THE IMMERSED INTERFACE METHOD
FOR FLOW AROUND NON-SMOOTH BOUNDARIES
AND ITS PARALLELIZATION

Approved by:

Dr. Sheng Xu
Associate Professor of Mathematics

Dr. Johannes Tausch
Professor of Mathematics

Dr. Barry Lee
Associate Professor of Mathematics

Dr. Weihua Geng
Assistant Professor of Mathematics

Dr. Paul Krueger
Professor of Mechanical Engineering

THE IMMERSED INTERFACE METHOD FOR FLOW AROUND NON-SMOOTH BOUNDARIES AND ITS PARALLELIZATION

A Dissertation Presented to the Graduate Faculty of the

Dedman College

Southern Methodist University

in

Partial Fulfillment of the Requirements

for the degree of

Doctor of Philosophy

with a

Major in Applied Mathematics

by

Yang Liu

B.S., Applied Mathematics, University of Science and Technology Beijing
B.S., Applied Mathematics, University of Texas at Arlington
M.S., Applied Mathematics, Southern Methodist University

May 20, 2017

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my advisor Prof. Sheng Xu for the continuous support of my Ph.D study and related research, for his patience, motivation, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis.

Besides my advisor, I would like to thank the rest of my thesis committee: Prof. Barry Lee, Prof. Johannes Tausch, Prof. Paul Kruger and Prof. Weihua Geng, for their insightful questions, comments and encouragement. I am very grateful to all the other faculty and staff members from SMU department. Their endless instructions, advise, help and support are all very important and valuable to my student life here. I would also like to thank the US National Science Foundation for its financial support to my research work.

Last but not the least, I would like to thank my parents Yajun Liu and Xiaolei Zhao, my beloved Hongni Wang and all my friends for supporting me throughout writing this thesis and my life in general.

Liu, Yang B.S., Applied Mathematics, University of Science and Technology Beijing
B.S., Applied Mathematics, University of Texas at Arlington
M.S., Applied Mathematics, Southern Methodist University

The Immersed Interface Method
for Flow Around Non-smooth Boundaries
and Its Parallelization

Advisor: Dr. Sheng Xu
Doctor of Philosophy degree conferred May 20, 2017
Dissertation completed May 20, 2017

In the immersed interface method (IIM), the boundaries of objects in a fluid are treated as immersed interfaces in the fluid. Singular forces are used to represent the effects of the objects on the fluid, and jump conditions induced by the singular forces are incorporated into numerical schemes to simulate the flow. Previously, the immersed interface method for simulating smooth rigid objects with prescribed motion in 2D & 3D incompressible viscous flows has been developed by Xu [72–74]. In this thesis, we extend the method for rigid objects with non-smooth boundaries by computing necessary jump conditions using line segment representation of 2D objects. We also present the parallelization strategy for the development of a high-performance program for distributed-memory parallel computing with Message Passing Interface(MPI). Different tests are performed, and numerical results and comparisons are given to study the accuracy, efficiency and robustness of our method.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

This thesis is dedicated to my family.

Chapter 1

INTRODUCTION

1.1. Problem Statement

In the field of computational fluid mechanics, one important topic is how to resolve moving boundaries and their effects on fluid flow accurately and efficiently. For example, biolocomotion is very popular in the past several decades and the study of insect flight aerodynamics has attracted lots of researchers. When a butterfly flaps its wings, we would like to know what is the velocity and pressure around the wings and the fluid force and torque. Think about it further, if we design a wing by ourselves, how can we simulate the aerodynamics of the flapping wings. If the wings have complex geometries, how can we examine if it works as we expect. All the thoughts bring us to a big question that, can we design a method, which is suitable to test aerodynamics of moving objects with complex geometries, and the problem can be solved accurately and efficiently. Driven by the question above, we started the study of immersed interface method for flow around objects with non-smooth boundaries.

This work is difficult and challenging. Because the object could be moving or static, we need to think about whether to use Lagrangian method to focus on the object or use Eulerian method to focus on the space. For the objects in the flow field, we need develop a method to couple the movement with the flow field, and think about how to handle the domain inside and outside of the objects. In addition, the method should be able to handle objects with different complex geometries and can solve the problem stably. Assume we put hundreds of moving objects into the flow field, the method should be efficient to solve the problem and the computational cost should not be increased dramatically with more objects. Besides, the method should have the good capacity to be easily implemented into a high-performance

program to speed up the problem solving process. For complicated physics problems with a large computational domain, a serial program is not enough to handle all the work, but a highly efficient parallel method is not easy to develop.

For the past decades, many numerical methods have been developed to address problems in this area and they all have their advantages and disadvantages. We will give a brief review in the next section.

1.2. Literature Review

In the Lagrangian methods, we will track the movement of objects as well as the physical properties. The computational mesh will be regenerated with the movement of objects. One of the most popular mesh generation method is called Chimera mesh or overset mesh. The main idea is to decompose the complex geometry into a system of overlapping grids and interpolation is used to exchange the boundary information. This method has high quality under large displacements and efficient for high-order accurate methods. Details can be found in [39, 45, 60]. However, the computational cost is large due to the regeneration of grids for moving boundary problems. Most researchers would combine Lagrangian and Eulerian methods to solve the problems involving the fluid motion.

One common approach to resolve the moving boundaries problem is based on Cartesian grids. Objects will move in the fixed computational domain and the position will be calculated at each time step. Researchers have been working on developing new Cartesian grid methods to reduce the computational cost while maintain the accuracy and efficiency. Among the developed Cartesian grid methods, some can be applied to solve flow problems for moving objects with the prescribed motion. Examples of these methods include the immersed boundary method [19, 25, 26, 43, 44, 48], sharp interface method [65, 80], immersed interface method [29, 30, 34, 77], Lattice Boltzmann method [27], Russell and Wang's method [50], ghost cell methods [4, 64], etc. In the sharp interface method, a mixed Eulerian−Lagrangian framework is employed, which treats the immersed boundary as a sharp interface. Second-order accurate finite-volume method is used to discretize the Navier-Stokes equations and a

second-order accurate fractional-step scheme is used for time marching. Boundary motion can be properly produced by translating each boundary particle with the prescribed velocity. For the Lattice Boltzmann method, it can be simply considered as a numerical solver of the Boltzmann equation. A regular Eulerian grid is used for the flow domain and a Lagrangian grid is used to follow the moving objects in the flow field. The velocity field of the fluid and moving objects is solved by adding a force density term into the Lattice Boltzmann equation. In Russell and Wang's method, instead of solving the velocity and pressure directly, they solve the flow problem using a streamfunction–vorticity formulation and represent the embedded objects with discontinuities. In their method, they first solve the Poisson equation for streamfunction with discontinuities at the boundary. Then they solve a homogeneous inviscid problem using boundary method. Vorticity is distributed around object boundary to satisfy no-slip condition, and the vorticity is integrated in time within the effects of singular sources. For the ghost cell method, ghost cells are fictitious cells inside the object. The grid cells can be separated by the object boundary and finite difference approach can no longer be applied. Then ghost cells are needed and boundary conditions of the object can be implicitly incorporated through the ghost cells. Ghost cells can be updated by extrapolating values from the flow field and the boundary.

Among all the methods above, the immersed boundary method is most notable. The immersed boundary method was first introduced by Peskin in 1972 [43] to simulate blood flow in human heart. A detailed explanation can be found in [44]. The immersed boundary method treats the boundary of an immersed object as a set of Lagrangian fluid particles. A singular force is added to the Navier-Stokes equation and determined by fluid particles to represent the effects of the object on the fluid. The force distribution is described as a Dirac delta function. Because of the formulation of the Navier-Stokes equation, the immersed boundary method has the advantage that it can handle multiple moving objects easily and efficiently. A Cartesian grid method is used for fluid and Lagrangian grid used for the immersed boundary. The Naiver-Stokes equation can be solved with the communication between the fluid and immersed boundary. When the immersed boundary method was first

introduced, some disadvantages of the initial implementation were exposed, and one of the biggest is it only has first order accuracy. A lot of research have been done to improve the immersed boundary method in the recent years.

1.3. Immersed Interface Method

Motivated by the goal to achieve second-order accuracy, LeVeque and Li introduced the immersed interface method in 1994 [30, 31]. The immersed boundary method and immersed interface method share the same formulation, and the biggest difference is that in the immersed interface method the finite difference scheme is used to incorporate the jump conditions caused by the Dirac delta function. If the necessary jump conditions are known, then second-order or even higher order accuracy can be secured. The immersed interface method was first introduced to solve elliptic equations [30] and Stokes equation [31]. Later Wiegmann and Bube extented the immersed interface method to nonlinear parabolic equations and Poisson equations with piecewise smooth solutions [69, 70]. In [29, 34], the immersed interface method was extented to solve 2D incompressible Navier-Stokes equations and in [40] it was used to solve the 1D Schrödinger equation. Meanwhile, for the past decade the immersed interface method was developed based on both finite difference method [77, 79] and the finite element method [32, 35] to provide large potential for implementation. In the previous work of Xu [78], he systematically derived jump conditions of all first-, second-, and third-order derivatives of the velocity and the pressure by construction of singular force, as well as the jump conditions of first- and second-order temporal derivatives of the velocity for 3D incompressible Navier-Stokes equations. With these jump conditions, he implemented the immersed interface method to simulate 2D & 3D incompressible viscous flow with moving boundaries [77, 79]. His method is proved to be stable, accurate and efficient to handle single or multiple smooth moving objects.

The shortcoming of Xu's previous work is this method can only be applied to simulate flow around smooth objects, like a circular cylinder or a rounded plate. Previously, cubic splines were used to parametrize the immersed interface, but the interface of object

with complex/non-smooth boundaries cannot be represented in the same pattern. In order to overcome this weakness, our goal here is to develop the immersed interface method for incompressible viscous flow with complex/non-smooth boundaries in both stationary and moving conditions, which can be resolved stably, accurately and efficiently. In our current method, we use line segment panels to represent the interface instead of using cubic spline for surface parametrization. Instead of expressing jump conditions through construction of a singular force, here jump conditions are directly calculated based on the fluid field. We re-derive the principle jump conditions and first- and second-order Cartesian jump conditions for velocity and pressure, such that they can be applied to non-smooth objects. Besides, the new developed method can be easily modified and implemented into a high-performance parallel program. In this thesis there are sufficient details of method derivation, implementation and the design of the parallel program. Anyone who is interested in our method can program and test it.

1.4. Parallel/High-Performance Computing

In the past, programs are written in the way that the instructions will be executed one by one sequentially and can only be executed on one processor. We call this kind of program a serial program. But with the fast development of software and hardware in the past decades, the computers have much more power to handle heavy-duty computations. This has helped the researchers from the scientific computing field to develop more powerful methods, and CFD is a very good example. Nowadays, most of the numerical methods are developed based on the idea of parallel computing. By doing this, a big problem can be split into many small tasks and sent to different processors. Same instructions can be executed at the same time on different processors and an overall control is used to manage the computation. Parallel computing has the ability to solve very large problems which cannot be executed on only one processor and can save computational time. Parallel computing is widely used in almost all areas of science and engineering, in both academia and industry.

Parallel computers are mainly designed in two ways, one computer with multiple cores or multiple computers connected to each other using network connection. Based on the structure of computing sources, the parallel program can be identified as a shared memory program or distributed memory program. Shared memory programming is mainly for one computer with multiple cores, and all cores can pull out information from the same memory. *OpenMP* is one of the most popular application program interfaces(API) that supports it and details can be found in [12]. In distributed memory programming, each processor will store the information in local memory and exchange it with neighboring processors if necessary. The Message Passing Interface Standard(*MPI*) from Argonne National Laboratory is mainly designed for this purpose, as described in [1]:

> The Message Passing Interface Standard (*MPI*) is a message passing library standard based on the consensus of the *MPI* Forum, which has over 40 participating organizations, including vendors, researchers, software library developers, and users. The goal of the Message Passing Interface is to establish a portable, efficient, and flexible standard for message passing that will be widely used for writing message passing programs. As such, *MPI* is the first standardized, vendor independent, message passing library. The advantages of developing message passing software using *MPI* closely match the design goals of portability, efficiency, and flexibility. *MPI* is not an IEEE or ISO standard, but has in fact, become the industry standard for writing message passing programs on HPC platforms.

MPI and *OpenMP* have very good support for C++ and FORTRAN, and are widely used in high-performance computing/scientific computing field. They can also be mixed to create a hybrid *MPI*/*OMP* program. Due to time limits, the development of our parallel program is only based on *MPI*.

Traditionally, parallel programming is mainly developed based on the structure of the central processing units(CPU), but with the fast development of graphics processing units(GPU), more and more researchers are moving to study the possibility of parallel programming on

GPUs. CPUs and GPUs have different structures. Even though each core on a GPU is much slower than a CPU and also the memory cache is much smaller, GPU still has its own strength. Nowadays each CPU can only have no more than 100 cores, but GPUs could have a few thousands cores on a single unit, which shows the potential of highly powerful computing capacity. Some APIs are developed to help people develop parallel programs on GPUs. *OpenMP* supports GPU programming, and there are other popular APIs such as *CUDA*, *OpenCL* and *OpenACC*. In the CFD area, some numerical methods have been developed for parallel programming on GPUs. In [61, 62], Thibault used *CUDA* kernels to implement the projection algorithm to solve the Navier-Stokes equations for incompressible fluid flow. In [49], Rossinelli and Koumoutsakos implemented the vortex particle method for incompressible flow simulations on GPUs. In [11], Castonguay presented a high-order compressible viscous flow solver for mixed unstructured grids on multi-GPU. Some researchers are also working on combining CPU and GPU programing together to solve flow problems [21, 71].

Parallel programming has a high standard requirement for both hardware and software. Regular personal computers are usually not able to handle high-performance programs. SMU has its own high-performance facility, ManeFrame, open to all the faculties and students, below are the technical details as of this writing:

- 1084 nodes with 24 GB of RAM
- 20 nodes with 192 GB of RAM
- 1.2 PB high performance parallel Lustre file system
- All nodes have 8-core Intel® Xeon® CPU X5560 @ 2.80GHz 107 processors
- All nodes are connected by a 20Gbps DDR InfiniBand connection to the core backbone InfiniBand network
- Scientific Linux 6 (64 bit) operating system
- SLURM resource scheduler

All the parallel simulations in this thesis are conducted on ManeFrame.

1.5. Outline

This thesis is organized as follows. In chapter 2, we present the mathematical formulation of the governing equations and the modified finite difference schemes with jump conditions in the immersed interface method. In chapter 3, the new method of derivation for principle and Cartesian jump conditions will be explained in the 2D case. In chapter 4, we will talk about implementation of our current method, including the spatial discretization and temporal discretization. In chapter 5, we will present the parallelization of our method, including domain decomposition, data structure, communication and other improvements. In chapter 6, different flow problems are tested to study the stability, accuracy, efficiency and robustness of our current method. In chapter 7, conclusions will be given.

Chapter 2

GOVERNING EQUATIONS AND FINITE DIFFERENCE SCHEMES

In this chapter, we present the model of the immersed interface method for flow with non-smooth boundaries. In the first section, we will present the Governing equations used in the immersed interface method. In the second section, we will present the finite difference method used for the treatment of the computational domain.

2.1. Governing Equations

In the immersed interface method, we treat an immersed object boundary as an immersed interface, and the effect of the object on the fluids is represented as the singular force. Consider the incompressible viscous flow with an object, the non-dimensional 2D Navier-Stokes equations subject to singular force are given as below,

$$\frac{\partial \vec{u}}{\partial t} + \nabla \cdot (\vec{u}\vec{u}) = -\nabla p + \frac{1}{Re}\Delta \vec{u} + \vec{q} + \int_{\Gamma} \vec{f}\left(\vec{X}, t\right) \delta\left(\vec{x} - \vec{X}\right) dl \tag{2.1}$$

$$\nabla \cdot \vec{u} = 0 \tag{2.2}$$

where $\vec{u} = (u, v)$ is the velocity, p is the pressure. Re is the Reynolds number, Γ is the object boundary immersed in the fluid. $\vec{q} = (q_x, q_y)$ is the finite body force to enforce the rigid motion of the fluid enclosed by the object boundary. $\vec{f}$ is the density of the singular force, $\delta(\cdot)$ is the 2D Dirac δ function, $\vec{F} = \int_{\Gamma} \vec{f}\left(\vec{X}, t\right) \delta\left(\vec{x} - \vec{X}\right) dl$ is the singular force representing the effect of object on the fluid. $\vec{x} = (x, y)$ is the Cartesian coordinates, and $\vec{X} = (X, Y)$ is the Cartesian coordinates of boundary vertices. Multiple objects can be represented in the similar manner.

In our current method, the object has rigid boundaries and has the potential to be extended to deforming objects in the future. The objects can be any kind of shape, with smooth or non-smooth boundaries. The objects are either static or in prescribed motion, so

the position and relative movement of the object can be a function of time, which is

$$\vec{X} = \vec{X}(t) = (X(t), Y(t)) \tag{2.3a}$$

$$X(t) = x_c(t) + X_0 * \cos(\theta(t)) - Y_0 * \sin(\theta(t)) \tag{2.3b}$$

$$Y(t) = y_c(t) + X_0 * \cos(\theta(t)) + Y_0 * \sin(\theta(t)) \tag{2.3c}$$

$(x_c(t), y_c(t))$ is the Cartesian coordinates of a fixed reference point with respect to the boundary, (X_0, Y_0) is the initial coordinates of vertices, and $\theta(t)$ is the rotation angle of the object. Equations (2.1) and (2.2) are defined on the entire domain Ω as shown in Figure 2.1, where Ω^+ is the domain of the fluid and Ω^- is the domain inside the object. The computational domain is fixed and will not change with the time. $\vec{n}$ is the unit normal vector on the vertices and $\vec{\tau}$ is the unit tangential vector, where $\vec{\tau} = (\tau_x, \tau_y) = (-n_y, n_x)$.

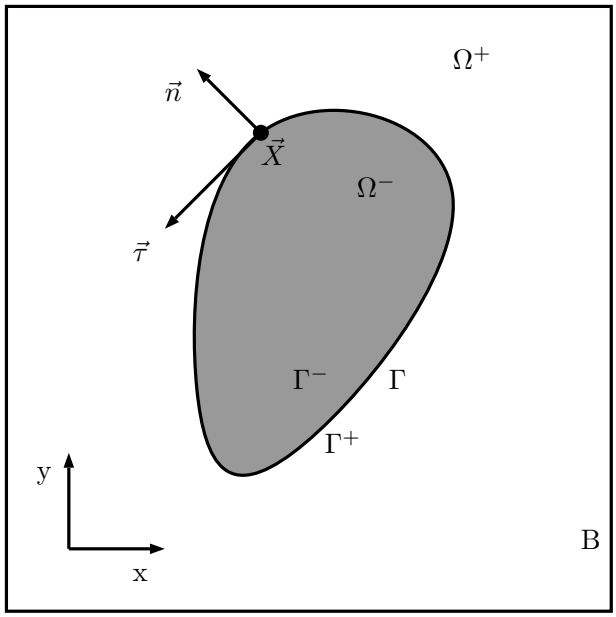


Figure 2.1: Geometric description of the immersed object.

2.2. Finite Difference Scheme

On the boundary, jump conditions are induced by the singular force and discontinuous body force. Different from the immersed boundary method which approximates the Dirac

δ function by using discretized smooth functions, the immersed interface method directly modifies a finite difference scheme to incorporate the jump conditions. Previously from the work of Xu and Wang [78], they presented the generalized Taylor expansion for a piecewise smooth function, which is expressed as below,

$$g\left(z_{i+1}^{-}\right)=\sum_{n=0}^{\infty}\frac{g^{(n)}(z_0^{+})}{n!}(z_{i+1}-z_0)^n+\sum_{l=1}^{i}\sum_{n=0}^{\infty}\frac{\left[g^{(n)}(z_l)\right]}{n!}(z_{i+1}-z_l)^n \tag{2.4}$$

where $g(z)$ is a piecewise smooth function as shown in Figure 2.2. $\left[g^{(n)}(z_l)\right]$ denotes the jump conditions along the z direction, $\left[g^{(n)}(z_l)\right]=g^{(n)}(z_l^{+})-g^{(n)}(z_l^{-})$. The proof for equation (2.4) was presented in [78]. Based on this, second order central finite difference schemes are given in equations (2.5a) and (2.5b). Interpolation scheme with incorporation of jump conditions is also developed and given in equation (2.6), where $g(z)$ is discontinuous at $z=\xi$ and $z=\eta$ as shown in Figure 2.2.

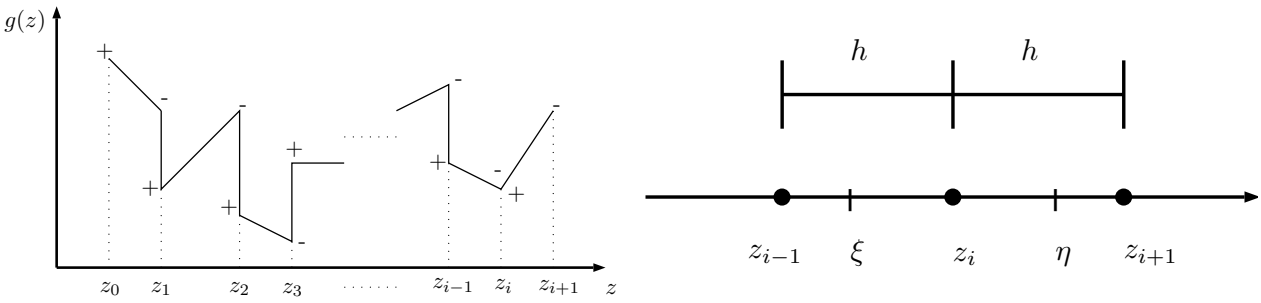


Figure 2.2: Examples for generalized Taylor expansion and finite difference scheme

$$\begin{aligned}\frac{dg\left(z_i^{-}\right)}{dz}&=\frac{g\left(z_{i+1}^{-}\right)-g\left(z_{i-1}^{+}\right)}{2h}+O\left(h^2\right)\\&\quad+\frac{1}{2h}\left(\sum_{n=0}^{2}\frac{-\left[g^n(\xi)\right]}{n!}(z_{i-1}-\xi)^n-\sum_{n=0}^{2}\frac{-\left[g^n(\eta)\right]}{n!}(z_{i+1}-\eta)^n\right)\end{aligned} \tag{2.5a}$$

$$\begin{aligned}\frac{d^2g\left(z_i^{-}\right)}{dz^2}&=\frac{g\left(z_{i+1}^{-}\right)-2g(z_i)+g\left(z_{i-1}^{+}\right)}{h^2}+O\left(h^2\right)\\&\quad+\frac{1}{h^2}\left(\sum_{n=0}^{3}\frac{-\left[g^n\left(\xi\right)\right]}{n!}\left(z_{i-1}-\xi\right)^n-\sum_{n=0}^{3}\frac{-\left[g^n\left(\eta\right)\right]}{n!}\left(z_{i+1}-\eta\right)^n\right)\end{aligned} \tag{2.5b}$$

$$g\left(z_i\right)=\frac{z_{i-1}-z_{i+1}}{2}+O(h^2)+\frac{1}{2}\left[\frac{\partial g(\xi)}{\partial z}\right]\left(z_{i-1}-\xi\right)-\frac{1}{2}\left[\frac{\partial g(\eta)}{\partial z}\right]\left(z_{i+1}-\eta\right) \tag{2.6}$$

Equation (2.5a) is the first-order derivative of function $g(z)$ with incorporation of jump conditions and equation (2.5b) is the second-order derivative of the function $g(z)$. The interpolation equation (2.6) also gives second-order accuracy. The development of the above finite difference schemes and interpolation scheme is very important because the whole method is developed based on the finite difference method and how to calculate and use these jump conditions is the key to the success of our method.

Chapter 3

FORMULATION OF JUMP CONDITIONS

Previously from the work of Xu and Wang [78], the formulation of necessary jump conditions for the immersed interface method in three-dimensional flow simulation has been systematically derived. The formulation of the jump conditions has been applied in the 2D immersed interface method in [72, 77] and the 3D immersed interface method in [79]. The method is proved to be stable, accurate and efficient. But, since cubic splines were used to parametrize the object boundary in 2D, these jump conditions can only be applied to objects with smooth boundaries, like circular cylinder or rounded plate. When it comes to non-smooth objects like square cylinder or rectangular wing, the previous method is no longer valid. Inspired by this, we have developed new formulation of the jump conditions which can be applied to non-smooth objects.

In our current method, there are two main types of jump conditions are needed: Principle jump conditions and Cartesian jump conditions. Principle jump conditions of velocity and pressure across the closed surface, and along their normal directions. Cartesian jump conditions are along the x and y directions of the Cartesian coordinates, and calculated using the principle jump conditions. Below are the jump conditions we need in the immersed interface method,

- Principle jump conditions
  $[\vec{u}]$, $\left[\frac{\partial \vec{u}}{\partial n}\right]$, $[\Delta \vec{u}]$, $[p]$, $\left[\frac{\partial p}{\partial n}\right]$, $[\Delta p]$

- Cartesian jump conditions
  $\left[\frac{\partial \vec{u}}{\partial x}\right]$, $\left[\frac{\partial \vec{u}}{\partial y}\right]$, $\left[\frac{\partial^2 \vec{u}}{\partial x^2}\right]$, $\left[\frac{\partial^2 \vec{u}}{\partial y^2}\right]$, $\left[\frac{\partial^2 \vec{u}}{\partial x \partial y}\right]$
  $\left[\frac{\partial p}{\partial x}\right]$, $\left[\frac{\partial p}{\partial y}\right]$, $\left[\frac{\partial^2 p}{\partial x^2}\right]$, $\left[\frac{\partial^2 p}{\partial y^2}\right]$, $\left[\frac{\partial^2 p}{\partial x \partial y}\right]$

In the previous approach [72,77–79], jump conditions are expressed in terms of singular force. In the 2D case, the formulation for singular force is shown as below,

$$f_n = \int \left(\frac{1}{Re} \frac{\partial \omega}{\partial n}|_{\Gamma^+} + [b_\tau] \right) J d\alpha \tag{3.1a}$$

$$f_\tau = -\frac{1}{Re} \left(\omega|_{\Gamma^+} - 2\dot{\theta} \right) \tag{3.1b}$$

f_n is the normal singular force and f_τ is the tangential singular force. ω is the vorticity, b_τ is the tangential body force, α is the Lagrangian parameter and $J = ||\frac{\partial \vec{X}}{\partial \alpha}||_2$. After f_n and f_τ known, the jump conditions can be expressed using the singular force. Example of principle jump conditions of pressure is shown below

$$[p] = f_n \tag{3.2a}$$

$$\left[\frac{\partial p}{\partial n} \right] = \frac{\partial f_\tau}{\partial \tau} + [b_n] \tag{3.2b}$$

Different from the previous work, we no longer need to calculate the singular force to express the jump conditions. Instead, the jump conditions are directly computed based on the velocity and pressure field. In the work of Xu and Pearson [76], they presented a method to compute the necessary Cartesian jump conditions from given principle jump conditions using a triangular mesh representation of a 3D interface. The triangular mesh representation is simpler and robuster than interface parametrization for complex or non-smooth interface. We have modified the method for computing Cartesian jump conditions using a line segment panel representation of a 2D interface. Here we focus on the development of the method for computing jump conditions.

In the first section, we will present the derivation of principle and Cartesian jump conditions for velocity $\vec{u}$. In the second section, we will present the derivation of principle and Cartesian jump conditions for pressure p. In the third section, we will present the derivation of fluid force calculation.

3.1. Jump Conditions for $\vec{u}$

For the velocity $\vec{u}$, we will first show the derivation of principle jump conditions then the Cartesian jump conditions, as Cartesian jump conditions are derived from its principle jump conditions.

3.1.1. Principle jump conditions

The principle jump conditions for $\vec{u}$ are $[\vec{u}]$, $\left[\frac{\partial \vec{u}}{\partial n}\right]$ and $[\Delta \vec{u}]$.

- $[\vec{u}]$

  The principle jump conditions of velocity is $[\vec{u}] = 0$, as $\vec{u}$ is finite and continuous at the interface.

- $\left[\frac{\partial \vec{u}}{\partial n}\right]$

  For the principle jump conditions of the derivative $\vec{u}$ along normal direction, it can be expressed as

$$\left[\frac{\partial \vec{u}}{\partial n}\right] = \frac{\partial \vec{u}}{\partial n}|_{\Gamma^+} - \frac{\partial \vec{u}}{\partial n}|_{\Gamma^-} \tag{3.3}$$

  As for the 3D case shown in [73], $\frac{\partial \vec{u}}{\partial n}|_{\Gamma^-} = \vec{\Omega} \times \vec{n}$, which is the formula for the rigid motion of an object. When it comes to 2D,

$$\frac{\partial \vec{u}}{\partial n}|_{\Gamma^-} = \dot{\theta} \cdot \vec{\tau}, \tag{3.4}$$

  where $\dot{\theta}$ is the angular velocity of rotation for the objects in the Cartesian system and is changing with time, $\dot{\theta} = \dot{\theta}(t)$.

  To calculate $\frac{\partial \vec{u}}{\partial n}|_{\Gamma^+}$, we applied one-sided finite difference scheme along the normal direction $\vec{n}$ as shown in Figure 3.1. It can be expressed as

$$\frac{\partial \vec{u}}{\partial n}|_{\Gamma^+} = \frac{-3\vec{u}(S_0) + 4\vec{u}(S_1) - \vec{u}(S_2)}{2\delta n} + O(\delta n^2) \tag{3.5}$$

  where δn is the distance between two adjacent points along normal direction $\vec{n}$ and $\delta n \geq \sqrt{\delta x^2 + \delta y^2}$ to avoid the two adjacent points are in the same grid cell. For

velocity $\vec{u}$ at vertex S_0 on the boundary, it is the prescribed velocity of the boundary,

$$u(S_0) = \dot{x_c} - \dot{\theta} \cdot (Y - y_c) \tag{3.6a}$$

$$v(S_0) = \dot{y_c} + \dot{\theta} \cdot (X - x_c) \tag{3.6b}$$

(x_c, y_c) is the Cartesian coordinates of a fixed reference point with respect to the boundary, $(\dot{x_c}, \dot{y_c})$ is the velocity of the reference point movement, and (X, Y) is the current coordinates of the vertex S_0. The velocity at points S_1 and S_2 are interpolated from four surrounding points from the Cartesian grid cell. For example, velocity at the point S_2 can be interpolated from points I, II, III, IV. If higher order accuracy for $\frac{\partial \vec{u}}{\partial n}|_{\Gamma^+}$ is required, we can add more points on normal direction to achieve that.

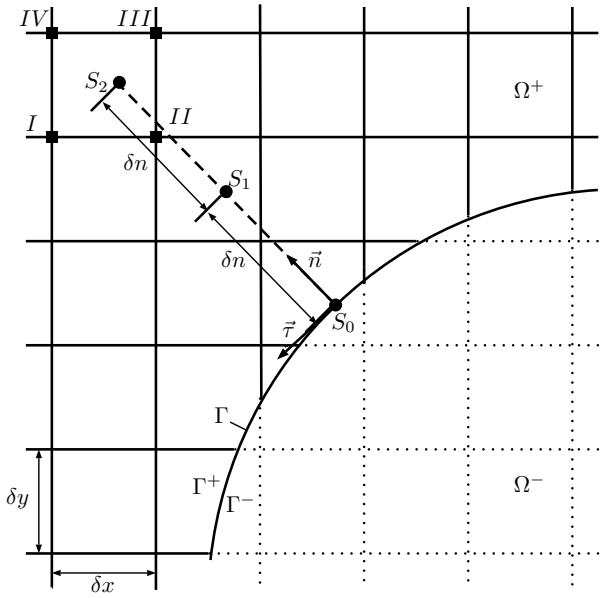


Figure 3.1: One-sided finite difference scheme

- $[\Delta \vec{u}]$

  For $[\Delta \vec{u}]$, it can be expressed using natural coordinates, which is given below,

$$[\Delta \vec{u}] = \left[\frac{\partial^2 \vec{u}}{\partial n^2}\right] + \kappa \left[\frac{\partial \vec{u}}{\partial n}\right] \tag{3.7}$$

  where κ is the curvature of the object boundary, $\left[\frac{\partial^2 \vec{u}}{\partial n^2}\right] = \frac{\partial^2 \vec{u}}{\partial n^2}|_{\Gamma^+} - \frac{\partial^2 \vec{u}}{\partial n^2}|_{\Gamma^-}$ and $\frac{\partial^2 \vec{u}}{\partial n^2}|_{\Gamma^-} = 0$.

For $\frac{\partial^2 \vec{u}}{\partial n^2}|_{\Gamma^+}$, similarly we can use one-sided finite difference scheme, which gives

$$\frac{\partial^2 \vec{u}}{\partial n^2}|_{\Gamma^+}= \frac{2\vec{u}\,(S_0) - 5\vec{u}\,(S_1) + 4\vec{u}\,(S_2) - \vec{u}\,(S_3)}{(\delta_n)^2} + O(\delta_n^2) \tag{3.8}$$

$\frac{\partial^2 \vec{u}}{\partial n^2}|_{\Gamma^+}$ and $\frac{\partial \vec{u}}{\partial n}|_{\Gamma^+}$ now are second order accurate. With the above derivations, we can achieve

$$[\Delta \vec{u}] = \frac{\partial^2 \vec{u}}{\partial n^2}|^+ + \kappa \left[\frac{\partial \vec{u}}{\partial n}\right] \tag{3.9}$$

which can be easy to calculate with information already known.

3.1.2. First-order Cartesian jump conditions

For the first-order Cartesian jump conditions, we need to calculate $\left[\frac{\partial \vec{u}}{\partial x}\right]$ and $\left[\frac{\partial \vec{u}}{\partial y}\right]$. As we know $[\vec{u}] = 0$, $\left[\frac{\partial \vec{u}}{\partial \tau}\right] = 0$, we can come to the two equations below

$$\left[\frac{\partial \vec{u}}{\partial \tau}\right] = \left[\frac{\partial \vec{u}}{\partial x}\right] \cdot \tau_x + \left[\frac{\partial \vec{u}}{\partial y}\right] \cdot \tau_y \tag{3.10a}$$

$$\left[\frac{\partial \vec{u}}{\partial n}\right] = \left[\frac{\partial \vec{u}}{\partial x}\right] \cdot n_x + \left[\frac{\partial \vec{u}}{\partial y}\right] \cdot n_y \tag{3.10b}$$

such that the following linear system can be built as

$$\begin{bmatrix} \tau_x & \tau_y \\ n_x & n_y \end{bmatrix} \begin{bmatrix} \left[\frac{\partial \vec{u}}{\partial x}\right] \\ \left[\frac{\partial \vec{u}}{\partial y}\right] \end{bmatrix} = \begin{bmatrix} 0 \\ \left[\frac{\partial \vec{u}}{\partial n}\right] \end{bmatrix}$$

Now we have the expression for the first-order Cartesian jump conditions

$$\left[\frac{\partial \vec{u}}{\partial x}\right] = \frac{-\tau_y}{\tau_x \cdot n_y - \tau_y \cdot n_x} \left[\frac{\partial \vec{u}}{\partial n}\right] \tag{3.11a}$$

$$\left[\frac{\partial \vec{u}}{\partial y}\right] = \frac{\tau_x}{\tau_x \cdot n_y - \tau_y \cdot n_x} \left[\frac{\partial \vec{u}}{\partial n}\right] \tag{3.11b}$$

3.1.3. Second-order Cartesian jump conditions

For the second-order Cartesian jump conditions, we need to calculate $\left[\frac{\partial^2 \vec{u}}{\partial x^2}\right]$, $\left[\frac{\partial^2 \vec{u}}{\partial y^2}\right]$ and $\left[\frac{\partial^2 \vec{u}}{\partial x \partial y}\right]$, and they are not as straightforward to calculate as the first-order Cartesian jump

conditions. Assume on any line segment panel with end points A and B as shown in Figure 3.2, we have

$$\frac{\partial}{\partial \tau}\left[\frac{\partial \vec{u}}{\partial x}\right]_A \approx \frac{1}{|AB|}\left(\left[\frac{\partial \vec{u}}{\partial x}\right]_B - \left[\frac{\partial \vec{u}}{\partial x}\right]_A\right) + O(|AB|),$$

and $\frac{\partial}{\partial \tau}\left[\frac{\partial \vec{u}}{\partial y}\right]$ can be approximated similarly. We can derive another linear system for second-order Cartesian jump conditions based on the equations below

$$[\Delta \vec{u}] = \left[\frac{\partial^2 \vec{u}}{\partial x^2}\right] + \left[\frac{\partial^2 \vec{u}}{\partial y^2}\right] \tag{3.12a}$$

$$\frac{\partial}{\partial \tau}\left[\frac{\partial \vec{u}}{\partial x}\right] = \left[\frac{\partial^2 \vec{u}}{\partial x^2}\right] \cdot \tau_x + \left[\frac{\partial^2 \vec{u}}{\partial x \partial y}\right] \cdot \tau_y \tag{3.12b}$$

$$\frac{\partial}{\partial \tau}\left[\frac{\partial \vec{u}}{\partial y}\right] = \left[\frac{\partial^2 \vec{u}}{\partial x \partial y}\right] \cdot \tau_x + \left[\frac{\partial^2 \vec{u}}{\partial y^2}\right] \cdot \tau_y \tag{3.12c}$$

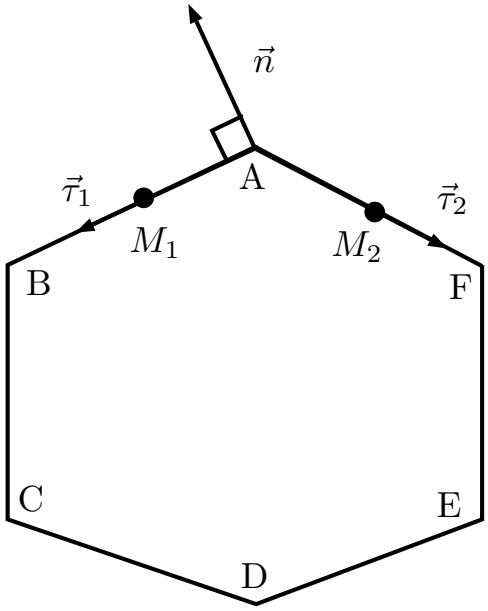


Figure 3.2: Representation of 2D interface in line segment panels

The system can be built as

$$\begin{bmatrix} 1 & 0 & 1 \\ \tau_x & \tau_y & 0 \\ 0 & \tau_x & \tau_y \end{bmatrix} \begin{bmatrix} \left[\frac{\partial^2 \vec{u}}{\partial x^2}\right] \\ \left[\frac{\partial^2 \vec{u}}{\partial x \partial y}\right] \\ \left[\frac{\partial^2 \vec{u}}{\partial y^2}\right] \end{bmatrix} = \begin{bmatrix} [\Delta \vec{u}] \\ \frac{\partial}{\partial \tau}\left[\frac{\partial \vec{u}}{\partial x}\right] \\ \frac{\partial}{\partial \tau}\left[\frac{\partial \vec{u}}{\partial y}\right] \end{bmatrix}$$

Based on this linear system we can derive

$$\left[\frac{\partial^2 \vec{u}}{\partial x^2}\right] = \frac{1}{\tau_x^2 + \tau_y^2}\left[\tau_x \frac{\partial}{\partial \tau}\left[\frac{\partial \vec{u}}{\partial x}\right] - \tau_y \frac{\partial}{\partial \tau}\left[\frac{\partial \vec{u}}{\partial y}\right] + \tau_y^2 \left[\Delta \vec{u}\right]\right] \tag{3.13a}$$

$$\left[\frac{\partial^2 \vec{u}}{\partial x \partial y}\right] = \frac{1}{\tau_x^2 + \tau_y^2}\left[\tau_y \frac{\partial}{\partial \tau}\left[\frac{\partial \vec{u}}{\partial x}\right] + \tau_x \frac{\partial}{\partial \tau}\left[\frac{\partial \vec{u}}{\partial y}\right] - \tau_x \tau_y \left[\Delta \vec{u}\right]\right] \tag{3.13b}$$

$$\left[\frac{\partial^2 \vec{u}}{\partial y^2}\right] = \frac{1}{\tau_x^2 + \tau_y^2}\left[\tau_y \frac{\partial}{\partial \tau}\left[\frac{\partial \vec{u}}{\partial y}\right] - \tau_x \frac{\partial}{\partial \tau}\left[\frac{\partial \vec{u}}{\partial x}\right] + \tau_x^2 \left[\Delta \vec{u}\right]\right] \tag{3.13c}$$

Now we have all the necessary formulas for principle and Cartesian jump conditions of velocity $\vec{u}$, and they can be easily coded into program.

3.2. Jump Conditions for pressure p

In this section, we will first show how to derive the Cartesian jump conditions of pressure derivatives, then we will present how to derive principle jump conditions.

3.2.1. First-order Cartesian jump conditions

For first-order Cartesian jump conditions $\left[\frac{\partial p}{\partial x}\right]$ and $\left[\frac{\partial p}{\partial y}\right]$, we can calculate $[\nabla p]$. As we know, ∇p term is in the Navier-Stokes equation,

$$\frac{\partial \vec{u}}{\partial t} + \nabla \cdot (\vec{u}\vec{u}) = -\nabla p + \frac{1}{Re}\Delta \vec{u} + \vec{q} + \int_\Gamma \vec{f}\left(\vec{X}, t\right) \delta\left(\vec{x} - \vec{X}\right) dl$$

Different from [76], where they compute $[\nabla p]$ by building a matrix problem, here we directly take jump conditions of the Navier-Stokes equations with the information already known as below,

$$[\vec{u}] = 0$$

$$\left[\frac{D\vec{u}}{Dt}\right] = \left[\frac{\partial \vec{u}}{\partial t}\right] + [\nabla \cdot (\vec{u}\vec{u})] = 0$$

$$\left[\vec{F}\right] = \left[\int_\Gamma \vec{f}\left(\vec{X}, t\right) \delta\left(\vec{x} - \vec{X}\right) dl\right] = 0$$

Then the Navier-Stokes equation can be reduced to

$$[\nabla p] = \frac{1}{Re}[\Delta \vec{u}] + [\vec{q}] \tag{3.15}$$

where $[\nabla p]$ is the first-order Cartesian jump condition, $[\Delta \vec{u}]$ is known from the previous section, $\vec{q}$ is piecewise defined and has jump conditions across the boundary [73]. As for body force $\vec{q}$, inside the boundary,

$$\vec{q}|_{\Gamma^-} = \ddot{\theta} \cdot \left(\vec{X} - \vec{x}_c \right) \tag{3.16}$$

where $\ddot{\theta}$ is the angular acceleration of object rotation. Outside boundary, $\vec{q}|_{\Gamma^+} = 0$. Thus the jump condition of $[\vec{q}]$ is

$$[\vec{q}] = -\ddot{\theta} \cdot \left(\vec{X} - \vec{x}_c \right) \tag{3.17}$$

Now we have the first-order Cartesian jump conditions as below

$$\left[\frac{\partial p}{\partial x} \right] = \frac{1}{Re} [\Delta u] - \ddot{\theta} \cdot (X - x_c) \tag{3.18a}$$

$$\left[\frac{\partial p}{\partial y} \right] = \frac{1}{Re} [\Delta v] - \ddot{\theta} \cdot (Y - y_c) \tag{3.18b}$$

3.2.2. Second-order Cartesian jump conditions

Similar as second-order Cartesian jump conditions for $\vec{u}$, we use the same strategy to calculate $\left[\frac{\partial^2 p}{\partial x^2} \right]$, $\left[\frac{\partial^2 p}{\partial y^2} \right]$ and $\left[\frac{\partial^2 p}{\partial x \partial y} \right]$. On panel AB, since $[\nabla p]_A$ and $[\nabla p]_B$ are already known, we can derive

$$\frac{\partial}{\partial \tau} [\nabla p]_A \approx \frac{[\nabla p]_B - [\nabla p]_A}{|AB|} + O(|AB|) \tag{3.19}$$

Build the linear system, we have

$$\begin{bmatrix} 1 & 0 & 1 \\ \tau_x & \tau_y & 0 \\ 0 & \tau_x & \tau_y \end{bmatrix} \begin{bmatrix} \left[\frac{\partial^2 p}{\partial x^2} \right] \\ \left[\frac{\partial^2 p}{\partial x \partial y} \right] \\ \left[\frac{\partial^2 p}{\partial y^2} \right] \end{bmatrix} = \begin{bmatrix} [\Delta p] \\ \frac{\partial}{\partial \tau} \left[\frac{\partial p}{\partial x} \right] \\ \frac{\partial}{\partial \tau} \left[\frac{\partial p}{\partial y} \right] \end{bmatrix}$$

The second-order Cartesian jump conditions for p can be derived as below

$$\left[\frac{\partial^2 p}{\partial x^2}\right] = \frac{1}{\tau_x^2+\tau_y^2}\left[\tau\frac{\partial}{\partial\tau}\left[\frac{\partial p}{\partial x}\right] - \tau_y\frac{\partial}{\partial\tau}\left[\frac{\partial p}{\partial y}\right] + \tau_y^2\left[\Delta p\right]\right] \tag{3.20a}$$

$$\left[\frac{\partial^2 p}{\partial x\partial y}\right] = \frac{1}{\tau_x^2+\tau_y^2}\left[\tau_y\frac{\partial}{\partial\tau}\left[\frac{\partial p}{\partial x}\right] + \tau_x\frac{\partial}{\partial\tau}\left[\frac{\partial p}{\partial y}\right] - \tau_x\tau_y\left[\Delta p\right]\right] \tag{3.20b}$$

$$\left[\frac{\partial^2 p}{\partial y^2}\right] = \frac{1}{\tau_x^2+\tau_y^2}\left[\tau_y\frac{\partial}{\partial\tau}\left[\frac{\partial p}{\partial y}\right] - \tau_x\frac{\partial}{\partial\tau}\left[\frac{\partial p}{\partial x}\right] + \tau_x^2\left[\Delta p\right]\right] \tag{3.20c}$$

3.2.3. Principle jump conditions

For principle jump conditions of pressure p, we need to calculate $[p]$, $\left[\frac{\partial p}{\partial n}\right]$ and $[\Delta p]$.

- $\left[\frac{\partial p}{\partial n}\right]$

  Since the first-order Cartesian jump condition is known, we have

$$\left[\frac{\partial p}{\partial n}\right] = [\nabla p]\cdot\vec{n} \tag{3.21}$$

- $[\Delta p]$

  By taking divergence of Navier-Stokes equation (2.1), we have the pressure Poisson equation

$$\Delta p = s_p + \nabla\cdot\left(\vec{q}+\vec{F}\right) \tag{3.22}$$

  where

$$s_p = -\left(\frac{\partial D}{\partial t} + \nabla\cdot(2\vec{u}D) - \frac{1}{Re}\Delta D\right) + 2\left(\frac{\partial u}{\partial x}\frac{\partial v}{\partial y} - \frac{\partial u}{\partial y}\frac{\partial v}{\partial x}\right)$$

$$D = \nabla\cdot\vec{u}$$

  The divergence free condition is better enforced here by including the terms with D. By taking jump conditions of equation (3.22), we have

$$[\Delta p] = 2\left[\frac{\partial u}{\partial x}\frac{\partial v}{\partial y}\right] - 2\left[\frac{\partial u}{\partial y}\frac{\partial v}{\partial x}\right] \tag{3.24}$$

- $[p]$

  For principle jump condition $[p]$, unlike the other principle jump conditions, it cannot be directly computed locally. Here we are using a different strategy. Since the first-order Cartesian jump condition $[\nabla p]$ is already known, then at any vertex on boundary,

we have

$$\left[\frac{\partial p}{\partial \tau_1}\right] = [\nabla p] \cdot \vec{\tau}_1 \tag{3.25a}$$

$$\left[\frac{\partial p}{\partial \tau_2}\right] = [\nabla p] \cdot \vec{\tau}_2 \tag{3.25b}$$

For example, on panel AB and AF as shown in Figure 3.2, Simpson's rule can be applied to achieve the equations below

$$\int_A^B \left[\frac{\partial p}{\partial \tau_1}\right] dl = [p]_B - [p]_A \approx \frac{|AB|}{6}\left(\left[\frac{\partial p}{\partial \tau_1}\right]_A + 4\left[\frac{\partial p}{\partial \tau_1}\right]_{M_1} + \left[\frac{\partial p}{\partial \tau_1}\right]_B\right) \tag{3.26a}$$

$$\int_A^F \left[\frac{\partial p}{\partial \tau_2}\right] dl = [p]_F - [p]_A \approx \frac{|AF|}{6}\left(\left[\frac{\partial p}{\partial \tau_2}\right]_A + 4\left[\frac{\partial p}{\partial \tau_2}\right]_{M_2} + \left[\frac{\partial p}{\partial \tau_2}\right]_F\right) \tag{3.26b}$$

where l is the length parameter along a line segment, M_1 is the center of panel AB and M_2 is the center of panel AF. $\left[\frac{\partial p}{\partial \tau_1}\right]_{M_1}$ and $\left[\frac{\partial p}{\partial \tau_2}\right]_{M_2}$ can be computed using the same strategy as we presented in previous sections. By adding equations (3.26a) and (3.26b) together, we have

$$[p]_B + [p]_F - 2[p]_A = rhs_A \tag{3.27}$$

where

$$\begin{aligned} rhs_A = &\frac{|AB|}{6}\left(\left[\frac{\partial p}{\partial \tau_1}\right]_A + 4\left[\frac{\partial p}{\partial \tau_1}\right]_{M_1} + \left[\frac{\partial p}{\partial \tau_1}\right]_B\right) \\ &+ \frac{|AF|}{6}\left(\left[\frac{\partial p}{\partial \tau_2}\right]_A + 4\left[\frac{\partial p}{\partial \tau_2}\right]_{M_2} + \left[\frac{\partial p}{\partial \tau_2}\right]_F\right) \end{aligned}$$

By repeating using the Simpson's rule for all vertices on the boundary, there will be

enough information to build the Topelitz matrix problem below:

$$\begin{bmatrix} -2 & 1 & 0 & \dots & 0 & 1 \\ 1 & -2 & 1 & \dots & 0 & 0 \\ 0 & 1 & -2 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & 1 & -2 & 1 \\ 1 & 0 & \dots & 0 & 1 & -2 \end{bmatrix} \begin{bmatrix} [p]_A \\ [p]_B \\ \vdots \\ [p]_F \end{bmatrix} = \begin{bmatrix} rhs_A \\ rhs_B \\ \vdots \\ rhs_F \end{bmatrix} \tag{3.28}$$

Notice that this matrix problem is singular and the solution is not unique. However, we are solving it for principle jump conditions of pressure p, that means

$$[p] = p|_{\Gamma^+} - p|_{\Gamma^-} \tag{3.29}$$

Inside the boundary, we have an analytical solution for pressure $p|_{\Gamma^-}$ of motion for rigid objects,

$$p|_{\Gamma^-} = -\frac{d^2x_c}{dt^2}X - \frac{d^2y_c}{dt^2}Y + \frac{1}{2}(\dot{\theta})^2\left((X-x_c)^2 + (Y-y_c)^2\right) + p_c \tag{3.30}$$

where p_c is an arbitrary constant. Then for the principle jump condition $[p]$, it is subject to an arbitrary constant too. In this way, we can safely assume that at point F, $[p]_F = 0$. Apply this result to matrix problem (3.28), we will have

$$\begin{bmatrix} -2 & 1 & \dots & 0 \\ 1 & -2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & \dots & 1 & -2 \end{bmatrix} \begin{bmatrix} [p]_A \\ [p]_B \\ \vdots \\ [p]_E \end{bmatrix} = \begin{bmatrix} rhs_A \\ rhs_B \\ \vdots \\ rhs_E \end{bmatrix} \tag{3.31}$$

Now this matrix problem is reduced to non-singular and can be solved easily by Gaussian elimination or other classical methods.

3.3. Fluid Force Calculation

As we know drag and lift coefficients are two important data for us to observe the behavior of any flow with objects. The previous approach for computing fluid force is based on singular force [77, 79], which cannot be applied here in our new method. In general, the fluid force $\vec{G}$ applied by a fluid to an object can be calculated by [79]

$$\vec{G} = \int_{\Gamma} \left(-p|_{\Gamma^+} \cdot \vec{n} + \frac{1}{Re} \left(\frac{\partial \vec{u}}{\partial n} \right) |_{\Gamma^+} \right) dl \tag{3.32}$$

We can find $p|_{\Gamma^+}$ easily by one-sided extrapolation. For $\left(\frac{\partial \vec{u}}{\partial n}\right) |_{\Gamma+}$, it can be computed by

$$\frac{\partial \vec{u}}{\partial n}|_{\Gamma^+} = \left[\frac{\partial \vec{u}}{\partial n} \right]_{\Gamma} + \frac{\partial \vec{u}}{\partial n}|_{\Gamma^-} \tag{3.33}$$

where

$$\left[\frac{\partial \vec{u}}{\partial n} \right] = \left[\frac{\partial \vec{u}}{\partial x} \right] \cdot n_x + \left[\frac{\partial \vec{u}}{\partial y} \right] \cdot n_y \tag{3.34a}$$

$$\frac{\partial \vec{u}}{\partial n}|_{\Gamma^-} = \dot{\theta} \cdot \tau \tag{3.34b}$$

$\left[\frac{\partial \vec{u}}{\partial x}\right]$ and $\left[\frac{\partial \vec{u}}{\partial y}\right]$ are already known as Cartesian jump conditions. Here l is the length parameter along the boundary Γ, then finally we can sum over the whole boundary using the Trapezoidal rule to find $\vec{G}$.

Chapter 4

IMPLEMENTATION OF THE IMMERSED INTERFACE METHOD

In this chapter, we will talk about the implementation of the new method. Even though we have established a sound theory for our method, there are still a lot of details need to be taken care of in the implementation. In the first section, we will talk about the spatial discretization. In the second section, we will present the temporal discretization.

4.1. Spatial Discretization

4.1.1. MAC/Staggered grid

In the implementation, a staggered Marker-And-Cell(MAC) method is used for spatial discretization. This method was developed by Francis Harlow and details can be found in [67]. In the staggered mesh, the computational domain is uniformly divided into square cells and the pressure is defined in each center of the cell, as shown in Figure 4.1. Velocity u is defined at the center of vertical edges of the cell and velocity v is defined at the center of horizontal edges of the cell. Even though it will be more complicated and takes more work to code by using a MAC grid, compared with using a collocated grid, using a MAC grid can help to improve the accuracy, can apply different boundary conditions easily, and it will be easier to couple the velocity to solve the pressure Poisson equation. In the left graph of Figure 4.1, the black disks where $m-1$, m and $m+1$ marked are vertices of the boundary, and the white open circles are intersection points of grid line and object boundary. When solving u, v and p, we cannot directly use the jump conditions at the vertices. The interpolation of jump conditions from the vertices to the intersection points is necessary. With the jump contributions on intersection points known, we can use the central finite difference schemes with the incorporation of jump conditions as we have discussed in chapter 2. In the 2D case,

the finite difference schemes are shown as below

$$\delta_x(\cdot)_{i,j} = \frac{(\cdot)_{i+\frac{1}{2},j} - (\cdot)_{i-\frac{1}{2},j}}{\Delta x} + c_x(\cdot)_{i,j} \tag{4.1a}$$

$$\delta_y(\cdot)_{i,j} = \frac{(\cdot)_{i,j+\frac{1}{2}} - (\cdot)_{i,j-\frac{1}{2}}}{\Delta y} + c_y(\cdot)_{i,j} \tag{4.1b}$$

$$\delta_{xx}(\cdot)_{i,j} = \frac{(\cdot)_{i+1,j} - 2(\cdot)_{i,j} + (\cdot)_{i-1,j}}{\Delta x^2} + c_{xx}(\cdot)_{i,j} \tag{4.1c}$$

$$\delta_{yy}(\cdot)_{i,j} = \frac{(\cdot)_{i,j+1} - 2(\cdot)_{i,j} + (\cdot)_{i,j-1}}{\Delta y^2} + c_{yy}(\cdot)_{i,j} \tag{4.1d}$$

Δx, Δy are spatial discretization steps, and c_x, c_y, c_{xx} and c_{yy} are jump contributions calculated from the jump conditions of the intersection points. If the grid line does not cross any boundary or panels of the object, then the jump contributions will be zero and a scheme will be a regular central finite difference scheme.

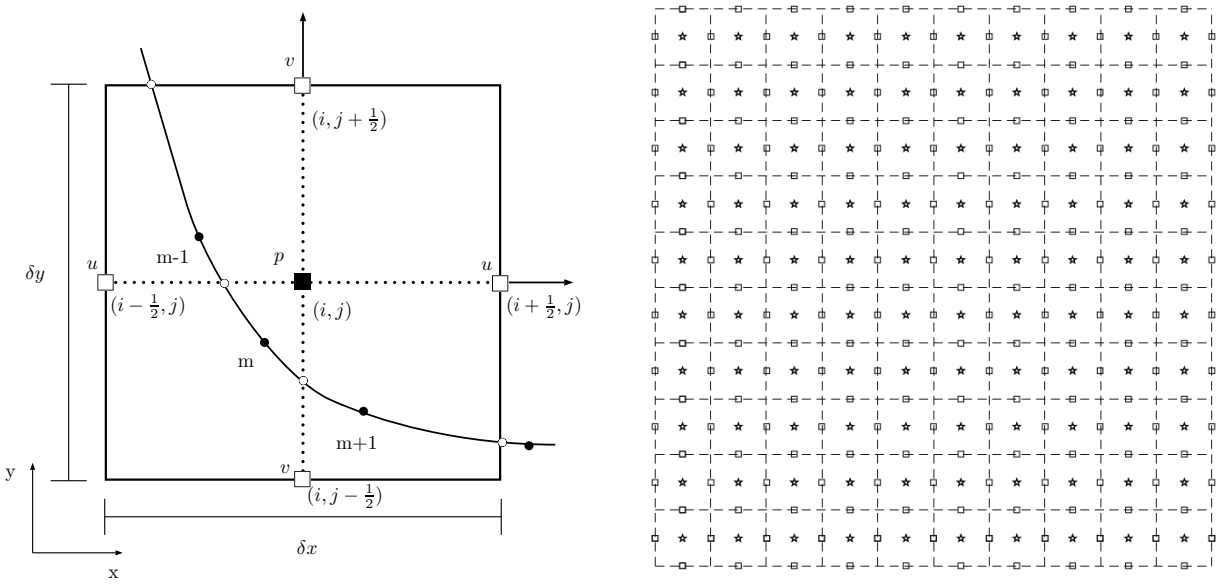


Figure 4.1: Velocity and pressure on a MAC grid

4.1.2. Object interface representation

As we mentioned earlier, currently we are using line segment panels to represent the interface of the objects. In the previous work of Xu and other researchers [33, 77, 79], surface

parametrization was used to form the object interface and periodic cubic spline was used to calculate the Lagrangian point coordinates. Since now we are going to solve problems for objects with complex or non-smooth boundaries, the previous method is no longer suitable. In our method, the information of the normal vector $\vec{n}$ and tangential vector $\vec{\tau}$ are frequently used. Under the new representation of the interface, we simply compute $\vec{\tau}$ and $\vec{n}$ based on the coordinates of vertices by the equations below

$$\vec{\tau} = (\tau_x, \tau_y) = \frac{\Delta \vec{X}}{||\Delta \vec{X}||_2}, \tag{4.2a}$$

$$\vec{n} = (n_x, n_y) = (\tau_y, -\tau_x)\,. \tag{4.2b}$$

where $\vec{X} = (X, Y)$ is the vertex of the object interface and normal vector $\vec{n}$ is computed by rotation of tangential vector $\vec{\tau}$. Then in implementation we have

$$\tau_x(m) = \frac{X_{m+1} - X_m}{\gamma} \tag{4.3a}$$

$$\tau_y(m) = \frac{Y_{m+1} - Y_m}{\gamma} \tag{4.3b}$$

$$\gamma = \sqrt{(X_{m+1} - X_m)^2 + (Y_{m+1} - Y_m)^2} \tag{4.3c}$$

$$n_x(m) = -\tau_x(m) \tag{4.3d}$$

$$n_y(m) = \tau_y(m) \tag{4.3e}$$

where m is the index of vertices. In this way, the boundary vertices can be set up based on the shape of objects. The vertices don't need to be equally distributed and we can use as many vertices as we want. The program has better robustness since it only needs to import vertices and curvature data, which can be setup using other software easily. In the calculation of $\vec{\tau}$, we will only do it in a counter-clockwise direction. When we calculate jump conditions, we actually calculate $\vec{n}$ in both the clockwise and counter-clockwise directions, and use the different $\vec{n}$ to compute jump conditions at the same vertex twice, then take the average to improve the accuracy and symmetry.

4.1.3. Pressure Poisson solver

Taking the divergence of the Navier-Stokes equation (2.1), we have the pressure Poisson equation as below

$$\Delta p = -\left(\frac{\partial D}{\partial t} + \nabla \cdot (2\vec{u}D) - \frac{1}{Re}\Delta D\right) + 2\left(\frac{\partial u}{\partial x}\frac{\partial v}{\partial y} - \frac{\partial u}{\partial y}\frac{\partial v}{\partial x}\right) + \nabla \cdot \left(\vec{q} + \vec{F}\right) \tag{4.4}$$

where $D = \nabla \cdot \vec{u}$. As we are using finite difference method, the pressure Poisson equation can be discretized as

$$(\triangle p)_{i,j} = \frac{p_{i-1,j} - 2p_{ij} + p_{i+1,j}}{\delta x^2} + \frac{p_{i,j-1} - 2p_{i,j} + p_{i,j+1}}{\delta y^2} + c_{ij} \tag{4.5}$$

where p_{ij} is the pressure at the center grid point (i, j), and c_{ij} is the jump contribution due to the incorporation of necessary jump conditions. Let L denote the Laplacian operator, $\underline{f}$ denote the vector formed by the discrete right hand side of equation (4.4), $\underline{c}$ denote the vector for the jump contribution, then

$$L\underline{p} + \underline{c} = \underline{f} \tag{4.6}$$

We have developed two different serial solvers to solve the pressure Poisson equation, which are a FFT solver and a Helmholtz iterative solver. We will describe our parallel solver later in the next chapter.

- FFT Solver

  In the previous work, the principle jump conditions of pressure p are assumed known, as we presented in the second chapter. Then the jump contribution $\underline{c}$ is independent of p. In this way, the FFT solver can be used to solve the pressure Poisson problem easily. FFT stands for the fast Fourier transformation and this algorithm is very powerful at reducing computational time.

- Helmholtz Iterative Solver

  Even though the FFT solver is a very powerful tool, it may have trouble when $\underline{c}$ is dependent on p. In other words, principle jump conditions cannot be derived in a simple and straightforward manner, for example in the two fluid problem the density

is discontinuous and $[p]$ cannot be calculated. In order to conquer this issue, we have developed the Helmholtz iterative solver. $[p]$ can be calculated by knowing $p|_{\Gamma^+}$ and $p|_{\Gamma^-}$. There exists an analytical solution for $p|_{\Gamma^-}$ with an arbitrary constant, and $p|_{\Gamma^+}$ can be calculated using one-sided extrapolation. Then we can find that the jump contribution $\underline{c}$ is linearly dependent on $\underline{p}$, and it can be written as $\underline{c} = C\underline{p} + \underline{c_0}$. Then equation (4.6) becomes

$$L\underline{p} + C\underline{p} = \underline{f} - \underline{c_0} \tag{4.7}$$

Now let's introduce the pseudo-time,

$$\frac{\partial \underline{p}}{\partial t} = L\underline{p} + \left(C\underline{p} + \underline{c_0}\right) - \underline{f} \tag{4.8}$$

As long as $\frac{\partial p}{\partial t}$ is 0, it will reach a steady solution, which is independent of the initial condition. Now we can introduce θ method at the same time and results in the following form

$$\gamma\left(\underline{p}^{n+1} - \underline{p}^{n}\right) = \theta L\underline{p}^{n+1} + (1-\theta)\, L\underline{p}^{n} + \left(C\underline{p}^{n} + \underline{c_0}\right) - \underline{f} \tag{4.9}$$

where γ is the reciprocal of the pseudo-time step Δt, and $\underline{p}^n$ is the grid pressure at the time $n\Delta t$. It can be rearranged as

$$(\theta L - \gamma I)\,\underline{p}^{n+1} = -\left((1-\theta)\, L + C + \gamma I\right)\underline{p}^{n} - \underline{c_0} + \underline{f} \tag{4.10}$$

In implementation, $\theta = 1$, then

$$(L - \gamma I)\,\underline{p}^{n+1} = \left(C\underline{p}^{n} + c_0\right) + \underline{f} - \gamma\underline{p}^{n} \tag{4.11}$$

By solving the above equation iteratively, p will finally reach to a steady point and then we have solved the problem. One thing to point out here is, because it is an iterative solver, the computational time will increase and it is much slower than using FFT solver. So if the principle jump condition is known, the FFT solver will be a better choice.

4.1.4. Boundary conditions

There are mainly three different types of boundary conditions used in our method for the far field boundaries of the computational domain, they are periodic boundary conditions, Dirichlet boundary conditions and Neumann boundary conditions.

- Periodic boundary condition

  Periodic boundary conditions are commonly seen in models with repeating nature in physical geometry and expected flow pattern.

  At west and east of the boundary: $\vec{u}_{0,j} = \vec{u}_{nx-1,j}$, $\vec{u}_{nx,j} = \vec{u}_{1,j}$

  At north and south of the boundary: $\vec{u}_{i,ny+1} = \vec{u}_{i,2}$, $\vec{u}_{i,0} = \vec{u}_{i,ny-1}$

  n_x and n_y are number of grid points along x and y directions.

- Dirichlet boundary condition

  Dirichlet boundary conditions are frequently used for models with fixed value on velocity or pressure at boundary, like lid-driven cavity flow or circular Couette flow, where the Dirichlet boundary conditions is applied for velocity. Since the actual physical domain is formed by the center lines of the grid cell, when we enforce the Dirichlet boundary condition, we actually force the average of u and v at boundary as what we want. For example, if we want to enforce $u = u_0$ on west physical boundary of the domain, we need set $\frac{u_{0,j}+u_{1,j}}{2} = u_0$. In implementation it will be $u_{0,j} = 2 * u_0 - u_{1,j}$, since $u_{1,j}$ will be calculated during the whole process but not $u_{0,j}$.

- Neumann boundary condition

  Neumann boundary conditions are often used for models with the normal gradient of a variable a constant at boundary. In implementation, $\frac{\partial p}{\partial n}$ is often derived from the Navier-Stokes equation. For example, on the west or east physical boundary, the Neumann boundary condition for pressure will be

  $$\frac{\partial p}{\partial x} = \frac{1}{Re}\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) - u\frac{\partial u}{\partial x} - v\frac{\partial u}{\partial y} \tag{4.12}$$

  For velocity, the setup is similar to a Dirichlet boundary. If we need to set at the west boundary $\frac{\partial u}{\partial x} = c$, then in implementation it will be $u_{0,j} = u_{1,j} - \Delta x \cdot c$.

- Boundary conditions for objects

  Besides the boundary conditions we have mentioned above, there is another condition we need to enforce, which is conditions on the objects. As explained, even though we treat the domain inside the objects as fluid in our method, we still use this body force to enforce the rigid motion of the objects. So for the fluid inside the objects, the divergence free condition and rigid motion still need to be applied. Here we used the equations below to enforce these conditions

$$D = \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + D_c = 0 \tag{4.13a}$$

$$u = \dot{x_c} - \dot{\theta} \cdot (Y - y_c) \tag{4.13b}$$

$$v = \dot{y_c} + \dot{\theta} \cdot (X - x_c) \tag{4.13c}$$

  where D_c is the jump contribution to the divergence inside the object. Without the above equations, the accuracy of the method cannot be guaranteed.

4.2. Temporal discretization

There are many different methods for temporal discretization, such as Runge-Kutta method, Crank-Nicolson method, Warming-Beam method, etc. Some methods are explicit, some are implicit, and some are mixed. Every method has its advantage and disadvantage. Here we used explicit Runge-Kutta method for temporal discretization because it is easy to implement and has very good accuracy and stability. The temporal discretization on the momentum equation is as follows

$$\frac{\mathbf{u}^{n+1} - \mathbf{u}^n}{\Delta t} = -\nabla \cdot (\mathbf{u}^n \mathbf{u}^n) - \nabla p^n + \frac{1}{Re} \Delta \mathbf{u}^n + \mathbf{q}^n + \mathbf{F}^n \tag{4.14}$$

On the right hand side of above equation, all the variables $\mathbf{u}, p, \mathbf{q}, \mathbf{F}$ are at time step n, and on the left hand side we have $\mathbf{u}$ at time step $n+1$ and n.

4.2.1. Runge-Kutta method

In our method, we used both third and fourth order Runge-Kutta method, which accordingly have third and fourth order of accuracy for temporal discretization. Runge-Kutta method is easy to code, has good accuracy and stability. The general form for an explicit Runge-Kutta method with s stages is as following

$$Y_i = y_n + k\sum_{j=1}^{i-1} a_{ij} f(t_n + c_j k, Y_j), \quad 1 \le i \le s \tag{4.15a}$$

$$y_{n+1} = y_n + k\sum_{i=1}^{s} b_i f(t_n + c_i k, Y_i) \tag{4.15b}$$

where Y_i is the intermediate approximations to the solution at time $t_n + c_i k$, and $k = t_{n+1} - t_n$. $c_i, a_{i,j}, b_i$ are the coefficients of the Runge-Kutta scheme. In our method, if we consider the momentum equation and prescribed movement of the objects, the equations can be written as

$$\frac{\partial \mathbf{u}}{\partial t} = \mathbf{Y}(\mathbf{u}, \mathbf{X}, p) \tag{4.16a}$$

$$\frac{\partial \mathbf{X}}{\partial t} = U(\mathbf{X}) \tag{4.16b}$$

$$\mathbf{Y}(\mathbf{u}, \mathbf{X}, p) = -\nabla \cdot (\mathbf{uu}) - \nabla p + \frac{1}{Re}\Delta \mathbf{u} + \mathbf{q} + \mathbf{F} \tag{4.16c}$$

For the third order Cartesian jump conditions, the scheme can be derived as follows

$$\mathbf{u}_1 = \mathbf{u}^n, \quad \mathbf{X}_1 = \mathbf{X}^n \tag{4.17a}$$

$$\mathbf{u}_2 = \mathbf{u}^n + \frac{\Delta t}{2}\mathbf{Y}(\mathbf{u}_1, \mathbf{X}_1, p_1), \quad \mathbf{X}_2 = \mathbf{X}^n + \frac{\Delta t}{2}U(\mathbf{X}_1) \tag{4.17b}$$

$$\mathbf{u}_3 = \mathbf{u}^n + \Delta t\mathbf{Y}(\mathbf{u}_2, \mathbf{X}_2, p_2), \quad \mathbf{X}_3 = \mathbf{X}^n + \Delta t U(\mathbf{X}_2) \tag{4.17c}$$

$$\mathbf{u}^{n+1} = \mathbf{u}^n + \frac{\Delta t}{6}(\mathbf{Y}(\mathbf{u}_1, \mathbf{X}_1, p_1) + 4\mathbf{Y}(\mathbf{u}_2, \mathbf{X}_2, p_2) + \mathbf{Y}(\mathbf{u}_3, \mathbf{X}_3, p_3)) \tag{4.17d}$$

$$\mathbf{X}^{n+1} = \mathbf{X}^n + \frac{\Delta t}{6}(U(\mathbf{X}_1) + 4U(\mathbf{X}_2) + U(\mathbf{X}_3)) \tag{4.17e}$$

Similarly for the fourth order Runge-Kutta method, the scheme is as following

$$\mathbf{u}_1 = \mathbf{u}^n, \quad \mathbf{X}_1 = \mathbf{X}^n \tag{4.18a}$$

$$\mathbf{u}_2 = \mathbf{u}^n + \frac{\Delta t}{2}\mathbf{Y}(\mathbf{u}_1, \mathbf{X}_1, p_1), \quad \mathbf{X}_2 = \mathbf{X}^n + \frac{\Delta t}{2}U(\mathbf{X}_1) \tag{4.18b}$$

$$\mathbf{u}_3 = \mathbf{u}^n + \frac{\Delta t}{2}\mathbf{Y}(\mathbf{u}_2, \mathbf{X}_2, p_2), \quad \mathbf{X}_3 = \mathbf{X}^n + \frac{\Delta t}{2}U(\mathbf{X}_2) \tag{4.18c}$$

$$\mathbf{u}_4 = \mathbf{u}^n + \Delta t\mathbf{Y}(\mathbf{u}_3, \mathbf{X}_3, p_3), \quad \mathbf{X}_4 = \mathbf{X}^n + \Delta t U(\mathbf{X}_3) \tag{4.18d}$$

$$\mathbf{u}^{n+1} = \mathbf{u}^n + \frac{\Delta t}{6}(\mathbf{Y}(\mathbf{u}_1, \mathbf{X}_1, p_1) + 2\mathbf{Y}(\mathbf{u}_2, \mathbf{X}_2, p_2) + 2\mathbf{Y}(\mathbf{u}_3, \mathbf{X}_3, p_3) + \mathbf{Y}(\mathbf{u}_4, \mathbf{X}_4, p_4)) \tag{4.18e}$$

$$\mathbf{X}^{n+1} = \mathbf{X}^n + \frac{\Delta t}{6}(U(\mathbf{X}_1) + 2U(\mathbf{X}_2) + 2U(\mathbf{X}_3) + U(\mathbf{X}_4)) \tag{4.18f}$$

In tests, since there is no big difference on the accuracy between RK3 and RK4, RK3 is mainly used because it takes less computational time.

4.2.2. CFL number

In order to guarantee the stability of the temporal discretization, we must be very careful on the setup of time step Δt. Considering the Reynolds number Re and Courant-Friedrichs-Lewy(CFL) condition, we cannot just set Δt always as a constant. Δt can be different with changes of the flow field. Here we have introduced three ways to compute Δt. The first one is our choice of time difference Δt_0 that $\Delta t = \Delta t_0$. The second Δt is calculated by the restriction of viscous term [46], given as

$$\Delta t_v = \frac{CFL_v \cdot Re}{\left(\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2}\right)} \tag{4.19}$$

The third Δt is calculated by the restriction of convective term [24], given as

$$\Delta t_c = \frac{CFL_c}{\frac{u_{max}}{\Delta x} + \frac{v_{max}}{\Delta y}} \tag{4.20}$$

u_{max} and v_{max} are the maximum velocity in the flow field. CFL_v and CFL_c are constants for the control of Δt. In the last, we compare all the Δt we have and use the smallest to make sure of stability.

$$\Delta t = min(\Delta t_0, \Delta t_c, \Delta t_v) \tag{4.21}$$

4.3. Method Summary

The main procedure of the current method can be summarized as below:

1. Initialize the flow field and interfaces

   At the beginning of the program, we need to initialize the computational geometry, flow field and object boundaries. The vertices and curvature data can be setup using other software like Matlab and imported into the program.

2. Calculate surface properties and geometric quantities

   In this step we need calculate the $\vec{n}$, $\vec{\tau}$ and decide whether the grid points are inside or outside of the object boundaries. The program will also calculate the intersection points where the object interface crosses the MAC grid lines.

3. Calculate principle and Cartesian jump conditions on boundary vertices

   In this step we follow the formulas derived in Chapter 3 to compute all the necessary jump conditions .

4. Incorporate jump conditions with finite difference scheme

   As principle and Cartesian jump conditions known, the jump contributions can be calculated and added to the finite difference scheme.

5. Solve the pressure field using FFT solver or Helmholtz iterative solver

   In this step right hand side of pressure Poisson equation needs to be calculated with the jump contributions, then the pressure filed can be solved using the solvers.

6. Update the velocity field by using Runge Kutta method for time marching

   Since pressure is known, we can update velocities using the momentum equation. If the object is in motion, then the object boundary coordinates, velocity, rotation angle and their derivatives respect to time will be calculated.

7. Go back to step 2 until finished

   If the simulation comes to the end time or other criteria is reached, the program stops and outputs all the data. If not, it goes back to step 2 and repeats the process above.

Chapter 5

PARALLELIZATION OF THE IMMERSED INTERFACE METHOD

In this chapter, we will present the main idea of parallelization for the immersed interface method. Even though we successfully extended our method for objects with non-smooth boundaries, the method will still be restricted by our serial program if the problem is very large or complicated. Driven by the downsides, we started the development of parallelization for the method, which has lots of improvements from our serial program. In the first section, we will present a brief introduction to the techniques we are using for the parallelization. In the second section, we will present the data structure. The design for communications is given in third section and stretched mesh is given in fourth section. We will talk about SMG Poisson solver in fifth section and the method to compute jump contributions in sixth section.

5.1. Introduction of Parallel Computing

In this section we will first give a short introduction to the domain decomposition of our method. Then parallel library *MPI* is introduced.

5.1.1. Domain decomposition

Before the introduction about domain decomposition, let's see a simple problem. Assume we have matrix A and vectors $\mathbf{x}$ and $\mathbf{b}$, the size of $\mathbf{x}$ and $\mathbf{b}$ is n, the size of A is $n \times n$, n is an even integer. We want to compute the residual vector $\mathbf{r}$ where $\mathbf{r} = \mathbf{b} - A\mathbf{x}$ shown as below,

$$\begin{bmatrix} \\ r_1 \\ \\ r_2 \\ \\ \vdots \\ \\ r_n \\ \\ \end{bmatrix} = \begin{bmatrix} -2 & 1 & \dots & 0 \\ \\ 1 & -2 & \dots & 0 \\ \\ \vdots & \vdots & \ddots & \vdots \\ \\ 0 & \dots & -2 & 1 \\ \\ 0 & \dots & 1 & -2 \end{bmatrix} \begin{bmatrix} \\ x_1 \\ \\ x_2 \\ \\ \vdots \\ \\ x_n \\ \\ \end{bmatrix} - \begin{bmatrix} \\ b_1 \\ \\ b_2 \\ \\ \vdots \\ \\ b_n \\ \\ \end{bmatrix}$$

As we can see from the above matrix, $r_i = (x_{i-1} - 2x_i + x_{i+1}) - b_i$. In the program, we don't even need to store matrix A. Assume we have two processors now, for processor $p1$ it only stores about half of the vector $\mathbf{x}$ and $\mathbf{b}$, which is $x[1 : \frac{n}{2} + 1]$ and $b[1 : \frac{n}{2} + 1]$, and processor $p2$ stores $x[\frac{n}{2} : n]$ and $b[\frac{n}{2} + 1 : n]$. Then for $p1$, it can compute $r[1 : \frac{n}{2}]$ and $p2$ can compute $r[\frac{n}{2} + 1 : n]$. At the end we can collect $r[1 : \frac{n}{2}]$ and $r[\frac{n}{2} + 1 : n]$ and put it together to get vector $\mathbf{r}$. If we ignore the communication between processors, then in theory the computational time will be reduced by half since each processor only computes half size of the vector $\mathbf{r}$ and both starts to compute at the same time. With more processors, the computational time can be reduced more.

In the area of parallel/high performance computing, one common idea is to decompose the computational domain into several sub-domains and distribute them to several processors, just as we did in the last paragraph and shown in Figure 5.1. During the computation, most of the processors will only store part of the information(blue and red dots as in Figure 5.1) from the whole computational domain, run calculation for local parts(blue dots), and exchange the local results with neighboring processors to get information (red dots) if necessary. By doing this, we can distribute a very large problem to as many processors as we want, and computational time can be largely saved if the parallel strategy is well designed.

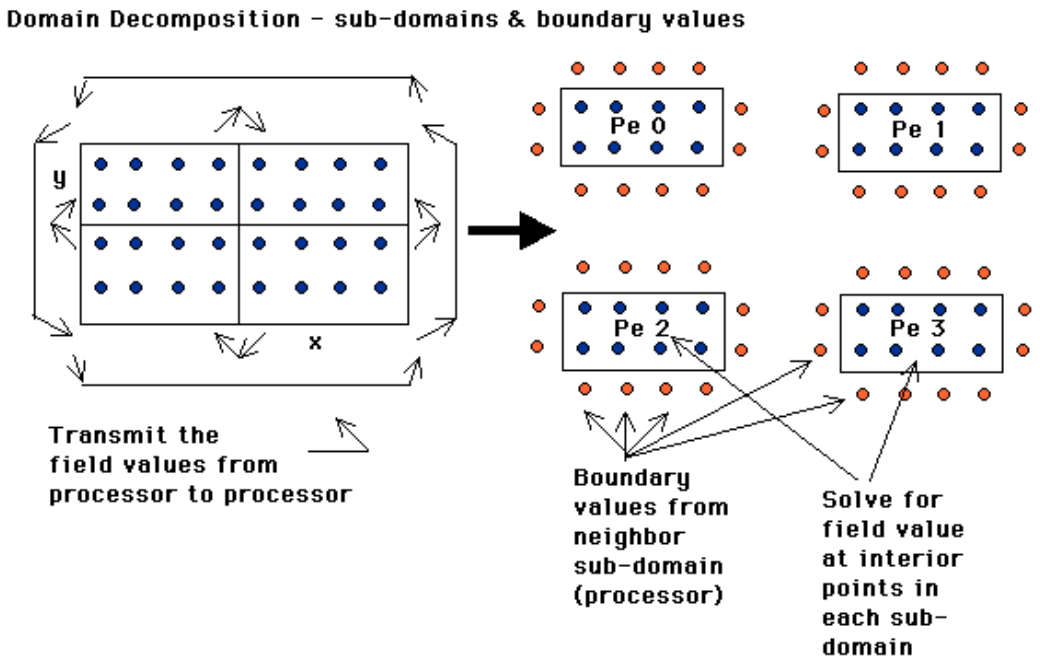


Figure 5.1: Domain decomposition. Source: http://physics.drexel.edu/

5.1.2. Message Passing Interface(*MPI*)

One important part need to noticed in parallel computing is the information exchange. Let's take a look again at the matrix problem we mentioned above, we have

$$r_i = x_{i-1} - 2x_i + x_{i+1} - b_i$$

Assume the size of vector $\mathbf{x}$ and $\mathbf{r}$ is n, and on processor $p1$, i is from 1 to $\frac{n}{2}$, and on $p2$, i is from $\frac{n}{2}+1$ to n. Then on the first processor, x_{i+1} is out of boundary when $i = \frac{n}{2}$ and on second processor x_{i-1} is out of boundary when $i = \frac{n}{2}+1$. Then information exchange for $x_{\frac{n}{2}}$ and $x_{\frac{n}{2}+1}$ is necessary between two processors so this computation can be finished correctly. With the help of Message Passing Interface(*MPI*) package, the information exchange can be performed fast and robust.

5.2. Data Structure

In our serial program, there are mainly four groups of variables stored in memory. The first group is the parameters used in the program, like Reynolds number, number of time steps, number of grids used, etc. The second group is the variables for flow field such as velocity, pressure and divergence. The third group is the jump conditions of vertices on the object boundary and intersections where object boundary crosses grid lines. The last group is the jump contributions to flow field, which are located on grid points. In our serial program, one obvious downside is that the size of all the variables are fixed. With finer resolution and more objects, both the computational cost and memory use can be very expensive. Now let's talk about the details of how we handle these variables in the parallel program.

5.2.1. Parameters

For the parameters, they are also recognized as global parameters and local parameters. Global parameters are common and identical in all processors, such as Reynolds number, number of time steps, setups for *MPI* library, etc. These are are stored in each processor. For local parameters, the value can be different in each processor, such as the beginning and ending indices for local subdomain in x and y directions. In our parallel program, we have parameters $ipbeg, ipend, jpbeg, jpend$,$iubeg, iuend, jubeg, juend$,$ivbeg, ivend, jvbeg, jvend$ to store store the beginning and ending indices for velocity and pressure field. We also have parameters like $nobj4proc$(number of objects in local processor), which are used to store local information about objects.

5.2.2. Flow field variables

In order to save memory, as shown in Figure 5.1, we divide the whole computational domain into several subdomains. Each subdomain only stores parts of the flow field with some extra layers prepared for information exchange, which we call ghost layers. In this way, the size of local flow field can be adjustable based on how many processors we used on each direction, and then local velocity and pressure can be allocated and we don't need to store

the whole flow field. In the parallel program, it will read how many grid points are set in total, and divides them by the number of processors on x & y directions, then we will know the actual size of local flow field variables and allocate them. One thing to notice here is, as shown in Figure 5.1, ghost layers are also necessary for the purpose of information exchange, so the actual local flow field variables are a little bigger. The code is shown below,

```
ALLOCATE(u(iubeg-nxghost:iuend+nxghost,jubeg-nyghost:juend+nyghost))
ALLOCATE(v(ivbeg-nxghost:ivend+nxghost,jvbeg-nyghost:jvend+nyghost))
ALLOCATE(p(ipbeg-nxghost:ipend+nxghost,jpbeg-nxghost:jpend+nxghost))
```

5.2.3. Jump conditions

As for jump condition variables, the decomposition will be different. For any object in the flow field, it has the possibility to stay in only one subdomain or can cross several subdomains, which is based on the shape of the object and its movement. What we did here is at each time step, if the object is moving, the program will keep asking each processor to detect which objects are in it and calculate the total number of vertices for these objects. At the same time, the program will also compute the number of intersection points. Then the jump conditions for vertices and intersections can be allocated dynamically. If the objects are moving, these variables will be allocated and freed at each time step to improve the memory use efficiency and the accuracy of the program can be guaranteed. For example, we have variables ujc, vjc and pjc to store principle and Cartesian jump conditions of velocity and pressure for vertices, which is shown as below,

```
ALLOCATE(ujc(1:6, 1:nvertex4proc(nobj4proc)))
ALLOCATE(vjc(1:6, 1:nvertex4proc(nobj4proc)))
ALLOCATE(pjc(1:8, 1:nvertex4proc(nobj4proc)))
```

where $nvertex4proc(nobj4proc)$ is the total number of vertices on the current processor. Similarly, we have variables for principle and Cartesian jump conditions of velocity and pressure on intersection points, where object boundary crosses center line or edge line of MAC grid, shown as below,

```
ALLOCATE(ujcxf(1:6,1:n_xf_int))
ALLOCATE(vjcxf(1:6,1:n_xf_int))
ALLOCATE(vjcxc(1:6,1:n_xc_int))
ALLOCATE(ujcxc(1:6,1:n_xc_int))
ALLOCATE(pjcxc(1:8,1:n_xc_int))
ALLOCATE(ujcyf(1:6,1:n_yf_int))
ALLOCATE(vjcyf(1:6,1:n_yf_int))
ALLOCATE(vjcyc(1:6,1:n_yc_int))
ALLOCATE(ujcyc(1:6,1:n_yc_int))
ALLOCATE(pjcyc(1:8,1:n_yc_int))
```

x_c and y_c are the center lines and x_f and y_f are the edge lines, n_xf_int is the number of intersection points on xf grid lines and similar for others.

5.2.4. Jump contributions

In our serial program, the variables for jump contributions are allocated in the beginning and the size is not adjustable. In our parallel program, it no longer stores any information of jump contributions. Since jump contributions are calculated from jump conditions on intersection points, as shown in equations (2.5a) and (2.5b), and they are located on MAC grid points, we can directly calculate jump contributions and add them to the grid points without taking extra memory.

5.3. Information Exchange/Communication

One big advantage of our method, when comes to parallelization, is that most of the computation can be performed locally, which means we can save time on information exchange. As we already know from chapter 3, most of the jump conditions can be calculated based on velocity and pressure stored on local processor. In our parallel program, there are mainly five parts of information exchange.

5.3.1. Ghost layers of flow field

First is the exchange of ghost layers of flow field variables u, v and p. As mentioned in section 5.1, since we are using a finite difference scheme. On ghost layers variables will not be

computed, but the information must be available for the calculation of internal grid points. This communication between processors can be performed using function *MPI_SendRecv* from *MPI* library very easily and efficiently. Let's see velocity u for example, which is shown in Figure 5.2. Assume the the total size of the computational domain is $n_x \times n_y$. We have 4 processors in total and 2 processors on each direction. As mentioned in last section, the local size for u will be $(iubeg - nxghost : iuend + nxghost, jubeg - nyghost : juend + nyghost)$, which is dependent on the total grid number and processors. For example in Figure 5.2, we assume $n_x = n_y = 10$ and $nxghost = nyghost = 2$. Since our current program is developed for 2D case, so we only need to think about information exchange with left/right/front/back neighbors. The code is shown as below

```
! x-direction
CALL MPI_SENDRECV(u(iuend-nxghost,jubeg-nyghost),1,uitype,right,0,
                  u(iubeg-nxghost+1,jubeg-nyghost),1,uitype,left,0,
                  comm2d,status,ierr)
CALL MPI_SENDRECV(u(iubeg+1,jubeg-nyghost),1,uitype,left,0,
                  u(iuend,jubeg-nyghost),1,uitype,right,0,
                  comm2d,status,ierr)
! y-direction
CALL MPI_SENDRECV(u(iubeg-nxghost,juend-nyghost),1,ujtype,back,0,
                  u(iubeg-nxghost,jubeg-nyghost+1),1,ujtype,front,0,
                  comm2d,status,ierr)
CALL MPI_SENDRECV(u(iubeg-nxghost,jubeg+1),1,ujtype,front,0,
                  u(iubeg-nxghost,juend),1,ujtype,back,0,
                  comm2d,status,ierr)
```

Function *MPI_SendRecv* will send local information to the neighboring processors and at the same time receive information from the same neighbor. So we only need to set the correct send and receive buffer and do this to four directions. In the first function call, $u(iuend - nxghost, jubeg - nyghost)$ is the send buffer. The index $iuend - nxghost, jubeg - nyghost$ is the starting index of the local internal points which will be sent to right neighbor and will be put on the left ghost layers of right neighbor. Here $uitype, ujtype$ is the *MPI* vector type set for information communication.

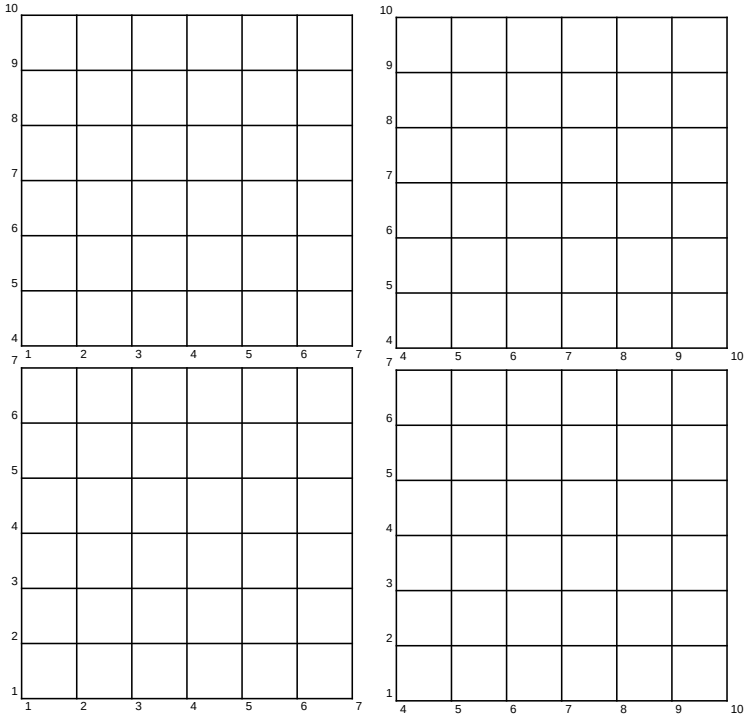


Figure 5.2: Domain decomposition of flow field u

5.3.2. Objects information

The second kind of communication is designed for the information of objects, i.e. vertices coordinates and curvature. For any processor, it only stores the information of objects if any object on it. Because an object could move to any position of the computational domain, as shown in Figure 5.3, the processor needs to find a way to know if it should receive any object from a neighbor and how many objects to receive. In order to solve this, we have developed a method for processors to detect the movement and pass the object information to neighbor if any object moves to any neighbor processor. The procedure is listed as following:

1. Determine if any object has moved into neighbor processor

```
! store old information of objects on neighboring processors
ALLOCATE(oldproc4obj(0:8,nobj4proc))
oldproc4obj=proc4obj
! reset proc4obj
proc4obj=0
! loop over the objects and neighbors to see
! if the object is on this neighbor
```

```
DO myobj=1,nobj4proc
   DO neighbor=0,8
      myproc=nearproc(neighbor)
      DO myvertex=nvertex4proc(myobj-1)+1,nvertex4proc(myobj)-1
         A(1)=xs(myvertex)
         A(2)=ys(myvertex)
         B(1)=xs(myvertex+1)
         B(2)=ys(myvertex+1)
         corner0=allcorner0(:,myproc)
         corner1=allcorner1(:,myproc)
         IF(panelinPE(A,B,corner0,corner1)) THEN
            proc4obj(neighbor,myobj)=1
            ! regard the object on myproc if one of its panels on myproc
            EXIT
         ENDIF
      END DO
   END DO
END DO
proc4obj=proc4obj-oldproc4obj
! After this operation, delete if value is -1,
! send if value is 1, and no action if value is 0
DEALLOCATE(oldproc4obj)
```

2. Determine number of objects to be sent to each neighbor processor

```
nobj4send=0
DO neighbor=1,8
   nobj4send(neighbor)=0
   DO myobj=1,nobj4proc
      IF(proc4obj(neighbor,myobj)==1) THEN
         nobj4send(neighbor)=nobj4send(neighbor)+1
      END IF
   END DO
END DO
```

3. Send and receive number of objects for exchange

```
nobj4recv=0
DO neighbor=2,8,2
! send from one direction and receive from
! its opposite direction, then alternate
   CALL MPI_SENDRECV(nobj4send(neighbor),1,MPI_INTEGER,&
```

```
        nearproc(neighbor),0,nobj4recv(neighbor-1),1,&
        MPI_INTEGER,nearproc(neighbor-1),0,comm2d,status,ierr)
   CALL MPI_SENDRECV(nobj4send(neighbor-1),1,MPI_INTEGER,&
        nearproc(neighbor-1),0,nobj4recv(neighbor),1,&
        MPI_INTEGER,nearproc(neighbor),0,comm2d,status,ierr)
END DO
```

4. Create a stack for objects and determine ending index of each object in object stack

```
DO neighbor=1,8
   nobj4send(neighbor)=nobj4send(neighbor-1)+nobj4send(neighbor)
END DO
ntotalobj4send=nobj4send(8)
DO neighbor=1,8
   nobj4recv(neighbor)=nobj4recv(neighbor-1)+nobj4recv(neighbor)
END DO
ntotalobj4recv=nobj4recv(8)

ALLOCATE(obj4send(2,ntotalobj4send))
obj4send=0
ALLOCATE(locindex4sendobj(ntotalobj4send))
locindex4sendobj=0
```

5. Pack objects for send in the stack

```
countobj=1
DO neighbor=1,8
   DO myobj=1,nobj4proc
      IF(proc4obj(neighbor,myobj)==1) THEN
         ! local object index
         locindex4sendobj(countobj)=myobj
         ! global object index
         obj4send(1,countobj)=obj4proc(myobj)
         ! number of vertices
         obj4send(2,countobj)=nvertex4proc(myobj)&
                              -nvertex4proc(myobj-1)
         countobj=countobj+1
      END IF
   END DO
END DO
```

6. Exchange objects info with neighbors

```
ALLOCATE(obj4recv(2,ntotalobj4recv))
obj4recv=0
DO neighbor=2,8,2
   countsend=nobj4send(neighbor)-nobj4send(neighbor-1)
   IF(countsend/=0) THEN
      CALL MPI_SEND(obj4send(1,nobj4send(neighbor-1)+1),&
      2*countsend,MPI_INTEGER,nearproc(neighbor),0,comm2d,ierr)
   END IF
   countrecv=nobj4recv(neighbor-1)-nobj4recv(neighbor-2)
   IF(countrecv/=0) THEN
      CALL MPI_RECV(obj4recv(1,nobj4recv(neighbor-2)+1),&
      2*countrecv,MPI_INTEGER,nearproc(neighbor-1),0,comm2d,status,ierr)
   END IF
   countsend=nobj4send(neighbor-1)-nobj4send(neighbor-2)
   IF(countsend/=0) THEN
      CALL MPI_SEND(obj4send(1,nobj4send(neighbor-2)+1),&
      2*countsend,MPI_INTEGER,nearproc(neighbor-1),0,comm2d,ierr)
   END IF
   countrecv=nobj4recv(neighbor)-nobj4recv(neighbor-1)
   IF(countrecv/=0) THEN
      CALL MPI_RECV(obj4recv(1,nobj4recv(neighbor-1)+1),&
      2*countrecv,MPI_INTEGER,nearproc(neighbor),0,comm2d,status,ierr)
   END IF
END DO
DEALLOCATE(obj4send)
```

7. Determine number of vertices to be received

```
ALLOCATE(nvertex4recv(0:ntotalobj4recv))
nvertex4recv=0
DO neighbor=1,8
   DO countobj=nobj4recv(neighbor-1)+1,nobj4recv(neighbor)
      nvertex4recv(countobj)=nvertex4recv(countobj-1)&
                             +obj4recv(2,countobj)
   END DO
END DO
ntotalvertex4recv=nvertex4recv(ntotalobj4recv)
```

8. Exchange vertices and curvatures

```
ALLOCATE(newvertex(3,ntotalvertex4recv))
DO neighbor=2,8,2
   DO countobj=nobj4send(neighbor-1)+1,nobj4send(neighbor)
      myobj=locindex4sendobj(countobj)
      myvertex=nvertex4proc(myobj-1)+1
      countvertex=nvertex4proc(myobj)-nvertex4proc(myobj-1)
      IF(countvertex/=0) THEN
         CALL MPI_SEND(vertex(1,myvertex),3*countvertex,&
         MPI_DOUBLE_PRECISION,nearproc(neighbor),0,comm2d,ierr)
      END IF
   END DO
   DO countobj=nobj4recv(neighbor-2)+1,nobj4recv(neighbor-1)
      myvertex=nvertex4recv(countobj-1)+1
      countvertex=nvertex4recv(countobj)-nvertex4recv(countobj-1)
      IF(countvertex/=0) THEN
         CALL MPI_RECV(newvertex(1,myvertex),3*countvertex,&
         MPI_DOUBLE_PRECISION,nearproc(neighbor-1),0,comm2d,status,ierr)
      END IF
   END DO
   DO countobj=nobj4send(neighbor-2)+1,nobj4send(neighbor-1)
      myobj=locindex4sendobj(countobj)
      myvertex=nvertex4proc(myobj-1)+1
      countvertex=nvertex4proc(myobj)-nvertex4proc(myobj-1)
      IF(countvertex/=0) THEN
         CALL MPI_SEND(vertex(1,myvertex),3*countvertex,&
         MPI_DOUBLE_PRECISION,nearproc(neighbor-1),0,comm2d,ierr)
      END IF
   END DO
   DO countobj=nobj4recv(neighbor-1)+1,nobj4recv(neighbor)
      myvertex=nvertex4recv(countobj-1)+1
      countvertex=nvertex4recv(countobj)-nvertex4recv(countobj-1)
      IF(countvertex/=0) THEN
         CALL MPI_RECV(newvertex(1,myvertex),3*countvertex,&
         MPI_DOUBLE_PRECISION,nearproc(neighbor),0,comm2d,status,ierr)
      END IF
   END DO
END DO
DEALLOCATE(locindex4sendobj)
```

9. Save old object data for update

```
ALLOCATE(oldobj4proc(nobj4proc))
ALLOCATE(oldnvertex4proc(0:nobj4proc))
oldobj4proc=obj4proc
oldnvertex4proc=nvertex4proc
DEALLOCATE(obj4proc)
DEALLOCATE(nvertex4proc)
nlocvertex=oldnvertex4proc(nobj4proc)
ALLOCATE(oldvertex(3,nlocvertex))
oldvertex=vertex

DEALLOCATE(vertex)
DEALLOCATE(xs)
DEALLOCATE(ys)
```

10. Count number of received repeated objects

```
nobj4repeat=0
DO myobj=1,ntotalobj4recv
   DO countobj=1,myobj-1
      IF(obj4recv(1,countobj)/=0 .AND.&
      obj4recv(1,myobj)==obj4recv(1,countobj)) THEN
         obj4recv(1,myobj)=0
         nobj4repeat=nobj4repeat+1
         EXIT
      END IF
   END DO
END DO
```

11. Determine number of objects out of the current processor

```
nobj4delete=0
DO myobj=1,nobj4proc
   nobj4delete=nobj4delete-proc4obj(0,myobj)
END DO
```

12. Update local object info

```
oldnobj4proc=nobj4proc
nobj4proc=oldnobj4proc-nobj4delete+ntotalobj4recv-nobj4repeat
ALLOCATE(obj4proc(nobj4proc))
ALLOCATE(nvertex4proc(0:nobj4proc))
obj4proc=0
nvertex4proc=0
countobj=1
DO myobj=1,oldnobj4proc
   IF(proc4obj(0,myobj)==-1) THEN
      oldobj4proc(myobj)=0
   ELSE
      obj4proc(countobj)=oldobj4proc(myobj)
      nvertex4proc(countobj)=nvertex4proc(countobj-1)&
               +(oldnvertex4proc(myobj)-oldnvertex4proc(myobj-1))
      countobj=countobj+1
   END IF
END DO
DO myobj=1,ntotalobj4recv
   IF(obj4recv(1,myobj)/=0) THEN
      obj4proc(countobj)=obj4recv(1,myobj)
      nvertex4proc(countobj)=nvertex4proc(countobj-1)&
                             +obj4recv(2,myobj)
      countobj=countobj+1
   END IF
END DO
DEALLOCATE(oldobj4proc)
nlocvertex=nvertex4proc(nobj4proc)
```

13. Update local vertices and curvatures based on old data and received data

```
ALLOCATE(vertex(3,nlocvertex))
ALLOCATE(xs(nlocvertex))
ALLOCATE(ys(nlocvertex))
countobj=1
DO myobj=1,oldnobj4proc
   IF(proc4obj(0,myobj)/=-1) THEN
      vertex(:,nvertex4proc(countobj-1)+1:nvertex4proc(countobj))=&
           oldvertex(:,oldnvertex4proc(myobj-1)+1:oldnvertex4proc(myobj))
      countobj=countobj+1
   END IF
END DO
```

```
DEALLOCATE(proc4obj)
DEALLOCATE(oldnvertex4proc)
DEALLOCATE(oldvertex)
DO myobj=1,ntotalobj4recv
   IF(obj4recv(1,myobj)/=0) THEN
      vertex(:,nvertex4proc(countobj-1)+1:nvertex4proc(countobj))=&
           newvertex(:,nvertex4recv(myobj-1)+1:nvertex4recv(myobj))
      countobj=countobj+1
   END IF
END DO

DEALLOCATE(obj4recv)
DEALLOCATE(nvertex4recv)
DEALLOCATE(newvertex)
```

14. Determine neighboring processors for each object

```
ALLOCATE(proc4obj(0:8,nobj4proc))
proc4obj=0
DO myobj=1,nobj4proc
   DO myobj_global = 1, nobj
      IF ( obj4proc(myobj) == object_list(myobj_global) ) THEN
         CALL objUpdate(myobj_global,0,objMove)
         xsc0     = objMove(1)
         ysc0     = objMove(2)
         theta0   = objMove(3)
      ENDIF
   ENDDO
   DO neighbor=0,8
      myproc=nearproc(neighbor)
      DO myvertex=nvertex4proc(myobj-1)+1,nvertex4proc(myobj)-1
         xs(myvertex)=xsc0+vertex(1,myvertex)*COS(theta0)&
                          -vertex(2,myvertex)*SIN(theta0)
         ys(myvertex)=ysc0+vertex(1,myvertex)*SIN(theta0)&
                          +vertex(2,myvertex)*COS(theta0)
         xs(myvertex+1)=xsc0+vertex(1,myvertex+1)*COS(theta0)&
                          -vertex(2,myvertex+1)*SIN(theta0)
         ys(myvertex+1)=ysc0+vertex(1,myvertex+1)*SIN(theta0)&
                          +vertex(2,myvertex+1)*COS(theta0)
         A(1)=xs(myvertex)
         A(2)=ys(myvertex)
         B(1)=xs(myvertex+1)
```

```
            B(2)=ys(myvertex+1)
            corner0=allcorner0(:,myproc)
            corner1=allcorner1(:,myproc)
            IF(panelinPE(A,B,corner0,corner1)) THEN
               proc4obj(neighbor,myobj)=1
               ! regard the object on myproc if one of its panels on myproc
               EXIT
            ENDIF
         END DO
      END DO
   END DO
```

In the development, *MPI_Send*, *MPI_Recv* and *MPI_SendRecv* are capable enough to handle the communication work. Non-blocking functions *MPI_iSend* and *MPI_iRecv* are not necessary as we need to guarantee the synchronization between all processors.

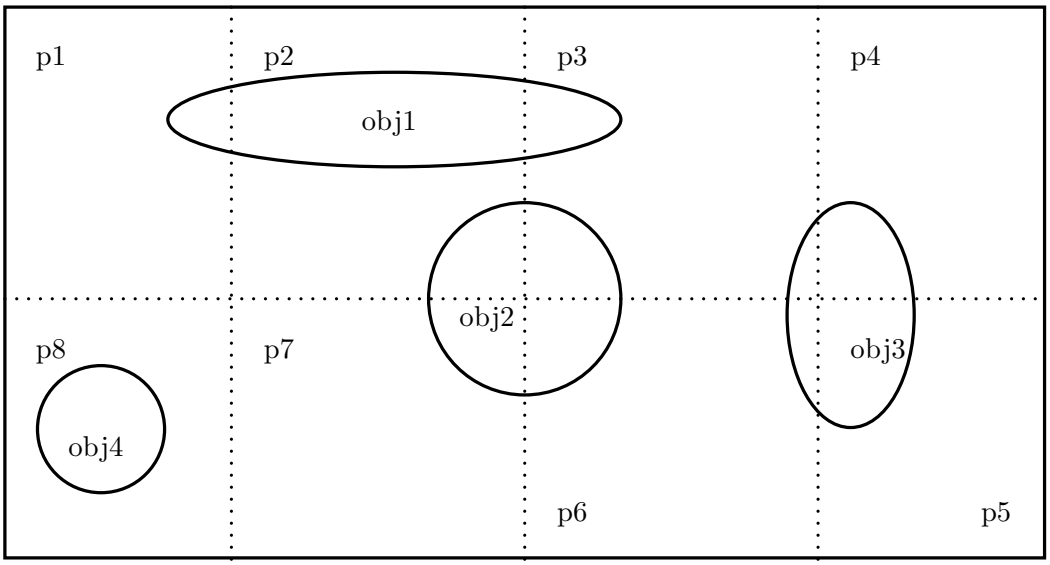


Figure 5.3: Objects on domain

5.3.3. Calculation of principle jump conditions for p

In this part, we are going to present the parallel strategy for calculation of principle jump

conditions for p, which is the simple matrix problem (3.31) below,

$$\begin{bmatrix} -2 & 1 & \dots & 0 \\ 1 & -2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & \dots & 1 & -2 \end{bmatrix} \begin{bmatrix} [p]_A \\ [p]_B \\ \vdots \\ [p]_E \end{bmatrix} = \begin{bmatrix} rhs_A \\ rhs_B \\ \vdots \\ rhs_E \end{bmatrix}$$

$$rhs_A = \frac{|AB|}{6}\left(\left[\frac{\partial p}{\partial \tau_1}\right]_A + 4\left[\frac{\partial p}{\partial \tau_1}\right]_{M_1} + \left[\frac{\partial p}{\partial \tau_1}\right]_B\right) + \frac{|AF|}{6}\left(\left[\frac{\partial p}{\partial \tau_2}\right]_A + 4\left[\frac{\partial p}{\partial \tau_2}\right]_{M_2} + \left[\frac{\partial p}{\partial \tau_2}\right]_F\right)$$

$$\left[\frac{\partial p}{\partial \tau}\right] = \left[\frac{\partial p}{\partial x}\right]\tau_x + \left[\frac{\partial p}{\partial y}\right]\tau_y$$

As we discussed in chapter 3, rhs_A can be calculated based on local information. $|AB|$ is the length of panel, $\left[\frac{\partial p}{\partial x}\right]$, $\left[\frac{\partial p}{\partial y}\right]$, τ_x and τ_y can be easily computed. Since this is a very simple matrix problem, in the development we used Gauss elimination method to solve it locally.

But the problem here is, for each processor, it can only compute parts of rhs vector if the object is partially on it. If the processor wants to solve this matrix problem locally, there is not enough information. What's even worse is for any processor, it may have more than one object on it. Let's see the example in Figure 5.3, objects 1∼4 are located in different processors from *p1* to *p8*. For each object, the processor only has parts of the rhs stored locally. So the question here is how can we collect right hand side vector rhs from the processor that the object occupies, so each processor can individually solve the problem. In order to solve this, we have developed a method to collect rhs. The main idea of the method is to use the feature of *MPI_Group* and *MPI_Allreduce*. For each object, a unique group will be created including the the processors which has this exact object on it. For example given $obj1$, a group will be created including $p1$, $p2$ and $p3$. For each processor, it can belong to different groups because it may have different objects on it. When computing the rhs, all values of the right hand side can be reduced to all the processors in this exact group, then

each processor can solve the matrix locally and take the parts of $[p]$ needed. The procedure is listed as following:

1. Define the FORTRAN structure for creating *MPI* group. The idea is for every object, we will create a *MPI_Group* involving all processors where this object is on those. *r_temp* will temporally store the *rhs* collected from all processors for each object and *p_rank* will store the processor ID

```
TYPE cg_group
    INTEGER:: group, comm
    DOUBLE PRECISION, DIMENSION(:), ALLOCATABLE:: r_temp
    INTEGER, DIMENSION(:), ALLOCATABLE:: p_rank, req
    INTEGER, DIMENSION(:,:), ALLOCATABLE:: STATUS
END TYPE cg_group
TYPE(cg_group):: group_mpi(nobj)
INTEGER:: rank_count, rank_total(1:nobj), &
          rank_temp(0:nprocs-1),rank_c(1:nobj)
```

2. Compute elements of *rhs* if the vertex for this element is located inside the local computational domain, then we will have parts of *rhs* ready for each object on each processor

3. Find ranks of each object group then create group

```
! Create global group0 including all preocessors
CALL MPI_COMM_GROUP(comm2d,group0,ierr)
! loop over all objects
DO myobj = 1,nobj
    IF object on local processor
        rank_c(myobj) = 1,
        EXIT               ! Exit to next object
    ELSE
        rank_c(myobj)=0
    ENDIF
    rank_temp = 0
    ! For each global object, gather rank for all processors
    CALL MPI_ALLGATHER(rank_c(myobj),1,MPI_INTEGER, &
            rank_temp,1,MPI_INTEGER,comm2d,ierr)
    ! Compute for each object, how many processors have it
    rank_total(myobj) = SUM(rank_temp)
```

```
        ! Assign the rank of processors to group_mpi(myobj)%p_rank
        ALLOCATE(group_mpi(myobj)%p_rank(rank_total(myobj)))
        rank_count = 0
        DO i = 0, nprocs-1
            IF(rank_temp(i) == 1) THEN
                rank_count = rank_count +1
                group_mpi(myobj)%p_rank(rank_count) = i
            ENDIF
        ENDDO
        ! create group for current global object
        CALL MPI_GROUP_INCL(group0,rank_total(myobj), &
            group_mpi(myobj)%p_rank,group_mpi(myobj)%group,ierr)
        CALL MPI_COMM_CREATE(comm2d,group_mpi(myobj)%group, &
            group_mpi(myobj)%comm,ierr)
    ENDDO
```

4. Gather *rhs* from all processors for each object

```
! loop over all objects
DO myobj = 1, nobj
! Since for any processor it may not have any specific object
! so here we need identify if the communicator is NULL or not
    IF(group_mpi(myobj)%comm /= MPI_COMM_NULL) THEN
        ! loop over local objects and find corresponding object
        DO myobj_local = 1, nobj4proc
           IF ( obj4proc(myobj_local) == object_list(myobj) ) THEN
              ! If global and local object matches, then create
              ! temporary r vector to store rhs
              ALLOCATE(group_mpi(myobj)%r_temp(nvertex4proc(myobj_local) &
                    -nvertex4proc(myobj_local-1)))
              ! update x_sol & r_new
              ! loop over all vertices of this object
              DO myvertex = nvertex4proc(myobj_local-1)+1,  &
                    nvertex4proc(myobj_local)
                 A = vertex(:,myvertex)
                 IF( (A(1)>xc(ipbeg)).AND.(A(2)>yc(jpbeg)).AND. &
                      (A(1)<xc(ipend+1)).AND.(A(2)<yc(jpend+1))) THEN
                    ! Reduce the value of the vertex
                    ! if it is inside the subdomain
                    CALL MPI_ALLREDUCE(jcp_rhs(myvertex), &
                        group_mpi(myobj)%r_temp(myvertex-nvertex4proc(myobj_local-1)), &
                        1, MPI_DOUBLE_PRECISION, &
                        MPI_SUM,group_mpi(myobj)%comm,ierr)
                 ELSE
                    ! Reduce zero to all processors if the vertex not on this processor
                    CALL MPI_ALLREDUCE(0.0d0, &
                    group_mpi(myobj)%r_temp(myvertex-nvertex4proc(myobj_local-1)), 1, &
                    MPI_DOUBLE_PRECISION,MPI_SUM,group_mpi(myobj)%comm,ierr)
                 ENDIF
              ENDDO
```

```
        END IF
      END DO
    ENDIF
  END DO
```

5. Update *rhs* with the information we have reduced from other processors

```
jcp_rhs(myvertex) = group_mpi(myobj)%r_temp(myvertex &
                        -nvertex4proc(myobj_local-1))
```

6. Free the object group and group communicator

```
DO myobj = 1,nobj
 IF(group_mpi(myobj)%comm /= MPI_COMM_NULL) THEN
    CALL MPI_COMM_FREE(group_mpi(myobj)%comm, ierr)
 ENDIF
 CALL MPI_GROUP_FREE(group_mpi(myobj)%group, ierr)
ENDDO
```

7. Solve the matrix prolbem locally using Gauss-Seidel

By the method above, $[p]$ can be easily solved. The disadvantage of this method is for every processor it will loop over all the objects no matter the object is on it or not. Depending on how the objects are indexed, there will be a potential bottleneck here that one processor may be waiting for other processors work on the same object. We have another way to solve this problem but have not implemented it into the program. For example in Figure 5.3, *obj*2 only takes 4 processors. Instead of creating a group for *obj*2, we can simply use *MPI_SendRecv* to pass *rhs* for those four processors. We can first use *MPI_SendRecv* to handle small objects which only takes up to 4 processors, then for bigger objects use the *MPI_Group* technique to solve for $[p]$.

In the beginning of solving this problem, we have thought another way to do it, which is using the conjugate gradient method(CG). The algorithm is shown as below,

1. $\mathbf{r}_0 := \mathbf{b} - \mathbf{A}\mathbf{x}_0$

2. $\mathbf{p}_0 := \mathbf{r}_0$

3. $k_0 := 0$

4. **while do**

5. $\alpha_k := \frac{\mathbf{r}_k{}^T \mathbf{r}_k}{\mathbf{p}_k{}^T \mathbf{A}\mathbf{p}_k}$

6. $\mathbf{x}_{k+1} := \mathbf{x}_k + \alpha_k \mathbf{p}_k$

7. $\mathbf{r}_{k+1} := \mathbf{r}_k - \alpha_k \mathbf{A}\mathbf{p}_k$

8. $if\ ||\mathbf{r}_{k+1}|| < tol,\ exit$

9. $\beta_k := \frac{\mathbf{r}_{k+1}{}^T \mathbf{r}_{k+1}}{\mathbf{r}_k{}^T \mathbf{r}_k}$

10. $\mathbf{p}_{k+1} := \mathbf{r}_{k+1} + \beta_k \mathbf{p}_k$

11. $k := k + 1$

12. **end while**

13. **return** x_{k+1}

$\mathbf{b}$ is the right hand side vector, $\mathbf{r}$ is the residual vector, $\mathbf{x}_0$ is the initial solution. Why we wanted to use CG at the beginning of the development is because it has this vector multiplication $\mathbf{r}_k{}^T \mathbf{r}_k$ which shows us the possibility to avoid the communication between processors and save computational time. However, for the exact matrix problem we have here, in calculation of $\mathbf{p}_k{}^T \mathbf{A}\mathbf{p}_k$, we still need information outside of local boundary, which means a ghost layer is still needed and as well as the communication between processors. Besides, if we want to use CG, this communication will be done during every while loop, which is a big cost on computational time. In order to avoid uncountable communication in while loops, we decide to pass the information between processors only once and then use classical Gauss elimination to solve the problem.

5.3.4. Collection communication

Besides the above information exchange, we have the collection communication between processors. For example, when we compute time step Δt, we have

$$\Delta t_v = \frac{CFL_v \cdot Re}{\left(\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2}\right)}$$

$$\Delta t_c = \frac{CFL_c}{\frac{u_{max}}{\Delta x} + \frac{v_{max}}{\Delta y}}$$

$$\Delta t = min(\Delta t_0, \Delta t_c, \Delta t_v)$$

As we know, u_{max} and v_{max} are the maximum value in the flow field. However, since each processor only stores parts of u and v, a collection communication is needed. The same situation is applied to Δx and Δy. In later sections of discretizaton for parallel program, we will talk about use of non-uniform grid in our method. Here in a word, Δx and Δy could be different on each processor and we need to find the minimum Δx and Δy across the whole computational domain. Another collection communication is necessary. In the development, *MPI_Gather*, *MPI_Allgather*, *MPI_Reduce*, *MPI_Allreduce* and *MPI_Bcast* could handle this kind of collective communication easily. For example, the FORTRAN syntax of *MPI_Allreduce* is designed as below,

```
MPI_ALLREDUCE(SENDBUF, RECVBUF, COUNTS, DATATYPE, OP, COMM, IERROR)
```

SENDBUF and *RECVBUF* are send and receive buffer, *COUNTS* is the size of data. *COMM* is the communicator and *IERROR* is the error message. What makes this function powerful is we have this *OP* operation handle in the syntax, which can be *MPI_Max*, *MPI_MIin*, *MPI_Product*, *MPI_Sum*, etc. By calling this function, we can automatically find the max or min value of a variable from all processors, or the sum of a number from all processors. In our method, we use *MPI_Allreduce* to find minimum Δx and Δy, then use *MPI_Reduce* to find u_{max} and v_{max} for the root processor. Then root processor calculates Δt and broadcasts it to all processors using *MPI_Bcast*.

Similar work has been done when we compute drag & lift coefficients.

$$\vec{G} = \int_{\Gamma} \left(-p|_{\Gamma^+} \cdot \vec{n} + \frac{1}{Re} \left(\frac{\partial \vec{u}}{\partial n} \right) |_{\Gamma^+} \right) dl$$

We have to summarize $\sum \left(-p|_{\Gamma^+} \cdot \vec{n} + \frac{1}{Re} \left(\frac{\partial \vec{u}}{\partial n} \right) |_{\Gamma^+} \right) dl$ over all panels, but results of $\left(-p|_{\Gamma^+} \cdot \vec{n} + \frac{1}{Re} \left(\frac{\partial \vec{u}}{\partial n} \right) |_{\Gamma^+} \right)$ could be on different processors as objects may across different processors. Here *MPI_Reduce* or *MPI_Allreduce* is necessary. For other collective communications, they are similar to the above here.

5.3.5. Parallel I/O

By far all the applications of *MPI* is for the calculation of our method, but when the computation is finished, all the information of velocity and pressure field is still distributed on different processors, as shown in Figure 5.1. The question here is how can we collect information from all the processors and then move to the post-processing stage. Here we use the I/O functions from *MPI*. For example, let's see the output of pressure field. The codes are shown as below,

```
CALL MPI_TYPE_CREATE_SUBARRAY(2, psizes, psubsizes, pstarts,&
     MPI_ORDER_FORTRAN,MPI_DOUBLE_PRECISION,pfiletype,ierr)
CALL MPI_Type_commit(pfiletype,ierr)
CALL MPI_Info_create(info,ierr)
CALL MPI_Info_set(info,'collective_buffering','true',ierr)
CALL MPI_File_open(comm2d,'OUTPUT/p.dat',&
      MPI_MODE_WRONLY + MPI_MODE_CREATE,info,pfh,ierr)
CALL MPI_FILE_SET_VIEW(pfh,zero_off,&
      MPI_DOUBLE_PRECISION,pfiletype,'native',MPI_INFO_NULL,ierr)
CALL MPI_FILE_WRITE_ALL(pfh,psubarray,np,&
      MPI_DOUBLE_PRECISION,MPI_STATUS_IGNORE,ierr)
CALL MPI_FILE_CLOSE(pfh,ierr)
```

First, we have to calculate the total size of the computational domain and local size of the subdomain, then create a new data type using *MPI_Type_create_subarray* and *MPI_Type_commit*. Then we need to create an info object using *MPI_Info_create* and *MPI_Info_set* to set collective communication and prepare for the open of data file. Finally we can open the file, write the pressure data with all the previous setup, and finally close the file. In our tests,

this I/O process has been proved to be very efficient and there is no conflict between each processor when writing data to the same data file.

5.4. Mesh Stretching

In this section, we will talk about some details of discretization for the parallel program. As we introduced mesh stretching in the parallel program, many formulas have been modified and details will be presented here.

5.4.1. Mesh stretching

In order to improve the robustness and accuracy, in our parallel program the stretched/non-uniform mesh is introduced instead of using a uniform mesh. Stretched mesh is widely used in CFD problems as it improves the resolution and avoids increasing the computational cost. In our parallel program, all algorithms are redesigned based on stretched mesh. Here we actually keep two Cartesian grids, one is uniform Cartesian grids on (ξ, η), the other one is stretched on $(x(\xi), y(\eta))$. x is a function of ξ, and y is a function of η. In the program, we usually set $\xi = -1 : 1$ and $\eta = -1 : 1$. Then the range of x and y is the range of real computational domain.

For example, in the lid-driven Cavity flow test(6.2), the result could be more accurate if there are more points in the area which is close to the wall on $\xi = -1, 1$ and $\eta = -1, 1$, and the real computational domain is $x = 0 : 1$ and $y = 0 : 1$. So here we use the stretching method as below:

$$X(\xi) = \frac{\tanh(c\xi)}{2\tanh(c)} + 0.5 \tag{5.3a}$$

$$Y(\eta) = \frac{\tanh(c\eta)}{2\tanh(c)} + 0.5 \tag{5.3b}$$

Where c is a constant to control the stretching ratio, which is defined as $Ratio = \frac{\Delta x_{max}}{\Delta x_{min}}$. As you can see from Figure 5.4a, ξ is uniformly distributed between -1 and 1, but for x the points are more close to $x = 0$ and $x = 1$ and loose in the middle around $x = 0.5$. Figure 5.4b shows the stretching Cartesian grids we have used in the parallel test of lid-driven Cavity

flow. As you can see the grid points are more gathered close to the wall and loose in the center. The wall looks like very thick because there are more points located there.

For different tests, the stretching method is different and it depends on the problem itself. Besides, we also need to compute x_ξ, $x_{\xi\xi}$, ξ_x, ξ_{xx}, y_η, $y_{\eta\eta}$, η_y and η_{yy}. These results will be used in the finite difference schemes.

5.4.2. Finite difference schemes

As we know, $x = x(\xi)$ and $y = y(\eta)$. Assume a function ϕ is defined on the Cartesian grids and it is a function of x and y, then $\phi = \phi(x, y)$. Then for the first and second order derivatives, they will be

$$\frac{\partial \phi}{\partial x} = \frac{\partial \phi}{\partial \xi}\frac{d\xi}{dx} = \xi_x \frac{\partial \phi}{\partial \xi} \tag{5.4a}$$

$$\frac{\partial \phi}{\partial \xi} = \frac{\partial \phi}{\partial x}\frac{dx}{d\xi} = x_\xi \frac{\partial \phi}{\partial x} \tag{5.4b}$$

$$\frac{\partial^2 \phi}{\partial x^2} = \frac{\partial}{\partial x}\left(\frac{\partial \phi}{\partial \xi}\frac{d\xi}{dx}\right) = \xi_{xx}\frac{\partial \phi}{\partial \xi} + (\xi_x)^2 \frac{\partial^2 \phi}{\partial \xi^2} \tag{5.4c}$$

$$\frac{\partial^2 \phi}{\partial \xi^2} = \frac{\partial}{\partial \xi}\left(\frac{\partial \phi}{\partial x}\frac{dx}{d\xi}\right) = x_{\xi\xi}\frac{\partial \phi}{\partial x} + (x_\xi)^2 \frac{\partial^2 \phi}{\partial x^2} \tag{5.4d}$$

In our program, except $\frac{\partial \phi}{\partial x}$ and $\frac{\partial^2 \phi}{\partial x^2}$ are necessary, sometimes we also need $\frac{\partial \phi}{\partial \xi}$ and $\frac{\partial^2 \phi}{\partial \xi^2}$. For $\frac{\partial \phi}{\partial y}$, $\frac{\partial^2 \phi}{\partial y^2}$, $\frac{\partial \phi}{\partial \eta}$ and $\frac{\partial^2 \phi}{\partial \eta^2}$, the process is similar.

Now the question is how can we discretize them. Since we are using MAC method, we have x and y in both center lines and edge lines of the MAC cell, which we call x_c, x_f, y_c and y_f. $(x_c)_i = x_i$ and $(x_f)_i = x_{i+\frac{1}{2}}$ as shown in Figure 4.1, and similar for y_c and y_f. For the first order derivatives, we have three ways to discretize them depending on where we will use them. If function ϕ is defined on the vertical lines x_c that pass the center of MAC cell, then the discretization will be

$$Central: \left(\frac{\partial \phi}{\partial x}\right)_i = \frac{\phi_{i+1} - \phi_{i-1}}{2\Delta\xi}(\xi_x)_i \tag{5.5a}$$

$$Forward: \left(\frac{\partial \phi}{\partial x}\right)_i = \frac{\phi_{i+1} - \phi_i}{\Delta\xi}(\xi_x)_{i+\frac{1}{2}} \tag{5.5b}$$

$$Backward: \left(\frac{\partial \phi}{\partial x}\right)_i = \frac{\phi_i - \phi_{i-1}}{\Delta \xi} \left(\xi_x\right)_{i-\frac{1}{2}} \tag{5.5c}$$

For the second order derivative, the discretization is shown below:

$$\begin{aligned}
\frac{\partial^2 \phi}{\partial x^2} &= \frac{\phi_{i+1} - \phi_i}{\left(x_\xi\right)_{i+\frac{1}{2}} \left(x_\xi\right)_i \Delta \xi^2} - \frac{\phi_i - \phi_{i-1}}{\left(x_\xi\right)_{i-\frac{1}{2}} \left(x_\xi\right)_i \Delta \xi^2} \\
&= \left(\xi_x\right)_{i+\frac{1}{2}} \left(\xi_x\right)_i \frac{\phi_{i+1} - \phi_i}{\Delta \xi^2} - \left(\xi_x\right)_{i-\frac{1}{2}} \left(\xi_x\right)_i \frac{\phi_i - \phi_{i-1}}{\Delta \xi^2} \\
&= \frac{\left(\xi_x\right)_i}{\Delta \xi^2} \left[\left(\phi_{i+1} - \phi_i\right) \left(\xi_x\right)_{i+\frac{1}{2}} - \left(\phi_i - \phi_{i-1}\right) \left(\xi_x\right)_{i-\frac{1}{2}}\right] \\
&= \frac{\left(\xi_x\right)_i}{\Delta \xi^2} \left[\left(\xi_x\right)_{i-\frac{1}{2}} \phi_{i-1} - \left(\left(\xi_x\right)_{i-\frac{1}{2}} + \left(\xi_x\right)_{i+\frac{1}{2}}\right) \phi_i + \left(\xi_x\right)_{i+\frac{1}{2}} \phi_{i+1}\right]
\end{aligned} \tag{5.6}$$

Of course there is another way to discretize the second order derivative, which is based on equation

$$\frac{\partial^2 \phi}{\partial x^2} = \xi_{xx} \frac{\partial \phi}{\partial \xi} + \left(\xi_x\right)^2 \frac{\partial^2 \phi}{\partial \xi^2}$$

But when we setup the matrix coefficients of pressure Poisson problem, the first discretization will be easier to code since the second discretization involves both first and second order derivatives of ϕ and ξ.

5.5. Pressure Solver

As we mentioned in section 4.1.3, we used FFT and Helmholtz iterative solver to solve the pressure Poisson equation in serial. FFT is a very powerful method, but it is not trival to develop a parallel version for it. Besides, since we are using stretched mesh in the parallel program, FFT does not work any more. Instead, in our parallel program we use the multigrid method to solve the the pressure Poisson problem and *Hypre* library is used in the development of the program.

5.5.1. Multigrid method

In the scientific computing field, no matter what we are studying, there is always a high possibility that we will come to solve the matrix problem $A\mathbf{x} = \mathbf{b}$. The matrix A could be dense or sparse, symmetric or even semi-positive definite. In order to solve this problem,

people have been studying it since as early as 19th century. There are some very famous methods that are widely used by researchers, like Gauss elimination method, successive over-relaxation(SOR) method, singular value decomposition(SVD), etc. There are also iterative methods like generalized minimum residual method(GMRES), conjugate gradient(CG), and the method we used here, which is multigrid method(MG).

In the multigrid method, the main idea is to accelerate the convergence of classical iterative method by solving the problem on coarser grids. For many standard iterative methods, they possess the property that they are effective at eliminate the high-frequency or oscillatory components of the error and leaving the low-frequency components unchanged. But what we want is to make this method effective on all error components. First thing we know is, on a coarser grid, the remaining smooth error components will look more oscillatory, which means the smooth error components on a fine grid will be less smooth on a coarse grid. This suggests we can use relaxation method on a coarser grid to make it more effective. The second thing we know is, assuming we have $\bar{\mathbf{x}}$ is the exact solution of $A\mathbf{x} = \mathbf{b}$ and $\mathbf{u}$ is the approximation result, then the error $\mathbf{e} = \mathbf{u} - \bar{\mathbf{x}}$ satisfies $A\mathbf{e} = \mathbf{r} = \mathbf{b} - A\mathbf{u}$. The relaxation on $A\mathbf{x} = \mathbf{b}$ is equivalent to relaxation on $A\mathbf{e} = \mathbf{r}$ with the special initial guess $\mathbf{e} = \mathbf{0}$. Then by solving $\mathbf{e}$, we can make a correction to the result $\mathbf{u}$ and let the result be more accurate. Inspired by this, we can repeat the relaxation process on a coarser and coarser grid until it reaches to the coarsest grid. Now let me introduce a classical multigrid method scheme, which is the V-Cycle scheme, to demonstrate how it works. Assume we have this function MG, ν_1 and ν_2 are the total number of smooth operations for pre-relaxation and post-relaxation. h means operation on fine grid and $2h$ means operation on coarsen grid. Matrix A is the stencil, $\mathbf{u}_0$ is the initial guess and $\mathbf{b}$ is the right hand side. The scheme is shown below

$\mathbf{u}^h = MG(A^h, \mathbf{u}_0^h, \mathbf{b}^h, \nu_1, \nu_2)$

1. Pre-relaxation: $\mathbf{u}^h = smooth^{\nu_1}(A^h, \mathbf{u}^h, \mathbf{b}^h)$

2. Get residual $\mathbf{r}^h = \mathbf{b}^h - A^h\mathbf{u}^h$

3. Coarsen: $\mathbf{r}^{2h} = I_h^{2h}\mathbf{r}^h$

4. If reaches the coarsest grid:

5. Solve: $A^{2h}\delta^{2h} = \mathbf{r}^{2h}$

6. Else:

7. Recursion: $\delta^{2h} = MG(A^{2h}, 0, \mathbf{r}^{2h}, \nu_1, \nu_2)$

8. End If

9. Correction: $\mathbf{u}^h = \mathbf{u}^h + I_{2h}^h\delta^{2h}$

10. Post-relaxation: $\mathbf{u}^h = smooth^{\nu_2}(A^h, \mathbf{u}^h, \mathbf{b}^h)$

11. Return $\mathbf{u}^h$

In the V-cycle here, we used idea of recursion to help us improve the speed. By calling the same function $MG(A^{2h}, 0, \mathbf{r}^{2h}, \nu_1, \nu_2)$ for solving δ^{2h} inside the function, the program will go to a coarser level again instead of just returning to the last finer level. There are more details of the multigrid method can be found in [8, 51] if anyone is interested.

5.5.2. *Hypre* library

Hypre library provides a suite of scalable linear solvers for large-scale scientific computing in C, C++ and FORTRAN. It is developed by Lawrence Livermore national library mainly based on multigrid methods and it uses *MPI* for communication between processors. It solves problems faster than traditional methods at large scale and has good support of both structured and unstructured grids problems. This library has been widely used by many institutes and researchers, and the solvers are proved to be stable, accurate, efficient and stable.

The setup of *Hypre* is very easy and steps are as following:

1. Create the 2D/3D grid and set the extents

2. Build the stencil and set the stencil element

3. Build the stencil of matrix A for pressure Poisson problem $A\mathbf{x} = \mathbf{b}$

   Since our parallel program is developed for a 2D problem, we need modify equation (5.6). The result is shown as below:

$$\left(\frac{\partial^2\phi}{\partial x^2} + \frac{\partial^2\phi}{\partial y^2}\right)_{i,j} = \frac{(\xi_x)_i}{\Delta\xi^2}\left[(\xi_x)_{i-\frac{1}{2}}\,\phi_{i-1,j} - \left((\xi_x)_{i-\frac{1}{2}} + (\xi_x)_{i+\frac{1}{2}}\right)\phi_{i,j} + (\xi_x)_{i+\frac{1}{2}}\,\phi_{i+1,j}\right] + \frac{(\eta_y)_j}{\Delta\eta^2}\left[(\eta_y)_{j-\frac{1}{2}}\,\phi_{i,j-1} - \left((\eta_y)_{j-\frac{1}{2}} + (\eta_y)_{j+\frac{1}{2}}\right)\phi_{i,j} + (\eta_y)_{j+\frac{1}{2}}\,\phi_{i,j+1}\right] \tag{5.7}$$

   As shown in this equation, a five-point stencil is established,

$$\begin{bmatrix} & back & \\ left & center & right \\ & front & \end{bmatrix}$$

$$left = \frac{(\xi_x)_i}{\Delta\xi^2}(\xi_x)_{i-\frac{1}{2}}, \quad right = \frac{(\xi_x)_i}{\Delta\xi^2}(\xi_x)_{i+\frac{1}{2}} \tag{5.8a}$$

$$front = \frac{(\eta_y)_j}{\Delta\eta^2}(\eta_y)_{j-\frac{1}{2}}, \quad back = \frac{(\eta_y)_j}{\Delta\eta^2}(\eta_y)_{j+\frac{1}{2}} \tag{5.8b}$$

$$center = -\frac{(\xi_x)_i}{\Delta\xi^2}\left((\xi_x)_{i-\frac{1}{2}} + (\xi_x)_{i+\frac{1}{2}}\right) - \frac{(\eta_y)_j}{\Delta\eta^2}\left((\eta_y)_{j-\frac{1}{2}} + (\eta_y)_{j+\frac{1}{2}}\right) \tag{5.8c}$$

   On the boundary, the stencil will be dependent on the boundary conditions and the setup could be different.

4. Setup the solution vector $\mathbf{x}$ and right-hand-side vector $\mathbf{b}$

5. Choose the solver and preconditioners from different multigrid methods and set solver parameters, such as tolerance and maximum iteration number

6. Assign values of vector $\mathbf{x}$ and $\mathbf{b}$ to the solver, assemble the problem with the solver we choose and matrix A

7. Solve the problem of $A\mathbf{x} = \mathbf{b}$

8. Return the computation result to $\mathbf{x}$ and retrieve other desired information

9. Free the solver

In the actual use of *Hypre*, the solver we chose is SMG, which stands for semi-coarsening multigrid method. In the 2D SMG solver, it operates semi-coarsening on x-direction and line relaxation in the y-direction. It is a very robust method and can give excellent convergence results. More details of SMG solver can be found in [9, 20, 53].

5.5.3. Compatibility condition

Even though our method is theoretically correct, in the actual computation the errors will always be generated. The errors may be small and not significant, but it actually destroys the compatibility condition for the Neumann problem, affects our use of *Hypre* library and the computation often failed. So what we are doing here is to force the right hand side vector to satisfy the compatibility condition before pass it to the *Hypre* SMG solver.

For problem $A\mathbf{x} = \mathbf{b}$, it has a solution if and only if $\mathbf{b}$ is orthogonal to the solution of $A^T\mathbf{z} = \mathbf{0}$, which is $\mathbf{b}^T\mathbf{z} = \mathbf{0}$. Since in reality $\mathbf{b}^T\mathbf{z} \neq \mathbf{0}$, then we need to find $\hat{\mathbf{b}}$ to satisfy $\hat{\mathbf{b}}^T\mathbf{z} = \mathbf{0}$. So here we let

$$\hat{\mathbf{b}} = \mathbf{b} - \frac{\mathbf{b}^T\mathbf{z}}{\mathbf{z}^T\mathbf{z}}\mathbf{z} \tag{5.9}$$

then

$$\hat{\mathbf{b}}^T\mathbf{z} = \mathbf{b}^T\mathbf{z} - \frac{\mathbf{b}^T\mathbf{z}}{\mathbf{z}^T\mathbf{z}}\mathbf{z}^T\mathbf{z} = \mathbf{b}^T\mathbf{z} - \mathbf{b}^T\mathbf{z} = \mathbf{0} \tag{5.10}$$

All we need to do is find $\mathbf{z}$, such that $\hat{\mathbf{b}}^T\mathbf{z} = \mathbf{0}$. This $\mathbf{z}$ can be found with $A^T\mathbf{z} = \mathbf{0}$. Matrix A is the 5-point stencil we have discussed in the previous part of this section, and in a general discretization form $A^T\mathbf{z} = \mathbf{0}$ will be

$$A_{i-1,j}z_{i-1,j} + A_{i+1,j}z_{i+1,j} + A_{i,j}z_{i,j} + A_{i,j+1}z_{i,j+1} + A_{i,j-1}z_{i,j-1} = 0 \tag{5.11}$$

By bringing the actual $A_{i,j}$ to the equation above, we can build the following equations(the proof is skipped):

$$z_{ij} = p_i q_j \tag{5.12a}$$

$$p_1 = 1, \quad q_1 = 1 \tag{5.12b}$$

$$p_i = \frac{(x_{i+\frac{1}{2}} - x_{i-\frac{1}{2}})(x_2 - x_0)}{(x_{\frac{3}{2}} - x_{\frac{1}{2}})(x_1 - x_0)} p_1, \quad i = 2 : n_x - 1 \tag{5.12c}$$

$$q_j = \frac{(y_{j+\frac{1}{2}} - y_{j-\frac{1}{2}})(y_2 - y_0)}{(y_{\frac{3}{2}} - y_{\frac{1}{2}})(y_1 - y_0)} q_1, \quad j = 2 : n_y - 1 \tag{5.12d}$$

$$p_{n_x} = \frac{(x_{n_x+\frac{1}{2}} - x_{n_x-\frac{1}{2}})(x_{n_x+1} - x_{n_x})(x_2 - x_0)}{(x_{n_x+1} - x_{n_x-1})(x_{\frac{3}{2}} - x_{\frac{1}{2}})(x_1 - x_0)} p_1 \tag{5.12e}$$

$$q_{n_y} = \frac{(y_{n_y+\frac{1}{2}} - y_{n_y-\frac{1}{2}})(y_{n_y+1} - y_{n_y})(y_2 - y_0)}{(y_{n_y+1} - y_{n_y-1})(y_{\frac{3}{2}} - y_{\frac{1}{2}})(y_1 - y_0)} q_1 \tag{5.12f}$$

n_x and n_y is the number of grid points on each direction, and vector $\mathbf{z}$ is built based on the coordinates of stretching grid. With a different stretching method, the value of $\mathbf{z}$ will be different.

5.6. Jump Contributions

Previously our method was based a uniform mesh and central finite difference scheme can be easily applied. Now we are using stretched mesh, the finite difference schemes are still be able to be used but some modifications are needed. In chapter 2, we have equations (2.5a) and (2.5b) for first and second order central finite difference schemes with incorporation of jump conditions, now let's see an example how they will be modified for stretched mesh.

5.6.1. Jump contribution of pressure

As shown in Figure 5.5, assume function $p(x)$ is defined on the whole domain and has jump conditions at a, b, c, d, and $x = x(\xi)$. Our goal here is to find $\left(\frac{\partial^2 p}{\partial x^2}\right)_i$. First, in order to use finite difference scheme, we should change it to an expression with uniform mesh.

$$\left(\frac{\partial^2 p}{\partial x^2}\right) = \frac{\partial}{\partial \xi}\left(\frac{\partial p}{\partial x}\right) \cdot \frac{d\xi}{dx} \tag{5.13}$$

Apply finite difference scheme on the equation above, we have

$$\left(\frac{\partial^2 p}{\partial x^2}\right)_i = \frac{(\xi_x)_i}{\Delta \xi}\left(\left(\frac{\partial p}{\partial x}\right)_{i+\frac{1}{2}} - \left(\frac{\partial p}{\partial x}\right)_{i-\frac{1}{2}} + correction\ terms\right) \tag{5.14}$$

Then our current job is to find $\left(\frac{\partial p}{\partial x}\right)_{i+\frac{1}{2}}$ and $\left(\frac{\partial p}{\partial x}\right)_{i-\frac{1}{2}}$. By applying the finite difference scheme again, we have

$$\begin{aligned}\left(\frac{\partial p}{\partial x}\right)_{i+\frac{1}{2}} &= \left(\frac{\partial p}{\partial \xi}\right)_{i+\frac{1}{2}} (\xi_x)_{i+\frac{1}{2}} \\ &= (\xi_x)_{i+\frac{1}{2}} \left(\frac{p_{i+1}-p_i}{\Delta\xi} + \frac{1}{\Delta\xi}\left(-\sum_{n=0}^{2}\frac{[p^{(n)}(c)]}{n!}(\xi_i - x2\xi(c)) - \sum_{n=0}^{2}\frac{[p^{(n)}(d)]}{n!}(\xi_{i+1} - x2\xi(d))\right)\right)\end{aligned} \tag{5.15a}$$

$$\begin{aligned}\left(\frac{\partial p}{\partial x}\right)_{i-\frac{1}{2}} &= \left(\frac{\partial p}{\partial \xi}\right)_{i-\frac{1}{2}} (\xi_x)_{i-\frac{1}{2}} \\ &= (\xi_x)_{i-\frac{1}{2}} \left(\frac{p_i-p_{i-1}}{\Delta\xi} + \frac{1}{\Delta\xi}\left(-\sum_{n=0}^{2}\frac{[p^{(n)}(a)]}{n!}(\xi_{i-1} - x2\xi(a)) - \sum_{n=0}^{2}\frac{[p^{(n)}(b)]}{n!}(\xi_i - x2\xi(b))\right)\right)\end{aligned} \tag{5.15b}$$

where function $x2\xi$ will change the grid coordinates from stretched mesh to uniform mesh. Similarly, we have function $\xi 2x$ in the program which will change grid coordinates from uniform mesh to stretched mesh. $[\cdot]^n$ here means the order of jump conditions.

- At $n = 0$, $[p^{(n)}] = [p]$
- At $n = 1$, $[p^{(n)}] = \left[\frac{\partial p}{\partial \xi}\right] = \left[\frac{\partial p}{\partial x}\right] x_\xi$
  It is important to be very careful here because since we are using a finite different scheme for uniform grids(ξ & η), we must use $\left[\frac{\partial p}{\partial \xi}\right]$ instead of $\left[\frac{\partial p}{\partial x}\right]$
- At $n = 2$, $[p^{(n)}] = \left[\frac{\partial}{\partial \xi}\frac{\partial p}{\partial \xi}\right] = \left[\frac{\partial}{\partial \xi}\frac{\partial p}{\partial x}\right] x_\xi = \left[\frac{\partial p}{\partial x}\right] x_{\xi\xi} + \left[\frac{\partial^2 p}{\partial x^2}\right] (x_\xi)^2$
  Similar as $n = 1$, we have to use $\frac{\partial}{\partial \xi}\left[\frac{\partial p}{\partial \xi}\right]$ instead of $\left[\frac{\partial^2 p}{\partial x^2}\right]$

With $\left(\frac{\partial p}{\partial x}\right)_{i+\frac{1}{2}}$ and $\left(\frac{\partial p}{\partial x}\right)_{i-\frac{1}{2}}$ known, equation (5.14) can be expressed as

$$\begin{aligned}\left(\frac{\partial^2 p}{\partial x^2}\right)_i &= (\xi_x)_i \frac{\partial}{\partial \xi}\left(\frac{\partial p}{\partial x}\right)_i \\ &= \frac{(\xi_x)_i}{\Delta\xi}\left[\left(\frac{\partial p}{\partial x}\right)_{i+\frac{1}{2}} - \left(\frac{\partial p}{\partial x}\right)_{i-\frac{1}{2}} + \left(-\sum_{n=0}^{2}\frac{\left[p_x^{(n)}(b)\right]}{n!}\left(\xi_{i-\frac{1}{2}} - x2\xi(b)\right)^n - \sum_{n=0}^{2}\frac{\left[p_x^{(n)}(c)\right]}{n!}\left(\xi_{i+\frac{1}{2}} - x2\xi(c)\right)^n\right)\right]\end{aligned} \tag{5.16}$$

- At $n = 0$, $\left[p_x^{(n)}\right] = \left[\frac{\partial p}{\partial x}\right]$

As the equation above is simply the finite difference scheme to use on $\frac{\partial p}{\partial x}$, so here at $n = 0$ we should use $\left[\frac{\partial p}{\partial x}\right]$ but not $[p]$.

- At $n = 1$, $\left[p_x^{(n)}\right] = \frac{\partial}{\partial \xi}\left[\frac{\partial p}{\partial x}\right] = \frac{\partial}{\partial x}\left[\frac{\partial p}{\partial x}\right] x_\xi = \left[\frac{\partial^2 p}{\partial x^2}\right] x_\xi$
  Similarly at $n = 1$, we must find first order derivative of $\frac{\partial p}{\partial x}$ on uniform grids ξ, $\frac{\partial}{\partial \xi}\left(\frac{\partial p}{\partial x}\right)$ should be used but not $\frac{\partial p}{\partial x}$ or $\frac{\partial^2 p}{\partial x^2}$.

- At $n = 2$, we set $\left[p_x^{(n)}\right] = 0$, because we didn't calculate third-order Cartesian jump conditions and second order is enough.

Now we have successfully conducted first and second order central finite difference schemes with incorporation of jump conditions under stretched mesh. Equations (5.15) and (5.16) can also be applied to the calculation of jump contributions for velocity, divergence and strain.

5.6.2. Jump contribution of interpolation

Assume function $g(\xi)$ is defined on Cartesian grids and has jump condition at D, as shown in Figure 5.6, then the interpolation at point B will be calculated from points A, D, C, which is shown as below

$$g_B = \frac{g_A + g_c}{2} + \frac{1}{2}[g_D] - \frac{1}{2}\left[\frac{\partial g_D}{\partial z}\right] h^- + \frac{1}{4}\left[\frac{\partial^2 g_D}{\partial z^2}\right] (h^-)^2 + O(h^2) \tag{5.17}$$

$$g_B = \frac{g_A + g_c}{2} - \frac{1}{2}[g_D] - \frac{1}{2}\left[\frac{\partial g_D}{\partial z}\right] h^+ - \frac{1}{4}\left[\frac{\partial^2 g_D}{\partial z^2}\right] (h^+)^2 + O(h^2) \tag{5.18}$$

The above equations are developed for a uniform mesh. The first equation is for D between A and B, and second equation is for D between B and C. When we compute interpolation g_B using stretched mesh, we are actually computing g_B under uniform mesh. The values is same though under different coordinates. But for the jump conditions, it will be different. First let's see the interpolation equations under uniform mesh,

$$g_B = \frac{g_A + g_c}{2} + \frac{1}{2}[g_D] - \frac{1}{2}\left[\frac{\partial g_D}{\partial \xi}\right] \xi^- + \frac{1}{4}\left[\frac{\partial^2 g_D}{\partial \xi^2}\right] (\xi^-)^2 + O(\xi^2) \tag{5.19a}$$

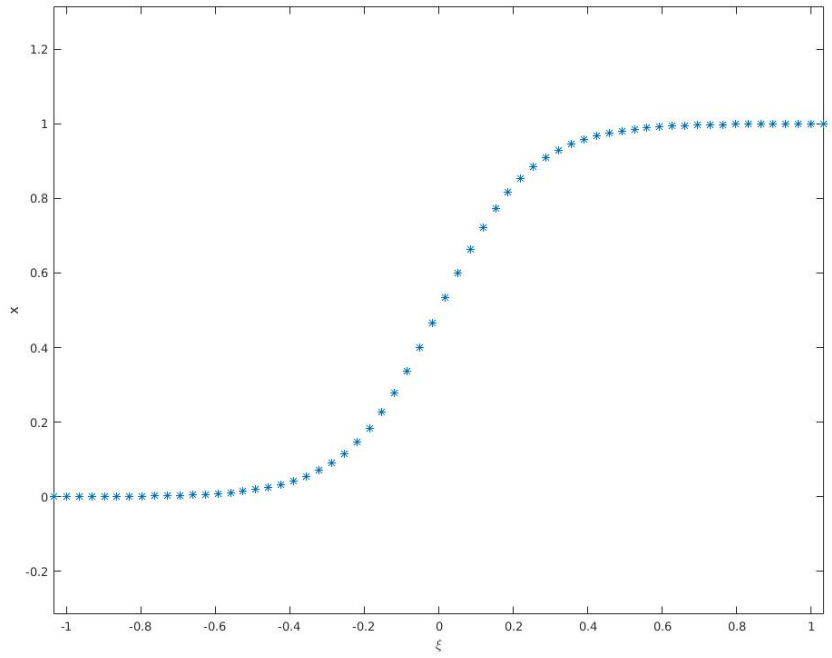
$$g_B = \frac{g_A + g_c}{2} - \frac{1}{2}[g_D] - \frac{1}{2}\left[\frac{\partial g_D}{\partial \xi}\right]\xi^+ - \frac{1}{4}\left[\frac{\partial^2 g_D}{\partial \xi^2}\right](\xi^+)^2 + O(\xi^2) \tag{5.19b}$$

As shown above, we need $[g_D]$, $\left[\frac{\partial g_D}{\partial \xi}\right]$, ξ^-, $\left[\frac{\partial^2 g_D}{\partial \xi^2}\right]$ and ξ^+. For ξ^+ and ξ^-, and we also need use function $x2\xi$ to transform grid coordinates from x to ξ. For $\left[\frac{\partial g_D}{\partial \xi}\right]$ and $\left[\frac{\partial^2 g_D}{\partial \xi^2}\right]$, we will apply equations (5.4). Finally we can derive the equation below for interpolation,

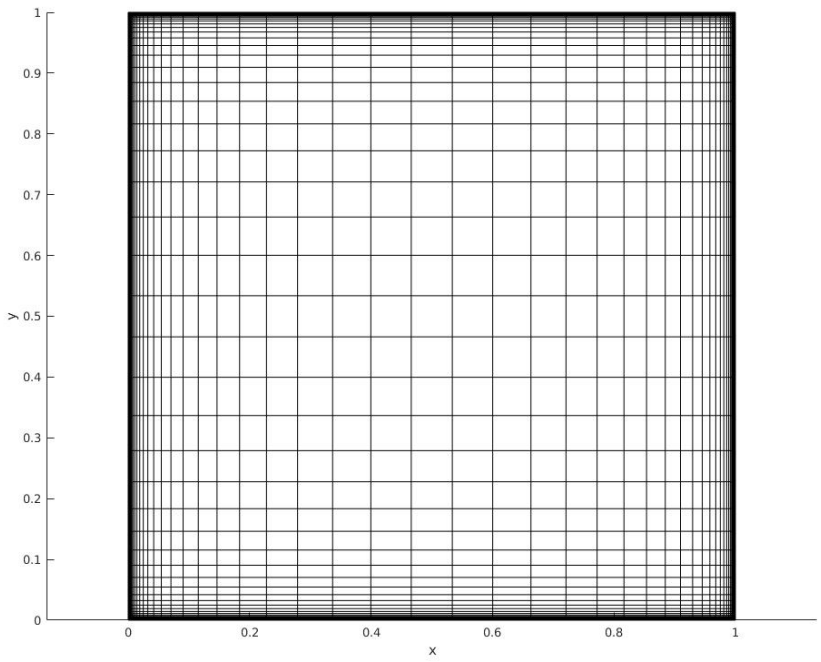
$$\begin{aligned} g_B &= \frac{g_A + g_c}{2} + \frac{1}{2}[g_D] - \frac{1}{2}\left[\frac{\partial g_D}{\partial \xi}\right]\xi^- + \frac{1}{4}\left[\frac{\partial^2 g_D}{\partial \xi^2}\right](\xi^-)^2 + O(\xi^2) \\ &= \frac{g_A + g_c}{2} + \frac{1}{2}[g_D] - \frac{1}{2}x_\xi(D)\left[\frac{\partial g_D}{\partial x}\right](x2\xi(D) - x2\xi(A)) \\ &+ \frac{1}{4}(x2\xi(D) - x2\xi(A))^2\left((x_\xi(D))^2\left[\frac{\partial^2 g_D}{\partial x^2}\right] + x_{\xi\xi}(D)\left[\frac{\partial g_D}{\partial x}\right]\right) + O(\xi^2), A < D < B \end{aligned} \tag{5.20a}$$

$$\begin{aligned} g_B &= \frac{g_A + g_c}{2} - \frac{1}{2}[g_D] - \frac{1}{2}\left[\frac{\partial g_D}{\partial \xi}\right]\xi^+ - \frac{1}{4}\left[\frac{\partial^2 g_D}{\partial \xi^2}\right](\xi^+)^2 + O(\xi^2) \\ &= \frac{g_A + g_c}{2} - \frac{1}{2}[g_D] - \frac{1}{2}x_\xi(D)\left[\frac{\partial g_D}{\partial x}\right](x2\xi(C) - x2\xi(D)) \\ &- \frac{1}{4}(x2\xi(C) - x2\xi(D))^2\left((x_\xi(D))^2\left[\frac{\partial^2 g_D}{\partial x^2}\right] + x_{\xi\xi}(D)\left[\frac{\partial g_D}{\partial x}\right]\right) + O(\xi^2), B < D < C \end{aligned} \tag{5.20b}$$

The above equations are mainly used for interpolation of velocity at center and edge points of the MAC cell.



(a) Stretched mesh $X = X(\xi)$, $c = 3$



(b) Stretching Cartesian Grids

Figure 5.4: Stretched mesh

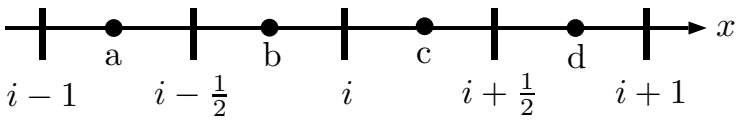


Figure 5.5: $p = p(x(\xi))$ with jumps on a, b, c, d

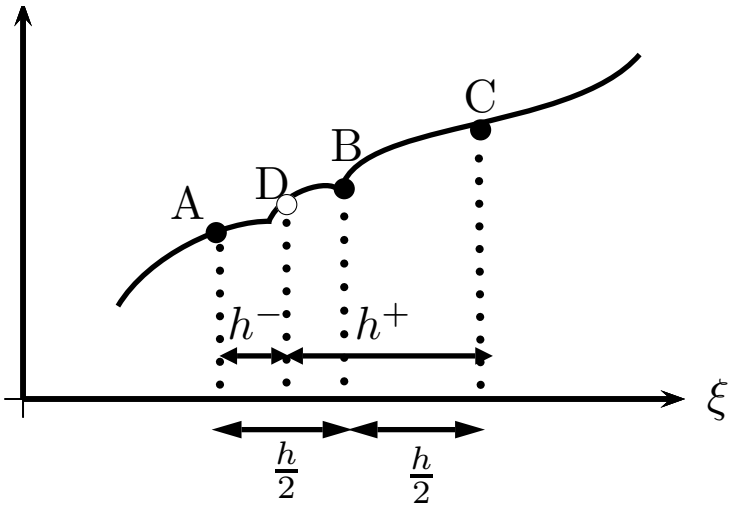


Figure 5.6: $g = g(\xi)$ with jumps at D

Chapter 6

NUMERICAL SIMULATIONS

In section, we present results of several numerical simulations to demonstrate the accuracy, robustness and efficiency of our method. In the development of our program, we first developed a serial program to prove our method is accurate and valid, then we developed a parallel program to improve the robustness and efficiency further. In this chapter, simulation results will be marked as serial or parallel. As mentioned in Chapter 1, all the parallel tests were simulated on SMU high-performance computing facility ManeFrame. The tests are as following:

- Poisson solver with jump conditions
- Lid-driven cavity flow
- Circular Couette flow
- Flow past single stationary circular cylinder
- Flow past single stationary square cylinder
- Flow past two square cylinders in tandem arrangement
- Flow around single hovering flapper
- Flow around multiple hovering flappers
- Cylinders rotating along a circle
- Flow past stationary triangle cylinder
- Flow past SMU mascot Peruna

6.1. Poisson Solver With Jump Conditions

In this section, the test of Poisson solver with involvement of jump conditions is presented. Previously in Xu and Pearson [76], they have developed the method for computing necessary Cartesian jump conditions in 3D using a triangular mesh, and simulation results of the Poisson solver was given to show the accuracy. In section 3.2, we have showed how to compute the necessary principle and Cartesian jump conditions in 2D using line segment representation for interfaces. In order to test the accuracy of the jump conditions and Poisson solver, a similar simulation is tested here. Figure 6.1 shows the geometry of this test. The circle and square in the middle are the objects we want to test. The size of computational domain is $[-3,3] \times [-3,3]$. The radius of the circular cylinder is 0.5 and the side length of square is 1. Assume Γ is the object boundary, then we construct the function $p(x,y)$ and it is discontinuous across the boundary Γ, which is shown as below

$$p(x,y) = \begin{cases} \sin(x)\sin(y), & outside\ \Gamma \\ e^{-(x+y)}, & inside\ \Gamma \end{cases} \tag{6.1}$$

We used this analytical expression for p to compute principle and Cartesian jump conditions, and then use the jump conditions to calculate jump contributions and finally solve the Poisson equation (4.5). By comparing the results with the analytical expression for p, we can find the accuracy of our method for computing the jump conditions and Poisson solver. Table 6.1 and 6.2 present the results and second order convergence is achieved in all tests. For the tests in table 6.2, the stretching method is $x(\xi) = \frac{3tanh(c\xi)}{tanh(3c)}$, c is constant to control the stretching ratio.

6.2. Lid-driven Cavity Flow

In the first example we showed the accuracy of the Poisson solver and here we want to examine the validation of our method for a flow problem without any object. 2D lid-driven cavity flow is a very good example. Assume there is a box full of fluid, and on the top of

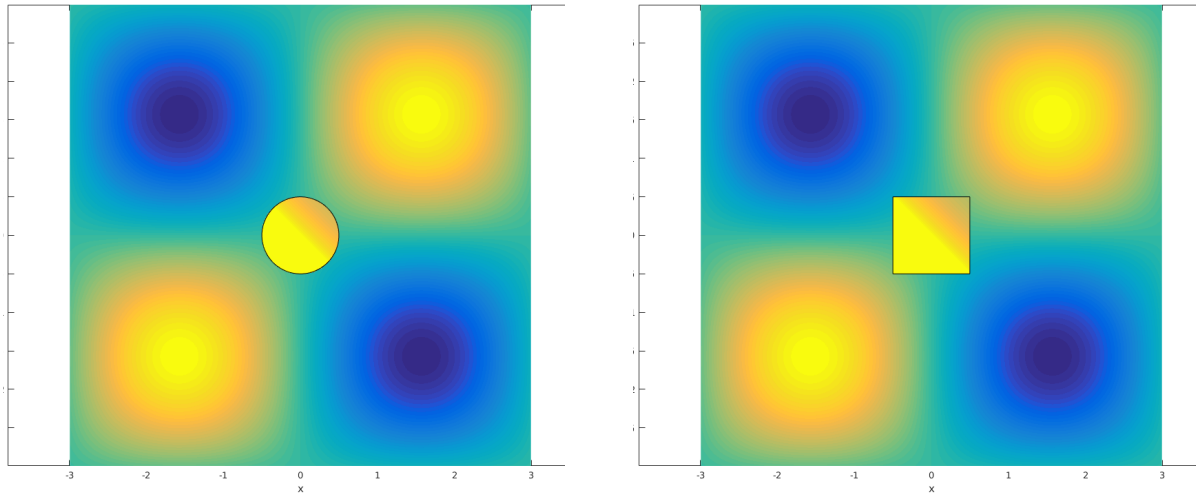


Figure 6.1: Geometry of Poisson solver test

box there is a lid driving the fluid moving from right to left. After some time, the motion of fluid inside the box will reach to a steady state.

6.2.1. Validation

For the validation test, the geometry is shown in Figure 6.2. In the test, the computational domain was $[0,1] \times [0,1]$. For velocity, Dirichlet boundary condition was applied on all sides. On the left, right and bottom, the boundary condition of velocity was 0. On top, $u = -1$ and $v = 0$. For pressure, Neumann boundary condition was applied on all sides. The simulation was tested at $Re = 100$ & 1000, and 4 cores were used. For $Re = 100$, 65×65 and 129×129 grids were used. For $Re = 1000$, 129×129 and 161×161 grids were used. The simulation results are given in table 6.3 and 6.4 and the velocity extrema along the center lines are shown, which are the key characteristics for the cavity flow. u_{max} is the maximum of u on the vertical line where $x = 0.5$ and y_{max} is its location. Similarly, v_{max} is the maximum of v on the horizontal line where $y = 0.5$ and x_{max} is its location, v_{min} is the minimum of v

Circular Cylinder						
Grid	Uniform			Non-uniform		
	Δx	Error	Convergence	$\Delta \xi$	Error	Convergence
129	0.04688	8.99E-04		0.04688	7.99E-04	
257	0.02344	2.26E-04	1.992	0.02344	2.07E-04	1.950
513	0.01172	5.64E-05	2.004	0.01172	5.33E-05	1.956
1025	0.00586	1.42E-05	1.992	0.00586	1.31E-05	2.021
2049	0.00293	3.61E-06	1.974	0.00293	3.16E-06	2.056

Table 6.1: Poisson solver test with circular cylinder, 4 cores

Square Cylinder						
Grid	Uniform			Non-uniform		
	Δx	Error	Convergence	$\Delta \xi$	Error	Convergence
129	0.04688	1.70E-03		0.04688	1.57E-03	
257	0.02344	3.98E-04	2.097	0.02344	3.25E-04	2.271
513	0.01172	1.09E-04	1.873	0.01172	1.01E-04	1.689
1025	0.00586	2.56E-05	2.087	0.00586	2.63E-05	1.936
2049	0.00293	6.83E-06	1.906	0.00293	6.41E-06	2.039

Table 6.2: Poisson solver test with square cylinder, 4 cores

on the horizontal line where $y = 0.5$ and x_{min} is its location. The stretching method here was $x(\xi) = \frac{tanh(c(\xi-0.5))}{2tanh(0.5c)} + 0.5$. By comparing the results with other people's work, good agreement can be found. Figure 6.3 gives the stream function of the lid-driven cavity flow at $Re = 1000$.

6.2.2. Parallel speedup and efficiency

Since our final goal is to develop an accurate, efficient and robust parallel program, we need to look at the parallel speedup and efficiency of our program. In this test, 1 to 256 cores were used. The computational domain was $[0, 10] \times [0, 10]$ with a 1601×1601 grid. 2000 time steps were used for $Re = 1000$. The boundary condition was same as the validation

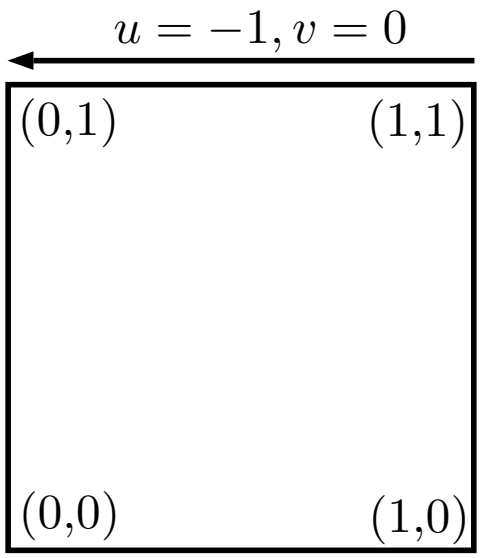


Figure 6.2: Geometry of lid-driven cavity flow

Reference	Grid	u_{max}	y_{max}	v_{max}	x_{max}	v_{min}	x_{min}
Ref. [5]	64x64	0.214042	0.4581	0.1795728	0.7630	-0.2538030	0.1896
Ref. [15]	64x64	0.21315	-	0.17896	-	-0.25339	-
Present	65x65	0.213179	0.453846	0.179253	0.761538	-0.253665	0.192308
Ref. [23]	129x129	0.21090	0.4531	0.17527	0.7656	-0.24533	0.1953
Ref. [10]	129x129	0.2106	0.4531	0.1786	0.7656	-0.2521	0.1875
Present	129x129	0.213741	0.46124	0.17943	0.763566	-0.253428	0.189922

Table 6.3: Lid-driven cavity flow at $Re = 100$

test. Figure 6.4 shows the total computational time(seconds) of the parallel program and the computational time which the SMG solver takes during the simulation. Figure 6.5 and table 6.5 show the parallel speedup of our program. Figure 6.6 and table 6.6 show the parallel efficiency of the program and percentage of SMG time compared with total computational time. The parallel speedup and efficiency are computed by the formulas below:

$$speedup = \frac{sequential\ time}{parallel\ time} \tag{6.2a}$$

$$efficiency = \frac{speedup}{processors} \tag{6.2b}$$

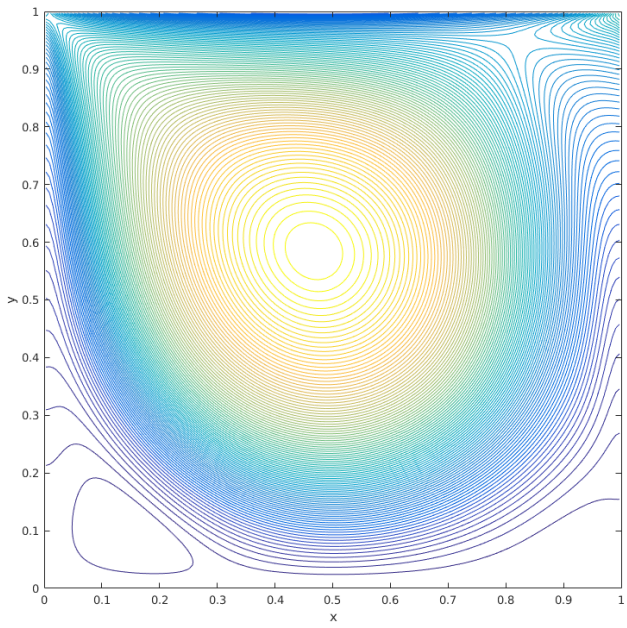


Figure 6.3: Stream function of lid-driven cavity flow

As we can find from Figure 6.5 and table 6.5, for 1 to 8 cores, the parallel speedup for total computational time increased slowly. For 8 to 80 cores, the speedup was increasing, and then for more than 80 cores it was decreasing. The highest speedup is 12.4633. The same result also happens to the SMG solver. In this set of tests here, we compared the SMG computational time with the total time. For every single test, as we can find from table 6.6 and figure 6.6, the SMG solver takes more than 90% of the total computational time. Since SMG is an iterative solver, it is reasonable to take more computational time. Besides, there could be communication between different cores inside the SMG solver, but since we are not familiar with the coding details of SMG, we cannot record the iterative time, communication time and time for other possible computation from the SMG solver. Our own communication between ghost layers only takes up to 0.52% of the total time, which is very efficient. Besides, figure 6.6 shows us that the overall efficiency is almost identical to the efficiency of SMG solver. We can see the speedup and efficiency of our method is mainly dependent on the SMG solver.

Reference	Grid	u_{max}	y_{max}	v_{max}	x_{max}	v_{min}	x_{min}
Ref. [5]	128x128	0.3885698	0.1717	0.3769447	0.8422	-0.5270771	0.0908
Ref. [23]	129x129	0.38289	0.1719	0.37095	0.8437	-0.51550	0.0937
Present	129x129	0.385671	0.174419	0.374273	0.841085	-0.523069	0.096899
Present*	129x129	0.394481	0.174858	0.383777	0.84033	-0.534601	0.093218
Ref. [5]	160x160	0.3885698	0.1717	0.3769447	0.8422	-0.5270771	0.0908
Present	161x161	0.386791	0.177019	0.375471	0.841615	-0.524442	0.0900621
Present*	161x161	0.396790	0.174384	0.386014	0.843847	-0.537264	0.091206

*: stretching mesh was used.

Table 6.4: Lid-driven cavity flow at $Re = 1000$

6.2.3. Scalability Test

In this part, we tested our parallel program's scalability using large number of cores. The computational domain is $[0, 64] \times [0, 64]$, and 8193×8193 grid points were used, 16 to 1024 of cores were tested. The results is shown in table 6.7. We can find that SMG solver takes more than 90% of the total computational time in each test and the parallel program is stable to use up to 1024 cores. It is difficult to find any trend on the computational time when increasing the number of cores.

6.3. Circular Couette Flow

In this section, the simulation of steady circular Couette flow is given. The simulation results can help us examine our method's accuracy and convergence when an object is involved. In this test, we assume there are two circular cylinders rotating at different speeds and there is fluid between the cylinders. The geometry is shown in Figure 6.7. Here r is the radius of the circular cylinder and Π is the angular velocity of the cylinder rotation. In this test, $r_1 = 0.5$ and $r_2 = 2$, $\Pi_1 = 1$ and $\Pi_2 = -1$. The actual computational domain was the rectangular $l_x \times l_y$, where $l_x = l_y = 2$. Only the inside circular cylinder was included in the computational domain. On the boundary, analytical Dirichlet boundary conditions were used for velocity and analytical Neumann boundary conditions were used for pressure.

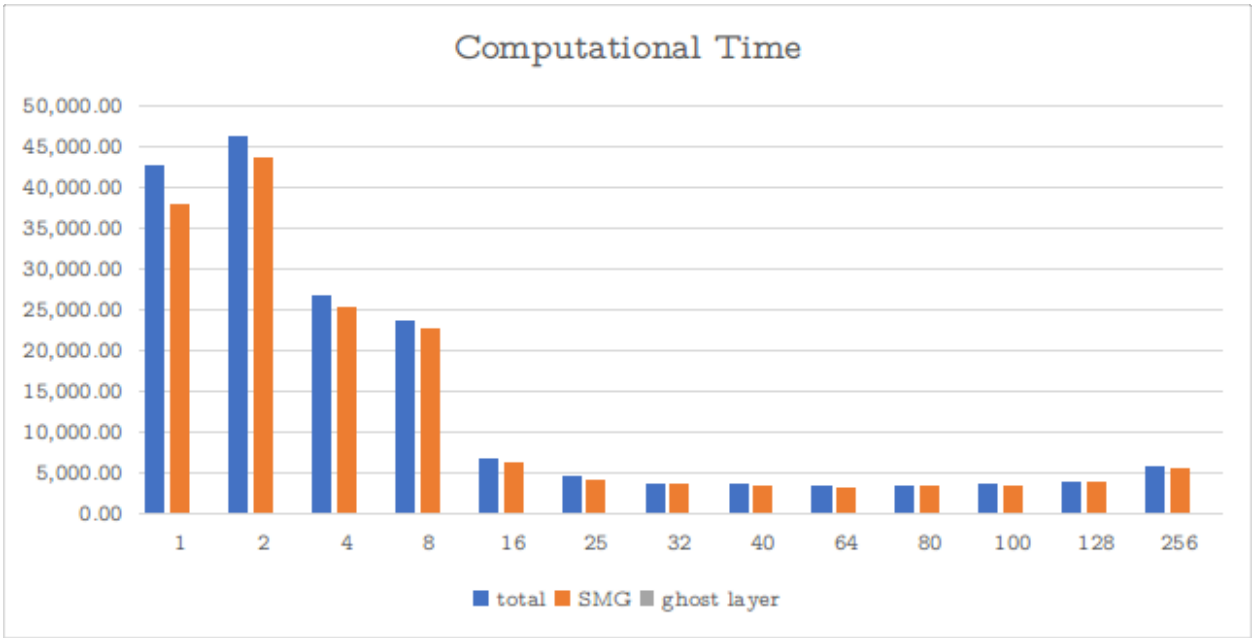


Figure 6.4: Computational time for cavity flow with different number of processors

The analytical solution of the flow between the two cylinders can be found and is given below

$$u = -\left(A_1 + \frac{A_2}{r^2}\right) y, \tag{6.3a}$$

$$v = \left(A_1 + \frac{A_2}{r^2}\right) x, \tag{6.3b}$$

$$p = \frac{A_1^2 r^2}{2} - \frac{A_2^2}{2r^2} + A_1 A_2 \cdot ln(r^2) + p_c, \tag{6.3c}$$

where $A_1 = \frac{\Pi_2 r_2^2 - \Pi_1 r_1^2}{r_2^2 - r_1^2}$, $A_2 = \frac{(\Pi_1 - \Pi_2) r_1^2 r_2^2}{r_2^2 - r_1^2}$, $r^2 = x^2 + y^2$, and p_c is an arbitrary constant. As the analytical solution is already known, we can compare our computational results to verify our method's accuracy and order of convergence. The Reynolds number used here was 10.

On the computational domain boundary B, we used Dirichlet boundary conditions for velocity and Neumann boundary conditions for pressure. Table 6.8 and 6.9 present the convergence analysis. From the tables we can find second order convergence is achieved for u and v by increasing the number of grid points, and first order convergence is achieved for p. The stretching method used in Table 6.9 was $x(\xi) = \frac{sinh(c\xi)}{2sinh(c)}$.

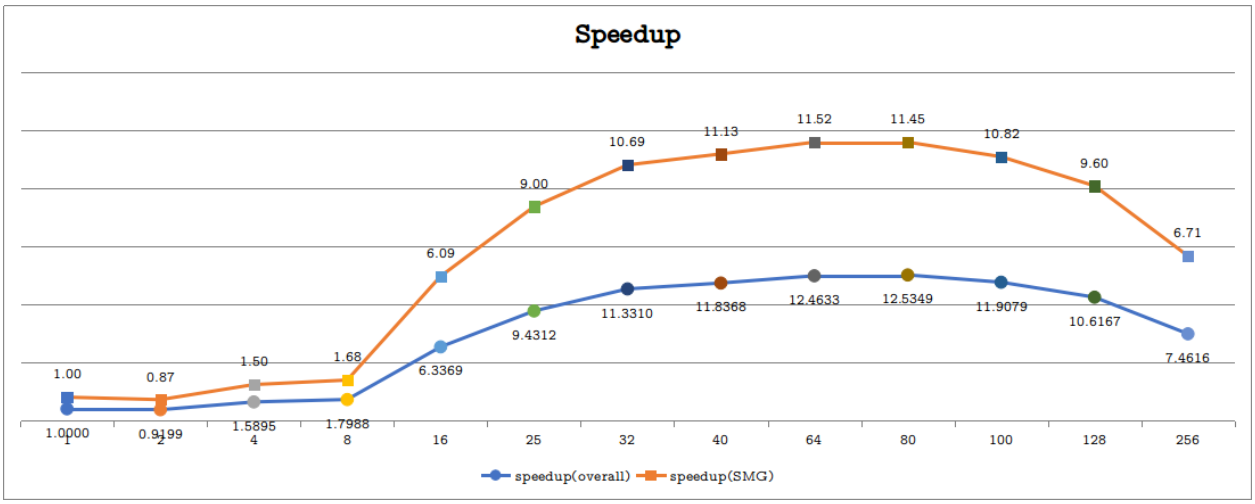


Figure 6.5: Cavity flow parallel speedup

6.4. Flow Past Circular Cylinder

In this section, the test of flow past a stationary circular cylinder is presented. Many researchers have been working on this problem and numerous experimental and numerical experiments have been conducted. This test can help us check the validation of our method.

6.4.1. Geometry of the computational domain

In this part, flow past a stationary circular cylinder was tested at $Re = 20$, 40, 100 and 200, and the geometry is shown in Figure 6.8. The domain size was $[-8, 24] \times [-8, 8]$. The spatial resolution for $Re = 20$ and 40 was $N_x \times N_y \times M_s = 961 \times 481 \times 256$, and the spatial resolution for $Re = 100$ and 200 was $N_x \times N_y \times M_s = 1601 \times 801 \times 256$. N_x and N_y was the number of grid points on x and y directions and M_s is the number of vertices on object boundary. The CFL coefficient for time step was set as $CFL_c = CFL_v = 0.5$.

6.4.2. Boundary conditions

The initial setup for flow field was uniform flow with $u = 1$ in the x direction and $v = 0$ in y direction.

- Inlet: $u = 1$, $v = 0$ and $\frac{\partial p}{\partial x} = \frac{1}{Re}\frac{\partial^2 u}{\partial x^2}$.

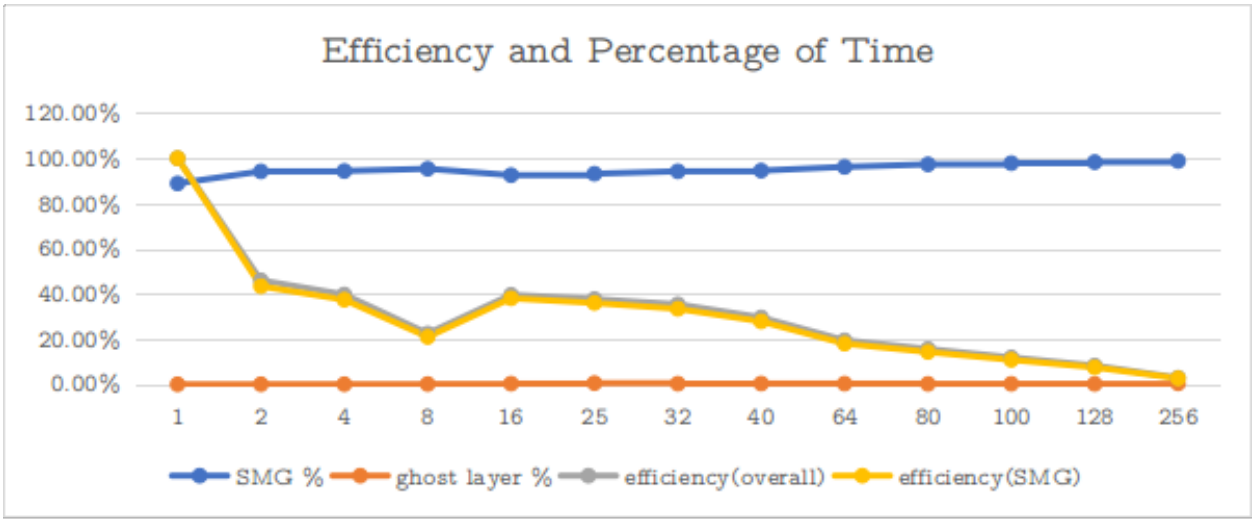


Figure 6.6: Cavity flow parallel efficiency and percentage of total time

- Outlet: $\frac{\partial u}{\partial x} = 0$, $\frac{\partial v}{\partial x} = 0$ and $\frac{\partial p}{\partial x} = \frac{1}{Re}\frac{\partial^2 u}{\partial x^2}$.
- Top: $\frac{\partial u}{\partial y} = 0$, $\frac{\partial v}{\partial x} = 0$ and $\frac{\partial p}{\partial x} = \frac{1}{Re}\frac{\partial^2 u}{\partial y^2}$.
- Bottom: $\frac{\partial u}{\partial y} = 0$, $\frac{\partial v}{\partial x} = 0$ and $\frac{\partial p}{\partial x} = \frac{1}{Re}\frac{\partial^2 u}{\partial y^2}$.

6.4.3. $Re = 20, 40$

At $Re = 20$ and 40, the flow separates when meets the object and reattaches behind the object in some distance. Figure 6.9 is the stream function contours for flow past cylinder at $Re = 20$. Behind the cylinder, two recirculation vortices are formed in the wake, where the length L and a, b are defined. This phenomena is as expected, which has been shown in [36]. At $Re = 20$ & 40, the lift coefficient approaches to 0. The drag coefficient C_d is a constant as the flow is steady. In Table 6.10, C_d, L, a, b are compared with other experimental and numerical results. Good agreement can be seen.

6.4.4. $Re = 100, 200$

At $Re = 100$ & 200, the flow is unsteady and the famous Von Karman vortex street is formed in the wake as expected, which can be observed from Figure 6.10. Table 6.11 compares the results of drag coefficient C_d, lift coefficient C_l and Strouhal number S_t with previous

processors	speedup(overall)	efficiency(overall)	speedup(SMG)	efficiency(SMG)
1	1.0000	100.00%	1.0000	100.00%
2	0.9185	45.92%	0.8646	43.23%
4	1.5895	39.74%	1.4968	37.42%
8	1.7988	22.49%	1.6765	20.96%
16	6.3369	39.61%	6.0870	38.04%
25	9.4312	37.72%	8.9970	35.99%
32	11.3310	35.41%	10.6870	33.40%
40	11.8368	29.59%	11.1277	27.82%
64	12.4633	19.47%	11.5154	17.99%
80	12.5349	15.67%	11.4484	14.31%
100	11.9079	11.91%	10.8164	10.82%
128	10.6167	8.29%	9.5986	7.50%
256	7.4616	2.91%	6.7116	2.62%

Table 6.5: Cavity flow parallel speedup and efficiency

numerical results, which shows our results are within good ranges. Figure 6.11 shows the drag and lift coefficients evolve with time at $Re = 100$ and $Re = 200$. As we can tell from Figure 6.11, the drag coefficient frequency is twice as the lift coefficient frequency, which is the same as the vortex shedding frequency or Strouhal frequency. This observation also matches our expectation. The vortices appear in two lines in the wake, which is unsteady and asymmetric. Each of the upper and lower vortices will bring one drag period to the object separately but together they only bring one lift period to the object. We can also notice that the average drag coefficient is non-zero but the average lift coefficient is zero.

6.5. Flow Past Square Cylinder

In this section we present the simulation of flow past a stationary square cylinder. As the main purpose of our new method is simulating flow past objects with non-smooth boundaries, square cylinder is a good example. The geometry of this test is shown in Figure 6.8. We have tested low Reynolds numbers Re up to 100 under different blockage ratios B, where B

processors	total(s)	SMG(s)	SMG %	ghost layer(s)	ghost layer %
1	42659.55	37912.52	88.87%	0.04	0.00%
2	46372.01	43681.23	94.20%	17.87	0.04%
4	26838.74	25329.13	94.38%	11.30	0.04%
8	23715.16	22614.28	95.36%	30.89	0.13%
16	6731.97	6228.43	92.52%	19.84	0.29%
25	4523.24	4213.93	93.16%	23.56	0.52%
32	3764.87	3547.54	94.23%	10.58	0.28%
40	3603.98	3407.03	94.54%	12.36	0.34%
64	3422.80	3292.33	96.19%	10.67	0.31%
80	3403.27	3311.61	97.31%	6.62	0.19%
100	3582.46	3505.11	97.84%	7.80	0.22%
128	4018.14	3949.78	98.30%	7.23	0.18%
256	5717.21	5648.82	98.80%	24.73	0.43%

Table 6.6: Cavity flow percentage of time

is ratio of the symmetric square edge length to the domain width, $B = \frac{D}{H}$. The blockage ratio is a key parameter for researchers to compare numerical and experimental results.

6.5.1. Numerical tests setup

- $B = 0.05, 0.0625, 0.067$

  The domain size was $[-10, 30] \times [-\frac{D}{2B}, \frac{D}{2B}]$ and spatial resolution was $N_x \times N_y \times M_s = 1200 \times \frac{30D}{B} \times 256$.

- CFL coefficients for time step

  The CFL coefficients for time step was set as $CFL_c = CFL_v = 0.5$.

6.5.2. Boundary conditions

The initial setup for flow field was uniform flow with $u = 1$ in the x direction and $v = 0$ in y direction.

Cores	total	SMG	SMG %	Ghost Layers	Ghost Layers %
16	29585.72	26954.63	91.11%	14.50	0.05%
32	14115.29	12789.52	90.61%	10.30	0.07%
64	7455.57	6730.72	90.28%	8.12	0.11%
128	4789.93	4401.82	91.90%	16.12	0.34%
256	3359.69	3122.98	92.95%	18.39	0.55%
400	11226.45	10725.75	95.54%	280.89	2.50%
512	13221.24	11960.81	90.47%	912.41	6.90%
625	18440.04	18211.99	98.76%	55.37	0.30%
800	24727.19	23631.67	95.57%	408.81	1.65%
1000	5802.49	5601.13	96.53%	67.62	1.17%
1024	12766.39	12560.84	98.39%	47.12	0.37%

Table 6.7: Scalability test for Cavity flow

n	Δx	$\|\|e_u\|\|_\infty$	order	$\|\|e_v\|\|_\infty$	order	$\|\|e_p\|\|_\infty$	order
31	0.1290	3.91×10^{-2}	-	4.03×10^{-2}	-	2.50×10^{-2}	-
61	0.0656	7.24×10^{-3}	2.49	7.20×10^{-3}	2.54	1.63×10^{-2}	0.63
121	0.0331	1.71×10^{-3}	2.11	1.73×10^{-3}	2.09	8.29×10^{-3}	0.99
241	0.0166	3.81×10^{-4}	2.18	3.81×10^{-4}	2.19	4.70×10^{-3}	0.82

Table 6.8: Circular Couette flow at $Re = 10$, uniform mesh, 4 cores

- Inlet: $u = 1$, $v = 0$ and $\frac{\partial p}{\partial x} = \frac{1}{Re}\frac{\partial^2 u}{\partial x^2}$.
- Outlet: $\frac{\partial u}{\partial x} = 0$, $\frac{\partial v}{\partial x} = 0$ and $\frac{\partial p}{\partial x} = \frac{1}{Re}\frac{\partial^2 u}{\partial x^2}$.
- Top: $u = 1$, $v = 0$ and $\frac{\partial p}{\partial x} = \frac{1}{Re}\frac{\partial^2 u}{\partial y^2}$.
- Bottom: $u = 1$, $v = 0$ and $\frac{\partial p}{\partial x} = \frac{1}{Re}\frac{\partial^2 u}{\partial y^2}$.

6.5.3. $Re < 100$

At $Re < 40$, the flow reaches a steady state. Figure 6.12 is stream function contours for flow past cylinder at $Re = 1.5$, 5, 20 & 40. Similar to the circular cylinder, for $Re = 20$

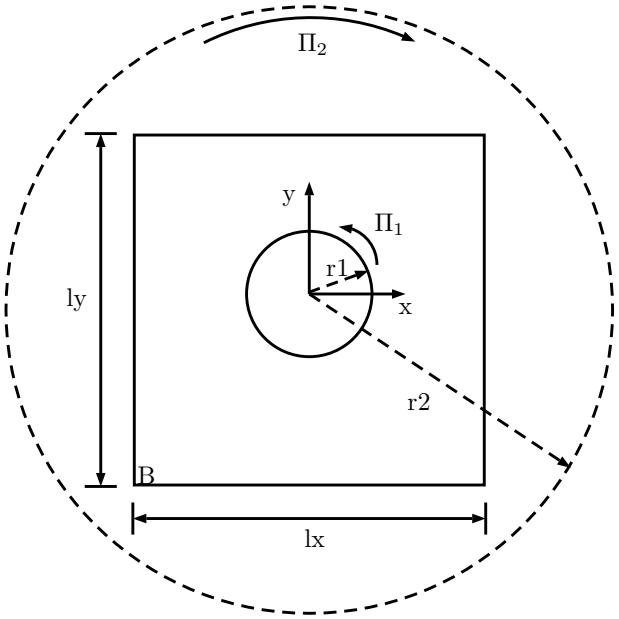


Figure 6.7: Geometry of circular Couette flow

n	$\Delta\xi$	$\|\|e_u\|\|_\infty$	order	$\|\|e_v\|\|_\infty$	order	$\|\|e_p\|\|_\infty$	order
31	0.0645	4.37×10^{-2}	-	5.35×10^{-2}	-	7.55×10^{-2}	-
61	0.0328	1.16×10^{-2}	1.96	1.16×10^{-2}	2.25	3.24×10^{-2}	1.25
121	0.0165	1.69×10^{-3}	2.81	1.71×10^{-3}	2.79	1.69×10^{-2}	0.95
241	0.0083	3.88×10^{-4}	2.14	3.89×10^{-4}	2.16	7.93×10^{-3}	1.10

Table 6.9: Circular Couette flow at $Re = 10$, stretching mesh, 4 cores

& 40, the flow separates and two symmetric recirculation vortices are formed in the wake behind the square cylinder. The bubble size increases with larger Reynolds number. From the study of Sen [54], steady flow starts separating at $Re \geq 1.17$. In our example as shown in Figure 6.12, we didn't reproduce the same results for $B = 0.05$, 0.0625 and 0.067. From the work of Breuer [7], separation can be observed at $Re \sim 5$ similar to cylinder case. In our method the separation was observed at Re between 7 and 8.

At $Re < 40$, the lift coefficient approaches to 0 and the drag coefficient C_d is constant. L is defined same as circular cylinder example. In Table 6.12, L/D and C_d are compared with other experimental and numerical results. Similar as circular cylinder case, the results from

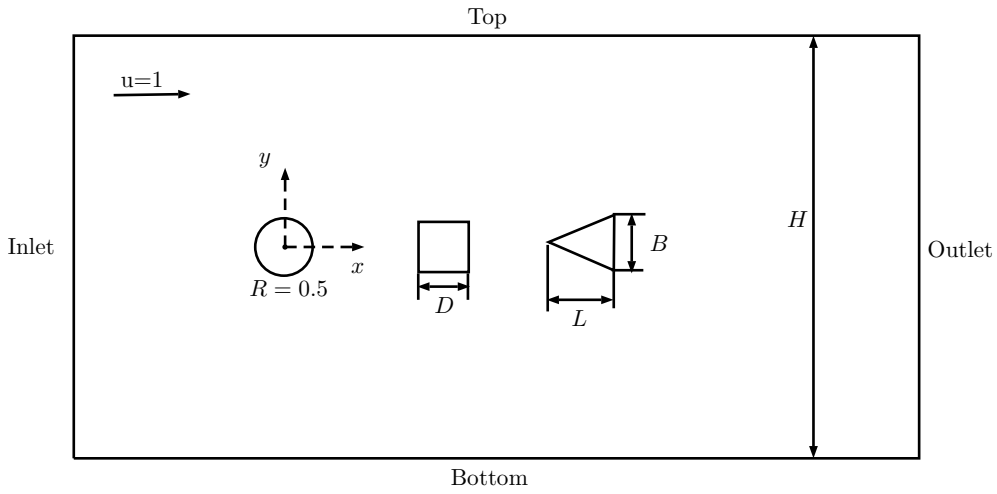


Figure 6.8: Geometry of flow past stationary circular/square/triangular cylinder

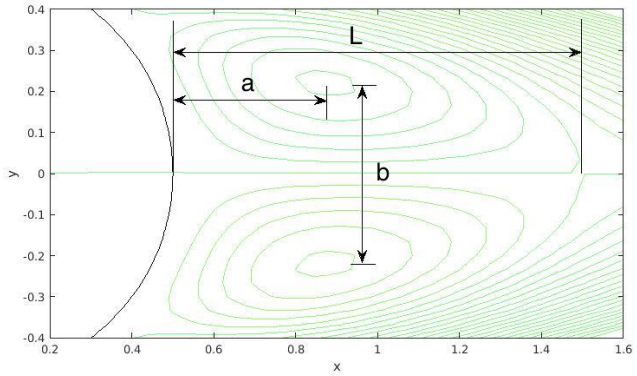


Figure 6.9: Streamfunction contours at $Re = 20$, serial

the parallel program are slightly bigger than results from serial program and other people's work. This is also due to the asymmetry of the parallel program, which I will explain later. Except this, all the results are in a reasonable range and good agreement can be seen.

6.5.4. $Re = 100, 200$

At $Re = 100$ & 200, similar to the circular cylinder example, the flow is unsteady and the famous Von Karman vortex street is formed in the wake as expected, shown in Figure 6.14. Table 6.13 compares the results of average drag coefficient $\bar{C_d}$ and Strouhal number S_t at different blockage ratio B with other people's work, which shows good agreement. Figure

	$Re = 20$				$Re = 40$			
	L	a	b	C_d	L	a	b	C_d
Ref. [63]	-	-	-	2.22	-	-	-	1.48
Ref. [14]	0.93	0.33	0.46	0	2.13	0.76	0.59	-
Ref. [16]	0.94	-	-	2.05	2.35	-	-	1.52
Ref. [22]	0.91	-	-	2.00	2.24	-	-	1.50
Ref. [77]	0.92	-	-	2.23	2.21	-	-	1.66
Ref. [37]	0.93	0.36	0.43	2.16	2.23	0.71	0.59	1.61
Present(serial)	0.98	0.37	0.43	2.06	2.4	0.74	0.6	1.56
Present(parallel)	0.98	0.37	0.43	2.09	2.4	0.74	0.6	1.59

Table 6.10: Flow characteristics of flow past a circular cylinder at $Re = 20$ & 40

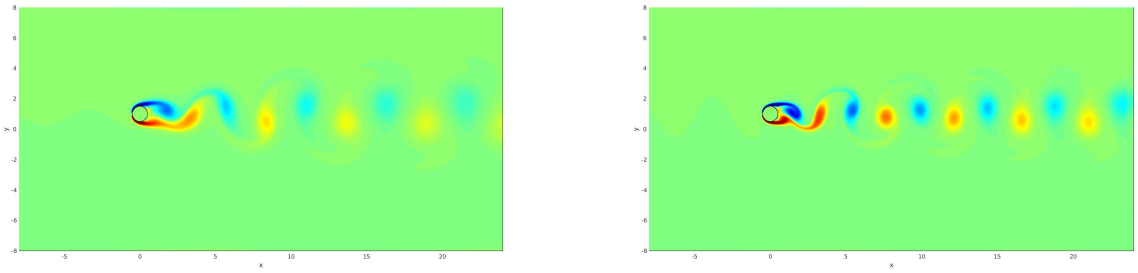


Figure 6.10: Vorticity field at $Re = 100$ & 200, serial

6.13 shows time evolution for drag and lift coefficients for the current method. Similarly, we can still find the drag coefficient frequency is twice as the lift coefficient frequency. Besides, the average drag coefficient is non-zero but average lift coefficient is zero.

6.5.5. Asymmetry

In this part, we will explain the asymmetry issue from our parallel program. During the development, we didn't notice the symmetry problem until we start the tests of flow past cylinders. In the earlier stage, we proved our method is second order accurate in velocity and first order accurate in pressure, shown in the circular Couette flow test. We didn't notice any symmetry issue as we only compared our results with analytical solution to see

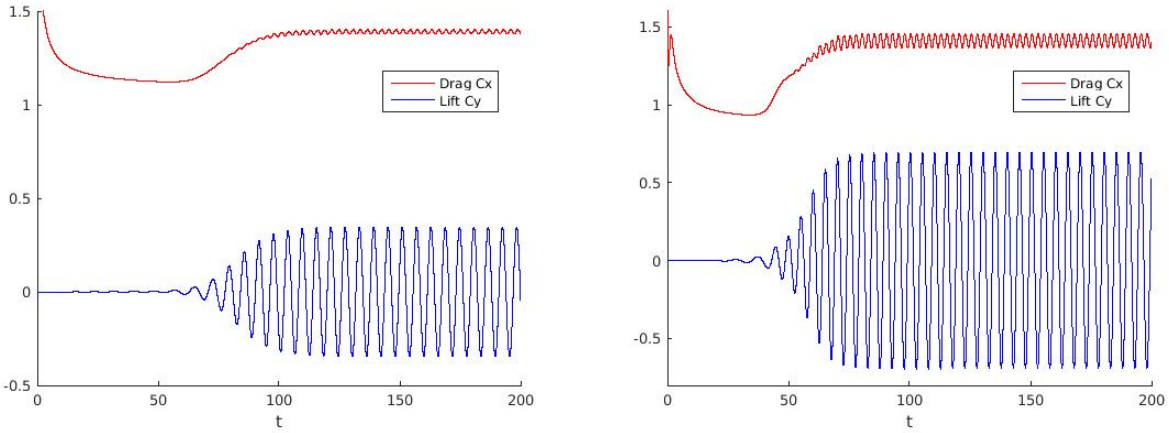


Figure 6.11: Drag and lift coefficients evolution with time at $Re = 100$ & 200, serial

	$Re = 100$			$Re = 200$		
	C_d	C_l	S_t	C_d	C_l	S_t
Ref. [6]	1.36 ± 0.015	± 0.250	-	1.40 ± 0.050	± 0.75	-
Ref. [50]	1.43 ± 0.009	± 0.322	0.172	1.45 ± 0.036	± 0.63	0.201
Ref. [77]	1.32 ± 0.013	± 0.250	0.171	1.42 ± 0.040	± 0.66	0.202
Ref. [37]	1.38 ± 0.010	± 0.337	0.169	1.37 ± 0.046	± 0.70	0.199
Ref. [28]	1.37 ± 0.009	± 0.323	0.169	1.34 ± 0.030	± 0.43	0.200
Present(serial)	1.39 ± 0.012	± 0.346	0.169	1.41 ± 0.043	± 0.69	0.200
Present(parallel)	1.40 ± 0.017	± 0.345	0.169	1.45 ± 0.001	± 0.69	0.200

Table 6.11: Flow characteristics of flow past a circular cylinder at $Re = 100$ & 200

the accuracy. Then in the flow past cylinders, the first unexpected result is that the lift coefficients at tests of $Re < 40$ are bigger than those from the serial program. For example, in the flow past a square cylinder test with $Re = 40$ and $B = 0.05$, the lift coefficient $C_l = 0.0039$, which is slightly bigger than zero. Besides, we noticed the drag coefficient C_d is bigger than our serial program's result and other researchers' results. The asymmetry can also be observed from the graph of streamlines, shown in Figure 6.15. This error can come from multiple sources: domain decomposition, communication between processors, incorrect algorithms for parallelization, methods to maintain compatibility condition, SMG pressure Poisson solver, truncation error and computer round-off error.

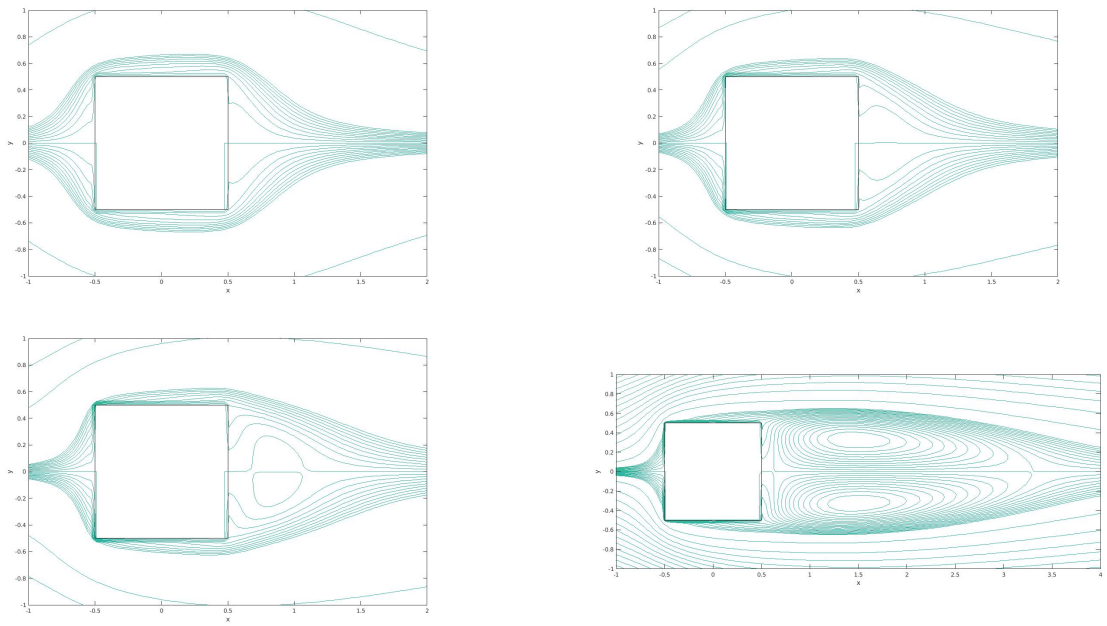


Figure 6.12: Streamfunction contours at $Re = 1.5$, 5, 20, 40, serial

The first thing we did is to use different number of processors for the same test. The exact same result is achieved no matter how many processors are used(even 1), which means there is no issue from the domain decomposition or algorithms. Since we are using a uniform mesh, the method for maintain compatibility will not bring any asymmetry. Then we checked the communication between processors, the function in this part are also working correctly. So the error can only come from the SMG solver, truncation error and computer round-off error. We cannot avoid round-off error, but since we are using double precision for any floating variables in our program, the round-off error is not significant. In order to locate whether the error is from SMG solver or truncation error, we did another test.

At the beginning of our program, it will initialize flow field, compute the jump conditions, solve the pressure Poisson equation, update the velocity and use it as initial flow field. In order to figure out what causes the asymmetry, the program only runs the initialization step. Now we have results of pressure p, right hand side of Poisson equation $rhsp$ and velocity u and v. Because the computational domain is symmetric along the x-axis, we can split the p, $rhsp$, u and v into mirrored two parts and compare the difference between them. In Figure

	B	L/D		C_d		
		$Re = 10$	$Re = 40$	$Re = 5$	$Re = 10$	$Re = 40$
Ref. [41]	0.067	-	2.7000	4.8140	-	1.8990
Ref. [54]	0.067	-	2.7348	5.2641	-	1.8565
Present(serial)	0.067	0.61	2.77	5.1334	3.1446	1.7664
Present(parallel)	0.067	0.61	2.77	5.2595	3.5609	1.9385
Present(serial)	0.0625	0.65	2.79	5.0616	3.3133	1.7518
Present(parallel)	0.0625	0.66	2.80	5.1859	3.5219	1.9219
Ref. [17]	0.050	-	2.8220	4.8400	-	1.7670
Ref. [54]	0.050	-	2.8065	4.9535	-	1.7871
Present(serial)	0.050	0.62	2.86	4.8759	3.03	1.7154
Present(parallel)	0.050	0.62	2.86	4.9950	3.4215	1.8797

Table 6.12: Flow characteristics of flow past a square cylinder at $Re = 5$, 10 & 40

6.16 and 6.17, the same tests were done with different tolerances for the SMG solver. For $rhsp$, maximum difference in the mirrored two parts has an amplitude of 10^{-10}. But for the difference of p, u and v, with smaller tolerance, the difference is smaller. So we can safely say the asymmetry issue comes from the SMG solver.

As the SMG solver can cause the asymmetry, one way to improve the symmetry is to use smaller tolerance. SMG is an iterative solver, by lowering the tolerance it will use more iterations and increase the computational time. Here we tried four different tolerance for the test of flow past a square cylinder at $B = 0.05$ and $Re = 40$. 50000 time steps were used and we compared some characteristics in table 6.14.

From this table, the drag and lift coefficients are almost identical in the four tests, and same as the maximum difference by comparing the mirrored parts of p, u and v. Using a smaller tolerance cannot bring us a better symmetry in the test results. The reason why symmetry cannot be improved is because it is limited by the order of accuracy of our method. With a smaller tolerance, the truncation error will become the most significant error source. During the computation, the method itself will carry the asymmetry from the SMG solver and magnify it to the order of truncation error. A good thing can be observed here is

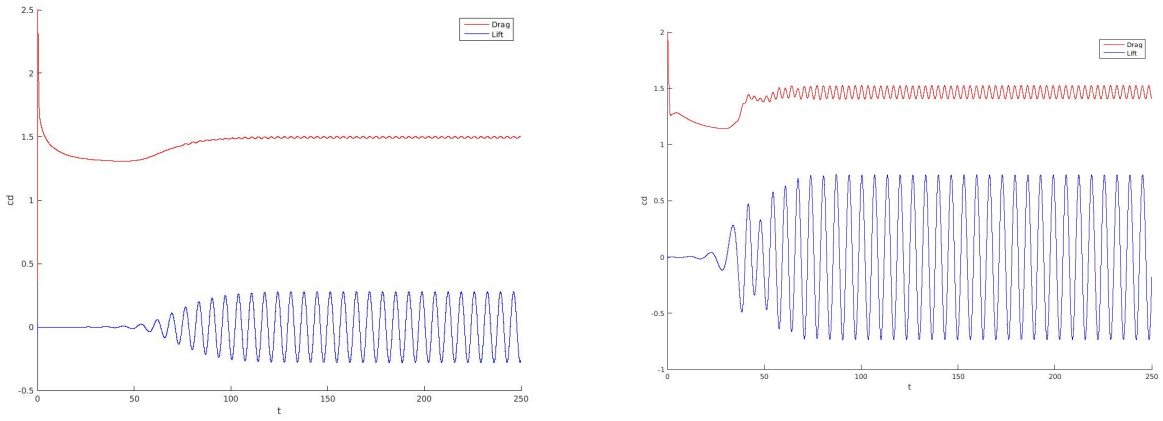


Figure 6.13: Fluid force evolution at $Re = 100$ & 200, $B = 0.05$, serial

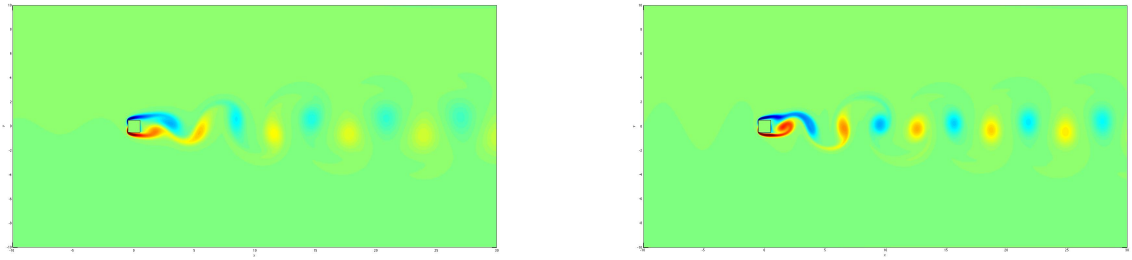


Figure 6.14: Vorticity field at $Re = 100$ & 200, serial

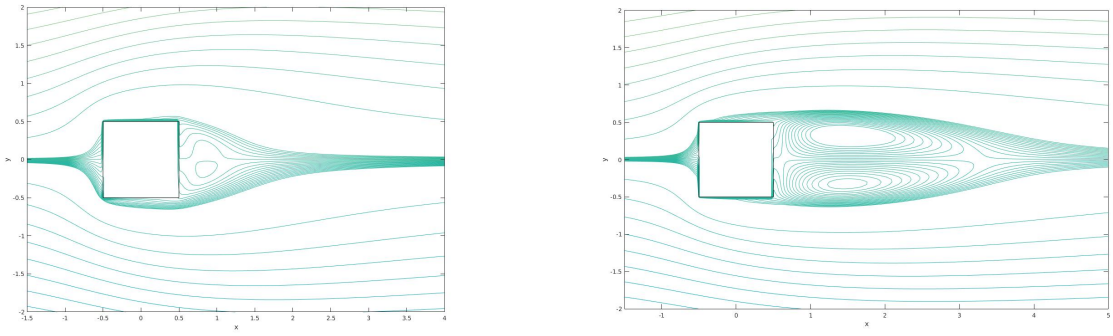
by using a smaller tolerance, the computational time doesn't increase dramatically, which means the SMG solver is very efficient. Right now we don't come up a clear solution to solve the asymmetry issue. In theory, we can change the SMG solver to another parallel Poisson solver with better symmetry preservation. We can also modify our FFT solver from the serial program into a parallel FFT solver, or we could solve only half of the computational domain, if the problem is symmetric in nature.

6.5.6. Parallel speedup and efficiency

In this test here, we study the parallel speedup and efficiency. We used 1 to 256 cores, 1200×600 grid numbers, 5000 time steps, $Re = 40$ and $B = 0.05$ in this test, and result is shown in Figure 6.18. As we can see, up to 24 cores, the speedup is increasing, and then it

	B	$\bar{C}_d$	S_t
Ref. [59]	0.050	1.4770	0.1460
Ref. [47]	0.056	1.5300	0.1540
Ref. [56]	0.050	1.4936	0.1488
Ref. [57]	0.050	1.5100	0.1470
Ref. [52]	0.050	1.4878	0.1486
Ref. [54]	0.050	1.5287	0.1452
Present(serial)	0.050	1.4941	0.1479
Present(parallel)	0.050	1.5984	0.1460

Table 6.13: Flow characteristics of flow past a square cylinder at $Re = 100$



(a) $Re = 10$ (b) $Re = 40$

Figure 6.15: Streamline function, parallel

is decreasing. The highest speedup is 7.01 when 24 cores used. For more than 24 cores, the parallel speedup drops.

6.6. Flow Past Two Square Cylinders

In this example, we tested if our method can handle multiple objects accurately and in a robust manner. The most popular type of object arrangement in the tests are side-by-side, tandem and staggered arrangement. In order to compare the numerical results with other people's work, here we choose tandem arrangement for two square cylinders. The geometry is shown in Figure 6.19. Here D is the side length of the square cylinder and we

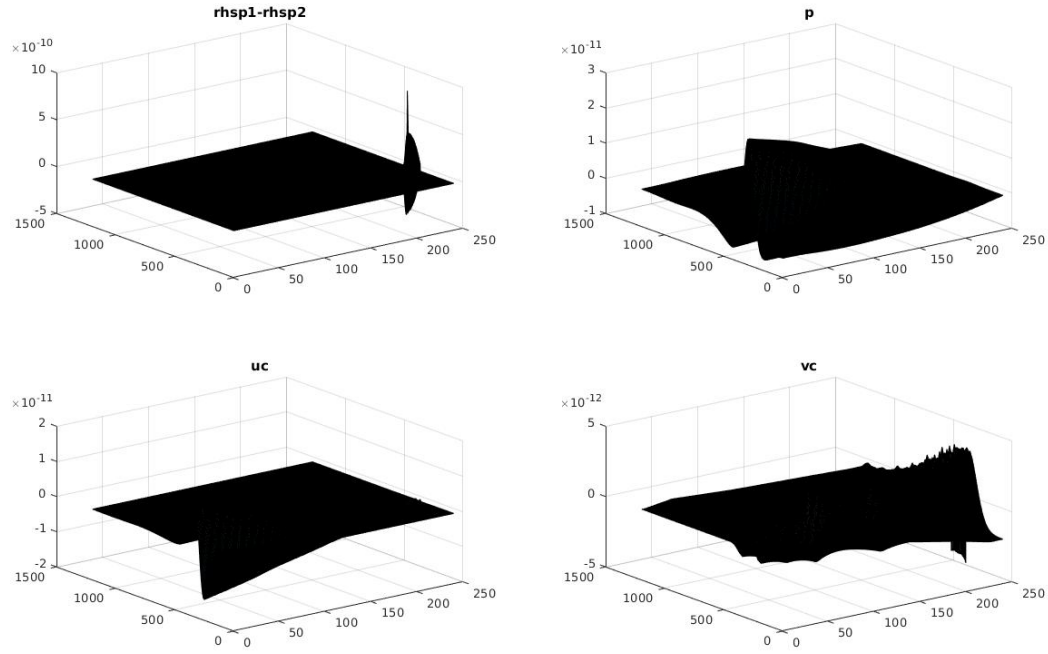


Figure 6.16: Error between mirrored p, $rhsp$, u and v with $tol = 1 \times 10^{-12}$

set $D = 1$. G is the distance between two square cylinders. We mark the left cylinder as upstream cylinder(UC) and right cylinder as downstream cylinder(DC). We use the same resolution and boundary conditions as circular cylinder example. In Table 6.15, we compared our results at $Re = 100$ and $G = 5$ with other numerical results. In Chatterjee [13], they used the PISO algorithm based finite volume solver in a collocated grid system to simulate the flow at different low Reynolds numbers and different G. In Sohankar [58], they used finite volume method based on the SIMPLEC algorithm and a non-staggered grid. They also simulated flow past two tandem square cylinders with different blockage ratios at low Reynolds number. Good agreement can be seen in Table 6.15. In Figure 6.20, the evolution of the drag and lift coefficients for both cylinders is shown. We can see the vertex shedding frequency for both cylinders is same. In Figure 6.21, the pressure field and vorticity field are shown. Besides the above tests, we also simulated the same test at $Re = 200$. In Figure 6.22, the evolution of drag and lift coefficients at $Re = 200$ is shown. In Figure 6.23, the pressure field and vorticity field at $Re = 200$ are shown.

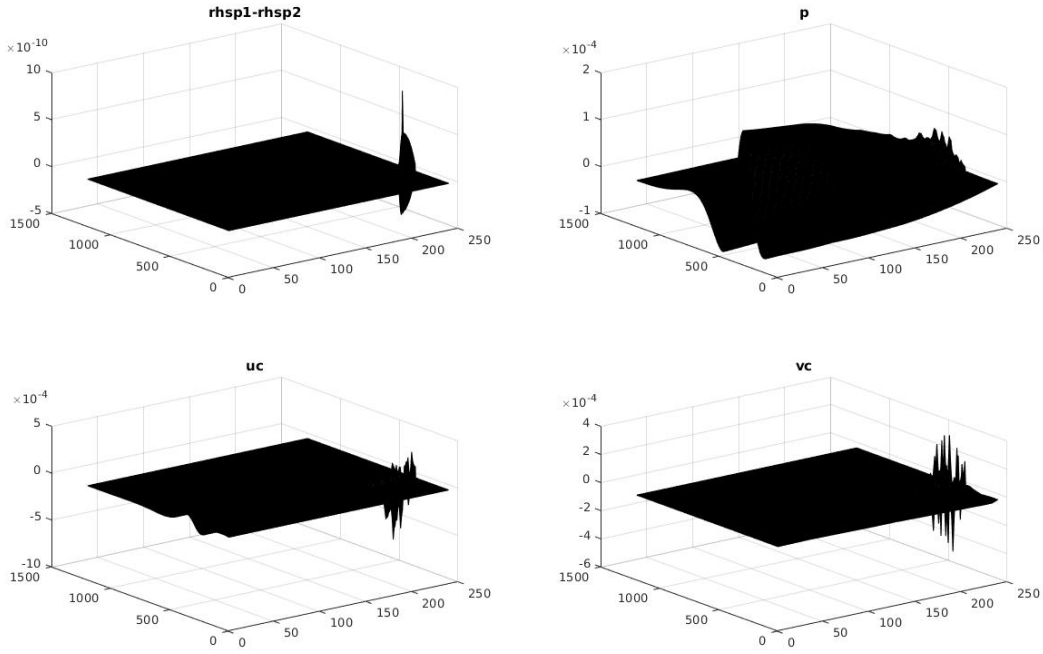


Figure 6.17: Error between mirrored p, $rhsp$, u and v with $tol = 1 \times 10^{-3}$

6.7. Flow Around A Hovering Flapper

In this section, we simulate the movement of a flapping rounded plate and a flapping rectangular plate. The geometry is shown in Figure 6.24. The spatial resolution is $N_x \times N_y \times M_s = 512 \times 512 \times 256$. The motion of the flapper is prescribed by

$$x_c(t) = 1.25(\cos(0.8t) + 1)\sin(\frac{\pi}{3}) \tag{6.4a}$$

$$y_c(t) = 1.25(\cos(0.8t) + 1)\cos(\frac{\pi}{3}) \tag{6.4b}$$

$$\theta(t) = \frac{3\pi}{4} + \frac{\pi}{4}\sin(0.8t)(1 - \exp(-t)) \tag{6.4c}$$

The time step is fixed in the computation, which is $\delta t = 3.927 \times 10^{-3} \approx \frac{T_f}{2000}$, and $T_f = \frac{2\pi}{0.8}$ is the flapping period of the hovering wing.

Previously the simulation of flow around a hovering rounded plate has been done in [66, 77]. Here we use the same Reynolds number $Re = 157$ and same object geometry and spatial resolution. Figure 6.25 shows the evolution of drag and lift coefficients of the rounded

tolerance	1.0×10^{-3}	1.0×10^{-6}	1.0×10^{-9}	1.0×10^{-12}
computational time(second)	33236.27	35743.56	38265.38	42808.63
unit time	1.00	1.08	1.15	1.29
C_d	1.9222	1.9219	1.9219	1.9219
C_l	0.0039	0.0039	0.0039	0.0039
max dif. in mirrored p	0.00750642	0.00750771	0.00750772	0.00750772
max dif. in mirrored u	0.00351327	0.00351345	0.00351345	0.00351345
max dif. in mirrored v	0.00168647	0.00168658	0.00168658	0.00168658

Table 6.14: Flow characteristics of flow past a square cylinder at $Re = 40$ and $B = 0.05$ with different tolerance

	C_d		S_t	
	UC	DC	UC	DC
Ref. [58]	1.539	1.292	0.143	0.143
Ref. [13]	1.5328	1.3202		
Present(serial)	1.4793	1.3102	0.1380	0.1379
Present(parallel)	1.4793	1.3102	0.1380	0.1379

Table 6.15: Flow characteristics of flow past two tandem square cylinders at $Re = 100$, $G = 5$

plate in 10 flapping periods and Figure 6.26 shows the evolution of drag and lift coefficients of the rectangular plate.

Comparing Figure 6.25 and Figure 6.26, we can find that the drag and lift coefficients for rectangular plate will be a little bigger than the rounded plate. This is expected because the geometry of the object is different, and rectangular plate will experience more drag and lift force when it moves in a fixed pattern. Figure 6.26 shows the results from serial program and Figure 6.27 comes from the results of parallel program. In Figure 6.26 the noise is noticeable and in Figure 6.27 the noise is even worse and hard to recognize the pattern of drag and lift coefficients. Russell and Wang have observed same noise phenomena in [50]. In [55], Seo and Mittal pointed out this is due to the violation of the geometric conservation law near the immersed boundary. They adopted a cut-cell based approach to strictly enforce geometric

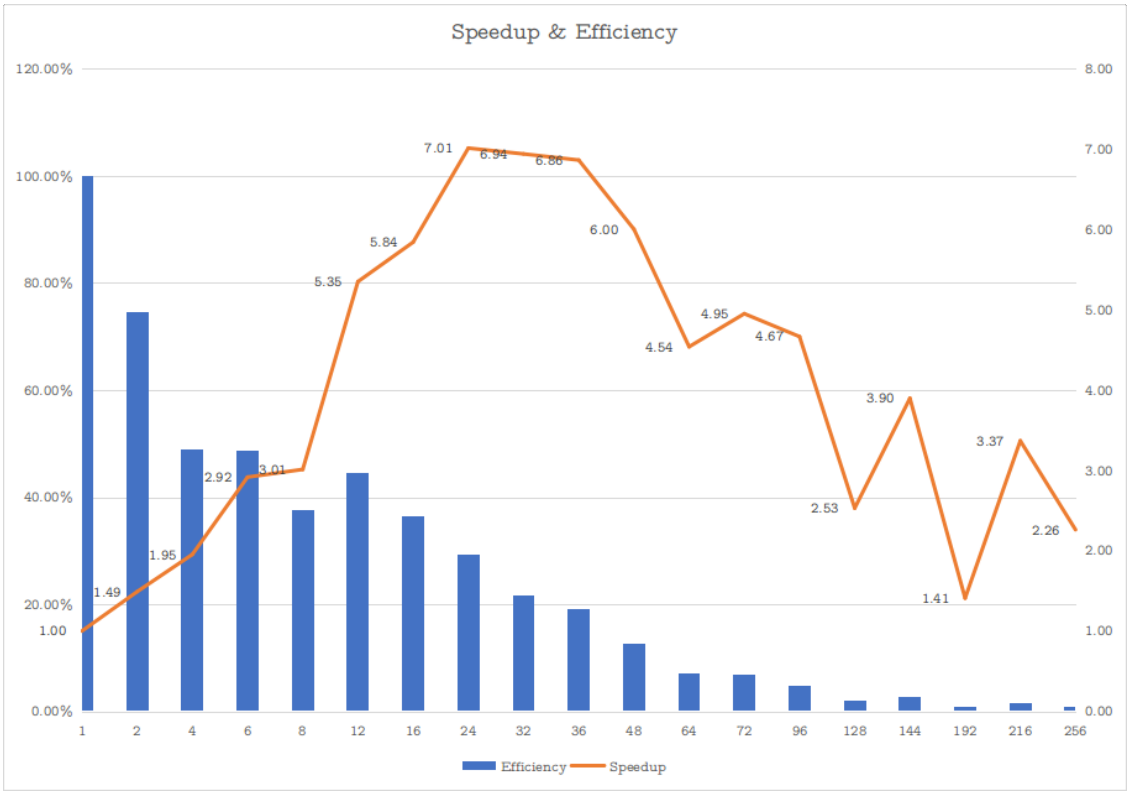


Figure 6.18: Parallel speedup and efficiency for flow past a square cylinder

conversation and reduced the noise successfully. Further investigation is necessary to remove the noise.

In Figure 6.28, we compared flow fields for the rounded plate at $Re = 157$ and $t \approx 10T_f$. Good agreement can be achieved for pressure and vorticity. In Figure 6.29, we compared flow fields for rounded plate and rectangular plate under our new method. The difference is big for both pressure and vorticity, but there are still some similarities between two objects.

6.8. Flow Around Multiple Hovering Flappers

In this example, we investigated the efficiency of simulation for flow around multiple moving rounded plate flappers and rectangular plate flappers. The same geometry and settings are used as our single flapper example. We tested both computational efficiency by adding more objects and parallel speedup and efficiency.

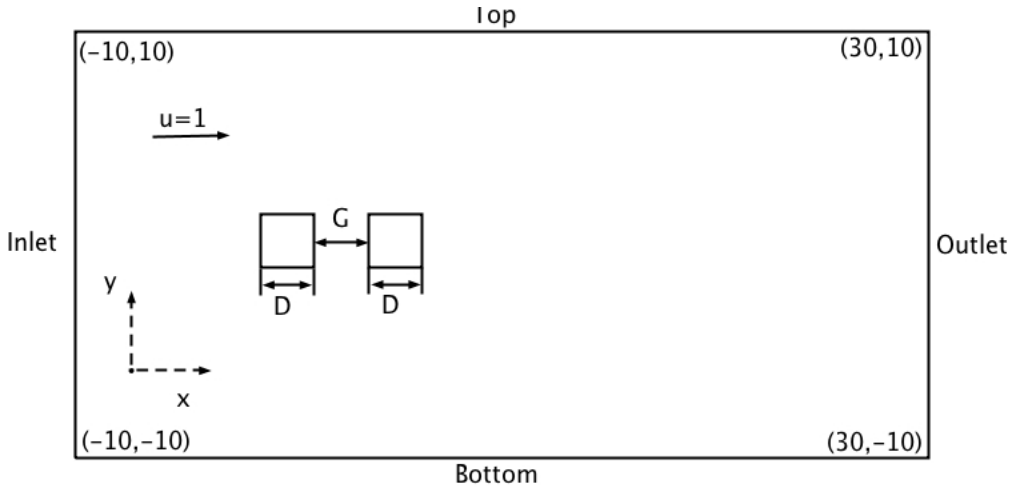


Figure 6.19: Geometry of flow past two square cylinders

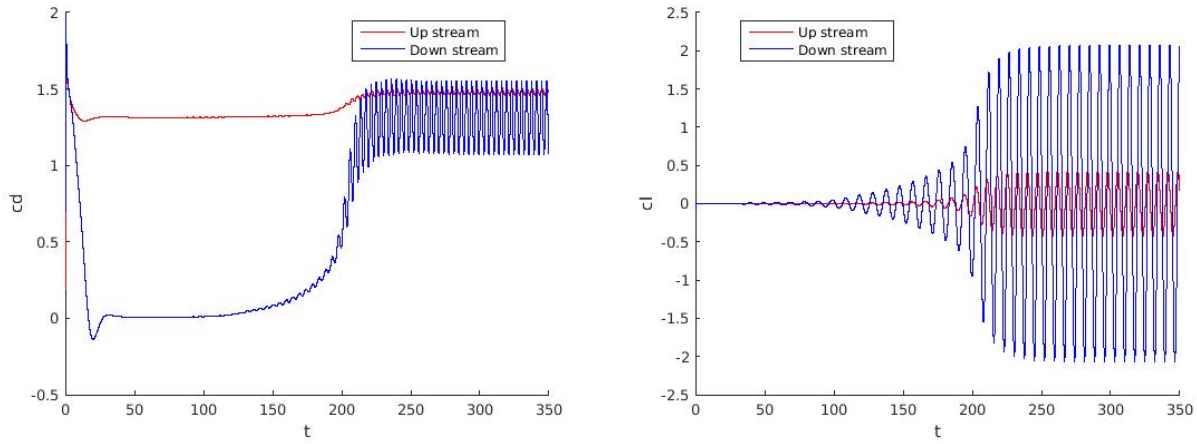


Figure 6.20: Fluid force evolution for two square cylinders at $Re = 100$, $G = 5$, serial

6.8.1. Efficiency of around multiple hovering flappers

In this test, we simulated up to 4 objects at the same time, and the motion of them were given by

$$x_c(l) = x_{c0}(l) + 1.25(\cos(0.8t) + 1)\sin(\frac{\pi}{3}), \tag{6.5a}$$

$$y_c(l) = y_{c0}(l) + 1.25(\cos(0.8t) + 1)\cos(\frac{\pi}{3}), \tag{6.5b}$$

$$\theta(l) = \frac{3\pi}{4} + \frac{\pi}{4}\sin(0.8t)(1 - \exp(-t)), \tag{6.5c}$$

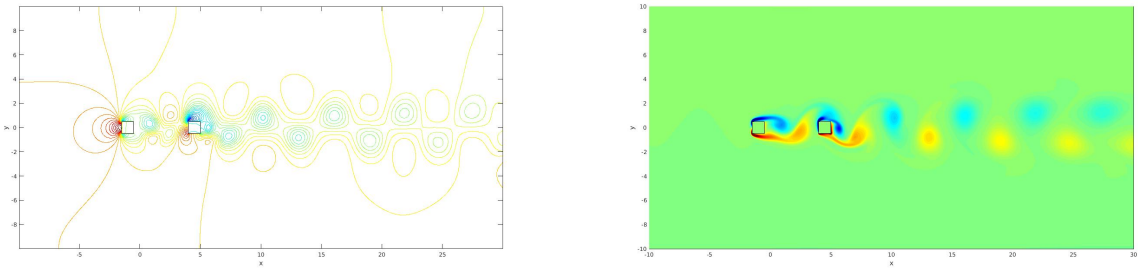


Figure 6.21: Flow field for two square cylinders at $Re = 100$, $G = 5$, serial

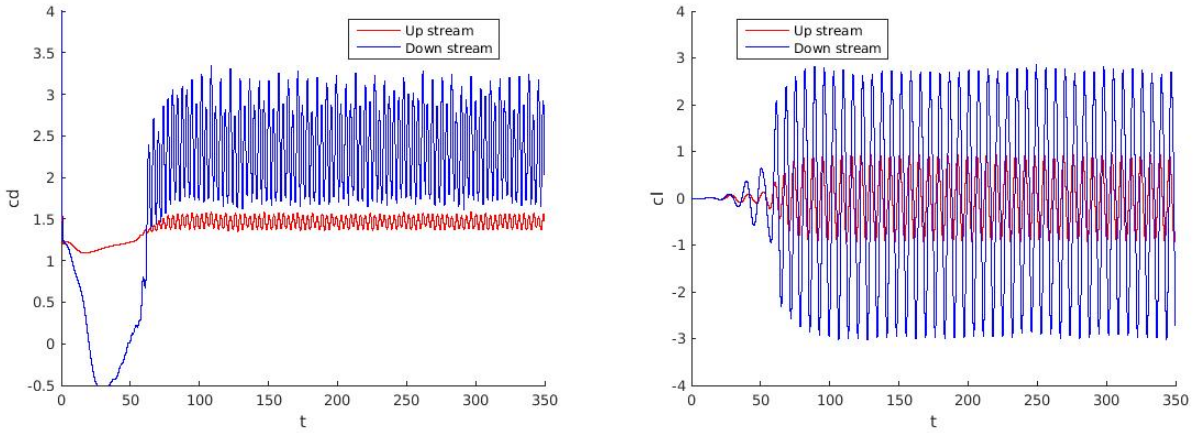


Figure 6.22: Fluid force evolution for two square cylinders at $Re = 200$, $G = 5$, serial

where $(x_{c0}(l), y_{c0}(l)) = (0,0)$, $(0,-3)$, $(-3,0)$ and $(-3,-3)$ for $l = 1,2,3,4$. The spatial resolution was $N_x \times N_y \times M_s = 512 \times 512 \times 256$. The time step was still $\delta t = 3.927 \times 10^{-3} \approx \frac{T_f}{2000}$, and $T_f = \frac{2\pi}{0.8}$ was the flapping period.

Figure 6.30 shows the vorticity field of simulation for flow around multiple rectangular flappers. Table 6.16 gives the relative computational time for different number of flappers with 10 flapping periods. We can find as the number of flappers increases, the increment on the computational time is proportional to the number of vertices. For 4 rectangular flappers case, we double the resolution so that's why the time spent is much bigger than other cases. Over all, a good efficiency of our method can be seen.

In table 6.17, we present the same simulation under our parallel program, and there are some difference from the last test. In this test, We used 1025×1025 grids, 32 cores and 20000

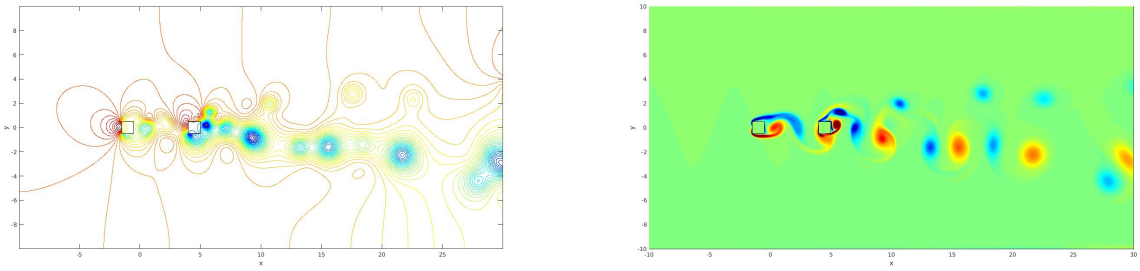


Figure 6.23: Flow field for two square cylinders at $Re = 200$, $G = 5$, serial

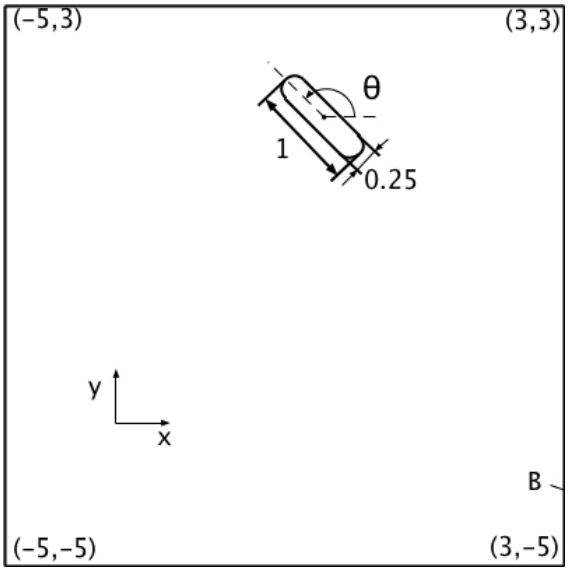


Figure 6.24: Geometry of flow around a flapper

time steps. Figure 6.31 gives a better view to study the relation between computational time and number of objects. We can also find that when the number of flappers increases, the increment on the computational time is proportional to the number of vertices.

6.8.2. Parallel speedup and efficiency

In this test, we present the results of parallel speedup and efficiency for simulating four hovering rectangular flappers. We used 1025×1025 grids and 20000 time steps. The number of processors was from 1 to 256. Table 6.18 and Figure 6.32 present the result of the overall computational time and the time for the SMG solver, computing principle jump condition

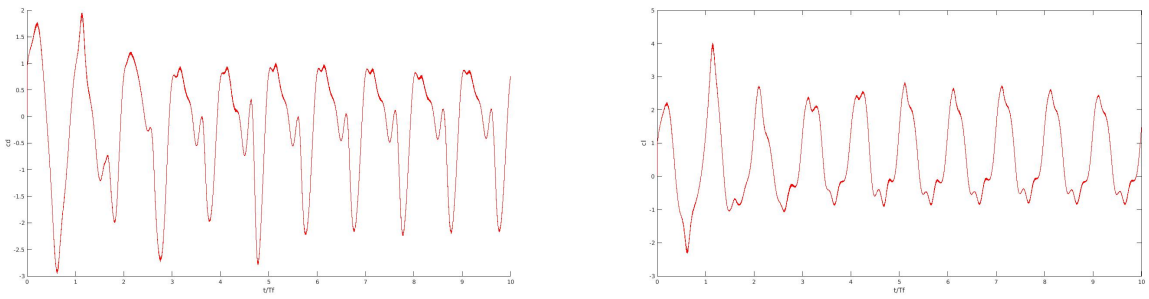


Figure 6.25: Drag and lift coefficients for rounded plate, serial

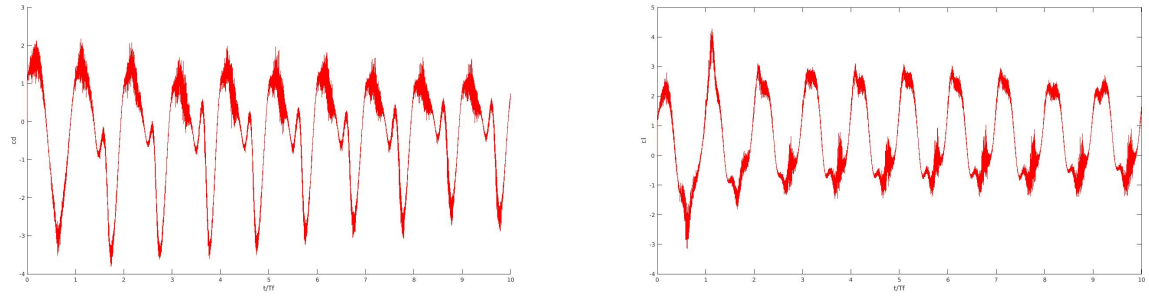


Figure 6.26: Drag and lift coefficients for rectangular plate, serial

$[p]$, communication between ghost layers and communication of objects. It also gives the percentage of each communication in the overall computational time. Figure 6.33 gives a better view for the percentage. As we can see, the SMG solver takes more than 80% of the overall time. The communication to compute $[p]$ will be second most time intensive with a maximum of 7.96%. The communication of ghost layers and objects take less than 1% of the overall time, which is not significant. Table 6.19 and Figure 6.34 present the results of speedup and efficiency for this test. For 1 to 24 processors, the speedup is increasing, and from 32 processors it is decreasing. The maximum speedup is 7.0343 at 24 cores. Similarly, we can find the overall parallel speedup and efficiency is almost identical to the SMG solver and is mainly dependent on the behavior of SMG solver.

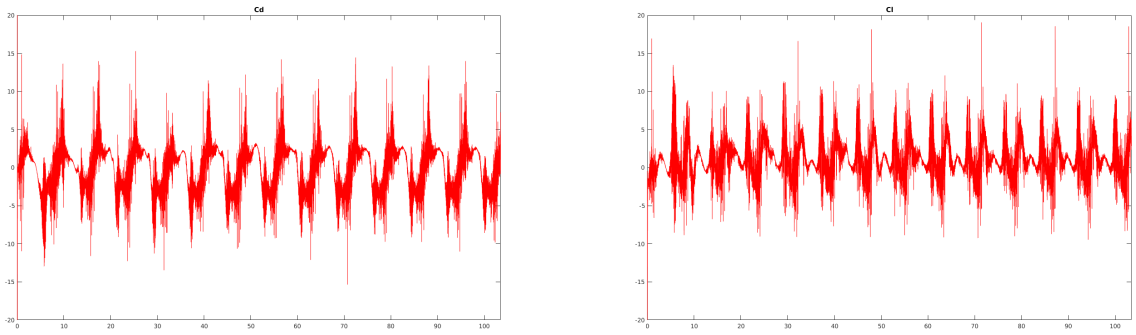


Figure 6.27: Drag and lift coefficients for rectangular plate, parallel

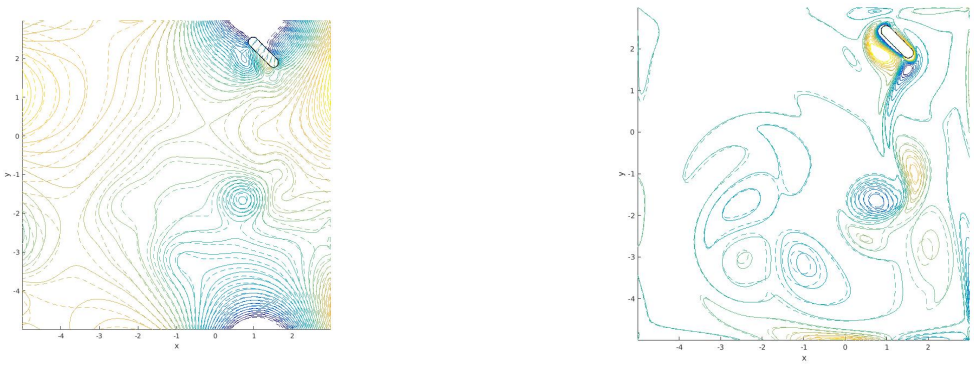


Figure 6.28: Comparison of flow fields around a flapper at $Re = 157$ and $t \approx 10T_f$. solid line: current, dashed line: previous. Serial results.

6.8.3. Scalability tests

In this part, we tested our parallel program's scalability for using different number of cores and involving large number of objects.

In the first test, we tested 16 to 1024 of flappers moving at the same time. The computational domain was $[-10, 70] \times [-10, 70]$, 256 cores and 500 time steps were used. The geometry is shown in Figure 6.35 and results are shown in table 6.20. We can find that by increasing the number of objects, the computational time for computing $[p]$ increases and for 1024 objects it takes more than 80% of the total computational time. As we mentioned in previous chapter, it has a large potential to reduce the time when solving for $[p]$ by recognizing the size of objects and use different method to finish the communication.

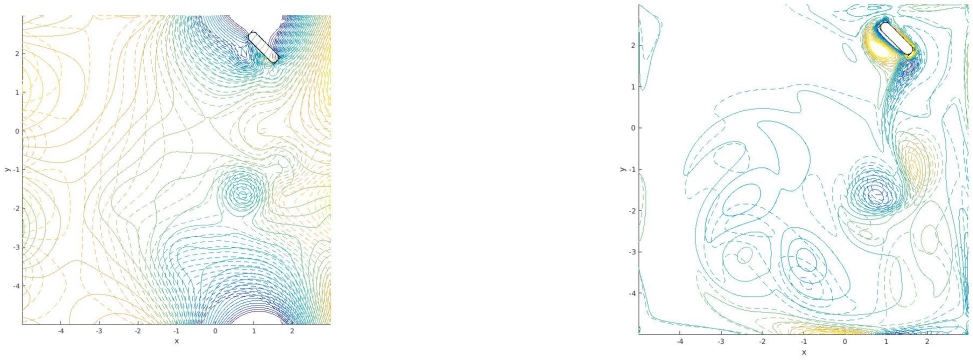


Figure 6.29: Comparison of flow fields around a flapper at $Re = 157$ and $t \approx 10T_f$. solid line: rounded plate, dashed line: rectangular plate. Serial results.

	1	2	3	4
Rounded plate	1	1.0509	1.1534	1.1977
rectangular	1.0195	1.0790	1.1526	1.4100

(The computational time corresponding to the unit value is 0.636 hours.)

Table 6.16: Relative computational time for different number of flappers(serial)

For communication of ghost layers and objects, they are all under 1% of the total time for different number of objects, which is efficient.

In the second test, we simulated 1024 flappers with different number of cores from 16 to 800. The results is given in table 6.21. From the table, it is difficult to recognize any trend on computational time by increasing the number of cores. We also tried to use more than 800 cores, but the simulation would fail and the reason is unclear.

In the last test here, we simulated the exact same test three times just to see if the hpc facility will affect the computational time. In this test, we used 1024 objects and 256 cores. By running the test three times, we have three different results on total computational time, given in table 6.22. In the table you can find it is impossible to ignore the difference on the computational time. Even though the three tests are exactly the same, HPC itself will affect the computational time. The reason why the behavior can be so different is unclear.

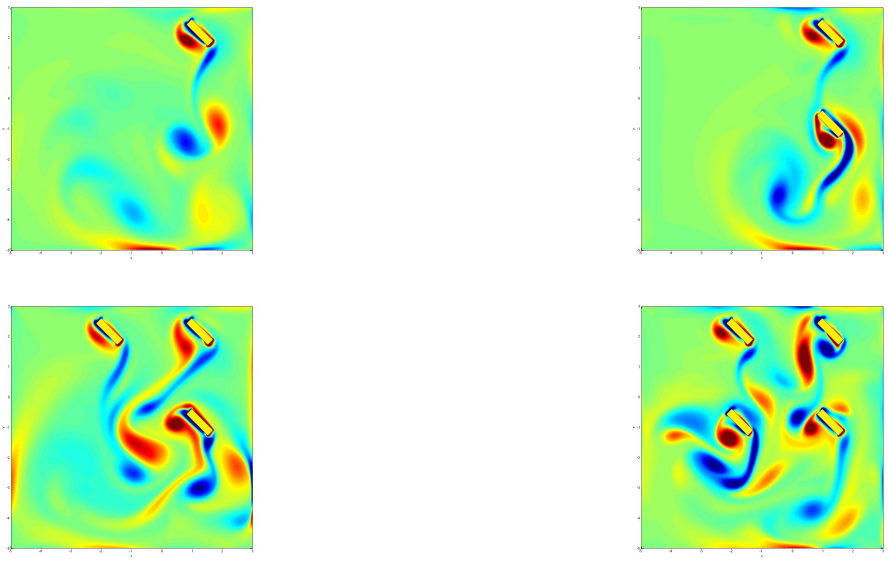


Figure 6.30: Vorticity field of multiple rectangular plate flappers, serial

	1	2	3	4
Rounded plate	1.0000	1.0179	1.0186	1.1872
rectangular	1.0348	1.0543	1.1181	1.2013

(The computational time corresponding to the unit value is 4.2332 hours.)

Table 6.17: Relative computational time for different number of flappers(parallel)

6.9. Cylinders Rotating Along A Circle

In this section, we tested up to 8 circular and square cylinders rotating with the same speed along a circle to test the efficiency of simulating multiple moving objects. The geometry is shown in Figure 6.36, which also shows the vorticity field. The computational domain was $[-4, 4] \times [-4, 4]$. The moving pattern for the objects is shown as below,

$$\theta = \frac{\pi}{4}(i-1) + \frac{\pi}{4}t, \quad i = 1, 8$$

$$x_c = 2\cos(\theta)$$

$$y_c = 2\sin(\theta)$$

In table 6.23, the result of relative computational time for different number of objects is

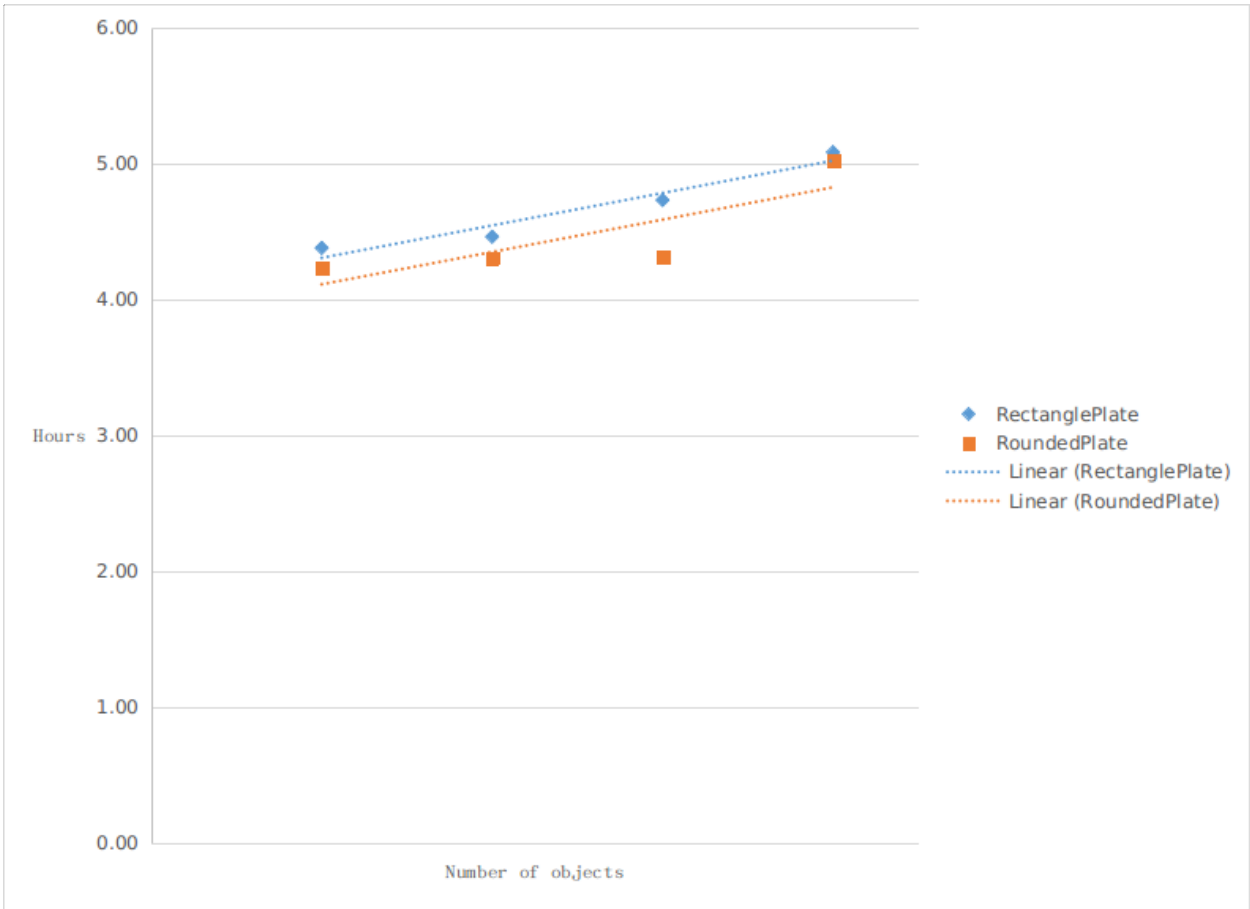


Figure 6.31: Relative computational time for different number of flappers

shown. We used 8 cores, 10000 time steps, 513×513 grids, $Re = 20$ and $\Delta t = 0.0002$. We can find as the number of object increases, the increment on the computational time is proportional to the number of vertices, as shown in Figure 6.37. Our method is able to handle multiple moving objects efficiently.

6.10. Flow Past Triangle Cylinder

In this example we want to simulate flow past a stationary triangle cylinder at different ratios R and different Reynolds numbers, to see how the drag coefficient will be affected. The geometry is shown in Figure 6.8, where $R = \frac{L}{B}$, $B = 1$, $L = [0.1, 10]$ and $Re = [10, 50]$. Figure 6.38 shows drag coefficients under different Reynolds number Re and different ratio R. From the figure we can see that as the Reynolds number increases, the drag coefficient

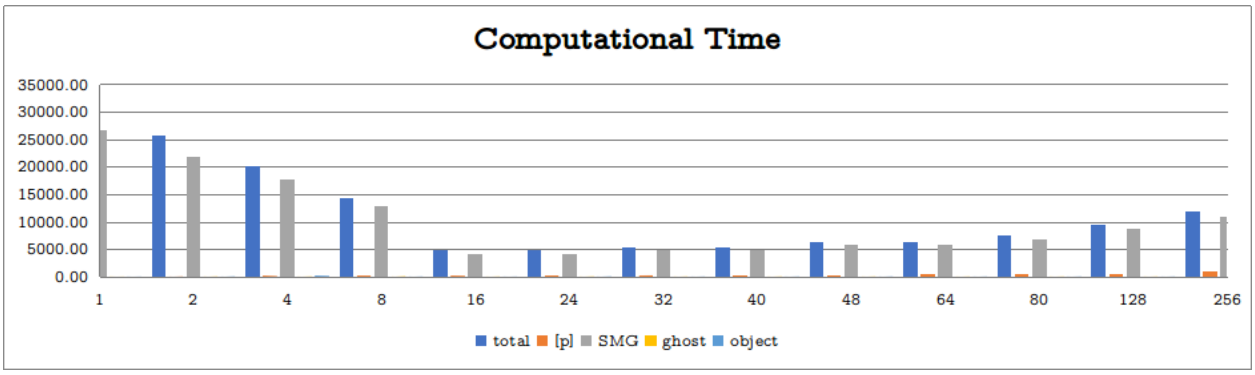


Figure 6.32: Computational time of hovering flappers at different processors

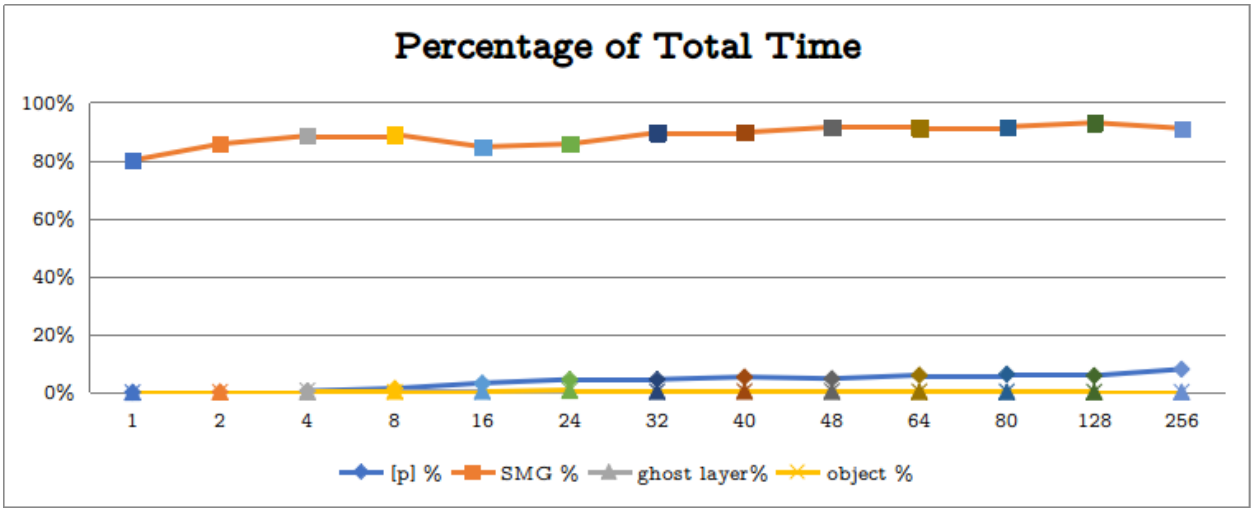


Figure 6.33: Percentage of computational time for hovering flappers at different processors

decreases. The most notable thing here is the change of drag coefficient with different R. When R increases from 0.1 to 1, the drag coefficient decreases. When $R = 1.732$, which is the equilateral triangle, the drag coefficient is the smallest for $Re = 30, 40, 50$. When Reynolds number increases from 2 to 10, the drag coefficient increases again.

6.11. Flow Past SMU Mascot Peruna

In the last section here, the test of flow past the SMU mascot Peruna is given. Since there is no available results we can compare, this test is mainly conducted to prove the robustness of our test. The mustang logo has very complex geometry and is a good example to see how

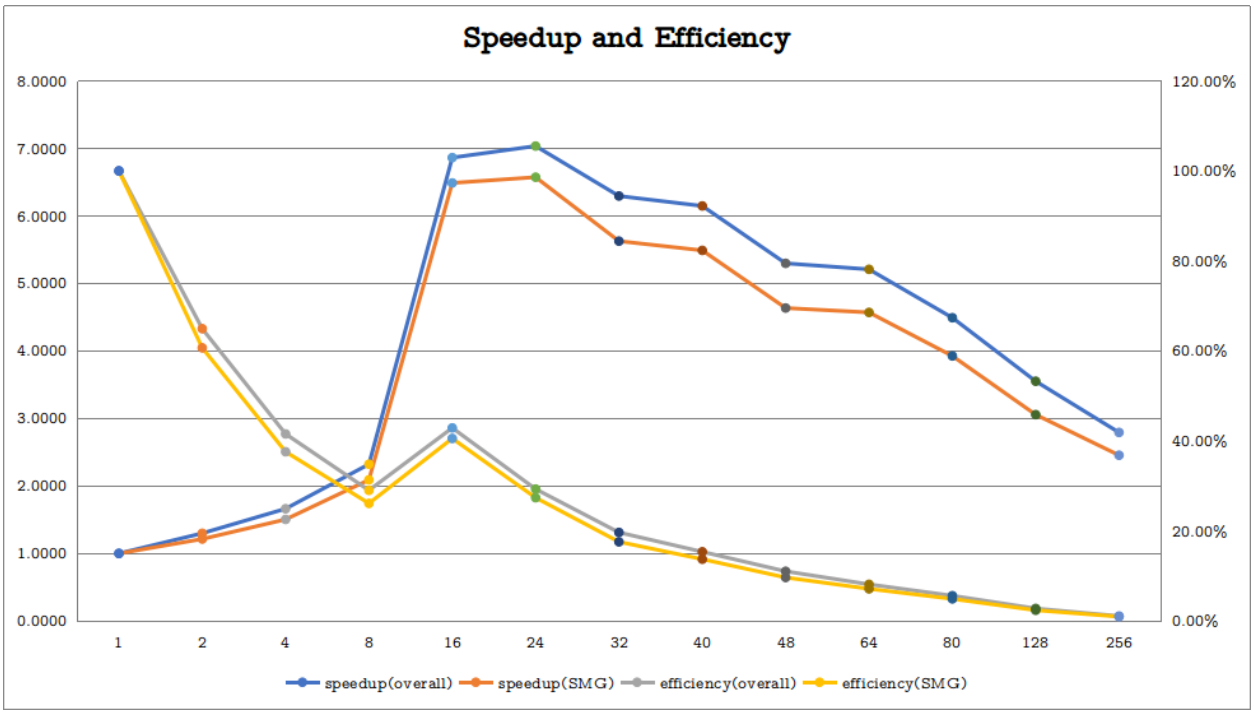


Figure 6.34: Parallel speedup & efficiency of hovering flappers at different processors

our method handle non-smooth geometries. In the test, the flow past the Peruna is from right to left at $Re = 1000$. In figure 6.39, the vorticity field and stream function are given.

n	total(s)	[p](s)	[p] %	SMG(s)	SMG %	ghost layer(s)	ghost layer %	object(s)	object %
1	33206.13	2.18	0.01%	26573.87	80.03%	0.10	0.00%	7.15	0.02%
2	25568.05	27.30	0.11%	21897.91	85.65%	24.99	0.10%	39.26	0.15%
4	19977.47	97.12	0.49%	17670.82	88.45%	23.28	0.12%	125.76	0.63%
8	14302.75	189.39	1.32%	12715.73	88.90%	55.71	0.39%	21.14	0.15%
16	4838.35	152.96	3.16%	4095.88	84.65%	17.70	0.37%	8.68	0.18%
24	4720.59	209.15	4.43%	4043.04	85.65%	29.64	0.63%	36.61	0.78%
32	5275.51	233.44	4.42%	4723.32	89.53%	16.37	0.31%	17.00	0.32%
40	5401.67	284.85	5.27%	4841.33	89.63%	20.32	0.38%	19.41	0.36%
48	6268.57	294.37	4.70%	5732.89	91.45%	18.21	0.29%	29.30	0.47%
64	6375.17	376.65	5.91%	5814.61	91.21%	14.58	0.23%	18.46	0.29%
80	7392.84	464.47	6.28%	6769.79	91.57%	13.67	0.18%	20.78	0.28%
128	9351.65	537.65	5.75%	8695.12	92.98%	11.86	0.13%	18.93	0.20%
256	11891.95	946.92	7.96%	10825.96	91.04%	13.21	0.11%	19.38	0.16%

Table 6.18: Parallel speedup & efficiency of hovering flappers at different processors

n	speedup	efficiency	speedup(SMG)	efficiency(SMG)
1	1.0000	100.00%	1.0000	100.00%
2	0.9380	46.90%	0.8446	42.23%
4	1.6622	41.55%	1.5038	37.60%
8	2.3217	29.02%	2.0898	26.12%
16	6.8631	42.89%	6.4880	40.55%
24	7.0343	29.31%	6.5727	27.39%
32	5.2180	16.31%	4.6603	14.56%
40	6.1474	15.37%	5.4890	13.72%
48	5.2972	11.04%	4.6353	9.66%
64	5.2087	8.14%	4.5702	7.14%
80	4.4917	5.61%	3.9254	4.91%
128	3.5508	2.77%	3.0562	2.39%
256	2.7923	1.09%	2.4546	0.96%

Table 6.19: Parallel speedup & efficiency of hovering flappers at different processors

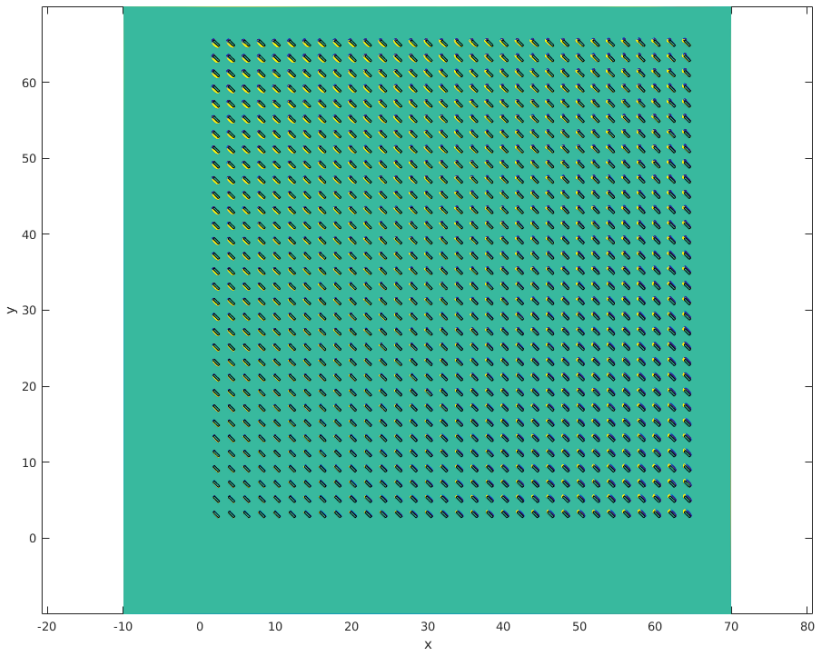


Figure 6.35: Geometry of 1024 hovering flappers

Objects	total	[p]	[p] %	SMG	SMG %	ghost	ghost %	object	object %
16	9687.72	530.17	5.47%	8716.85	89.98%	23.06	0.24%	13.45	0.14%
32	15814.48	788.82	4.99%	14549.80	92.00%	86.52	0.55%	60.71	0.38%
64	7006.46	1160.30	16.56%	5383.13	76.83%	23.90 2	0.34%	12.75	0.18%
128	11329.23	3271.53	28.88%	7597.99	67.07%	24.42	0.22%	12.33	0.11%
256	11623.42	5668.91	48.77%	5297.57	45.58%	47.03	0.40%	26.97	0.23%
512	21202.55	12904.14	60.86%	7213.60	34.02%	64.41	0.30%	48.80	0.23%
1024	57206.17	46342.58	81.01%	8765.44	15.32%	161.57	0.28%	143.32	0.25%

Table 6.20: Scalability test for different number of hovering flappers

Cores	total	[p]	[p] %	SMG	SMG %	ghost	ghost %	object	object %
16	55165.03	9483.77	17.19%	23590.23	42.76%	2586.74	4.69%	81.10	0.15%
64	19355.75	9322.37	48.16%	5671.40	29.30%	604.59	3.12%	46.78	0.24%
256	25641.38	21581.11	84.17%	3005.13	11.72%	76.98	0.30% %	63.86	0.25%
400	29277.38	25596.70	87.43%	3003.37	10.26%	37.85	0.13%	42.57	0.15%
625	60216.61	56875.73	94.45%	2823.46	4.69%	19.23	0.03%	25.09	0.04%
800	55503.99	38776.50	69.86%	16264.92	29.30%	173.91	0.31%	186.22	0.34%

Table 6.21: 1024 hovering flappers with different cores

Test	total	[p]	[p] %	SMG	SMG %	ghost	ghost %	object	object %
1	33954.87	28399.59	83.64%	4475.89	42.76%	115.41	0.34%	99.84	0.29%
2	34520.36	27423.43	79.44%	6024.05	29.30%	199.54	0.58%	172.39	0.50%
3	29402.48	20872.46	70.99%	7497.61	11.72%	71.93	0.24%	66.44	0.23%

Table 6.22: Test for checking HPC's influence on computational time by running same test three times

Figure 6.36: Geometry of flow around multiple cylinders rotating around a center

# of Objects	1	2	3	4	5	6	7	8
Circular	1.0000	1.0242	1.0393	1.0648	1.0797	1.0977	1.1051	1.1283
Square	1.0041	1.0251	1.0446	1.0593	1.0827	1.0926	1.1139	1.1276

(The computational time corresponding to the unit value is 0.8752 hours.)

Table 6.23: Relative computational time for different number of objects

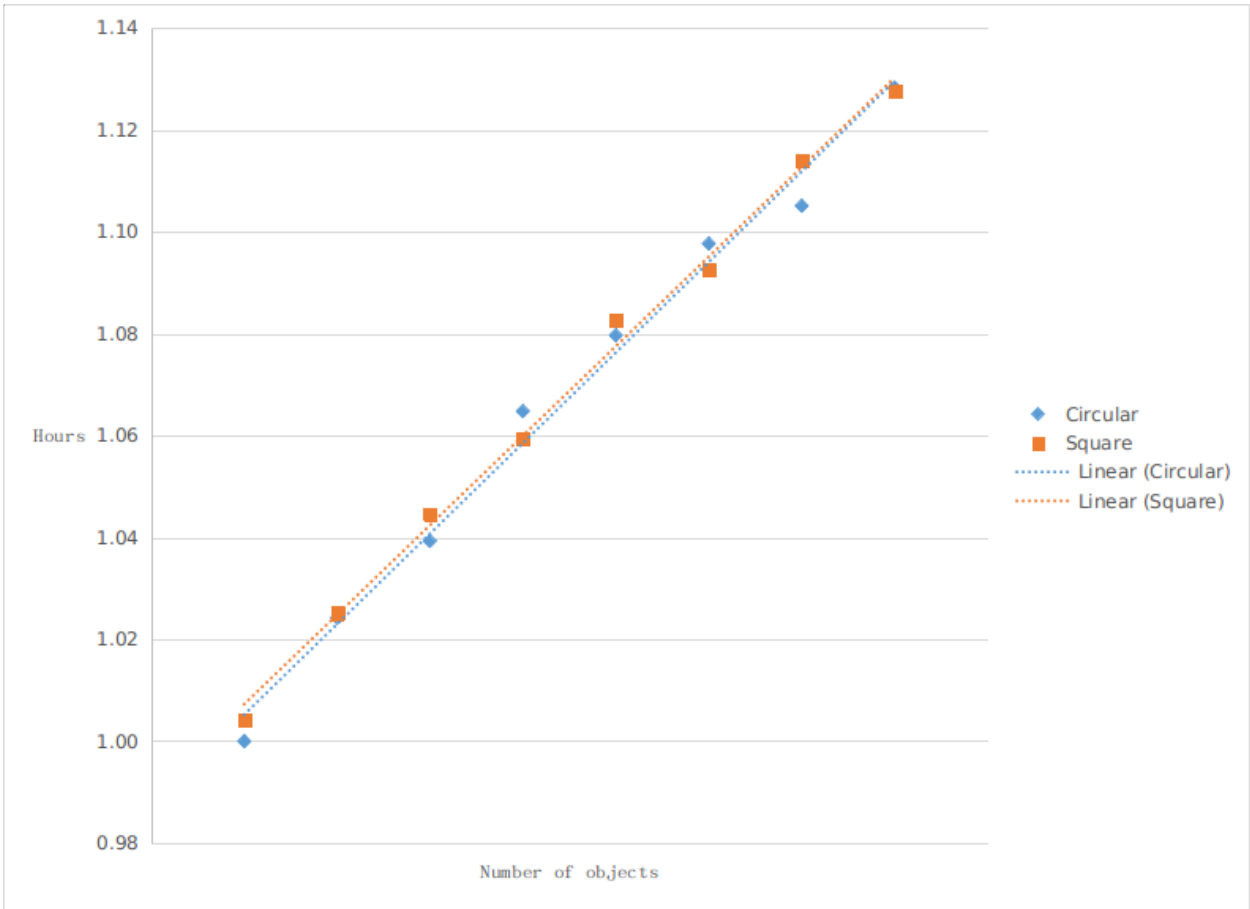


Figure 6.37: Relative computational time for different number of objects

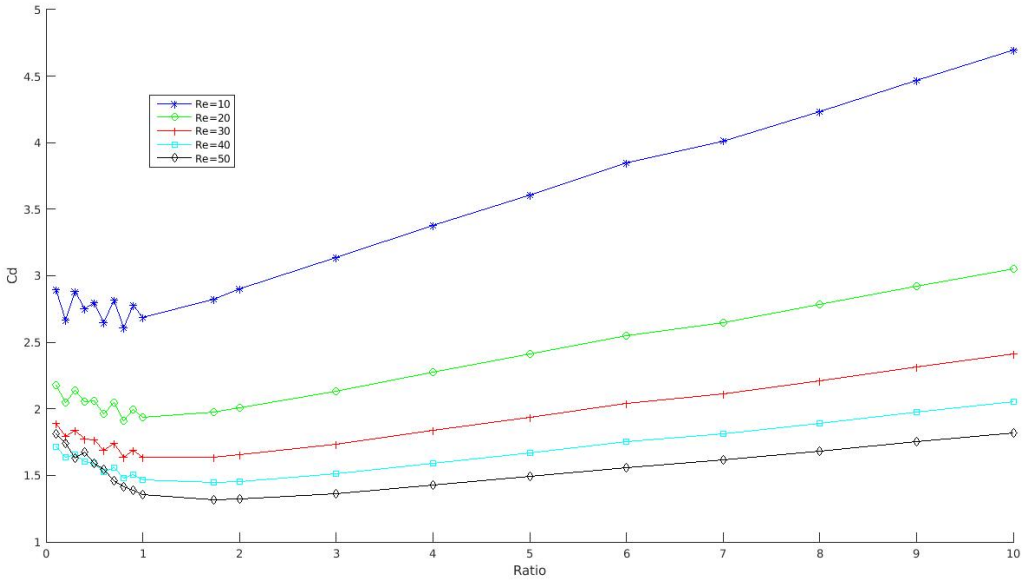
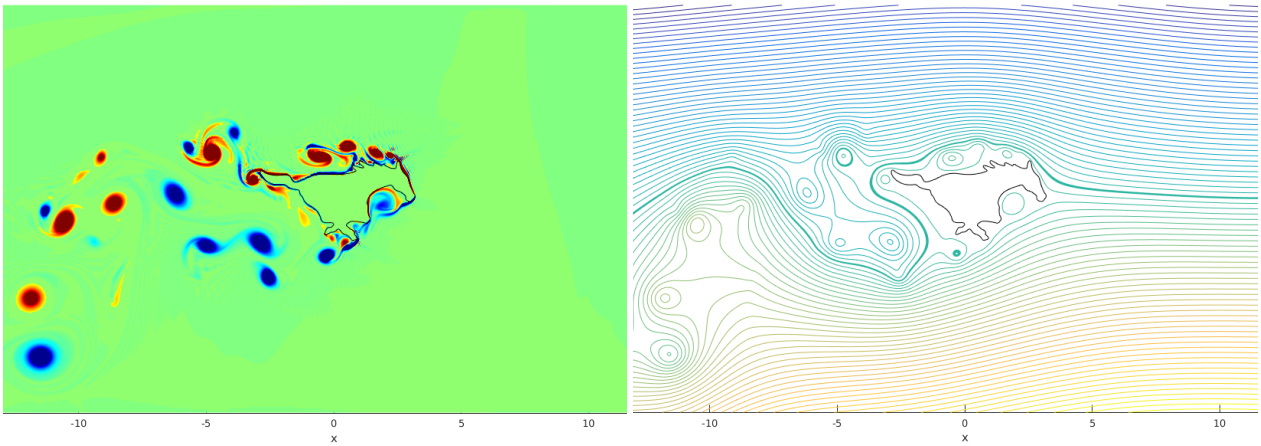


Figure 6.38: Drag coefficients vs. ratio with different Reynolds number



(a) Vorticity field (b) Stream function

Figure 6.39: Flow past Peruna from right to left at $Re = 1000$

Chapter 7

SUMMARY AND CONCLUSIONS

In this thesis, we have presented the immersed interface method for simulating incompressible viscous flow around non-smooth boundaries. This method couples the Navier-Stokes equation and pressure Poisson equation to approximate solutions and is developed based on a finite difference scheme. The key elements of this method are listed as below:

- Line segment panel representation of object boundaries is used in our method.
- A generalized second-order central finite difference method is used to incorporate the jump conditions and used in the discretization.
- An method to compute principle and Cartesian jump conditions.
- A third-order Runge Kutta method is used for time integration.
- A stretched mesh is introduced and all algorithms for computing jump conditions and jump contributions are modified based on it.
- A staggered grid is used for representation of computational domain.
- A semi-coarsening multigrid method(SMG) is used to solve the pressure Poisson equation.
- The parallel program utilizes the *MPI* library for communication between different cores.
- A data structure is designed and different parallel strategies are developed for different communications in the program.

Compared to previous methods [72, 77–79], our method has better robustness and efficiency. It presents a brand new approach to solve the flow problems with non-smooth objects, which is traditionally difficult to compute.

Different simulations were performed to investigate our method's stability, accuracy, robustness and efficiency during the development.

- The Poisson solver test helps us examine the accuracy of the jump condition calculations, the accuracy of Poisson solver and the ability to solve problems under stretched mesh. The second order accuracy is achieved for solving the Poisson equation.
- The lid-driven cavity flow test examines our method's validation for flow problems without any object, and good results are achieved.
- The circular Couette test is performed to test the accuracy for flow problems with objects. The test results show our method is second order accurate in the infinity norm for the velocity and first-order accurate in the infinity norm for pressure. Good accuracy has been seen in both uniform and stretched grids.
- The tests of flow past a circular cylinder, a square cylinder and two square cylinders show our method is stable and results are valid for Reynolds number between 5 and 1000 by comparing with previous work. Asymmetry issues were observed from the results of the parallel program which will affect the results. This asymmetry is carried from the SMG solver.
- The multiple hovering rounded plate and rectangle plate flappers test shows good efficiency of our method. The extra cost to handle additional objects is proportional to the number of the vertices used to represent the objects. The test of cylinders translating along a circle also shows good efficiency when simulating multiple objects moving in the domain.
- We also tested the parallel speedup and efficiency in cavity flow, flow past a square cylinder and multiple hovering flappers. The SMG solver has a good efficiency when using high tolerance. The time for communication between different cores is not significant compared with the SMG solver.

- The test of flow past the SMU mustang logo shows good robustness of our method and ability to handle objects with complex geometries.

Overall, our method is stable at all the Reynolds number have been tested and shows good agreement with prior work. The parallel speedup and parallel efficiency are dependent on the SMG solver.

Most of the test results have matched our expectation, but there are still some weakness we can improve. For the parallel program, because we are using SMG solver, the asymmetry is introduced when solving the pressure Poisson equation. The computational time is also mainly dependent on the SMG solver. Besides, our method has the potential to be more efficient when we collect the right hand side vector in section 5.3.3 for solving the $Ax = b$ problem to achieve the information of $[p]$. When handling large number of objects moving at the same time, this could be a potential bottleneck.

In the future, this method can be improved to handle flows with larger Reynolds number and be even more efficient. Because of the good accuracy, robustness and efficiency, this method can be applied to simulate insects flight aerodynamics or has the potential for applications in many other areas.

BIBLIOGRAPHY

[1] B. Barney. Message passing interface (mpi). lawrence livermore national laboratory, h ttps. *computing. llnl. gov/tutorials/mpi/(available online, 2010)*, 2009.

[2] A. J. Bergou, S. Xu, and Z. J. Wang. Passive wing pitch reversal in insect flight. *Journal of Fluid Mechanics*, 591, 2007.

[3] J. Bernsdorf, F. Durst, and M. Schäfer. Comparison of cellular automata and finite volume techniques for simulation of incompressible flows in complex geometries. *International Journal for Numerical Methods in Fluids*, 29(3):251–264, 1999.

[4] P. A. Berthelsen and O. M. Faltinsen. A local directional ghost cell approach for incompressible viscous flow problems with irregular boundaries. *Journal of computational physics*, 227(9):4354–4397, 2008.

[5] O. Botella and R. Peyret. Benchmark spectral results on the lid-driven cavity flow. *Computers & Fluids*, 27(4):421–433, 1998.

[6] M. Braza, P. Chassaing, and H. H. Minh. Numerical study and physical analysis of the pressure and velocity fields in the near wake of a circular cylinder, 1986.

[7] M. Breuer, J. Bernsdorf, T. Zeiser, and F. Durst. Accurate computations of the laminar flow past a square cylinder based on two different methods: lattice-boltzmann and finite-volume. *International Journal of Heat and Fluid Flow*, 21(2):186–196, 2000.

[8] W. L. Briggs, V. E. Henson, and S. F. McCormick. *A multigrid tutorial*. SIAM, 2000.

[9] P. N. Brown, R. D. Falgout, and J. E. Jones. Semicoarsening multigrid on distributed memory machines. *SIAM Journal on Scientific Computing*, 21(5):1823–1834, 2000.

[10] C.-H. Bruneau and C. Jouron. An efficient scheme for solving steady incompressible navier-stokes equations. *Journal of Computational Physics*, 89(2):389–413, 1990.

[11] P. Castonguay, D. Williams, P. Vincent, M. Lopez, and A. Jameson. On the development of a high-order, multi-gpu enabled, compressible viscous flow solver for mixed unstructured grids. In *20th AIAA Computational Fluid Dynamics Conference*, page 3229, 2011.

[12] B. Chapman, G. Jost, and R. Van Der Pas. *Using OpenMP: portable shared memory parallel programming*, volume 10. MIT press, 2008.

[13] D. Chatterjee and B. Mondal. Forced convection heat transfer from tandem square cylinders for various spacing ratios. *Numerical Heat Transfer, Part A: Applications*, 61(5):381–400, 2012.

[14] M. Coutanceau and R. Bouard. Experimental determination of the main features of the viscous flow in the wake of a circular cylinder in uniform translation. part 1. steady flow. *Journal of Fluid Mechanics*, 79(02):231–256, 1977.

[15] G. Deng, J. Piquet, P. Queutey, and M. Visonneau. Incompressible flow calculations with a consistent physical interpolation finite volume approach. *Computers & fluids*, 23(8):1029–1047, 1994.

[16] S. C. R. Dennis and G.-Z. Chang. Numerical solutions for steady flow past a circular cylinder at Reynolds numbers up to 100. *J. Fluid Mech.*, 42(03):471, 2006.

[17] A. Dhiman, R. Chhabra, A. Sharma, and V. Eswaran. Effects of reynolds and prandtl numbers on heat transfer across a square cylinder in the steady flow regime. *Numerical Heat Transfer, Part A: Applications*, 49(7):717–731, 2006.

[18] A. El Yacoubi, S. Xu, and Z. Jane Wang. Computational study of the interaction of freely moving particles at intermediate Reynolds numbers. *Journal of Fluid Mechanics*, 705:134–148, 2012.

[19] E. Fadlun, R. Verzicco, P. Orlandi, and J. Mohd-Yusof. Combined immersed-boundary finite-difference methods for three-dimensional complex flow simulations. *Journal of Computational Physics*, 161(1):35–60, 2000.

[20] R. D. Falgout and J. E. Jones. Multigrid on massively parallel architectures. In *Multigrid Methods VI*, pages 101–107. Springer, 2000.

[21] C. Feichtinger, J. Habich, H. Köstler, G. Hager, U. Rüde, and G. Wellein. A flexible patch-based lattice boltzmann parallelization approach for heterogeneous gpu–cpu clusters. *Parallel Computing*, 37(9):536–549, 2011.

[22] B. Fornberg. A numerical study of steady viscous flow past a circular cylinder. *Journal of Fluid Mechanics*, 98(04):819–855, 1980.

[23] U. Ghia, K. N. Ghia, and C. Shin. High-re solutions for incompressible flow using the navier-stokes equations and a multigrid method. *Journal of computational physics*, 48(3):387–411, 1982.

[24] M. Kang, R. P. Fedkiw, and X.-D. Liu. A boundary condition capturing method for multiphase incompressible flow. *Journal of Scientific Computing*, 15(3):323–360, 2000.

[25] J. Kim, D. Kim, and H. Choi. An immersed-boundary finite-volume method for simulations of flow in complex geometries. *Journal of Computational Physics*, 171(1):132–150, 2001.

[26] M.-C. Lai and C. S. Peskin. An immersed boundary method with formal second-order accuracy and reduced numerical viscosity. *Journal of Computational Physics*, 160(2):705–719, 2000.

[27] P. Lallemand and L.-S. Luo. Lattice boltzmann method for moving boundaries. *Journal of Computational Physics*, 184(2):406–421, 2003.

[28] D. V. Le, B. C. Khoo, and J. Peraire. An immersed interface method for viscous incompressible flows involving rigid and flexible boundaries. *J. Comput. Phys.*, 220(1):109–138, 2006.

[29] L. Lee and R. J. LeVeque. An immersed interface method for incompressible navier–stokes equations. *SIAM Journal on Scientific Computing*, 25(3):832–856, 2003.

[30] R. J. Leveque and Z. Li. The immersed interface method for elliptic equations with discontinuous coefficients and singular sources. *SIAM Journal on Numerical Analysis*, 31(4):1019–1044, 1994.

[31] R. J. LeVeque and Z. Li. Immersed interface methods for stokes flow with elastic boundaries or surface tension. *SIAM Journal on Scientific Computing*, 18(3):709–735, 1997.

[32] Z. Li. The immersed interface method using a finite element formulation. *Applied Numerical Mathematics*, 27(3):253–267, 1998.

[33] Z. Li and K. Ito. *The immersed interface method: numerical solutions of PDEs involving interfaces and irregular domains*, volume 33. Siam, 2006.

[34] Z. Li and M.-C. Lai. The immersed interface method for the navier–stokes equations with singular forces. *Journal of Computational Physics*, 171(2):822–842, 2001.

[35] Z. Li, T. Lin, and X. Wu. New cartesian grid methods for interface problems using the finite element formulation. *Numerische Mathematik*, 96(1):61–98, 2003.

[36] J. H. Lienhard. *Synopsis of lift, drag, and vortex frequency data for rigid circular cylinders*. Technical Extension Service, Washington State University, 1966.

[37] M. N. Linnick and H. F. Fasel. A high-order immersed interface method for simulating unsteady incompressible flows on irregular domains. *J. Comput. Phys.*, 204(1):157–192, 2005.

[38] S. McKee, M. F. Tomé, V. G. Ferreira, J. a. Cuminato, a. Castelo, F. S. Sousa, and N. Mangiavacchi. The MAC method. *Comput. Fluids*, 37(8):907–930, 2008.

[39] R. L. Meakin. Composite overset structured grids. In *Handbook of Grid Generation*. CRC Press, 1998.

[40] E. Momox, N. Zakhleniuk, and N. Balkan. Solution of the 1d schrödinger equation in semiconductor heterostructures using the immersed interface method. *Journal of Computational Physics*, 231(18):6173–6180, 2012.

[41] B. Paliwal, A. Sharma, R. Chhabra, and V. Eswaran. Power law fluid flow past a square cylinder: momentum and heat transfer characteristics. *Chemical engineering science*, 58(23):5315–5329, 2003.

[42] P. P. Patil and S. Tiwari. Effect of blockage ratio on wake transition for flow past square cylinder. *Fluid Dynamics Research*, 40(11-12):753–778, 2008.

[43] C. S. Peskin. Flow patterns around heart valves: a numerical method. *Journal of computational physics*, 10(2):252–271, 1972.

[44] C. S. Peskin. The immersed boundary method. *Acta numerica*, 11:479–517, 2002.

[45] N. A. Petersson. Hole-cutting for three-dimensional overlapping grids. *SIAM Journal on Scientific Computing*, 21(2):646–665, 1999.

[46] A. Prosperetti and G. Tryggvason. *Computational methods for multiphase flow*. Cambridge university press, 2009.

[47] J. Robichaux, S. Balachandar, and S. Vanka. Three-dimensional floquet instability of the wake of square cylinder. *Physics of Fluids (1994-present)*, 11(3):560–578, 1999.

[48] A. M. Roma, C. S. Peskin, and M. J. Berger. An adaptive version of the immersed boundary method. *Journal of computational physics*, 153(2):509–534, 1999.

[49] D. Rossinelli and P. Koumoutsakos. Vortex methods for incompressible flow simulations on the gpu. *The Visual Computer*, 24(7):699–708, 2008.

[50] D. Russell and Z. J. Wang. A Cartesian grid method for modeling multiple moving objects in 2D incompressible viscous flow. *J. Comput. Phys.*, 191(1):177–205, 2003.

[51] Y. Saad. *Iterative methods for sparse linear systems*. SIAM, 2003.

[52] A. K. Sahu, R. Chhabra, and V. Eswaran. Two-dimensional unsteady laminar flow of a power law fluid across a square cylinder. *Journal of Non-Newtonian Fluid Mechanics*, 160(2):157–167, 2009.

[53] S. Schaffer. A semicoarsening multigrid method for elliptic partial differential equations with highly discontinuous and anisotropic coefficients. *SIAM Journal on Scientific Computing*, 20(1):228–242, 1998.

[54] S. Sen, S. Mittal, and G. Biswas. Flow past a square cylinder at low reynolds numbers. *International Journal for Numerical Methods in Fluids*, 67(9):1160–1174, 2011.

[55] J. H. Seo and R. Mittal. A sharp-interface immersed boundary method with improved mass conservation and reduced spurious pressure oscillations. *Journal of computational physics*, 230(19):7347–7363, 2011.

[56] A. Sharma and V. Eswaran. Heat and fluid flow across a square cylinder in the two-dimensional laminar flow regime. *Numerical Heat Transfer, Part A: Applications*, 45(3):247–269, 2004.

[57] A. Singh, A. De, V. Carpenter, V. Eswaran, and K. Muralidhar. Flow past a transversely oscillating square cylinder in free stream at low reynolds numbers. *International journal for numerical methods in fluids*, 61(6):658–682, 2009.

[58] A. Sohankar and A. Etminan. Forced-convection heat transfer from tandem square cylinders in cross flow at low reynolds numbers. *International Journal for Numerical Methods in Fluids*, 60(7):733–751, 2009.

[59] A. Sohankar, C. Norberg, and L. Davidson. Low-reynolds-number flow around a square cylinder at incidence: study of blockage, onset of vortex shedding and outlet boundary condition. *International journal for numerical methods in fluids*, 26(1):39–56, 1998.

[60] N. Suhs, S. Rogers, and W. Dietz. Pegasus 5: an automated pre-processor for overset-grid cfd. In *32nd AIAA Fluid Dynamics Conference and Exhibit*, page 3186, 2002.

[61] J. Thibault and I. Senocak. Cuda implementation of a navier-stokes solver on multi-gpu desktop platforms for incompressible flows. In *47th AIAA aerospace sciences meeting including the new horizons forum and aerospace exposition*, page 758, 2009.

[62] J. C. Thibault and I. Senocak. Accelerating incompressible flow computations with a pthreads-cuda implementation on small-footprint multi-gpu platforms. *The Journal of Supercomputing*, 59(2):693–719, 2012.

[63] D. J. Tritton. Experiments on the flow past a circular cylinder at low Reynolds numbers, 1959.

[64] Y.-H. Tseng and J. H. Ferziger. A ghost-cell immersed boundary method for flow in complex geometry. *Journal of computational physics*, 192(2):593–623, 2003.

[65] H. Udaykumar, R. Mittal, P. Rampunggoon, and A. Khanna. A sharp interface cartesian grid method for simulating flows with complex moving boundaries. *Journal of Computational Physics*, 174(1):345–380, 2001.

[66] Z. J. Wang. Two dimensional mechanism for insect hovering. *Phys. Rev. Lett.*, 85(10):2216–2219, 2000.

[67] J. E. Welch, F. H. Harlow, J. P. Shannon, and B. J. Daly. The mac method-a computing technique for solving viscous, incompressible, transient fluid-flow problems involving free surfaces. Technical report, Los Alamos Scientific Lab., Univ. of California, N. Mex., 1965.

[68] J. E. Welch, F. H. Harlow, J. P. Shannon, and B. J. Daly. The mac method. Technical report, Los Alamos Scientific Laboratory of the University of California, 1966.

[69] A. Wiegmann and K. P. Bube. The immersed interface method for nonlinear differential equations with discontinuous coefficients and singular sources. *SIAM Journal on Numerical Analysis*, 35(1):177–200, 1998.

[70] A. Wiegmann and K. P. Bube. The explicit-jump immersed interface method: finite difference methods for pdes with piecewise smooth solutions. *SIAM Journal on Numerical Analysis*, 37(3):827–862, 2000.

[71] M. Xu, F. Chen, X. Liu, W. Ge, and J. Li. Discrete particle simulation of gas–solid two-phase flows with multi-scale cpu–gpu hybrid computation. *Chemical engineering journal*, 207:746–757, 2012.

[72] S. Xu. The immersed interface method for simulating prescribed motion of rigid objects in an incompressible viscous flow. *Journal of Computational Physics*, 227(10):5045 – 5071, 2008.

[73] S. Xu. Singular forces in the immersed interface method for rigid objects in 3D. *Applied Mathematics Letters*, 22(6):827–833, 2009.

[74] S. Xu. A boundary condition capturing immersed interface method for 3D rigid objects in a flow. *Journal of Computational Physics*, 230(19):7176–7190, 2011.

[75] S. Xu and M. P. Martin. Assessment of inflow boundary conditions for compressible turbulent boundary layers. *Physics of Fluids*, 16(7):2623–2639, 2004.

[76] S. Xu and G. D. Pearson. Computing jump conditions for the immersed interface method using triangular meshes. *Journal of Computational Physics*, 302:59–67, 2015.

[77] S. Xu and Z. J. Wang. An immersed interface method for simulating the interaction of a fluid with moving boundaries. *Journal of Computational Physics*, 216(2):454–493, 2006.

[78] S. Xu and Z. J. Wang. Systematic Derivation of Jump Conditions for the Immersed Interface Method in Three-Dimensional Flow Simulation. *SIAM Journal on Scientific Computing*, 27(6):1948–1980, 2006.

[79] S. Xu and Z. J. Wang. A 3D immersed interface method for fluid-solid interaction. *Computer Methods in Applied Mechanics and Engineering*, 197(25-28):2068–2086, 2008.

[80] T. Ye, R. Mittal, H. Udaykumar, and W. Shyy. An accurate cartesian grid method for viscous incompressible flows with complex immersed boundaries. *Journal of computational physics*, 156(2):209–240, 1999.