of $200 \times 200 \times 320$ is tested by running on 128–2048 processes. The timings are performed as follows. In each case, we take the average execution time of the first 100 time steps that start from $t = 0$. For the purpose of comparison among all cases, the time step is fixed to 5×10^{-3}. The speedup vs. number of processes is shown in Fig. 17. From this figure, a perfect linear speedup is observed for the process numbers in the range of 128–1024. For the process number of 2048, the curve slightly diverges from the ideal and the parallel efficiency is around 91%.

The weak scalability of the parallel code with the presence of immersed boundary is also evaluated. Similar to that of the previous subsection, a series of cases with increasing problem size and process number are tested. The number of unknowns per process is approximately a constant $(16{,}500 \pm 100)$ for each case. Time step sizes are chosen for each case such that the nominal CFL numbers remain constant among all cases. The number of Lagrangian points used in each case ensures that the inter-point distance is comparable with the size of the Eulerian grid. The computational settings for the series of cases are listed in Table 3.

For this computational setting, the CPU time for a given physical time duration can be estimated by

$$T = N_1(n_p)[N_{CG}(n_p)T_1(n_p) + T_2(n_p) + T_3(n_p)], \tag{27}$$

Table 3
Computational settings for the weak scalability test of sphere sedimentation problem.

	n	M	n_p	h	Δt
1	$100 \times 100 \times 160$	575	128	0.0667	0.005
2	$126 \times 126 \times 200$	902	256	0.0529	0.004
3	$160 \times 160 \times 250$	1393	512	0.0417	0.0031
4	$200 \times 200 \times 320$	2071	1024	0.0334	0.0025

where T_2 is the time for computing the Lagrangian force (solving Eq. (5) on the master process); T_3 is the time for communication between the master and the slaves. We assume that the execution time (on the master) for assembling the matrix of Eq. (5) and updating the position of the sphere (Eqs. (21)–(26) is negligible. These assumptions have been verified by our test below.

For the scaling of T_2 with n (or n_p), we notice that the number of unknowns (M) for Eq. (5) is proportional to $n^{2/3}$ and the initial residue is independent of n; the execution time for each CG iteration is also proportional to the number of unknowns (since a sequential CG solver is used). Thus, $T_2 \propto (n^{2/3})^{1/3} \cdot n^{2/3} \propto n^{8/9}$. Due to the fact that the communication between the master and the slaves involves all Eulerian grid points, it is reasonable to assume that $T_3 \propto n$. According to the analysis of the previous subsection, the first term in the square bracket of Eq. (27) is approximately a constant independent of n. Although T_2 and T_3 increase faster than the first term with the increase of n, since the multiplicative factors in them are very small, the first term is still the dominating one for certain range of n_p.

The total execution time T (for a given physical time duration) vs. the process number is shown in Fig. 18. From the figure it is seen that the execution time scales approximately as $n_p^{1/3}$, which is the same as that for the 3D cavity problem. This scaling law is in agreement with the aforementioned statement. A breakdown of the execution time in one time advancing step is shown in Fig. 19. The time spent in solving the flow field, computing the immersed boundary force and exchanging data between the master and the slaves are plotted in the figure. From this figure, we can conclude that the most time-consuming part is the solution of the flow field in all cases. In each time step, the time spent in solving NS equations is at least one order of magnitude larger than that

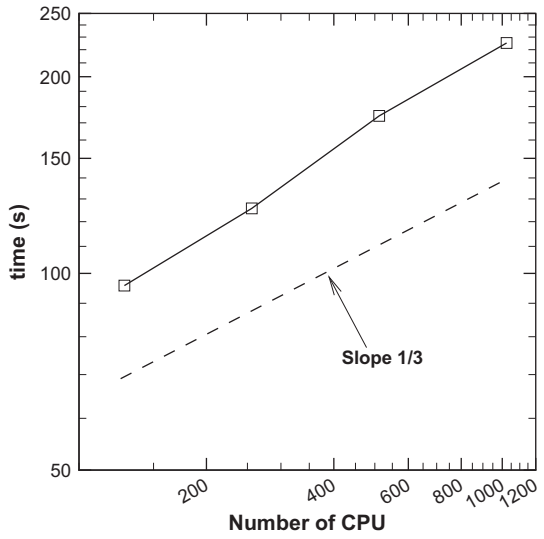

Fig. 18. The total execution time for a given physical time duration as a function of process number in the weak scalability test of sphere sedimentation problem.

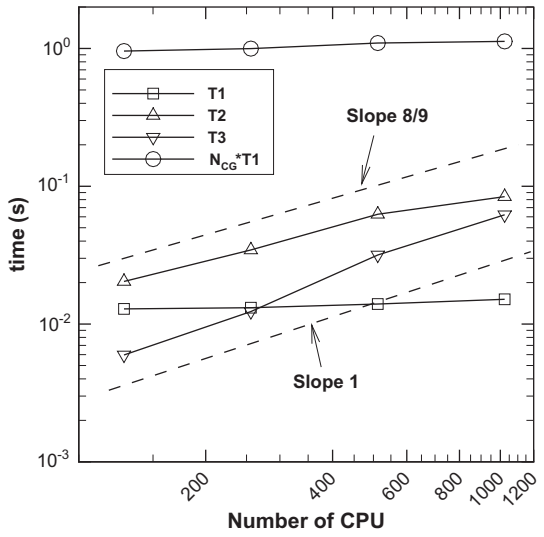

Fig. 19. A breakdown of the execution time in one time advancing step for weak scalability test of sphere sedimentation problem. The time spent in solving the flow field, computing the immersed boundary force and exchanging data between the master and the slaves are shown.

in computing the force and exchanging the data. The time for solving NS equation in one time step $N_{CG}*T_1$ is almost a constant (which varies no more than 5%) with the increase of process number. This is in agreement with the analysis above. In the entire range of process numbers, the time used in the data exchange (communication) is less than that used in the force computing. When the process number is small, the time spent in communication is one order magnitude smaller than that in computing the force. However, with the increase of process number, the communication time increases faster than the time for force computing. This is also in agreement with the aforementioned scaling laws of T_2 and T_3. In case where 1024 processes are used, the communication time and force-computing time are of the same order in magnitude.

It is seen that for the computations with the inclusion of immersed boundary and the number of unknowns up to ten million, at least 85% of the total execution time is spent in solving the flow field and thus the grid-dependent convergence rate of the CG solver is still the bottleneck in achieving a good weak scalability. However, if the computational scale is much larger or more Lagrangian points are deployed (such as in the particle-laden flow), the three terms in the bracket of Eq. (27) can be of the same order in magnitude. Under such circumstances, some additional measures are mandatory to reduce the communication and the force-computing time. For example, by using a parallel CG solver for Eq. (5), the dependency of the execution time for each CG iteration on n can be eliminated, thus $T_2 \propto n^{2/9}$; and by restricting the Eulerian grid points that involved in the communication to those adjacent to the immersed boundary, T_3 may scale as $n^{2/3}$.

In the tests above, the ratio of the number of Lagrangian unknowns to that of the Eulerian unknowns is only about 0.02%. Next, we would like to evaluate the efficiency of the proposed 'gathering and scattering' strategy in cases where the computational cost related to the Lagrangian variables becomes larger. For this purpose, we consider the sedimentation of 112 spheres in a container. The dimensions of the container are the same as those in the previous tests. Initially, the centers of the 112 spheres are placed at the nodes of a $4 \times 4 \times 7$ matrix (in x, y, and z dimensions). The gap between two neighboring spheres is $0.4D$. The gap between the four side-walls and nearest spheres is $0.735D$; while that between the top and bottom walls and nearest spheres is $0.3D$. The schematic drawing of the configuration of this problem is shown in Fig. 20. In this problem, the ratio of the number of Lagrangian unknowns to that of the Eulerian unknowns is around 2%, which is the ratio often encountered in the state-of-art interface-resolved simulations of particulate flows (such as that in [30]).

The weak scalability test is also conducted for this problem by running on 64–1024 processes. The resolutions of the Eulerian meshes are the same as those in the sedimentation of a single sphere (see Table 3). In each case, the number of Lagrangrain points has been increased by a factor of 112. The timing is also the same as that in the test of the sedimentation of a single sphere.

The total execution time T (for a given physical time duration) vs. the number of processes is shown in Fig. 21. From the figure it is seen that the execution time scales much higher than $n_p^{1/3}$. We conjecture that this is due to the sequential solution of Eq. (5)in the master process. This is confirmed by using a parallel solver for the solution of the Lagrangian force. More specifically speaking, for the weak stability tests running on 64, 128, 512

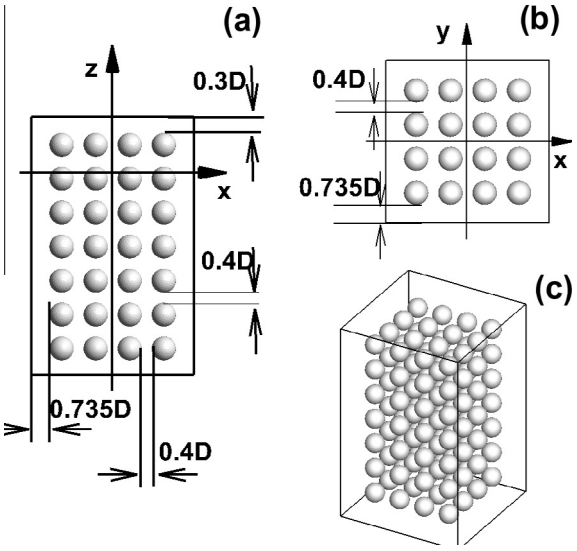

Fig. 20. The schematics drawing of the configuration for the sedimentation of 112 spheres in a container: (a) side view, (b) top view, (c) perspective view.

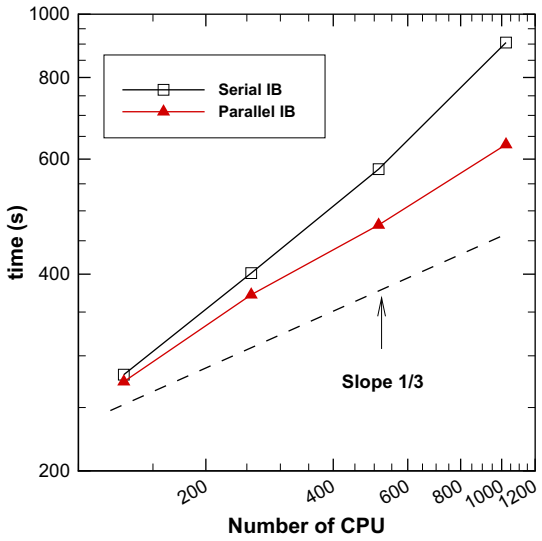

Fig. 21. The total execution time for a given physical time duration as a function of process number in the weak scalability test of the sedimentation of 112 spheres in a container.

and 1024 processes, respectively, 2, 4, 6 and 8 processes are used to solve Eq. (5). This ensures that 25,000–32,000 Lagrangian unknowns are allocated to each process in all the testing cases. The result indicates that the scaling of $n_p^{1/3}$ can be restored by the parallel solution of Eq. (5).

A breakdown of the execution time in one time advancing step is shown in Fig. 22. It is seen that the aforementioned scaling laws are still valid. Comparing with that in the sedimentation of a single sphere, the execution time in this problem has increased by one order of magnitude. This is primarily due to the drastic increase in the number of iterations in the CG solver. From this figure, it is also seen that for the case of 1024 processes, the time elapsed in solving Eq. (5) is comparable with that in the Eulerian solver. Thus it is absolutely necessary to use parallel solver for the solution of Lagrangian forces. Moreover, the '2/9' scaling law aforementioned

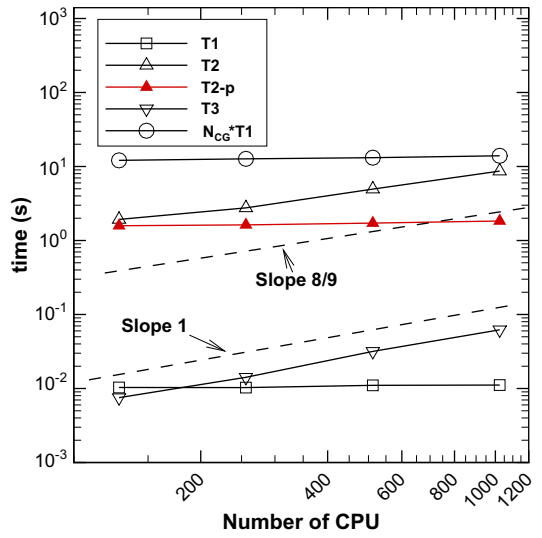

Fig. 22. A breakdown of the execution time in one time advancing step for weak scalability test of the sedimentation of 112 spheres in a container. The time spent in solving the flow field, computing the immersed boundary force and exchanging data between the master and the slaves are shown. 'T2-p' denotes the time spent in solving the Lagrangian forces if a parallel solver is used.

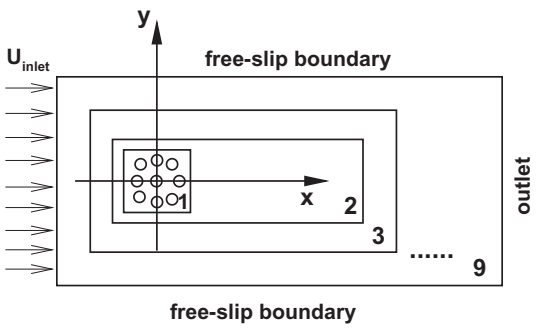

Fig. 23. A schematic drawing of the problem of flow through and around a circular array of cylinders.

(which looks like a constant on a logarithm plot) can be achieved if a parallel solver is used in the solution of Eq. (5). We can see from Fig. 22 that the time for data communication is still very small when comparing with those for the solution of Eulerian and Lagrangian unknowns. Thus 'gathering and scattering' is still an effective strategy in dealing with this problem. In some extreme cases, such as dense particle-laden flows, the time for data communication can become the bottleneck if we apply the same strategy (with a parallel solver for the solution of Lagrangian forces). Under such circumstances, the 'master and slave' strategy proposed by Uhlmann [16] can be more advantageous.

4.3. Flow through and around a circular array of cylinders

In this test, we perform a high resolution two-dimensional simulation to study the local and global effect of a circular array of cylinders on an incident uniform flow [31]. The objective of this test is to demonstrate the performance of parallel code in handling locally-refined mesh with hanging-nodes.

In this problem, 133 cylinders are arranged in a series of concentric rings to allow an even distribution (see Fig. 23). The ratio of array diameter D_G to cylinder diameter D is 21. As a result, the average void fraction $\varphi = N_C(D/D_G)^2 = 0.3$, where Nc is the number of cylinders. The characteristic Reynolds number of the array is $Re_G = 2100$ (for an isolated cylinder, the Reynolds number $Re_D = 100$). Locally-refined meshes with hanging nodes are used in the simulations to arrow different mesh resolutions to be deployed in the computational domain (see Fig. 23). Totally *nine* levels of mesh refinement are employed. For the simulations performed in this work, the values of parameters regarding the domain size and mesh resolution are summarized in Table 4. Comparing with the use of simple (uniform) Cartesian mesh, the mesh local-refinement strategy results in significant save of grid points.

A uniform streamwise flow is imposed at the inlet and the static pressure at the outlet is set to zero. At the top and bottom walls, a free-slip boundary is prescribed. The uniform streamwise velocity is used as the initial condition for the entire domain. The time step is chosen to ensure that the maximum CFL number never exceeds 0.5 (much smaller time steps are used during the starting procedure to avoid stability problem). Fig. 24 shows the wake structure behind the array of cylinders after periodicity is reached. It is seen

Table 4
Parameters for the computational domain and mesh in the simulation of flow over an array of cylinders.

Domain size	*Refinement levels*	h_{min}	h_{max}	n	M
3456D × 512D	9	0.01D	2.56D	7.27 m	41,762

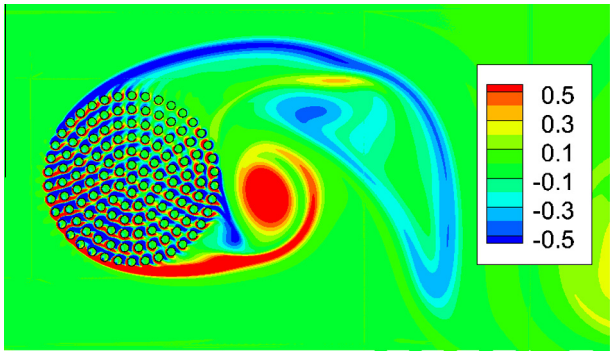

Fig. 24. The vorticity contours in the wake behind a circular array of cylinders.

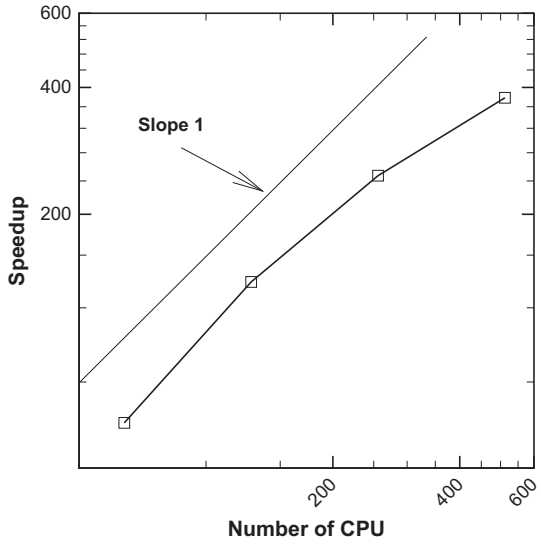

Fig. 25. The speedup vs. processes in the simulations of flow through and around a circular array of cylinders.

that the vortex shedding pattern is very similar to that around a large cylinder of diameter D_G. This is consistent with the result in [31], where a body-fitted unstructured mesh was used. Some quantitative results are also in agreement with those obtained in [31]. The Strouhal number and drag coefficient are 0.24 and 1.72 respectively in the present work, while the corresponding reference values are 0.25 and 1.77 in [31].

To evaluate the strong scalability of the code in the studying of this problem, we perform a series of simulations by running on 64–512 processes. The timings are performed as follows. In each case, we take the average execution time of the first 400 time steps (starting from $t = 0$, with the time step being fixed to 0.005). To avoid an initial velocity jump at the surfaces of the cylinders, we use the following inlet boundary condition for this test:

$$U(t)|_{inlet} = \begin{cases} \sin\left(\frac{\pi}{2}t\right), & 0 \leqslant t < 1.0; \\ 1.0, & t \geqslant 1.0. \end{cases} \tag{28}$$

The speedup vs. number of processes is shown in Fig. 25. From this figure, it is seen that a satisfactory speed-up is achieved for the number of processes up to 512 (where the parallel efficiency is approximately 80%). In this problem, the number of Lagrangian unknowns is only 0.5% of that of the Eulerian unknowns. The computational cost in the master process (related to the Lagrangian variables) is small as compared with that in the Eulerian flow solver, even if the sequential solver is used to solve Eq. (5). Moreover, the time required in the data communication is very small. Thus the strategy of 'gathering and scattering' is also very effective in dealing with this problem.

5. Discussion and conclusions

We have parallelized an immersed boundary solver for the simulation of flows in complex and moving geometries. The code parallelization is based on the domain decomposition strategy and programmed using MPI. The parallel algorithm designed in this paper ensures an effective overlapping of communication and computation, which is the key factor for achieving high performance on clusters of workstations. In addition, a 'gathering and scattering' strategy is implemented to simplify the parallelization of force computing at immersed boundaries. Numerical tests have been performed on problems with and without the presence of immersed boundary. Good parallel efficiency (in term of strong scalability) has been achieved for problem size up to 10 million and number of processes in the range of 16–2048 processes. The ideal (linear) speedup can be reached if the number of unknowns per process is larger than 2000.

In the present parallel implementation, we have not taken additional advantage of the shared memory in the communication (e.g. using MPI across nodes and OpenMP within nodes). Supposedly such mixed OPENMP/MPI programming is of little use due to the fact that the communication overhead with MPI is effectively hidden by the overlapping of communication and computation.

The major limitation of the present code is that the parallel performance in terms of weak scalability is not very good. As the problem size increases, the number of iterations required for convergence in the CG solver increases. To provide a solution with an (almost) grid-independent convergence rate, the parallel multigrid method will be explored in the future. Furthermore, if the computational scale becomes very large (in the order of several billion grid points), mandatory measures should be taken to further reduce the overhead (both in computation and in communication) related to the treatment of Lagrangian forces at the immersed boundaries.

Acknowledgements

This work is supported by Chinese Academy of Sciences under Project Nos. KJCX-SW-L08 and KJCX3-SYW-S01, National Natural Science Foundation of China under Project Nos. 10702074,10872201 and 11023001. The authors also like to thank the National Supercomputing Center in Tianjin (NSCC-TJ) for the allocation of computing time.

References

[1] Peskin CS. The immersed boundary method. Acta Numer 2002;11:479–517.
[2] Mittal R, Iaccarino G. Immersed boundary methods. Annu Rev Fluid Mech 2005;37:239–61.
[3] De Tullio MD, De Palma P, Iaccarino G, Pascazio G, Napolitano M. An immersed boundary method for compressible flows using local grid refinement. J Comput Phys 2007;225:2098–117.
[4] You D, Mittal R, Wang M, Moin P. Computational methodology for large-eddy simulation of tip-clearance flows. AIAA J 2004;42:271–9.
[5] Borazjani I, Ge L, Sotiropoulos F. Curvilinear immersed boundary method for simulating fluid structure interaction with complex 3D rigid bodies. J Comput Phys 2008;227:7587–620.
[6] Kang S, Iaccarino G, Ham F, Moin P. Prediction of wall-pressure fluctuation in turbulent flows with an immersed boundary method. J Comput Phys 2009;228:6753–72.
[7] Tai CH, Zhao Y. Parallel unsteady incompressible viscous flow computations using an unstructured multigrid method. J Comput Phys 2003;192:277–311.
[8] Grismer MJ, Strang WZ, Tomaro RF, Witzeman FC. Cobalt: a parallel, implicit, unstructured Euler/Navier–Stokes solver. Adv Eng Softw 1998;29:365–73.

[9] Karimian SAM, Straatman AG. Discretization and parallel performance of an unstructured finite volume Navier–Stokes solver. Int J Numer Meth Fluids 2006;52:591–615.
[10] Mavriplis D, Pirzadeh S. Large-scale parallel unstructured mesh computations for 3D high-lift analysis. AIAA J Aircraft 1999;36:987–98.
[11] Ramamurti R, Lohner R. A parallel implicit incompressible flow solver using unstructured meshes. Comput Fluids 1996;5:119–32.
[12] Gropp WD, Kaushik DK, Keyes DE, Smith BF. High-performance parallel implicit CFD. Parallel Comput 2001;27:337–62.
[13] Aumann P, Barnewitz H, Schwarten H, Becker K, Heinrich R, Roll B, et al. MEGAFLOW, parallel complete aircraft CFD. Parallel Comput 2001;27:415–40.
[14] Dolean V, Lanteri S. Parallel multigrid methods for the calculation of unsteady flows on unstructured grids: algorithmic aspect and parallel performances on clusters of PCs. Parallel Comput 2004;30:503–25.
[15] Koubogiannis DG, Poussoulidis LC, Rovas DV, Giannakoglou KC. Solution of flow problems using unstructured grids on distributed memory platforms. Comput Method Appl Mech Eng 1998;160:89–100.
[16] Uhlmann M. Simulation of particulate flows on multi-processor machines with distributed memory. CIEMAT technical report no. 1039, Madrid, Spain, ISSN 1135-9420; 2003.
[17] Wang ZL, Fan JR, Luo K. Parallel computing strategy for the simulation of particulate flows with immersed boundary method. Sci China Ser E – Tech Sci 2008;51:1169–76.
[18] Wang SZ, Zhang X. An immersed boundary method based on discrete stream function formulation for two- and three-dimensional incompressible flows. J Comput Phys 2011;230:3479–99.
[19] Chang W, Giraldo F, Perot B. Analysis of an exact fractional step method. J Comput Phys 2002;180:183–99.
[20] Karypis G, Kumar V. METIS: a software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. METIS user's manual (version 4.0); 1998.
[21] Su SW, Lai MC, Lin CA. An immersed boundary technique for simulating complex flows with rigid boundary. Comput Fluids 2007;36:313–24.
[22] Perot B, Nallapati R. A moving unstructured staggered mesh method for the simulation of incompressible free-surface flows. J Comput Phys 2003;184:192–214.
[23] Karypis G, Kumar V. A fast and high quality scheme for partitioning irregular graphs. SIAM J Sci Comput 1999;20:259–392.
[24] Ghia U, Ghia KN, Shin CT. High-Re solutions for incompressible flows using the Navier–Stokes equations and a multigrid method. J Comput Phys 1982;48:387–411.
[25] Bosshard C, Bouffanais R, Deville M, Gruber R, Latt J. Computational performance of a parallelized three-dimensional high-order spectral element toolbox. Comput Fluids 2011;44:1–8.
[26] ten Cate A, Nieuwstad CH, Derksen JJ, van den Akker HEA. Particle imaging velocimetry experiments and Lattice–Boltzmann simulations on a single sphere settling under gravity. Phys Fluids 2002;14:4012–25.
[27] Saff EB, Kuijlaars ABJ. Distributing many points on a sphere. Math Intell 1997;19:5–11.
[28] Breugem WP. A combined soft-sphere collision/immersed boundary method for resolved simulations of particulate flows. In: Proceedings of the ASME 2010 Third joint US-European fluids engineering summer meeting, Montreal, Quebec, Canada, 1–5 August 2010 (FEDSM-ICNMM 2010-30634).
[29] Feng ZG, Michaelides EE. Proteus: a direct forcing method in the simulations of particulate flows. J Comput Phys 2005;220:20–51.
[30] Uhlmann M. An immersed boundary method with direct forcing for the simulation of particulate flows. J Comput Phys 2005;209:448–76.
[31] Nicolle A, Eames I. Numerical study of flow through and around a circular array of cylinders. J Fluid Mech 2011;679:1–31.