Solving incompressible Navier-Stokes equations on heterogeneous parallel architectures

Yushan Wang

Université Paris-Sud

ÉCOLE DOCTORALE 427 : INFORMATIQUE PARIS SUD

Laboratoire de Recherche en Informatique

THÈSE DE DOCTORAT

INFORMATIQUE

par

Yushan WANG

Solving Incompressible Navier-Stokes Equations on Heterogeneous Parallel Architectures

Date de soutenance: 09/04/2015

Composition du jury:

Directeur de thèse :	Marc BABOULIN	Professeur (Université Paris-Sud, Orsay)
Co-directeur de thèse :	Olivier LE MAÎTRE	Directeur de Recherche (LIMSI/CNRS, Orsay)
Rapporteurs :	Fabienne JÉZÉQUEL	Maître de Conférences (LIP6, Paris)
	Masha SOSONKINA	Professeur (Old Dominion University, USA)
Examinateurs :	Abdel LISSER	Professeur (Université Paris-Sud, Orsay)
	Michel KERN	Chargé de Recherche (INRIA, Le Chesnay)
Membre invité :	Yann FRAIGNEAU	Ingénieur de Recherche (LIMSI/CNRS, Orsay)

Résumé

Dans cette thèse, nous présentons notre travail de recherche dans le domaine du calcul haute performance en mécanique des fluides. Avec la demande croissante de simulations à haute résolution, il est devenu important de développer des solveurs numériques pouvant tirer parti des architectures récentes comprenant des processeurs multi-cœurs et des accélérateurs. Nous nous proposons dans cette thèse de développer un solveur efficace pour la résolution sur architectures hétérogènes CPU/GPU des équations de Navier-Stokes (NS) relatives aux écoulements 3D de fluides incompressibles.

Tout d'abord nous présentons un aperçu de la mécanique des fluides avec les équations de NS pour fluides incompressibles et nous présentons les méthodes numériques existantes. Nous décrivons ensuite le modèle mathématique, et la méthode numérique choisie qui repose sur une technique de prédiction-projection incrémentale.

Nous obtenons une distribution équilibrée de la charge de calcul en utilisant une méthode de décomposition de domaines. Une parallélisation à deux niveaux combinée avec de la vectorisation SIMD est utilisée dans notre implémentation pour exploiter au mieux les capacités des machines multi-cœurs. Des expérimentations numériques sur différentes architectures parallèles montrent que notre solveur NS obtient des performances satisfaisantes et un bon passage à l'échelle.

Pour améliorer encore la performance de notre solveur NS, nous intégrons le calcul sur GPU pour accélérer les tâches les plus coûteuses en temps de calcul. Le solveur qui en résulte peut être configuré et exécuté sur diverses architectures hétérogènes en spécifiant le nombre de processus MPI, de threads, et de GPUs.

Nous incluons également dans ce manuscrit des résultats de simulations numériques pour des benchmarks conçus à partir de cas tests physiques réels. Les résultats obtenus par notre solveur sont comparés avec des résultats de référence. Notre solveur a vocation à être intégré dans une future bibliothèque de mécanique des fluides pour le calcul sur architectures parallèles CPU/GPU.

Mots clés: équations de Navier-Stokes, méthode de prédiction-projection, calcul haute performance, parallélisation multi-niveaux, calcul sur GPU.

Abstract

In this PhD thesis, we present our research in the domain of high performance software for computational fluid dynamics (CFD). With the increasing demand of high-resolution simulations, there is a need of numerical solvers that can fully take advantage of current manycore accelerated parallel architectures. In this thesis we focus more specifically on developing an efficient parallel solver for 3D incompressible Navier-Stokes (NS) equations on heterogeneous CPU/GPU architectures.

We first present an overview of the CFD domain along with the NS equations for incompressible fluid flows and existing numerical methods. We describe the mathematical model and the numerical method that we chose, based on an incremental prediction-projection method.

A balanced distribution of the computational workload is obtained by using a domain decomposition method. A two-level parallelization combined with SIMD vectorization is used in our implementation to take advantage of the current distributed multicore machines. Numerical experiments on various parallel architectures show that this solver provides satisfying performance and good scalability.

In order to further improve the performance of the NS solver, we integrate GPU computing to accelerate the most time-consuming tasks. The resulting solver can be configured for running on various heterogeneous architectures by specifying explicitly the numbers of MPI processes, threads and GPUs.

This thesis manuscript also includes simulation results for two benchmarks designed from real physical cases. The computed solutions are compared with existing reference results. The code developed in this work will be the base for a future CFD library for parallel CPU/GPU computations.

Keywords: Navier-Stokes equations, prediction-projection method, Helmholtz solver, Poisson solver, high performance computing, multi-level parallelization, GPU computing.

Acknowledgements

First of all, I would like to thank Marc Baboulin and Olivier Le Maître for acting as advisors for my thesis. I appreciate very much their help in this multidisciplinary collaborative research work and their availability.

I would also like to thank Yann Fraigneau, with whom I had detailed and fruitful discussions about the code SUNFLUIDH.

I also thank the referees of my thesis, Fabienne Jézéquel and Masha Sosonkina, for constructive comments on the manuscript.

I express my gratitude to Franck Cappello (Joint-Laboratory on Extreme Scale Computing) for funding my visit to Argonne National Laboratory, USA, in August 2013.

I want to thank Jack Dongarra and Stanimire Tomov for giving me access to their computing resources at Innovative Computing Laboratory (University of Tennessee, USA).

I acknowledge the Texas Advanced Computing Center (TACC) at The University of Texas at Austin for providing HPC resources that have contributed to the research results reported within this thesis.

Finally, I want to express my gratitude to Joël Falcou, Karl Rupp and the colleagues at LRI, who enabled me to progress in my thesis.

Contents

List of Figures

Introduction

This PhD thesis is a collaboration between the laboratories LRI[1] and LIMSI[2], in the framework of the CALIFHA project, funded by Région Île-de-France and Digitéo[3]. The main objective of this project was to take advantage of state-of-the-art parallel architectures (e.g. multicore, GPUs) in solving Navier-Stokes equations for incompressible fluid flows.

The Navier-Stokes equations play a major role in the domain of fluid dynamics since they describe a large class of fluid flows. Finding an analytical solution of the Navier-Stokes equations is one of the seven millennium problems listed by the Clay Mathematics Institute. If we do not have an analytical solution, then computing efficiently numerical solutions for these equations becomes essential. There exist many types of Navier-Stokes solvers based on various numerical methods based for instance on finite differences [21], finite elements [37, 90], finite volumes methods [31] or discontinuous Galerkin methods [84]. The importance of Navier-Stokes equations also results from their numerous fields of application. The numerical simulation of these equations is used for instance in aircraft design [53], weather forecast [45, 75], among other domains. For these application domains, the Navier-Stokes equations are used to model different types of fluid flows. For example, in the modeling of the earth atmosphere, the fluid (here the air) is considered as compressible while in the simulation of ocean tides, the fluid (water) is considered as incompressible.

In this document, we consider the Navier-Stokes equations applied to incompressible fluid flows, based on a finite difference discretization and an operator splitting method [52], the idea behind this operator splitting method being to divide the original boundary value problem into a set of subproblems that are easier to solve. Based on the Helmholtz-Hodge decomposition [99], we split the Navier-Stokes equations into an Helmholtz-like equation for an auxiliary velocity field and a Poisson equation for the auxiliary pressure. The temporal solutions are then computed via successive updates. In our work, we focus

[1] Laboratoire de Recherche en Informatique, http://www.lri.fr
[2] Laboratoire d'Informatique pour la Mécanique et les Sciences de l'Ingénieur, http://www.limsi.fr
[3] http://www.digiteo.fr

more specifically on solving the sparse linear systems arising from the discretization of Helmholtz and Poisson equations which represent the major part of the computational time in our Navier-Stokes solver. Our main objective in designing and implementing our solver was to combine the different paradigms of parallel programming (vectorization through SIMD extension, MPI message passing, multithreading with OpenMP, GPU programming with CUDA) in order to compute efficiently solutions on current heterogeneous multicore/GPU architectures. This required to identify the most time-consuming kernels in the solver and what is the most appropriate architecture/programming model to obtain the best performance for each kernel. In particular, a major concern in our algorithmic and programming choices was to exploit parallelism as much as we can but also to minimize data-communication that represents the main limitation to performance in current parallel systems. Our work in developing software for Navier-Stokes equations will be validated by numerical experiments on state-of-the-art parallel machines, including the Stampede system which is ranked #7 in the Top500 list[4].

In Chapter 1, we present the numerical method that we use for solving Navier-Stokes equations. We start with a brief overview of computational fluid dynamics and the Navier-Stokes equations. Then we give the detail of the mathematical model used in our 3D Navier-Stokes solver. For this model, we describe the spatial/temporal discretization and the proposed numerical methods. We present the derivation of the Helmholtz-like and Poisson equations as well as their numerical methods such as alternating direction implicit method [54], partial diagonalization [78] and some iterative methods.

In Chapter 2, we describe the algorithms and implementations of the Navier-Stokes solver for current multicore machines. We divide the computational domain into subdomains and computations on each subdomain are performed in parallel. We associate each subdomain to an MPI process representing a computational node of the parallel architecture. On each node, the computations are optimized using multithreading techniques. We also apply vectorization techniques to accelerate tridiagonal systems which represent the most time-consuming tasks in the Helmholtz and Poisson solvers. Performance results are presented using two different parallel machines.

In Chapter 3, we further extend the Navier-Stokes solver to the use of accelerators, namely Graphics Processing Units (GPU). We designed GPU routines for the Helmholtz and Poisson solvers and we obtained satisfactory speedups. Then we added GPU functionalities into the global Navier-Stokes solver, that can adapt automatically according to the architecture and the presence of GPUs. From the user point of view, the Navier-Stokes solver does not need to be modified to run on different architectures. The input

[4] http://www.top500.org/

parameters to be specified by the user are the number of MPI processes, the number of OpenMP threads, and possibly the number of GPUs.

In Chapter 4, we present numerical simulations on real physical problems in the domain of fluid dynamics using our CPU/GPU hybrid Navier-Stokes solver. The Taylor-Green vortex problem enables us to evaluate the accuracy of the solver with different Reynolds numbers. In the second benchmark, we simulate the flow motion around a square cylinder. Iterative methods are used in this case due to the presence of obstacle in the domain. The computed results for both benchmarks are compared with published results to validate numerically our approach. By testing our solver on these problems, we illustrate that our solver provides accurate solutions and can be applied to a wide range of applications.

Finally, we give concluding remarks and we propose possible research tracks that deserve further investigations.

Chapter 1

Navier-Stokes equations

In this chapter, we start by giving a brief history of computational fluid dynamics. We then introduce the Navier-Stokes (NS) equations considered in the thesis. We finally present the numerical method for the solution of the three-dimensional NS equations and finally discuss its discretization in the case of second order finite difference methods on Cartesian grids.

1.1 Computational fluid dynamics and Navier-Stokes equations

Computational fluid dynamics, abbreviated as CFD, is a branch of fluid mechanics which aims at solving numerically a system of partial differential equations that describes the motion of fluids. The evolutions in the CFD field is closely related to the development of modern computers architectures. Although the use of computers did not come into practice until the 1940s, the conceptual beginning of CFD can be traced back as early as 1917, when Lewis Fry Richardson (1881-1953, English mathematician, physicist and meteorologist) pioneered modern mathematical techniques of weather forecasting. One of Richardson's most celebrated achievements is his retroactive attempt to forecast the weather during a single day, 20 May 1910, by direct computation. Richardson's forecast failed dramatically because he did not apply smoothing techniques to the data [67]. When these are applied, Richardson's forecast revealed to be essentially accurate.

However, the real development of CFD started in the 1940s when Kopal compiled massive tables of the supersonic flow over sharp cones by numerically solving the governing

equations [59]. Since the 1950s, with the improvements achieved in the computer hardware, especially in the storage and execution speed, CFD began to play an important role in many scientific and engineering applications, including fluid mechanics.

1.1.1 Fluid mechanics

Fluid mechanics is a branch of physics that studies fluids (liquids, gases, and plasmas) and the forces that are applied on them. Fluid mechanics can be divided into three categories: fluid statics concerns the study of fluids at rest; fluid kinematics concerns the study of fluids in motion; and finally fluid dynamics which concerns the study of the effect of forces in fluid flows.

Early fluid mechanics studies can to traced back to ancient Greece. "Any object, wholly or partially immersed in a fluid, is buoyed up by a force equal to the weight of the fluid displaced by the object." This is the famous Archimedes' principle stated by Archimedes of Syracuse in his treatise on hydrostatics, On Floating Bodies, which is considered to be the first major work in fluid mechanics.

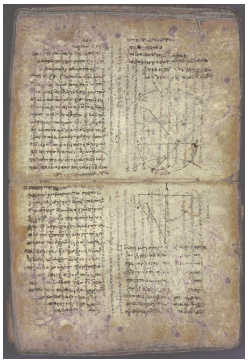


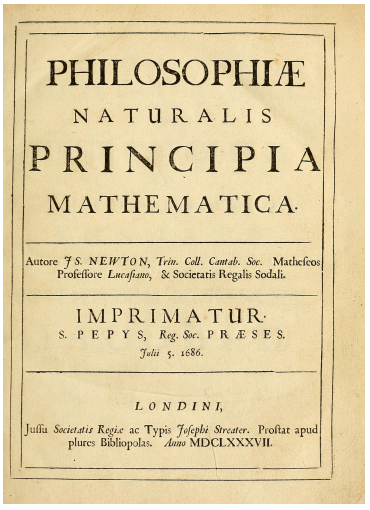
FIGURE 1.1: A page from "On Floating Bodies".

Leonardo da Vinci(1452-1519) was the first to visualize the motion of fluid particles in a flow, with his famous sketch (shown by Fig. 1.2) of a free water jet issuing from a square hole into a pool. This is perhaps the world's first use of visualization as a scientific tool to study a turbulent flow. Leonardo wrote (translated by Ugo Piomelli, University of Maryland), "Observe the motion of the surface of the water, which resembles that of hair, which has two motions, of which one is caused by the weight of the hair, the other by the direction of the curls; thus the water has eddying motions, one part of which is due to the principal current, the other to the random and reverse motion."

Sir Isaac Newton(1642-1727), an English physicist and mathematician widely recognized as one of the most influential scientists of all time and as a key figure in the scientific revolution, also played an important role in fluid mechanics. In his book *Principia*, Newton

FIGURE 1.2: A free water jet issuing from a square hole into a pool by Leonardo da Vinci.

formulated the laws of motion which dominated the scientists' views of the physical universe for the next three centuries. He also introduced the notion of Newtonian fluids in which the viscous stresses arising from its flow, at every point, are linearly proportional to the local strain rate, *i.e.* the rate of change of its deformation over time [12, 77]. In other words, in a Newtonian fluid the internal forces are proportional to the rates of change of the fluid's velocity vector as one moves away from the point in question in various directions.



PHILOSOPHIÆ
NATURALIS
PRINCIPIA
MATHEMATICA.

Autore *Is. NEWTON, Trin. Coll. Cantab. Soc.* Matheſeos Profeſſore *Lucaſiano*, & Societatis Regalis Sodali.

IMPRIMATUR.
S. PEPYS, *Reg. Soc.* PRÆSES.
Julii 5. 1686.

LONDINI,
Juſſu *Societatis Regiæ* ac Typis *Joſephi Streater.* Proſtat apud plures Bibliopolas. *Anno* MDCLXXXVII.

FIGURE 1.3: Title page of "Principia", first edition (1687).

From the publication of *Principia* in 1687 to the 1950's, researches in fluid mechanics were mainly divided theoretical and experimental approaches. In theoretical researches, scientists focus mainly on the derivation of governing equations for the flow and on the subsequent solution of these equations, while experimentalists perform measurements on real flows to investigate the behavior of the flows and their dynamics and provide material in order to confirm and validate theories. Before the emergence of computers, the resolution of the governing equations was limited to simple situations for which explicit solutions could be derived analytically, limiting essentially computable predictions to linear situations.

After the 50s and the starting availability of computers, the CFD approach started to have an increasing impact on fluid dynamics. Today, computational approaches are used to support theoretical investigations and is considered as a substitute to the experimental approach in many situations. This success is mostly due to the continuous increase in the computational resources available and the development of ever more efficient numerical methods dedicated to the resolution of the equations describing flows. Another success reason for the success of CFD is its relatively cheap cost (compared to the experimental approach) and also the possibility to conduct simulations for situations that can not be physically controlled in a real experiment. Examples of CFD application domains are the calculation of forces on aircrafts, the determination of the mass flow rate of petroleum through pipelines, weather forecast, the simulation of nebulae in interstellar space, the hydrodynamical modeling nuclear weapon, ...

1.1.2 Equations of fluid dynamics

In this section, we briefly recall the principles for the derivations for the fundamental governing equations [12] of fluid dynamics.

We start be summarizing the principal properties of fluids and flows:

- **Compressibility.** All fluids are compressible to some extent, that is, changes in pressure or temperature will result in changes in density. However, in many situations the changes in pressure and temperature are sufficiently small that the changes in density can be neglected. In this case the flow can be modeled as an incompressible flow [27, 106]. For flow of gases, to determine whether to use compressible or incompressible fluid models, the Mach number[1] of the flow is to be evaluated. As a rough guide, compressible effects can be ignored at Mach numbers below approximately 0.3. For liquids, whether the incompressible assumption is valid depends on the fluid properties (specifically the critical pressure and temperature of the fluid) and the flow conditions (how close to the critical pressure the actual flow pressure becomes). Acoustic problems always require allowing compressibility, since sound waves are compression waves involving changes in pressure and density of the medium through which they propagate. In this thesis we only consider flows of incompressible fluids.

- **Viscosity.** We can find in [68] a simple definition of (dynamic) viscosity: all fluids offer resistance to any force tending to cause one layer to move over another.

[1] In fluid mechanics, Mach number, named after Austrian physicist and philosopher Ernst Mach(1838-1916), is a dimensionless quantity representing the ratio of the characteristic fluid velocity and the local speed of sound.

Viscosity, often noted by μ, is the fluid property responsible for this resistance. It is a matter of common experience that, under particular conditions, one fluid offers greater resistance to flow than another. Liquids such as tar, treacle and glycerine can not be rapidly poured or easily stirred, and are usually spoken of as thick. On the other hand, so-called thin liquids such as water, petrol and paraffin flow much more readily. In fact, for fluids in motion, the ability to transmit a shear force introduces the property of dynamic viscosity. It is found empirically that for a large class of fluids the shear stress is directly proportional to the velocity gradient with the dynamic viscosity as the constant of proportionality. Such fluids are recognized as Newtonian fluids. In this thesis we only consider Newtonian fluids.

- **Steady and unsteady flows.** Steady-state flow refers to the condition where the fluid properties at a point in the system do not change over time. Otherwise, the flow is said unsteady (sometimes called transient). Whether a particular flow is steady or unsteady, can depend on the chosen frame of reference. For instance, the laminar flow over a translating sphere can be steady in the frame of reference that is stationary with respect to the sphere. In a frame of reference that is stationary with respect to a background flow, the flow is unsteady.

The partial differential equations formulation of the fluid flow is based on the continuum hypothesis and expresses the fundamental conservation laws of physics. From the microscopic perspective, a fluid consists of molecules which are individually in a state of random motion. By the continuum hypothesis, only the large-scale (macroscopic) motion of these molecules is perceived; therefore the various properties of the fluid in motion are assumed to vary continuously with position and time. Usually, the physical properties of interest are the density, the pressure, the temperature and the velocity (or the momentum).

There are two ways to define a coordinate system to describe a fluid in motion, namely the Eulerian description and the Lagrangian description. In the Eulerian description, the values of the velocity and thermodynamical properties are specified at a fixed location in the space-time domain, *i.e.* are defined as functions of (x, y, z, t). The alternative Lagrangian description traces along time individual particles (fluid parcels) from their positions and thermodynamical properties which are dependent variables. We shall adopt the Eulerian description throughout this thesis.

In deriving the governing equations of fluid dynamics, it is postulated that mass, momentum and energy are conserved. In order to derive the mathematical formulations of the corresponding conservation laws, we consider a closed control volume V of the

fluid domain, which is fixed in space and time in an Eulerian coordinate system. The boundary of V is an orientable surface S, with the unit vector $\mathbf{n}$ normal to S pointing from the inside of V toward the outside.

1.1.2.1 Conservation of mass

The mass conservation states that the rate of accumulation of mass in a volume V is exactly balanced by the mass flux across its boundary S; it expresses as

$$\frac{\partial}{\partial t}\int_V \rho\, dV + \int_S \rho \mathbf{n}\cdot\mathbf{u}\, dS = 0, \tag{1.1}$$

where ρ is the density and $\mathbf{u}$ the fluid velocity vector. The surface integral in Eq. (1.1) can be converted into a volume integral by Gauss' divergence theorem[2], hence Eq. (1.1) can be written as

$$\int_V \left[\frac{\partial}{\partial t} + \nabla\cdot(\rho\mathbf{u})\right] dV = 0, \tag{1.2}$$

which gives the integral form of mass conservation. From now on we further assume that all functions considered are continuous and sufficiently differentiable. Since V is arbitrary, we must have

$$\frac{\partial \rho}{\partial t} + \nabla\cdot(\rho\mathbf{u}) = 0. \tag{1.3}$$

By simple vector analysis, Eq. (1.3) implies

$$\frac{\partial \rho}{\partial t} + \mathbf{u}\cdot\nabla\rho + \rho\nabla\cdot\mathbf{u} = 0, \tag{1.4}$$

or

$$\frac{D\rho}{Dt} + \rho\nabla\cdot\mathbf{u} = 0, \tag{1.5}$$

where $\dfrac{D\rho}{Dt} = \dfrac{\partial\rho}{\partial t} + \mathbf{u}\cdot\nabla\rho$ is called the material derivative or substantial derivative of ρ. The material derivative is often used in fluid dynamics, which specifies the rate of change of a physical quantity when moving with the fluid flow.

The Eqs. (1.3) and (1.5) are often referred to as forms of the continuity equation. If the fluid is incompressible, the density ρ is constant with respect to both space and time, hence $\dfrac{D\rho}{Dt} = 0$ and

$$\nabla\cdot\mathbf{u} = 0, \tag{1.6}$$

which is the incompressibility condition for the flow field.

[2]Suppose V is a subset of $\mathbb{R}^n$ (in the case of $n = 3$, V represents a volume in 3D space) which is compact and has a piecewise smooth boundary S. If $\mathbf{F}$ is a continuously differentiable vector field defined on a neighborhood of V, then we have: $\int_V \nabla\cdot\mathbf{F}\, dV = \int_S \mathbf{F}\cdot\mathbf{n}\, dS$

1.1.2.2 Conservation of momentum

The conservation of momentum states that the rate of accumulation of momentum in V plus the flux of momentum out through S is equal to the rate of gain of momentum due to body forces and surface stresses. Mathematically it leads to

$$\frac{\partial}{\partial t}\int_V \rho\,\mathbf{u}\,dV + \int_S \rho\,(\mathbf{n}\cdot\mathbf{u})\mathbf{u}\,dS = \int_V \rho\,\mathbf{f}\,dV + \int_S \mathbf{n}\cdot\sigma\,dS, \tag{1.7}$$

where $\mathbf{f}$ is the body force and σ is the stress tensor. Using Gauss' divergence theorem, Eq. (1.7) yields

$$\int_V \left[\frac{\partial}{\partial t}(\rho\mathbf{u}) + \nabla\cdot(\rho\mathbf{u}\mathbf{u})\right] dV = \int_V (\rho\,\mathbf{f} + \nabla\cdot\sigma)\,dV, \tag{1.8}$$

where $\mathbf{uu}$ stands for the dyadic or tensor product. From Eq. (1.8), we again use the argument that the volume V is arbitrary to obtain

$$\frac{\partial}{\partial t}(\rho\mathbf{u}) + \nabla\cdot(\rho\mathbf{u}\mathbf{u}) = \rho\,\mathbf{f} + \nabla\cdot\sigma, \tag{1.9}$$

and by vector analysis

$$\mathbf{u}\frac{\partial\rho}{\partial t} + \rho\frac{\partial\mathbf{u}}{\partial t} + \rho\mathbf{u}\cdot\nabla\mathbf{u} + \mathbf{u}\nabla\cdot(\rho\mathbf{u}) = \rho\,\mathbf{f} + \nabla\cdot\sigma. \tag{1.10}$$

The continuity Eq. (1.3) implies that the first and last terms on the left hand side of Eq. (1.10) must cancel, hence

$$\rho\frac{\partial\mathbf{u}}{\partial t} + \rho\mathbf{u}\cdot\nabla\mathbf{u} = \rho\,\mathbf{f} + \nabla\cdot\sigma, \tag{1.11}$$

or in terms of material derivative,

$$\rho\frac{D\mathbf{u}}{Dt} = \rho\,\mathbf{f} + \nabla\cdot\sigma. \tag{1.12}$$

Eq. (1.11) and (1.12) are usually referred to as the equation of motion.

For Newtonian fluids, the constitutive relationship for the stress tensor σ is given by Newton's law as

$$\begin{cases} \sigma = -p\mathbf{I} + \tau, & \text{(1.13a)} \\ \tau = \lambda(\nabla\cdot\mathbf{u})\mathbf{I} + 2\mu\,\mathbf{D}(\mathbf{u}), & \text{(1.13b)} \end{cases}$$

where τ is the deviatoric stress tensor, $\mathbf{D}(\mathbf{u}) = \frac{1}{2}\left[\nabla\mathbf{u} + \nabla\mathbf{u}^T\right]$ is the rate-of-strain tensor, p is the pressure, μ the dynamic viscosity, and λ the second coefficient of viscosity.

Introducing Eqs. (1.13a) and (1.13b) into Eq. (1.12) yields

$$\rho\frac{D\mathbf{u}}{Dt} = \rho\,\mathbf{f} - \nabla p + \mu\Delta\mathbf{u} + (\lambda+\mu)\nabla(\nabla\cdot\mathbf{u}) + (\nabla\cdot\mathbf{u})\nabla\lambda + 2\ \mathbf{D}(\mathbf{u})\cdot\nabla\mu. \tag{1.14}$$

In the case of an incompressible fluid with constant viscosity, the equation of motion reduces to

$$\frac{\partial\mathbf{u}}{\partial t} + \mathbf{u}\cdot\nabla\mathbf{u} = -\frac{1}{\rho}\nabla p + \nu\Delta\mathbf{u} + \mathbf{f}, \tag{1.15}$$

where $\nu = \frac{\mu}{\rho}$ is called kinematic viscosity. The Eq. (1.15) is one of the most frequently encountered governing equation in fluid dynamics, known as the Navier-Stokes equation, which was derived independently by Claude-Louis Navier (1785-1836), and George Gabriel Stokes (1819-1903). In the literature, Eqs. (1.6) and (1.15) are jointly referred to as the Navier-Stokes equations for viscous incompressible flow.

1.1.2.3 Conservation of energy

The conservation of energy in the control volume V means that the rate of accumulation of energy plus the flux of energy out through S is equal to the flux of heat in through S plus the rate of gain of energy due to surface stresses (dissipation). Mathematically this principle implies

$$\frac{\partial}{\partial t}\int_V \rho E\,dV + \int_S \rho E(\mathbf{n}\cdot\mathbf{u})\,dS = -\int_S \mathbf{n}\cdot\mathbf{q}\,dS + \int_S \mathbf{n}\cdot(\sigma\cdot\mathbf{u})\,dS. \tag{1.16}$$

In Eq. (1.16), $\mathbf{q}$ is the heat-flux vector and E is the total specific energy given by

$$E = e + \frac{1}{2}\mathbf{u}^2 - \mathbf{f}\cdot\mathbf{u}, \tag{1.17}$$

where e is the specific internal energy, $\frac{1}{2}\mathbf{u}^2$ the specific kinetic energy, and $-\mathbf{f}\cdot\mathbf{u}$ the specific potential energy. By using Gauss' divergence theorem we have

$$\int_V \left[\frac{\partial}{\partial t}(\rho E) + \nabla\cdot(\rho E\mathbf{u})\right] dV = \int_V \left[-\nabla\cdot\mathbf{q} + \nabla\cdot(\sigma\cdot\mathbf{u})\right] dV. \tag{1.18}$$

Again, since V is arbitrary, it comes

$$\frac{\partial}{\partial t}(\rho E) + \nabla\cdot(\rho E\mathbf{u}) = -\nabla\cdot\mathbf{q} + \nabla\cdot(\sigma\cdot\mathbf{u}). \tag{1.19}$$

It can be shown, in [83], that Eq. (1.19) leads to

$$\rho\frac{De}{Dt} = -\nabla\cdot\mathbf{q} - p\nabla\cdot\mathbf{u} + \tau : \nabla\mathbf{u}, \tag{1.20}$$

where the double dot product of two tensors S and T is defined as $S : T = \text{trace}(S, T)$. The Eq. (1.20) expresses the first law of thermodynamics. Further discussions on the energy Equation (1.20) can be found in [83].

1.1.3 The dimensionless Navier-Stokes equations

In this thesis, we shall restrict ourselves to the case of isothermal incompressible flows with uniform density ρ and viscosity ν. Within these restrictions, the energy equation is irrelevant (no thermodynamical effects) and the flow is entirely governed by Eqs. (1.6) and (1.15) which are conveniently recast in a dimensionless form.

Dimensionless quantities are constructed considering the fluid density ρ, viscosity ν, and characteristic (reference) length L and velocity U. Using these references, we define

$$\overline{\mathbf{x}} = \frac{\mathbf{x}}{L}, \quad \overline{\mathbf{u}} = \frac{\mathbf{u}}{L}, \quad \bar{t} = \frac{U}{L}t, \quad \overline{p} = \frac{p}{\rho U^2}, \quad \overline{\mathbf{f}} = \frac{L}{U^2}\mathbf{f}$$

where the upper bars denote the dimensionless character of the quantities. The incompressible Navier-Stokes equations can be recast in terms of the dimensionless quantities, yielding the dimensionless form of (1.6) and (1.15)

$$\begin{cases} \nabla \cdot \mathbf{u} = 0, & \text{(1.21a)} \\ \dfrac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} = -\nabla p + \dfrac{1}{\text{Re}} \Delta \mathbf{u} + \mathbf{f}, & \text{(1.21b)} \end{cases}$$

where for notational convenience we have denoted the dimensionless variables by the same symbols as the corresponding dimensional ones.

The non-dimensional number $\text{Re} = \dfrac{LU}{\nu}$, known as the Reynolds number, is one of the most important non-dimensional numbers in fluid dynamics. The magnitude of Re measures how large are the inertial effects compared to the effects of the viscous dissipation in a particular fluid flow. For $\text{Re} \ll 1$, one can neglect the nonlinearity (inertial effects) and the solution of the Navier-Stokes equations can be found in closed-form in many instances [62]. In many flows of interest Re is very large. For example, river flows have Reynolds number as high as $\text{Re} \approx 10^7$. For $\text{Re} \gg 1$ there is no stable steady solution of the equations of motion. The solutions are strongly affected by the nonlinearity, and the actual flow pattern is complicated, convoluted and vortical. When such inertial instabilities are fully developed, the flow is said turbulent.

1.2 Numerical methods for the incompressible Navier-Stokes equations

We consider the unsteady flow of a incompressible Newtonian fluid governed by the Navier-Stokes (NS) equations. Recall the dimensionless form of the NS equations:

$$\begin{cases} \dfrac{\partial \mathbf{u}}{\partial t} + (\mathbf{u}\cdot\nabla)\mathbf{u} = -\nabla p + \dfrac{1}{\mathrm{Re}}\Delta\mathbf{u}, & \text{(1.22a)} \\ \nabla\cdot\mathbf{u} = 0. & \text{(1.22b)} \end{cases}$$

There exist many numerical methods for the discretization and resolution of the Navier-Stokes equations. The most common ones are the Finite Difference (FD) [8, 20, 31, 46, 102], Finite Volume (FV) [31, 102], Finite Element (FE) [41, 82] and Spectral Element (SE) [19].

All of these methods are based on Eulerian formulations of the governing equations. Mesh-free methods, including Particles and Vortex Methods (VM) [24] and Smoothed Particle Hydrodynamics (SPH) [65] are on the contrary based on Lagrangian formulations. All these methods have their own advantages and disadvantages, making them more or less suitable for the discretization of a given problem.

For instance, Particle methods naturally deal with flows in unbounded domains, but require great effort to accurately enforce boundary conditions on solid surfaces. Regarding Eulerian methods, FD methods are often considered more simple to implementation, in the case of cartesian meshes, compared to other methods, but the treatment for curved domains and the introduction of adaptive strategies can become quickly cumbersome. FV methods are often chosen for they enforce the "physical" conservation properties of the continuous equations at the discrete level; on the contrary to FE approaches calls for a special discretization to be conservative. However, high-order FV schemes are difficult to derive, especially in the case of non cartesian meshes, while FE methods, in particular the Discontinuous Galerkin method [47], are naturally amenable to higher order discretization schemes (polynomial approximation) and flexible enough to accommodate general meshes with local adaptation. However, FE methods can suffer from numerical stability issues, requiring stabilization fixes, and the discrete functional spaces must be selected with care in order to ensure the well-posedness of the weak formulation (*e.g.* the famous inf-sup condition, see [28]).

As discussed below, they are two central difficulties when solving the incompressible Navier-Stokes equations for a Newtonian fluid. The first one concerns the non-linearities induced by the convective term. The second issue concerns the enforcement of the

mass conservation and the role of the pressure term which has no thermodynamical significance.

Concerning the non-linearity, it is very common to rely on explicit treatments (in time) of the convective term to end with the resolution of linear problems for the time-advancement of the solution and so avoid the need to solve a full set of non-linear equations. However, even when treated explicitly, the non-linearity continues to manifest itself by the emergence of small structure in the flow dynamic. When the convective effects are important (cases of high Reynolds number flows) the discretization must be fine enough to properly accounts for all the scales present in the dynamics. This means in practice that one must use finer and finer spatial and temporal discretizations when the Reynolds number increased, with large computational costs as a result.

Concerning the enforcement of the mass conservation, the divergence-free constraint introduces an algebraic constraint on the velocity field, and the pressure is understood as the Lagrange multiplier associated to this constraint. The coupled velocity-pressure problem, for an explicit treatment of the non-linear terms (Stokes problem), has a saddle-node structure which makes its resolution difficult. For instance, the use of preconditioners is critical for an efficient iterative resolution of such saddle-node problem. Different alternatives have then been proposed to enforce the divergence-free character of the velocity field. For instance, the artificial compressibility method [20] re-introduces a pseudo state equation relating the rate of change of the pressure, $\frac{\partial p}{\partial t}$ to the divergence of the velocity field. Tuning carefully this pseudo state equation allows to control the divergence of the velocity field, but introduces an additional stiffness in the governing equations. Alternatively, the prediction-projection approach [17, 20, 40, 58], considered in this thesis and detailed below in Section 1.3, introduces an auxiliary elliptic problem for the determination of the pressure with an exact enforcement of the divergence-free character of the velocity field.

In our work, we apply the incremental prediction-projection method which is originally implemented in the 3D Navier-Stokes solver SUNFLUIDH. Within this method, we will apply the alternating direction implicit and the partial diagonalization methods to solve the subproblems resulting from the prediction-projection method.

1.3 Incremental prediction-projection method

For the time derivation term approximation, we consider the second order Euler's scheme, also known as the two-step backward differentiation formula (BDF2) [9, 48],

with a constant time step Δt. We denote $\mathbf{u}^{(n)}$ and $p^{(n)}$ the approximation of the velocity and pressure fields at time $t_n = n\Delta t$. For the BDF2 formula, the approximation of the time derivative writes

$$\frac{\partial \mathbf{u}}{\partial t}^{(n+1)} = \frac{3\mathbf{u}^{(n+1)} - 4\mathbf{u}^{(n)} + \mathbf{u}^{(n-1)}}{2\Delta t} + \mathbf{O}(\Delta t^2).$$

Further, denoting $NL(\mathbf{u}) = \mathbf{u} \cdot \nabla \mathbf{u}$ the non-linear term of the momentum equation, and considering a implicit discretization of the NS equation, we have to solve at each time step the semi-discrete problem

$$\frac{3\mathbf{u}^{(n+1)} - 4\mathbf{u}^{(n)} + \mathbf{u}^{(n-1)}}{2\Delta t} + NL(\mathbf{u})^{(n+1)} = -\nabla p^{(n+1)} + \frac{1}{\mathrm{Re}}\Delta \mathbf{u}^{(n+1)}, \tag{1.23}$$

$$\nabla \cdot \mathbf{u}^{(n+1)} = 0. \tag{1.24}$$

The non-linear character of the semi-discrete problem is first removed by approximating explicitly $NL(\mathbf{u})^{(n+1)}$. Possible choices are linear time extrapolations, using for instance

$$NL(\mathbf{u})^{(n+1)} \approx NL(\mathbf{u})^{(*)} = 2\mathbf{u}^{(n)} \cdot \nabla \mathbf{u}^{(n)} - \mathbf{u}^{(n-1)} \cdot \nabla \mathbf{u}^{(n-1)},$$

or

$$NL(\mathbf{u})^{(n+1)} \approx NL(\mathbf{u})^{(*)} = \frac{3}{2}\mathbf{u}^{(n)} \cdot \nabla \mathbf{u}^{(n)} - \frac{1}{2}\mathbf{u}^{(n-1)} \cdot \nabla \mathbf{u}^{(n-1)}.$$

Substituting such an approximation for NL yields a linear problem for $u^{(n+1)}$ and $p^{(n+1)}$, but maintain the velocity-pressure coupling. The decoupling is achieved through an incremental prediction-projection method. In a first step, a prediction of $u^{(n+1)}$, denoted $\mathbf{u}^*$, is computed by solving

$$\frac{3\mathbf{u}^* - 4\mathbf{u}^{(n)} + \mathbf{u}^{(n-1)}}{2\Delta t} + NL(\mathbf{u})^{(*)} = -\nabla p^{(n)} + \frac{1}{\mathrm{Re}}\Delta \mathbf{u}^*, \tag{1.25}$$

that is making explicit the pressure and disregarding the divergence constraint. The efficient resolution of the (Helmholtz) prediction problem for $\mathbf{u}^*$ is a central contribution of the thesis and will be further discussed later in the chapter.

Whence the velocity prediction has been computed, u^* must be corrected to enforce the divergence free condition $\nabla \cdot \mathbf{u}^{(n+1)} = 0$ and the pressure must be updated to $p^{(n+1)}$. The correction equation for $\mathbf{u}^{(n+1)} - \mathbf{u}^*$ is obtained by taking the difference of Eqs. (1.23) and (1.25). It comes to

$$\frac{3\mathbf{u}^* - 3\mathbf{u}^{(n+1)}}{2\Delta t} = \nabla(p^{(n+1)} - p^{(n)}) + \frac{1}{\mathrm{Re}}\Delta(\mathbf{u}^* - \mathbf{u}^{n+1}). \tag{1.26}$$

Following the Helmholtz-Hodge decomposition (HHD) [99], the velocity correction is sought as the gradient of a potential ϕ, that is

$$\mathbf{u}^{(n+1)} = \mathbf{u}^* - \nabla\phi. \tag{1.27}$$

Using the divergence-free condition $\nabla \cdot \mathbf{u}^{(n)} = 0$, the correction potential is solution of the Poisson equation

$$\Delta\phi = \nabla \cdot \mathbf{u}^*. \tag{1.28}$$

The Poisson Eq. (1.28) along with appropriate boundary conditions allow us to compute the velocity field $\mathbf{u}^{(n+1)}$ using Eq. (1.27) and to update the pressure from Eq. (1.26):

$$p^{(n+1)} = p^{(n)} + \frac{3}{2\Delta t}\phi - \frac{1}{\mathrm{Re}}\nabla \cdot \mathbf{u}^*. \tag{1.29}$$

It is seen that the incremental prediction-projection approach amounts to solve at every time step a system of Helmholtz equations for the velocity prediction $\mathbf{u}^*$, followed by the resolution of a Poisson equation for the velocity correction potential ϕ. These are the main computationally expensive steps for the time-integration of the incompressible Navier-Stokes equations. The principal contribution of the thesis consists in proposing efficient implementations for these two steps. In Section 1.4 we present the Alternating Direction Implicit method used for the resolution of the Helmholtz system, and in Section 1.5 we discuss the resolution of the Poisson equation using Partial Diagonalization.

1.4 Solution of the Helmholtz system

The incremental prediction-projection method introduced above leads to a prediction step consisting in the resolution of the Helmholtz system (1.25) for $\mathbf{u}$, which can be recast in

$$\left(\mathbf{I} - \frac{2\Delta t}{3\mathrm{Re}}\Delta\right)\mathbf{u}^* = \mathbf{S}, \tag{1.30}$$

where

$$\mathbf{S} = \frac{4\mathbf{u}^{(n)} - \mathbf{u}^{(n-1)}}{3} - \frac{2\Delta t}{3}\left(NL(\mathbf{u})^{(*)} + \nabla p^{(n)}\right). \tag{1.31}$$

In this section, we introduce the Alternating Direction Implicit (ADI) method for the resolution of this system. The central idea of the ADI method is to approximate the 3-dimensional Helmholtz operator into a product of one-dimensional operators acting along the 3 spatial coordinates:

$$\left(\mathbf{I} - \alpha\frac{\Delta t}{\mathrm{Re}}\Delta\right) \approx \left(\mathbf{I} - \alpha\frac{\Delta t}{\mathrm{Re}}\Delta_x\right)\left(\mathbf{I} - \alpha\frac{\Delta t}{\mathrm{Re}}\Delta_y\right)\left(\mathbf{I} - \alpha\frac{\Delta t}{\mathrm{Re}}\Delta_z\right), \tag{1.32}$$

with $\alpha = 2/3$.

However, applying directly the approximation of the Helmholtz operator to Eq. (1.30) leads to a numerical error which is of the first order in time. The second order time-accuracy can be recovered by formulating the prediction problem in terms of the velocity increment $\delta\mathbf{u}^* = \mathbf{u}^* - \mathbf{u}^{(n)}$. This is achieved by rewriting the prediction problem as follows:

$$\frac{3(\mathbf{u}^* - \mathbf{u}^{(n)}) - \mathbf{u}^{(n)} + \mathbf{u}^{(n-1)}}{2\Delta t} + NL(\mathbf{u})^{(*)} = -\nabla p^{(n)} + \frac{1}{\mathrm{Re}}\Delta(\mathbf{u}^* - \mathbf{u}^{(n)}) + \frac{1}{\mathrm{Re}}\Delta\mathbf{u}^{(n)}, \quad (1.33)$$

leading to the Helmholtz system for the velocity increment

$$\left(\mathbf{I} - \frac{2\Delta t}{3\mathrm{Re}}\Delta\right)\delta\mathbf{u}^* = \delta\mathbf{S}, \quad (1.34)$$

where

$$\delta\mathbf{S} = \frac{\mathbf{u}^{(n)} - \mathbf{u}^{(n-1)}}{3} - \frac{2\Delta t}{3}\left(NL(\mathbf{u})^{(*)} + \nabla p^{(n)} - \frac{1}{\mathrm{Re}}\Delta\mathbf{u}^{(n)}\right). \quad (1.35)$$

As previously mentioned, the Helmholtz equation is then approximated using the ADI method, which consists of approximating the 3D operator by a product of one-dimensional operators

$$\left(\mathbf{I} - \alpha\frac{\Delta t}{\mathrm{Re}}\Delta_x\right)\left(\mathbf{I} - \alpha\frac{\Delta t}{\mathrm{Re}}\Delta_y\right)\left(\mathbf{I} - \alpha\frac{\Delta t}{\mathrm{Re}}\Delta_z\right)\delta\mathbf{u}^* = \delta\mathbf{S}. \quad (1.36)$$

With this operator decomposition, we can substitute Eq. (1.36) by a sequence of three one-dimensional equations:

$$\begin{cases} \left(\mathbf{I} - \dfrac{\alpha\Delta t}{\mathrm{Re}}\Delta_x\right)\mathbf{T}_1 = \delta\mathbf{S}, & (1.37a) \\ \left(\mathbf{I} - \dfrac{\alpha\Delta t}{\mathrm{Re}}\Delta_y\right)\mathbf{T}_2 = \mathbf{T}_1, & (1.37b) \\ \left(\mathbf{I} - \dfrac{\alpha\Delta t}{\mathrm{Re}}\Delta_z\right)\delta\mathbf{u}^* = \mathbf{T}_2. & (1.37c) \end{cases}$$

The main advantage of using the ADI method comes from the fact that for meshes consisting of orthogonal Cartesian grids supporting a centered second-order finite-difference spatial discretization, the discrete one-dimensional operators are well conditioned tridiagonal systems. These three diagonal systems can be efficiently solved with Thomas algorithm [92].

The previous development can also been applied to other time-discretization schemes. In this thesis, we also consider the Crank-Nicolson (CN) scheme for prediction problem.

The CN scheme for the prediction problem is written as:

$$\frac{\mathbf{u}^* - \mathbf{u}^{(n)}}{\Delta t} + NL(\mathbf{u})^{(*)} = -\nabla p^{(n)} + \frac{1}{2\mathrm{Re}}\left(\Delta \mathbf{u}^{(n)} + \Delta \mathbf{u}^*\right). \tag{1.38}$$

The incremental version of the CN scheme is

$$\left(\mathbf{I} - \frac{\Delta t}{2\mathrm{Re}}\Delta\right)\delta\mathbf{u}^* = \delta\mathbf{S}, \tag{1.39}$$

with

$$\delta\mathbf{S} = -\Delta t\left(\nabla p^{(n)} + NL(\mathbf{u})^{(*)} - \frac{1}{\mathrm{Re}}\Delta\mathbf{u}^{(n)}\right). \tag{1.40}$$

The ADI resolution of the incremental problem for the CN scheme can be carried-out in a similar fashion as for the BDF-2 scheme, using $\alpha = 1/2$ in the definition of the one-dimensional operators. Both CN and BDF2 schemes are implemented in the code. In this thesis, we will keep the BDF2 scheme for demonstrating purpose.

1.5 Solution of the Poisson equation

There exists many methods to solve the Poisson equation. In this section, we present a direct method based on the eigendecomposition [85] of square matrices. The method is called Partial Diagonalization. The applicability of partial diagonalization method is most advantageous in the case of separable problems discretized on meshes consisting of orthogonal Cartesian grids.

Given a real square non-singular matrix $A \in \mathbb{R}^{n\times n}$, the diagonalization of A corresponds to the factorization of A in

$$A = Q\,\Lambda\,Q^{-1},$$

where $\Lambda = \{\lambda_i\}_{i=1,2,\ldots,n}$ is a diagonal matrix whose diagonal elements are the eigenvalues of A, and Q is the square matrix whose $i-$th column is the eigenvector q_i of A.

Now take A the discrete Laplacian operator in Eq. (1.28). We can write formally

$$A = \Delta = \Delta_x + \Delta_y + \Delta_z,$$

where Δ_x, Δ_y and Δ_z are the discrete versions of the $\partial^2/\partial x^2$, $\partial^2/\partial y^2$ and $\partial^2/\partial z^2$ operators respectively.

The principle of the Partial diagonalization method is to diagonalize the components of Laplacian operator associated with 2 directions (more generally, diagonalization along $N-1$ directions if N is the number of spatial dimensions of the problem).

For instance, diagonalizing Δ_x, and Δ_y we obtain

$$(Q_x \Lambda_x Q_x^{-1} + Q_y \Lambda_y Q_y^{-1} + \Delta_z)\phi = S, \tag{1.41}$$

where we have denoted as S the right-hand-side vector of the discretized Poisson equation for simplicity.

Multiplying Eq. (1.41) with $Q_x^{-1}Q_y^{-1}$ and using the following properties of the Q_x and Q_y matrices,

$$\begin{cases} Q_x^{-1}Q_y^{-1} = Q_y^{-1}Q_x^{-1}, \\ Q_x^{-1}\Lambda_y^{-1} = \Lambda_y^{-1}Q_x^{-1}, \\ Q_y^{-1}\Lambda_x^{-1} = \Lambda_x^{-1}Q_y^{-1}, \end{cases}$$

we obtain

$$(\Lambda_x + \Lambda_y + \Delta_z)\tilde{\phi} = \tilde{S}, \tag{1.43}$$

where

$$\begin{cases} \tilde{\phi} = Q_y^{-1}Q_x^{-1}\phi, \\ \tilde{S} = Q_y^{-1}Q_x^{-1}S. \end{cases}$$

For a separable problem with regular Cartesian grids, the matrices Q_x and Q_y have specific structures that can be exploited for an efficient parallel implementation. In addition, for such grids and using the classical second order central finite difference scheme, the resulting operator $\Lambda_x + \Lambda_y + \Delta_z$ is tridiagonal and well conditioned, allowing the use of the Thomas Algorithm mentioned in Section 1.4. This method is then faster than any other iterative methods. An important remark concerning Eq. (1.43) is that when using homogeneous Neumann boundary conditions there will be zero eigenvalues which cause singularities in the system. These singularities can be removed by introducing artificial Dirichlet boundary conditions on the free-modes when discretizing the problem.

For non-separable problems, we use the successive over-relaxation (SOR) and multigrid methods, as described in Appendix A, which are also very successful for solving elliptic and hyperbolic partial differential equations and the linear systems that arise when they are discretized. As explained previously, in a domain with obstacles, the partial diagonalization method is not applicable. Thus an iterative method such as SOR or Jacobi will be used to solve the Poisson equation.

1.6 Spatial discretization

As stated above, the thesis considers simple computational domains consisting in right-rectangular prisms, with orthogonal principal directions aligned with the x, y, and z directions. The three-dimensional domain is discretized using a Cartesian grids resulting from the tensorization of one dimensional grids in the x, y and z, which supports centered second-order finite-difference schemes for the approximations of the first and second order derivatives in the respective directions. However, the velocity and pressure unknowns are defined over different Cartesian grids as explained below.

1.6.1 Staggered mesh

As illustrated in Fig. 1.4, for a two-dimensional mesh, the computational domain is discretized into rectangular parallelepipeds (shown with black solid lines, referred to as cells). The discrete velocity and pressure unknowns are defined along a staggered arrangement, in which the pressure points are located at the center of each cell, and the velocity components are defined on the center of the corresponding edges.

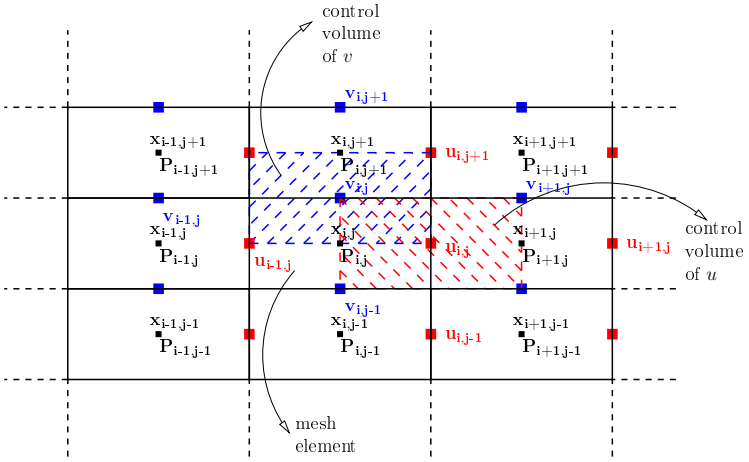


FIGURE 1.4: Staggered mesh showing the pressure (black squares at the cells' center) and velocity components unknowns (red and blue squares for the x and y components, respectively).

The staggered arrangement is widely used in CFD because it prevents spurious pressure oscillations [64]. For this type of arrangement, one can also easily construct second order approximations of the gradient, Laplacian, and divergence operators that verify the classical identities of vector analysis. This so-called mimetic discretization provides a better numerical stability for the prediction-projection method compared to a discretization where all the variables are defined on the same points of the mesh.

For the sake of simplicity, consider a two-dimensional rectangle domain D, of length L_x and heigth L_y. The position of each point of the domain is defined within a orthogonal Cartesian landmark in which the origin is placed in the left lower corner. The mesh is orthogonal Cartesian and the (two-dimensional) domain discretization have $N_x \times N_y$ mesh cells. The mesh associated to the pressure, that we note as $\mathcal{M}_c$, is such that:

- The discrete coordinates $x_c(i)$ and $y_c(j)$ define the center's position of mesh cell $\mathcal{M}_c(i,j)$ where all the scalar variables are defined. As the mesh discretization is Cartesian, x depends only on i and y depends only on j.
- The discrete coordinates $x_i(i)$ and $y_i(j)$ define the position of the superior interfaces of mesh cell (i,j) and serve to locate the velocity components. The relations between the centered coordinates and those of the interfaces are:

$$x_c(i) = \frac{x_i(i-1) + x_i(i)}{2}$$

$$y_c(i) = \frac{y_i(j-1) + y_i(j)}{2}$$

- The dimensions of the cell $\mathcal{M}_c(i,j)$ are defined by:

$$\Delta X_{c_i} = x_i(i) - x_i(i-1)$$

$$\Delta Y_{c_j} = y_i(j) - y_i(j-1)$$

- The control volume (the surface in 2D) of each mesh cell is defined as the product of cell dimensions in each direction:

$$\Delta V_{c_{i,j}} = \Delta X_{c_i} \cdot \Delta Y_{c_j}$$

From the definition of $\mathcal{M}_c$, it is possible to construct the shifted meshes $\mathcal{M}_u$ and $\mathcal{M}_v$ associated to velocity components u and v respectively. Theses meshes $\mathcal{M}_u$ and $\mathcal{M}_v$ are such that:

- For $\mathcal{M}_u$:

$$\begin{array}{lcl} \text{Coordinates of } u_{i,j} & : & (x_i(i), y_c(j)) \\ \text{Dimensions} & : & \Delta X_{u_i} = x_c(i+1) - x_c(i) \\ & & \Delta Y_{u_j} = y_i(j) - y_i(j-1) = \Delta Y_{c_j} \\ \text{Control volume} & : & \Delta V_{u_{i,j}} = \Delta X_{u_i} \cdot \Delta Y_{u_j} \end{array}$$

– For $\mathcal{M}_v$:

$$\begin{aligned} &\text{Coordinates of } v_{i,j} &&: (x_c(i), y_i(j)) \\ &\text{Dimensions} &&: \Delta X_{v_i} = x_i(i+1) - x_i(i) = \Delta X_{c_i} \\ &&&\quad \Delta Y_{v_j} = y_c(j+1) - y_c(j) \\ &\text{Control volume} &&: \Delta V_{v_{i,j}} = \Delta X_{v_i} \cdot \Delta Y_{v_j} \end{aligned}$$

Additional peripheral cells, usually named fictitious or ghost cells, are reserved for the treatment of the boundary conditions. Consider that the three grids have the same dimensions $N_x \times N_y$ (for the practical reason of programming arrays sizes) then the discrete interior points of the domain are:

$$\text{For } \mathcal{M}_c : [2, N_x - 1] \times [2, N_y - 1]$$

$$\text{For } \mathcal{M}_u : [2, N_x - 2] \times [2, N_y - 1]$$

$$\text{For } \mathcal{M}_v : [2, N_x - 1] \times [2, N_y - 2]$$

We note that the shifted meshes associated to the velocity components are defined with one cell less along the direction of the considered velocity component. In this case, mesh points associated to boundary conditions coincide with the boundaries of the domain. Otherwise, they would be outside of the domain in the fictitious cells (*cf.* Fig. 1.5).

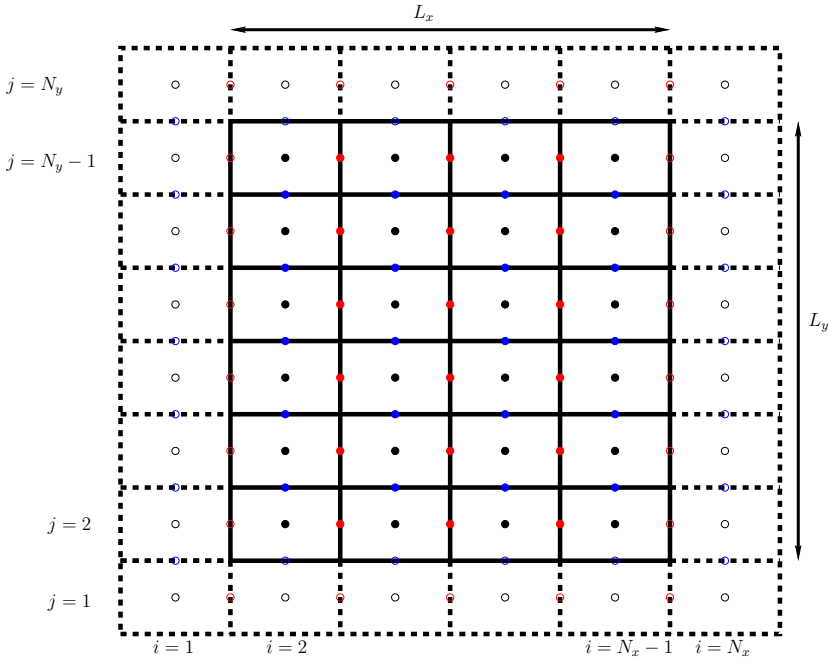


FIGURE 1.5: Locations of the pressure and velocity unknows: p in black, u-component in red, v-component in blue. Fictitious cells for the (pressure) boundary conditions are plotted with a dashed line.

1.6.2 Spatial discretization for the Navier-Stokes equation

Consider the momentum equation for incompressible flow:

$$\frac{\partial \mathbf{u}}{\partial t} + NL(\mathbf{u}) = -\nabla p + L(\mathbf{u}), \tag{1.45}$$

where NL is the non-linear convection term and L is the viscous diffusion term ($L(\mathbf{u}) = \frac{1}{\mathrm{Re}}\Delta\mathbf{u}$). In the following of the section, we present the centered second-order finite-difference schemes for the approximation of the different terms appearing in the discretization of the momentum equation, restricting ourselves to the two-dimensional case for simplicity, *i.e.* $\mathbf{u} = (u, v)$.

1.6.2.1 Pressure gradient discretization

Owing to the staggered arrangement of the variables on the mesh, the components of the pressure gradient appear in the equations for the respective velocity components. It allows for a straightforward approximation of the pressure gradient components at the corresponding velocity points. For a two-dimensional mesh, it comes

$$\left.\frac{\partial p}{\partial x}\right|_{(i,j)} = \frac{p(i+1,j) - p(i,j)}{\Delta X_{i+1/2}}, \tag{1.46}$$

$$\left.\frac{\partial p}{\partial y}\right|_{(i,j)} = \frac{p(i,j+1) - p(i,j)}{\Delta Y_{j+1/2}}. \tag{1.47}$$

1.6.2.2 Laplacian operator discretization

Owing to the Cartesian structure of the meshes, the discretization of the Laplacian operators appearing in the prediction and projection steps is immediate. For instance, the discretization of the viscous term

$$L(\mathbf{u})|_{(i,j)} = \left.\frac{1}{\mathrm{Re}}\Delta\mathbf{u}\right|_{(i,j)} = \left.\frac{1}{\mathrm{Re}}\left((\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}), (\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2})\right)\right|_{(i,j)}, \tag{1.48}$$

is performed on the mesh associated to the considered velocity component: the Laplacian of u is discretized on the mesh $\mathcal{M}_u$ and the Laplacian of v on the mesh $\mathcal{M}_v$. As a result, for the second order finite-difference scheme the approximations of the second order

operators for the u and v components are:

$$\left.\frac{\partial^2 u}{\partial x^2}\right|_{(i,j)} = \frac{u_{i+1,j} - u_{i,j}}{\Delta X_{i+1}\Delta X_{i+1/2}} - \frac{u_{i,j} - u_{i-1,j}}{\Delta X_i \Delta X_{i+1/2}}, \tag{1.49}$$

$$\left.\frac{\partial^2 u}{\partial y^2}\right|_{(i,j)} = \frac{u_{i,j+1} - u_{i,j}}{\Delta Y_{j+1/2}\Delta Y_j} - \frac{u_{i,j} - u_{i,j-1}}{\Delta Y_{j-1/2}\Delta Y_j}, \tag{1.50}$$

$$\left.\frac{\partial^2 v}{\partial x^2}\right|_{(i,j)} = \frac{v_{i+1,j} - v_{i,j}}{\Delta X_{i+1/2}\Delta X_i} - \frac{v_{i,j} - v_{i-1,j}}{\Delta X_{i-1/2}\Delta X_i}, \tag{1.51}$$

$$\left.\frac{\partial^2 v}{\partial y^2}\right|_{(i,j)} = \frac{v_{i,j+1} - v_{i,j}}{\Delta Y_{j+1}\Delta Y_{j+1/2}} - \frac{v_{i,j} - v_{i,j-1}}{\Delta Y_j \Delta Y_{j+1/2}}. \tag{1.52}$$

The discretization of the Laplacian equation of the projection step proceeds in a similar fashion, but for unknown correction potential Φ being collocated as for the pressure, *i.e.* on the mesh $\mathcal{M}_c$.

1.6.2.3 Convective term discretization

Concerning the convective term, two formulations are used in the present thesis: the conservative formulation $NL(\mathbf{u}) = \nabla \cdot (\mathbf{u} \otimes \mathbf{u})$ and the convective formulation $NL(\mathbf{u}) = (\mathbf{u} \cdot \nabla)\mathbf{u}$.

These two formulation, under the condition of incompressibility $\nabla \cdot \mathbf{u} = 0$, are equivalent:

$$\begin{aligned}\nabla \cdot (\mathbf{u} \otimes \mathbf{u}) - (\mathbf{u} \cdot \nabla)\mathbf{u} &= \begin{pmatrix} \frac{\partial(uu)}{\partial x} + \frac{\partial(uv)}{\partial y} + \frac{\partial(uw)}{\partial z} \\ \frac{\partial(vu)}{\partial x} + \frac{\partial(vv)}{\partial y} + \frac{\partial(vw)}{\partial z} \\ \frac{\partial(wu)}{\partial x} + \frac{\partial(wv)}{\partial y} + \frac{\partial(ww)}{\partial z} \end{pmatrix} - \begin{pmatrix} u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} + w\frac{\partial u}{\partial z} \\ u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} + w\frac{\partial v}{\partial z} \\ u\frac{\partial w}{\partial x} + v\frac{\partial w}{\partial y} + w\frac{\partial w}{\partial z} \end{pmatrix} \\ &= \begin{pmatrix} u\frac{\partial u}{\partial x} + u\frac{\partial v}{\partial y} + u\frac{\partial w}{\partial z} \\ v\frac{\partial u}{\partial x} + v\frac{\partial v}{\partial y} + v\frac{\partial w}{\partial z} \\ w\frac{\partial u}{\partial x} + w\frac{\partial v}{\partial y} + w\frac{\partial w}{\partial z} \end{pmatrix} = \mathbf{u}(\nabla \cdot \mathbf{u}) = \mathbf{0}. \end{aligned} \tag{1.53}$$

In the two-dimensional case, the convective term in conservative form is

$$\nabla \cdot (\mathbf{u} \otimes \mathbf{u})|_{(i,j)} = \left.\left(\frac{\partial(uu)}{\partial x} + \frac{\partial(uv)}{\partial y}, \frac{\partial(vu)}{\partial x} + \frac{\partial(vv)}{\partial y}\right)\right|_{(i,j)}, \tag{1.54}$$

leading to the discrete version

$$\left.\frac{\partial(uu)}{\partial x}\right|_{(i,j)} = \frac{(u_{i+1,j}+u_{i,j})^2-(u_{i-1,j}+u_{i,j})^2}{4\Delta X_{i+1/2}}, \tag{1.55}$$

$$\left.\frac{\partial(uv)}{\partial y}\right|_{(i,j)} = \frac{(u_{i,j}+u_{i,j+1})(v_{i,j}+v_{i+1,j})-(u_{i,j}+u_{i,j-1})(v_{i,j-1}+v_{i+1,j-1})}{4\Delta Y_j}, \tag{1.56}$$

$$\left.\frac{\partial(vu)}{\partial x}\right|_{(i,j)} = \frac{(u_{i,j}+u_{i,j+1})(v_{i,j}+v_{i+1,j})-(u_{i-1,j}+u_{i-1,j+1})(v_{i-1,j}+v_{i,j})}{4\Delta X_i}, \tag{1.57}$$

$$\left.\frac{\partial(vv)}{\partial y}\right|_{(i,j)} = \frac{(v_{i,j}+v_{i,j+1})^2-(v_{ii,j}+v_{i,j-1})^2}{4\Delta Y_{j+1/2}}. \tag{1.58}$$

Similarly, for the conservative form

$$(\mathbf{u}\cdot\nabla)\mathbf{u}|_{(i,j)} = \left.\left(u\frac{\partial u}{\partial x}+v\frac{\partial u}{\partial y}, u\frac{\partial v}{\partial x}+v\frac{\partial v}{\partial y}\right)\right|_{(i,j)}, \tag{1.59}$$

the discrete form is

$$\left.u\frac{\partial u}{\partial x}\right|_{(i,j)} = \frac{1}{2}u_{i,j}\left(\frac{u_{i,j}-u_{i-1,j}}{\Delta X_i}+\frac{u_{i+1,j}-u_{i,j}}{\Delta X_{i+1}}\right), \tag{1.60}$$

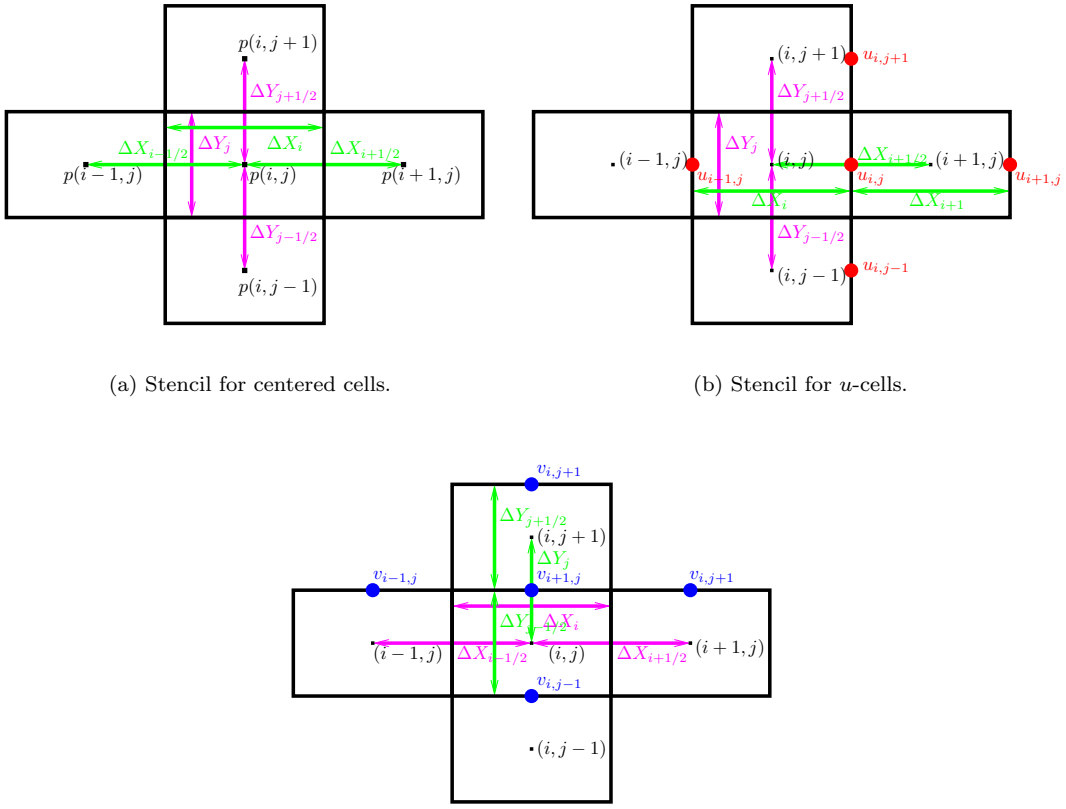
$$\left.v\frac{\partial u}{\partial y}\right|_{(i,j)} = \frac{1}{4}\left((v_{i,j}+v_{i+1,j})\frac{u_{i,j+1}-u_{i,j}}{\Delta Y_{j+1/2}}+(v_{i,j-1}+v_{i+1,j-1})\frac{u_{i,j}-u_{i,j-1}}{\Delta Y_{j-1/2}}\right), \tag{1.61}$$

$$\left.u\frac{\partial v}{\partial x}\right|_{(i,j)} = \frac{1}{4}\left((u_{i-1,j}+u_{i-1,j+1})\frac{v_{i,j}-v_{i-1,j}}{\Delta X_{i-1/2}}+(u_{i,j+1}+u_{i,j})\frac{v_{i+1,j}-v_{i,j}}{\Delta X_{i+1/2}}\right), \tag{1.62}$$

$$\left.v\frac{\partial v}{\partial y}\right|_{(i,j)} = \frac{1}{2}v_{i,j}\left(\frac{v_{i,j}-v_{i,j-1}}{\Delta Y_j}+\frac{v_{i,j+1}-v_{i,j}}{\Delta Y_{j+1}}\right). \tag{1.63}$$

Note that the discretization of the convective term may require a mixture of central differences and donor-cell discretization, *e.g.* up-winding, to ensure stability for strongly convective problems. Such stabilization schemes are not considered in the present work; we only remark that if needed they can be easily incorporate in the proposed framework, as stabilized convective schemes would only affect the discretization of the convective terms which are treated explicitly. In fact, the objective of the present thesis is to derive efficient computational approaches allowing for a fine enough spatial discretization such that stabilization methods, and their inherent numerical diffusivity, can be avoided.

The stencils for the discretization of the operators associated to the pressure and velocity cells are illustrated in Fig. (1.6).



(a) Stencil for centered cells.

(b) Stencil for u-cells.

(c) Stencil for v-cells.

FIGURE 1.6: Illustration of the stencils for the discretization of the operators appearing in the momentum equation in two-dimension.

1.7 Conclusion of Chapter 1

After a brief introduction of the incompressible Navier-Stokes equations for a Newtonian fluid, we have discussed in Section 1.3 a prediction-projection method for the time-integration of these equations. In particular, we have shown that this approach essentially amounts to the resolution at each time-step of Helmholtz and Poisson problems. Relying on the discretized operators provided in Section 1.6.2, one could readily construct the fully discretized version of the Helmholtz and Poisson problems. However, this direct discretization results in large discrete systems. In addition, when using fine grids the explicit treatment of the convective terms yields a stability constraint on the time-step, known as the Courant-Friedrichs-Lewy (CFL) condition [72], $\Delta t < \min(h/|u|)$ with h being the size of the spatial discretization and $|u|$ the local velocity modulus. As a result, a smaller and smaller time-step must be used when h decreases, making the simulation more and more demanding. This issue calls for efficient solvers in order to maintain reasonable computational times for the simulations. The remainder of the

thesis is dedicated to the design and implementation of fast solvers on different parallel hybrid architectures.

In the case of separable problems, the ADI and Partial Diagonalization techniques were introduced in Sections 1.4 and 1.5 respectively. These techniques allow us to recast the three-dimensional problems into sequences of one-dimensional ones with lower computational complexity. The crucial point is that when considering discretization over Cartesian grids, the discretization of the one-dimensional second order operators $\partial^2/\partial x^2$, $\partial^2/\partial y^2$, and $\partial^2/\partial z^2$ leads to tridiagonal systems (for instance, the operator in direction x involves unknowns with same (j,k) indices). This specificity allows for direct, fast (vectorized) and parallel solutions of the linear systems using Thomas algorithm, with significant computational saving compared to a direct solution of the corresponding three-dimensional problems. Two crucial aspects have been mainly investigated in view of an implementation on modern multicore computers. First, the efficient parallel implementation of the linear system solution and, second, the fast computation of the change of bases in the partial diagonalization method.

Finally, for non-separable problems, for instance in presence of solid obstacles inside the computational domain, the fully three-dimensional discrete systems must be considered. Due to the size of these systems, direct solvers are challenged in the case of very fine discretization meshes, and iterative solvers need be considered. We have also investigated the implementation of available iterative solvers on hybrid architectures. A quick overview of the iterative methods for the solution of linear systems is provided in Appendix A.

Chapter 2

Parallel algorithms for solving Navier-Stokes equations

Contents

As described in Chapter 1, the fine discretization of the Navier-Stokes equations represents a huge computational work. We use double precision variables in order to improve the accuracy. However this doubles memory usage comparing to single precision variables. We also want the mesh to be fine enough in such a way that the flow motion at small scale can be well captured by our numerical simulations. However, such fine discretization increases the size of the resulting linear systems and requires proper use of parallelism in the implementation. In this chapter, we present the Navier-Stokes solver and its implementation for multicore systems.

Section 2.1 describes the domain decomposition method used in the Navier-Stokes solver. We also explain how to compute the Schur complement in the solver. In Section 2.2, we discuss the interest of multi-level parallelism and why we choose the MPI/OpenMP programming model in our parallel implementation. In Section 2.3, we present the main structure of the NS solver and its sequential performance. Then in Section 2.4, we optimize the solver for tridiagonal systems using vectorization techniques. In Section 2.5, we describe some numerical experiments on both shared and distributed memory architectures including performance results. Finally, some concluding remarks are given in Section 2.6.

2.1 Domain decomposition approach

In the area of physical simulations, many problems arise from the discretization of partial differential equations. Because of the complexity of the computing domain, the domain decomposition method is often applied. This method solves a boundary value problem not on the original domain but on a set of subdomains. Different approaches exist to coordinate the solutions between subdomains. We choose the domain decomposition method for several reasons. The first reason is that the original boundary value problem is easier to solve on smaller and more regular subdomains. This is mainly the motivation of the domain decomposition method proposed by Schwarz [87]. However, the original Schwarz method, often called as alternating Schwarz method, consists in solving the same problem on subdomains, alternating from one to another. This procedure is sequential since solving the problem on one subdomain can not begin if the problem is not yet solved on its neighbors. Pierre-Louis Lions modified the alternating Schwarz method and proposed the parallel Schwarz method in [66]. In the parallel method, the problem is solved independently on each subdomain and the solutions on interfaces are updated after each iteration. The second reason is related to the increasing amount of data handled to solve the problem. As physicists want more and more realistic simulations, the degrees of freedom can be too high for the whole problem to fit into the computer memory.

Often referred as *divide-and-conquer* techniques, the first step of any domain decomposition method consists in dividing the domain. We can find in [25, 80, 94] more details on the types of partitioning. For example, when using the finite element discretization, the elements are mapped into subdomains resulting in a so-called *element-based* decomposition. In this way, all information related to one element is included in the same subdomain. The *edge-based* decomposition is often used in the finite volume discretization where edges cannot be split into two subdomains. The *vertex-based* partitioning

consists in dividing the vertex set into subsets and allowing edges and elements to be shared between subdomains. In the context of this thesis, we use an element-based decomposition. Fig 2.1 shows an example of a 3D domain decomposition of a rectangular domain (16 subdomains). Moreover, to balance the computational work on each subdomain, we make sure that the subdomains have equal size.

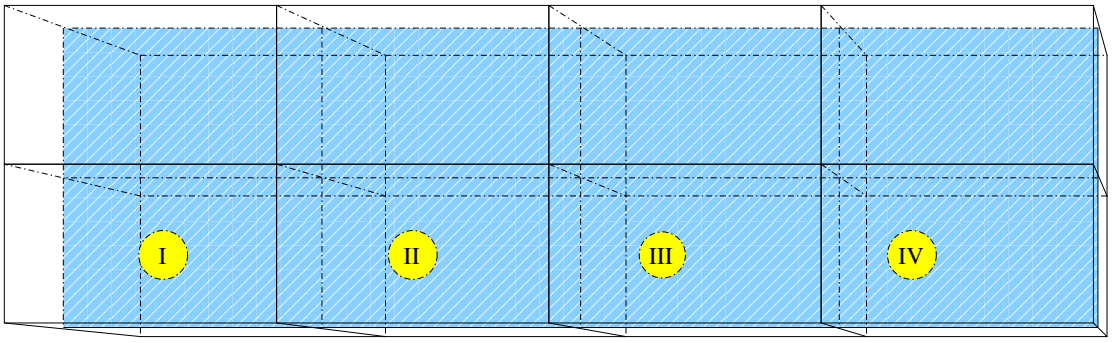


FIGURE 2.1: Example of 3D domain decomposition with a cartesian topology $4 \times 2 \times 2$

Another important aspect in domain decomposition methods is the existence of overlapping. We can have no overlapping, or overlapping with different sizes. In our work, we choose the overlapping to be of order one, which means that we introduce one layer of mesh cells to each subdomain. This layer can be either the fictive cells of the domain which are mainly used to deal with boundary conditions, or the real cells of the neighbor subdomain.

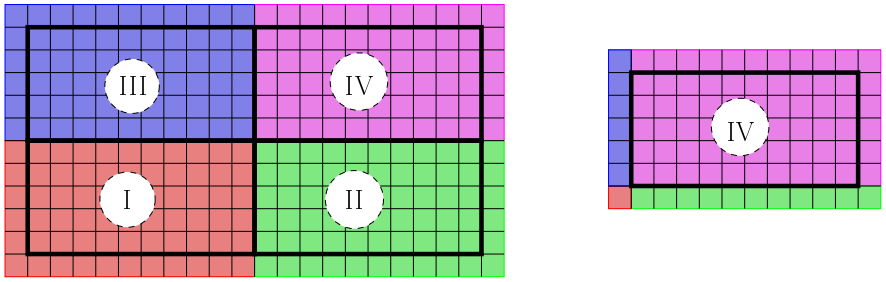


FIGURE 2.2: 2D domain decomposition with order 1 overlapping.

As illustrated in Fig. 2.2 (left), the domain is equally divided into four subdomains. When looking for instance at subdomain IV in Fig. 2.2 (right), the solid black rectangle surrounds the "real" cells while the cells outside the black rectangle are the boundary (fictive) cells and interface cells. For this example, we mention that we do not need the values on the four corners of the interface because of the differencing scheme we use. When updating the value on one cell, we only need the values of its left, right, upper and bottom neighbor cells in the 2D case (front and back cells are needed as well in the 3D case).

The domain decomposition in our NS solver serves as a "work distributor". As the domain is equally divided into subdomains, each MPI process has the same computational load. Also, the reason why each subdomain has one layer interface lies in the differencing scheme we chose. For example, when we compute the source term of the Helmholtz-like equation (1.34), we need the gradient of the pressure, Laplacian of the velocity and the convection term. In order to compute these quantities for one specific mesh cell in one subdomain, we need information from its four neighbor cells (six neighbors in the 3D case). If we do not have the interface layer, we will have to fetch information from neighboring subdomains in order to calculate on one subdomain which involve information exchange. Once the source term is computed, by using the ADI method, we obtain three tridiagonal systems for the Helmholtz problem. We note that each MPI process has one part of these systems. In the following, we show how these systems can be solved efficiently in parallel using the Schur complement method [107].

Let us consider a 2D domain with 2 subdomains (pink and blue) shown by Fig 2.3 where the variables are ordered by subdomains. Within each subdomain, the variables are ordered by row then by column. From Eq. (1.37a) for example, the resulting matrix can be depicted by Fig. 2.4. We can see that this system, each process has its own information about the system and we can not solve efficiently this system in parallel.

13	14	15	16	29	30	31	32
9	10	11	12	25	26	27	28
5	6	7	8	21	22	23	24
1	2	3	4	17	18	19	20

FIGURE 2.3: Ordering of variables in a 2D domain (two subdomains).

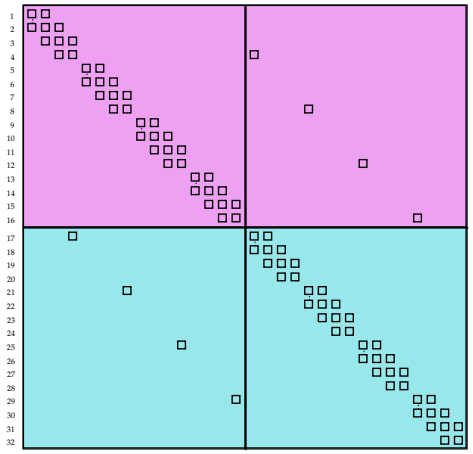


FIGURE 2.4: Matrix pattern using ordering from Fig. 2.3.

By changing the ordering of variables as shown in Fig. 2.5 (we order first the interior variables and then the interface variables), the resulting matrix (see Fig. 2.6) has a block structure that enables us to apply the Schur complement method.

10	11	12	28	32	22	23	24
7	8	9	27	31	19	20	21
4	5	6	26	30	16	17	18
1	2	3	25	29	13	14	15

FIGURE 2.5: Ordering of variables in a 2D domain (two subdomains, interior variables are numbered first).

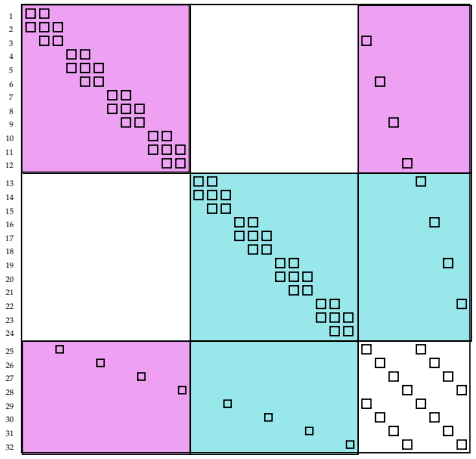


FIGURE 2.6: Matrix pattern using ordering from Fig. 2.5.

Given a linear system $Ax = b$, the Schur complement can be defined using an expression of the linear system in its block form :

$$\begin{pmatrix} B & E \\ F & C \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} \tag{2.1}$$

or equivalently,

$$Bx_1 + Ex_2 = b_1 \tag{2.2}$$

$$Fx_1 + Cx_2 = b_2 \tag{2.3}$$

From Eq. (2.2), the unknown x_1 can be expressed as

$$x_1 = B^{-1}(b_1 - Ex_2) \tag{2.4}$$

Then by substitution in Eq. (2.3), the following reduced system is obtained:

$$(C - FB^{-1}E)x_2 = b_2 - FB^{-1}b_1 \tag{2.5}$$

The matrix $S = C - FB^{-1}E$ is called the Schur complement matrix of system (2.1). If this matrix can be formed and Eq. (2.5) can be solved, the variable x_2 is then available. Once this variable is computed, the remaining variable x_1 can be computed from Eq. (2.4).

Let us go back to Fig. 2.6. We can consider that the block B represents in this pattern the two tridiagonal blocks. E and F are two sparse blocks and C is the bottom right block. We also notice that, except for C, the blocks B, E and F can be decoupled into two independent parts, which provide parallelism.

We mention that in the implementation of our NS solver, starting from Fig. 2.3, blocks E, F and C are built directly by choosing the right coefficients, leading to a matrix as represented in Fig. 2.6. The whole procedure consists of five steps:

- Identify interior and interface variables and construct blocks E, F and C.
- Compute right-hand side of Eq. (2.5).
- Construct the Schur complement S.
- Solve Eq. (2.5) to obtain interface values x_2.
- Solve Eq. (2.4) to obtain interior values x_1.

2.2 Multi-level parallelism

With the growing demand of three-dimensional models, parallelism became essential in numerical simulations. In this section, we present a brief description of the parallel programming models related to shared and distributed memory architectures.

2.2.1 Shared memory architecture

As the name suggests, a shared memory system is an architecture where all processors have the same access to a global memory. This means that the address space is the same for all processors. The main advantage of a shared memory architecture is that access to data depends very little on its location in memory. This property facilitates programming but one must pay attention to memory conflicts as well as the the data coherence to avoid incorrect results. One drawback of shared memory system is that it does not exploit data locality. When solving partial differential equations, the discretization often has an intrinsically local nature: data that are highly related are often stored nearby in the global memory.

There are several possibilities for programming shared memory architectures. For example, we can use threads as in Pthreads [18]. We can also use a different language for parallel programming, (e.g., Ada [3]) or specific extensions to existing programming language (C/C++, Fortran, *etc.*) [2, 74]. Another possibility is to use an existing programming language enriched with compiler directives to specify parallelism, for instance OpenMP [4].

Let us consider the domain decomposition that we presented in Section 2.1. When dealing with shared memory systems, we assign each subdomain to a processor - usually a processor in shared memory refers to a single compute core - and the computation can be performed in parallel by the each core.

2.2.2 Distributed memory architecture

A typical distributed memory system is main composed of many processors which are often identical and are connected via a network having a given topology. Each processor has its own memory, processing units *etc.*. Contrary to shared memory systems , processors only access to their own memory and the access to other processors memory is performed by communication.

To deal with communication among processors, one standard API is the Message Passing Interface (MPI) [32]. MPI is a standardized and portable message-passing model designed by researchers from academia and industry to be used on a wide variety of parallel computers. This standard defines the syntax and semantics of library routines useful for a wide range of users writing portable message-passing programs in different programming languages such as Fortran, C, C++ and Java. MPI is a specification, not an implementation. There are multiple implementations of MPI such as MPICH [39] and OpenMPI [34] etc.

Similarly to shared memory systems, when using MPI in a domain decomposition method, we divide the domain into subdomains according to the number of processors which are usually multicore processors.

2.2.3 Combining shared and distributed memory systems

A classical type of architecture, commonly used for clusters, is the distributed-shared memory model. In such a memory model, we have several nodes connected through some high-speed interconnection. A node can be either a single-processor computer, or a symmetric multiprocessors (SMP), or even a non uniform memory access (NUMA)

architecture [43]. The memory is considered to be shared on each node and is not directly addressable from other nodes.

An example of such a system is the Stampede[1] computer from Texas Advanced Computing Center (TACC) at University of Texas, Austin, USA (see Fig. 2.7).

FIGURE 2.7: Stampede system. Image from https://www.tacc.utexas.edu

This system is a 10 petaflop/s Dell Linux Cluster based on more than 6400 Dell PowerEdge server nodes. Each node of Stampede is composed of 2 Intel Xeon E5-2680 (Sandy Bridge) processors running at 2.7GHz with 8 cores each (16 cores total) and 32GB of global memory, and an Intel Xeon Phi coprocessor. The peak performance of the Intel Xeon E5 processors is about 2 petaflop/s, while the Xeon Phi coprocessors have an additional peak performance of more than 7 petaflop/s. Among the 6400 nodes of Stampede, we have also 128 compute nodes where we have a single NVIDIA K20 GPU on each node with 5 GB of on-board memory. Stampede is ranked #7 in the Top500 list[2] published in November 2014. From the computing resources of Stampede, we will use CPU nodes (in Chapter 2) and GPUs (in Chapter 3) but we will not use the Intel Xeon Phi coprocessors.

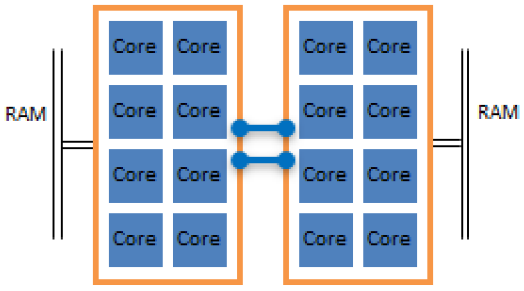


FIGURE 2.8: Illustration of a single node on Stampede.

Fig. 2.8 shows the architecture of a single SMP node on Stampede. On each node, we have 2 sockets to hold the Intel Sandy Bridge processors. Each socket has 8 cores and

[1] http://www.tacc.utexas.edu/stampede
[2] http://www.top500.org/

the local memory of the node is addressable from any core in any socket. As the memory is attached to sockets, the 8 cores sharing the socket have fastest access to the attached memory.

There are several parallel programming models for such an architecture: pure MPI, pure OpenMP, hybrid MPI+MPI and hybrid MPI+OpenMP. Each of these models has pros and cons. For example, with a pure MPI programming model where we have one MPI process on each core, any existing MPI program can run successfully without modifications and the MPI library does not need to support multiple threads. However, we lose performance with unnecessary intra-node communication and topology problem may occur. This problem is that, for a given application, the topology of MPI processes within a node can have an important effect on the performance. For example, given a 3D domain of size N^3 on a node with 32 cores, we apply a 1-dimensional data decomposition. The communication volume per core is about $2N^2$. If we apply a 3-dimensional data decomposition, we can have three times less communications. If we use the OpenMP-only model, although we will not have topology problem within a node, we will need the a virtual memory system. This will result in inter-node communication which are much slower than with MPI. The hybrid MPI+MPI model consists on using MPI for inter-node communications and the MPI-3.0 standard [69] for shared memory programming. The advantage of this model is that no message passing is performed inside of the SMP nodes and thread-safety is not an issue. However, we can encounter similar topology problem as in the pure MPI model. Also, unnecessary communications are presented in this model as in the pure MPI model. Finally, the model hybrid MPI+OpenMP has neither the topology problem nor the communication cost within an SMP node. The downside to this model is an increased code complexity and no performance guarantee compared to other models.

The optimal parallel programming model for distributed shared memory architectures depends on the application. However, the hybrid programming model is usually better than pure MPI and pure OpenMP models for the following reasons. First, we eliminate the domain decomposition at the node level. Second, we ensure automatic memory coherence at node level. Third, data movement is bounded between nodes. Last, we can synchronize on memory instead of using barriers.

Of course, the hybrid programming model has its drawbacks. For example, a multi-threaded algorithm created by aggregating MPI parallel components on a node will usually run slower than the MPI version algorithm. What's more, we add code complexity to the application when using hybrid programming model. Though the drawbacks, the hybrid programming model has become standard since not only does it balance the computational load of the applications, but also reduces memory traffic (especially for

memory-bound applications). We will use this model in developing our Navier-Stokes solver. We decompose the computational domain into subdomains of the number of available nodes. Within each subdomain, we use threads according to the number of cores for heavy computations.

2.3 General structure of the solver

We first present the main structure of the solver (see Fig. 2.9).

- Initialization
  - Data initialization: read the input file
  - Domain initialization: construct the subdomains
  - Fields initialization: allocate arrays
  - Operators initialization: construct the tridiagonal matrices
- Loop on time
  - Solve Helmholtz-like problem
  - Solve Poisson problem
  - Velocity update
  - Store results
- Finalize

FIGURE 2.9: Solver structure

In the data initialization step, we read inputs from an external file in which we specify the dimension, some physical quantities, and other characteristics of the problem. Then we initialize the computational domain and construct the set of subdomains according to the given number in the input file. In the field initialization step, we allocate the necessary memory for both velocity and pressure. In the final phase of initialization, we construct the operators (mainly tridiagonal) as described in Chapter 1.

Then we consider the solving phase which is inside a loop on time. This step, which has been clearly explained in Chapter 1, consists of solving a Helmholtz-like equation followed by a Poisson equation, and of updating the auxiliary variables by an explicit correction. We can choose to store the solution after each time iteration, or after a certain number of iterations in order to save memory space. For example, we can decide to store solutions every 500 iterations. If the time step is 0.001 second, then the final simulation will be constructed by the solutions obtained every 0.5 second. Of course, this can be changed, depending on the required accuracy.

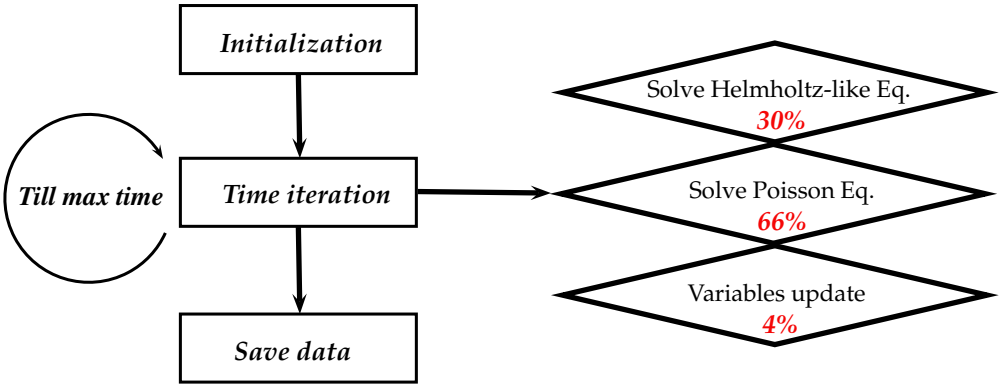


FIGURE 2.10: The main procedure of NS solver and the time percentage of each step.

We present in Fig. 2.10 the main procedure of the NS solver. The result depicted in this figure is obtained by executing an existing NS solver developed at LIMSI [33]. We use one core of the AMD Opteron 6172 processor, and compute for one iteration, the percentage of time spent in the three main tasks (velocity solving in Helmholtz problem, pressure solving in Poisson problem, and variables updating). We can see in this figure that the Helmholtz problem takes about 30% of the time in one iteration and Poisson problem is the most time consuming task which takes about 2/3 of the execution time. Improving the performance of both solvers can significantly improve the NS solver performance. In this chapter, we will talk about the tridiagonal solver. As explained previously in Chapter 1, solving a Helmholtz-like equation using ADI involves solving a set of tridiagonal systems. Thus, improving the performance of the tridiagonal solve will lead to better performance of the Navier-Stokes solver.

We would like to mention that in Fig. 2.10, we use the partial diagonalization method to solve the Poisson equation.

2.4 Accelerating the solution of the tridiagonal systems

We recall that, from Eqs. (1.37) and (1.43), the operators are all of a Laplacian form. Then, using the second order central differencing scheme ($\Delta_n u_i = \frac{u_{i-1} - 2u_i + u_{i+1}}{\Delta h^2}$), the discretization of these operators leads to diagonally dominant tridiagonal systems. We explain in this section how to improve these tridiagonal solves in our NS solver.

Let us consider a general diagonally dominant tridiagonal system (2.6) $Ax = s$ where $A \in \mathbb{R}^{m \times m}$ and $x, s \in \mathbb{R}^m$.

$$\begin{pmatrix} b_1 & c_1 & & & \\ \ddots & \ddots & \ddots & & \\ & a_i & b_i & c_i & \\ & & \ddots & \ddots & \ddots \\ & & & a_m & b_m \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_i \\ \vdots \\ x_m \end{pmatrix} = \begin{pmatrix} s_1 \\ \vdots \\ s_i \\ \vdots \\ s_m \end{pmatrix}. \tag{2.6}$$

This system can be solved using the Thomas algorithm given in Algorithm 1.This algorithm consists of two steps. The first step is a forward elimination where we eliminate the lower diagonal coefficients and transform the original matrix into an upper triangular one. The second step is a backward substitution that solves the upper triangular system. This requires $\mathcal{O}(n)$ operations.

Algorithm 1: Thomas algorithm.

Data: Diagonal matrix (a, b, c), RHS s.
Result: Solution x (stored in s_i).

Forward elimination: **for** $i = 2$ **to** m, **do**
  $b_i = b_i - \dfrac{c_{i-1} \times a_i}{b_{i-1}}$
  $s_i = s_i - \dfrac{s_{i-1} \times a_i}{b_{i-1}}$
end
Backward substitution: $s_m = \dfrac{s_m}{b_m}$
for $i = m - 1$ **to** 1, **do**
  $s_i = \dfrac{s_i - c_i \times s_{i+1}}{b_i}$
end

This algorithm actually corresponds to a Gaussian elimination without pivoting. In the LAPACK [7] linear algebra library, the solution of tridiagonal systems is implemented in the DGTSV routine (when using double precision arithmetic) which performs a Gaussian elimination with partial pivoting. Note that, if there is no need for pivoting, the Thomas algorithm and the DGTSV routine are similar. In the following, we explain how we can accelerate the Thomas algorithm by using vectorization techniques.

Since the late 90's, processor manufacturers provide specialized processing units called Single Instruction Multiple Data (SIMD) extensions. This new feature has allowed processors to exploit the latent data parallelism available in applications by executing a given instruction simultaneously on multiple data stored in a single special register. However, taking advantage of the SIMD extensions remains a complex task. We can find for instance in [29] a description of Boost.SIMD, a high-level C++ library to program SIMD architectures. Boost.SIMD provides both expressiveness and performance by using generic programming to handle vectorization in a portable way.

A first requirement to get performance is to store data in arrays that have been specially aligned in memory. The SIMD unit is able to read a certain number of bytes at a time and if it reads data from unaligned memory addresses, this request will involve two read operations. For instance in Fig. 2.11, assuming SIMD unit works with 8-byte units, trying to read 8 bytes from relative offset of 5 will be done by first reading bytes 0-7, and second reading bytes 8-15. As a result, the relatively slow memory access will become even slower.

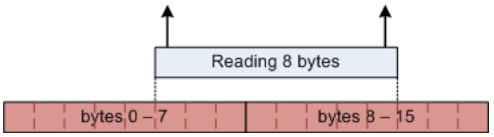


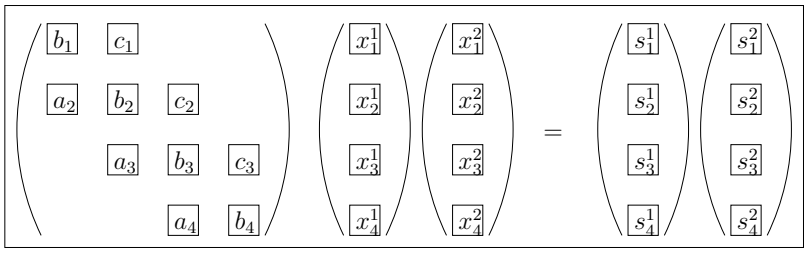
FIGURE 2.11: Read 8 bytes from unaligned memory.

To improve performance, we need to make sure that the arrays that we use in the tridiagonal system are aligned in memory. However, part of our NS solver is written in standard Fortran 95 which does not provide any memory alignment guarantee for arrays. The only way to use aligned memory is to allocate it with an external C/C++ function, such as *simd::allocator* in Boost.SIMD library. Once the C/C++ allocation is done, we can transfer this aligned memory allocation to Fortran by using the standard Fortran array pointer via the *ISO_C_BINDING* interface (see Algorithm 2).

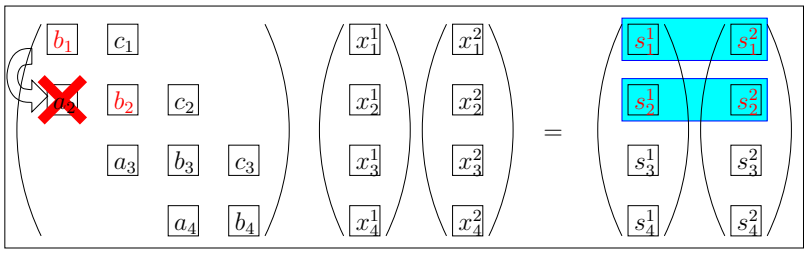
Algorithm 2: Aligned memory allocation in Fortran 95.

```
use ISO_C_BINDING

real(C_DOUBLE), pointer :: arr(:,:)
type(C_PTR) :: p

p = simd_alloc(int(L * M * N, C_SIZE_T))
    !... simd_alloc is the c++ allocator of Boost.SIMD
    !... L , M and N determine the form of the 3-D array
    !... C_SIZE_T is the number of elements to allocate

call c_f_pointer(p, arr, [L,M,N])
    !... associate pointer arr with allocated memory p
    !... use arr and arr(i,j) as usual

call simd_dealloc(p)
    !... free pointer p
```

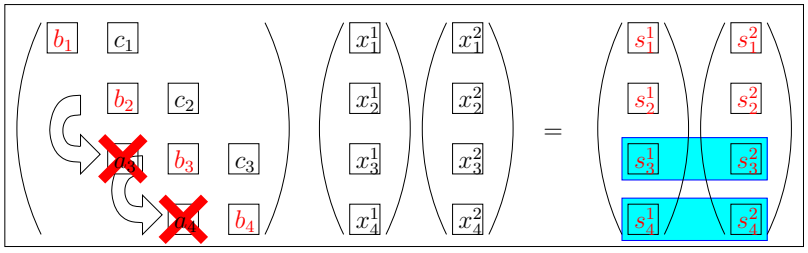
The interest of using vectorization comes from the fact that we handle multiple right-hand sides (RHS) in the tridiagonal systems. We illustrate the method in Fig. 2.12, where we consider the simplified case of two RHS.



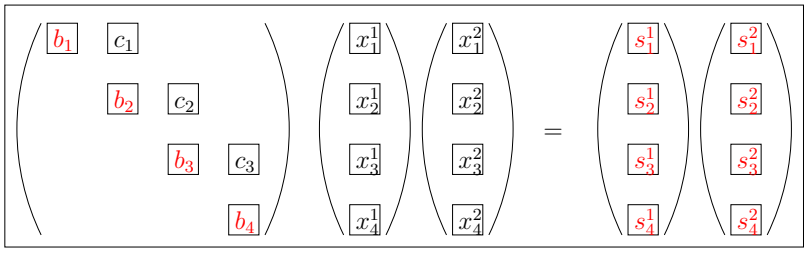
(a) Original system.



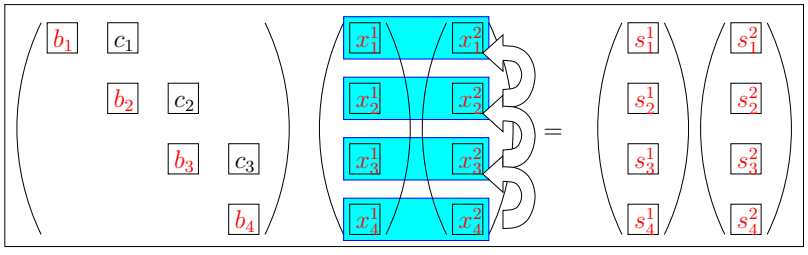
(b) First iteration.



(c) Second iteration.



(d) After elimination.



(e) Backward substitution.

FIGURE 2.12: Thomas algorithm with vectorization.

The main steps of our vectorized algorithm applied to the system given in Fig. 2.12(a) can be described as follows.

Forward elimination (Fig. 2.12(b), 2.12(c), 2.12(d)):

- load two register units with (s_i^1, s_{i+1}^1) and (s_i^2, s_{i+1}^2) for $i = 1, 3, ...n-1$,
- "shuffle" the two vectors to have (s_i^1, s_i^2), (s_{i+1}^1, s_{i+1}^2) (shown by the blue box).
- perform two Thomas iterations to eliminate a_i, a_{i+1} and update the corresponding diagonal and RHS:

$$\begin{aligned} b_i &= b_i - \frac{c_{i-1}a_i}{b_{i-1}} \\ (s_i^1, s_i^2) &= (s_i^1, s_i^2) - \frac{(s_{i-1}^1, s_{i-1}^2)a_i}{b_{i-1}} \\ b_{i+1} &= b_{i+1} - \frac{c_i a_{i+1}}{b_i} \\ (s_{i+1}^1, s_{i+1}^2) &= (s_{i+1}^1, s_{i+1}^2) - \frac{(s_i^1, s_i^2)a_{i+1}}{b_i} \end{aligned}$$

Backward substitution (Fig. 2.12(e)) :

- $(x_n^1, x_n^2) = \frac{(s_n^1, s_n^2)}{b_n}$,
- $(x_i^1, x_i^2) = \frac{(s_n^1, s_n^2) - c_i(x_{i-1}^1, x_{i-1}^2)}{b_i}$, for $i = n-1, ..., 1$
  Similarly to the RHS s, we compute two solutions with one operation.

The procedure is also described in Algorithm 3.

Algorithm 3: Thomas algorithm using SIMD.

- load two register units with two coefficients of the main diagonal and the lower diagonal;
- shuffle (permute) the data to have a 2×2 dense matrix;
- perform two Gaussian eliminations to eliminate the coefficients of the lower diagonal (see Fig. 2.12(c));
- solve backward the system with two RHS coefficients loaded in one register (see Fig. 2.12(e)).

Shuffling is a characteristic idiom of SIMD programming that replaces some class of complex memory access patterns by computation after simpler patterns. Depending on the extension, these shufflings can either be limited (like in SSE2 where shuffling

can only occurs piecewise inside a given register) or be random (like in Altivec where shuffles are in fact real complete byte permutations). Shuffling also enables idioms like deinterleaving, where scattered data can be fetched from main memory and brought back into a single, contiguous SIMD register. In our case, we use shuffling to aggregate values from the sparse representation of the system into a set of contiguous values in order to perform the backtracking and solving in a SIMD way. The code is expected to be faster due to less memory accesses (thus limiting cache misses) and as well as the augmented SIMD speedup layout of the data after the shuffling.

Our implementation extends this method to multiple RHS. Note that the memory alignment described previously enables here the "load" process to be performed efficiently. We mention that this vectorization approach can be also used in other parts of our solver, for instance to compute the source term faster. More generally, this technique enables us to get good speedups when performing computations on arrays that have no data dependencies.

2.5 Performance results for Navier-Stokes computations

2.5.1 Shared memory with pure MPI programming model

We performed some numerical experiments on a shared memory machine. The following experiments were carried out using a MagnyCours-48 system from University of Tennessee, Knoxville, USA. This machine has a NUMA architecture and is composed of four AMD Opteron 6172 (with a SSE-4a instruction set) running at 2.1GHz with twelve cores each (48 cores total) and 128GB of memory. Our solver is linked with the LAPACK and ScaLAPACK routines from the 10.3.6 version of the Intel MKL [51] library and communications are performed using OpenMPI 1.4.3. We use the pure MPI parallel programming model mentioned in Section 2.2 (no OpenMP directives) and one MPI process per core.

We consider a 3D vortex problem with mesh size 240^3, *e.g.* about 1.4×10^7 unknowns. The discretization sizes used in our simulations are chosen by physicists and they are considered to be realistic sizes for such testing problem. We represent the performance (in seconds) for one iteration of the NS solver including the Helmholtz and Poisson equations, and miscellaneous tasks (mainly I/O and velocity/pressure updates). The number of MPI processes varies from 1 to 48. We observe in Fig. 2.13 that the CPU time decreases significantly with the number of processes, showing then a good scalability of the solver. We observe that the Helmholtz equation represents about 30% of the global

computational time and this percentage remains the same when the number of processes increases.

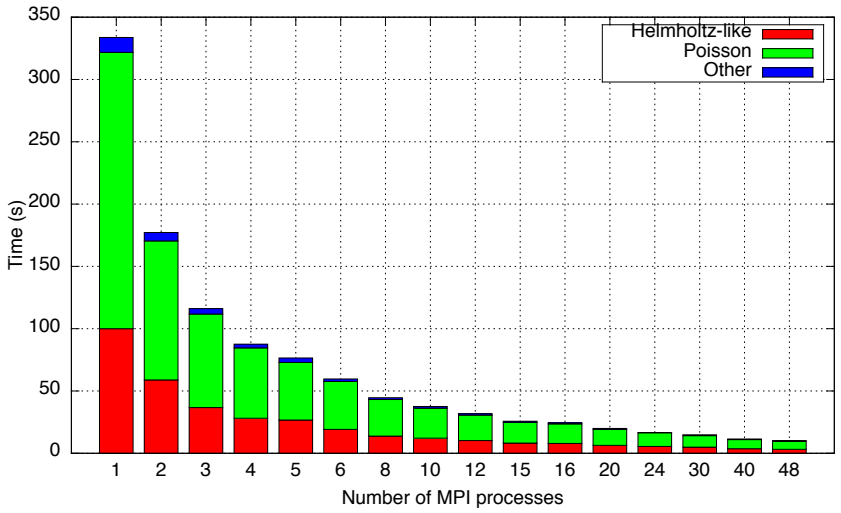


FIGURE 2.13: Time breakdown for one iteration of the NS solver.

Fig. 2.14 shows the parallel speedup of the solver. We obtain a speedup up of 33 using 48 cores. We also observe in Fig. 2.14 that the solution for the Poisson equation scales slightly better than for the Helmholtz equation. This is because in the Poisson equation we have only one tridiagonal system to solve (vs three systems in Helmholtz equation) and thus there is less information exchanged (as mentionned in Section 2.1, when we solve a tridiagonal system via the Schur complement method, we have a reordering of the variables that results in additionnal communication).

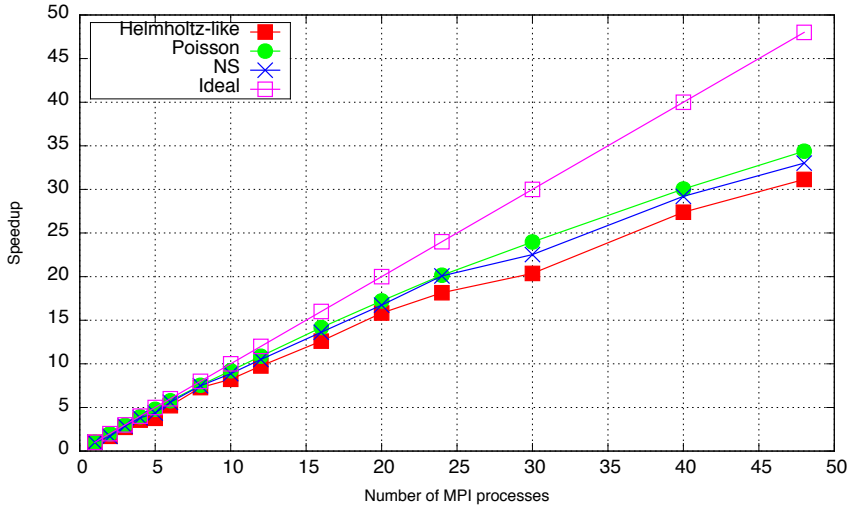


FIGURE 2.14: Strong scalability of NS solver.

In another experiment, we consider a 3-D vortex problem where the size increases with the number of threads and with a fixed mesh size per subdomain (weak scalability).

The local size is set to $240 \times 240 \times 10$ and 10 time iterations are performed. We observe in Fig. 2.15 that the time spent in solving Helmholtz and Poisson equations does not vary (about 60 and 110 milliseconds, respectively) because the number of unknowns computed is always $240^2 \times 10 = 5.76 \times 10^5$. However, the global CPU time increases linearly with the number of processes due to a larger amount of I/O after each iteration. Indeed, the numerical solution is stored after each time iteration in order to generate a detailed animation that visualizes the fluid movement. In this numerical test, we only divide the domain along one direction ($z-$direction here) to keep the shape of the interfaces. Thus, the amount of information to be exchanged (the number of interface elements) is $240^2 \times$ (number of MPI processes $- 1$).

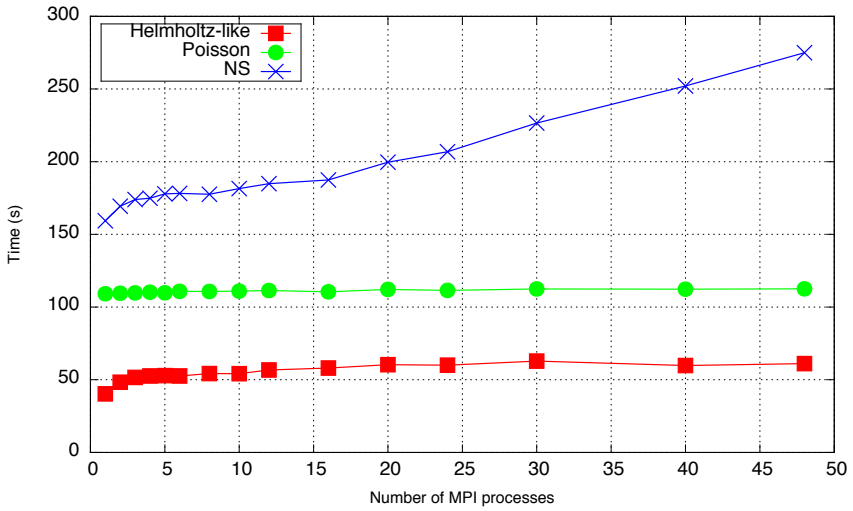


FIGURE 2.15: Weak scaling performance of the Navier-Stokes solver on shared memory.

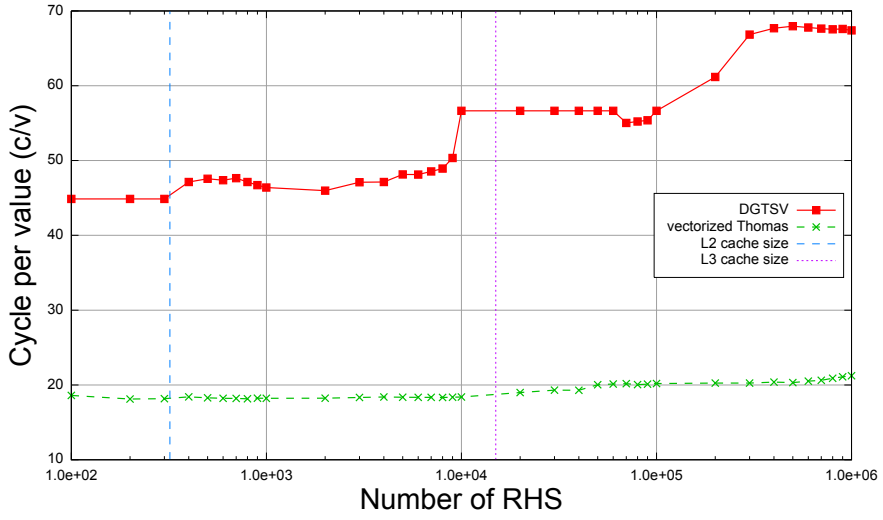
We now compare the performance of the vectorized Thomas algorithm (illustrated in Fig. 2.12) with the LAPACK 3.2 routine DGTSV from Netlib, linked with the MKL BLAS library, which solves a general tridiagonal system using Gaussian elimination with partial pivoting (note that, since our matrices are diagonally dominant, the routine DGTSV does not pivot and only the search for pivot is performed).

In Fig. 2.16(a), we plot the number of cycles required to compute one element of the result with respect to the number of RHS. This metric, computed as

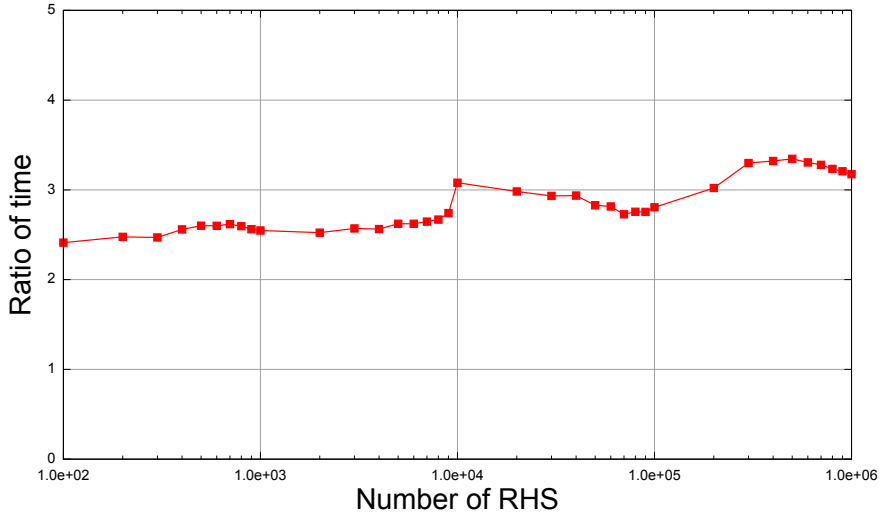
$$c = \frac{\text{execution time} \times \text{frequency}}{\text{number of elements}}$$

allows us to do a fine-grain analysis of both the impact of vectorization and the impact of memory access. For the curve related to DGTSV, we observe a cycle/value amount that jumps each time we hit the cache size (L2 and L3). This is due to the fact that every computation requires the full amount of memory access to be completed.

However, as shown in the second curve of Fig. 2.16(a), by using vectorized shuffle which replaces memory access by computation, the vectorized version of Thomas algorithm has a constant amount of CPU cycle/value up to the largest size (10^6 in our experiment). In Fig. 2.16(b), we plot the ratio of execution time for both DGTSV and vectorized Thomas algorithm. For large problem sizes (*e.g.* more than 10^6 unknowns), we observe that the vectorized version of Thomas algorithm outperforms DGTSV by a factor 3. In the remainder of this Chapter, this tridiagonal solver using SIMD extension will be used in the Navier-Stokes solver.



(a) Cycles per value for Thomas algorithm.

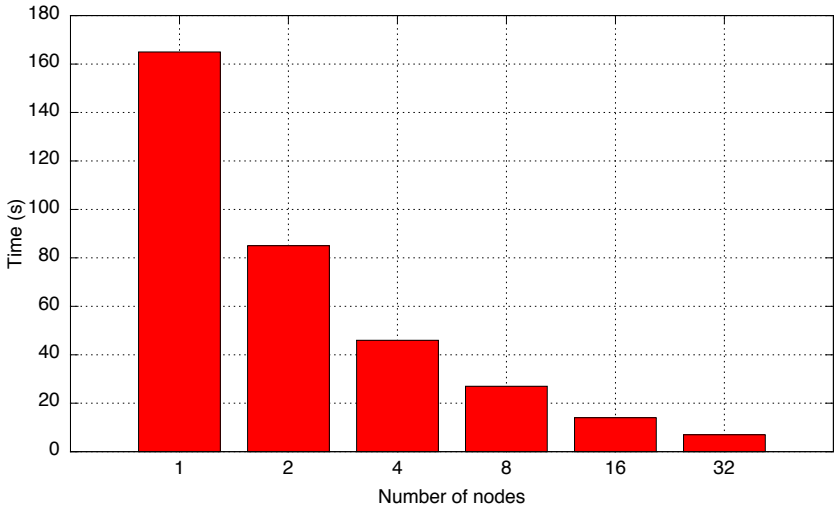


(b) Ratio $= \dfrac{\text{Time for LAPACK routine DGTSV}}{\text{Time for vectorized Thomas algorithm}}$.

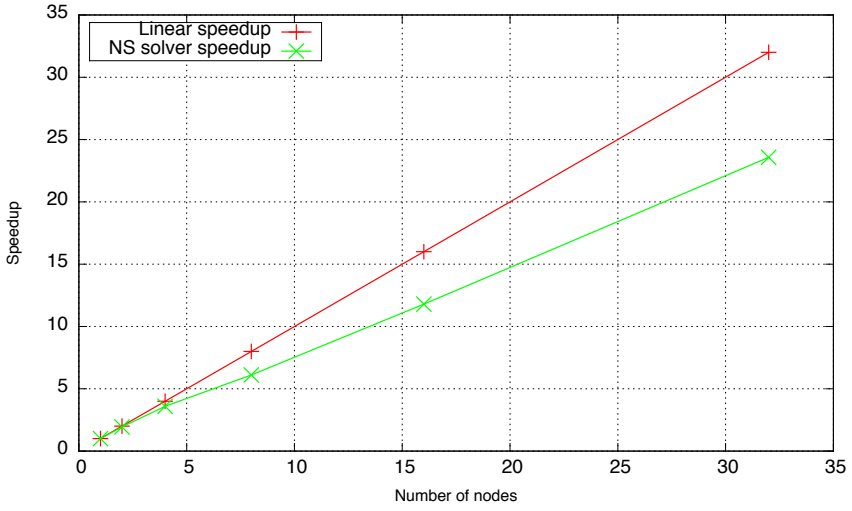
FIGURE 2.16: Performance of two implementations of Thomas algorithm. Matrix size = 100. Test carried out using Intel Xeon E5645.

2.5.2 Performance using MPI + OpenMP

We study in this section the performance of the Navier-Stokes solver on a cluster of multicore processors. We run the solver for 10 iterations and we set the global mesh size to 256^3.



(a) Time for solving NS equations.



(b) Parallel speedup of NS solver.

FIGURE 2.17: Performance of NS solver on the Stampede system.

We use the Stampede system described in Section 2.2.3 to run the code. Each node of Stampede is composed of two Intel Xeon E5 processors on which we use all the 16 cores for multithreading. We observe in Fig. 2.17(a) that the execution time drops with the

increasing number of nodes. This execution time is consistent with the time obtained in Section 2.5.1 (note that the problem size and the processor speed are slightly different).

The performance is good since we obtain, in Fig. 2.17(b) a parallel speedup of about 24 with 32 compute nodes (the speedup measures here the ratio of the execution time using one node and the time using various numbers of nodes). This speedup is similar to the one obtained in Section 2.5.1.

Next, we study the weak scalability of the solver. The mesh size per node is set to 64^3. We increase the number of nodes and measure the execution time for 10 iterations.

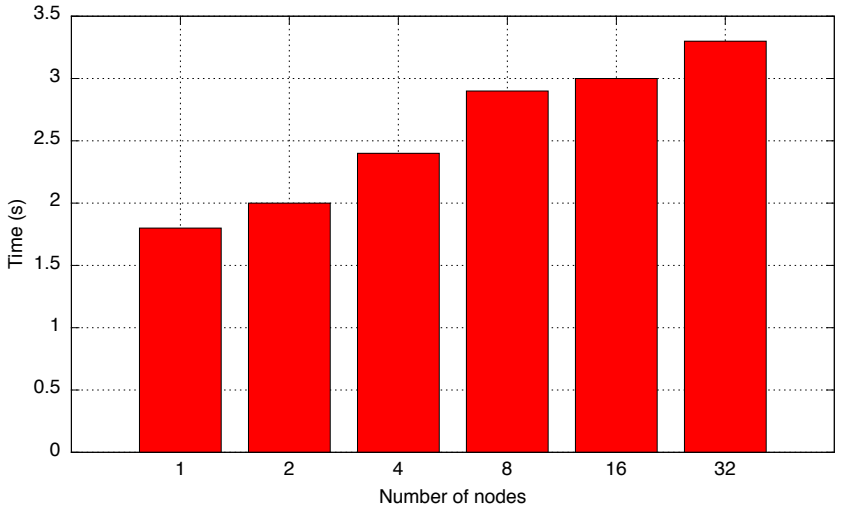


FIGURE 2.18: Weak scaling of NS solver (MPI-OpenMP implementation).

We observe in Fig. 2.18 that the execution time increases with the number of nodes. This increase of time is due to a larger amount of information exchanged as we increase the total mesh size. We did not store the solution after each iteration in order to observe the impact of increasing communication on the solver performance.

2.5.3 Performance comparison with an iterative method

We present in this section a performance comparison between the direct method based on partial diagonalization (see Section 1.5) and an iterative method for the solution of the Poisson equation described in Appendix A. The library Hypre [63] is used for the implementation of a Poisson solver based on the multigrid method and successive over-relaxation method.

Hypre is a library for solving large, sparse linear systems of equations on massively parallel computers. It contains several families of preconditioned algorithms. These algorithms include structured multigrid and element-based algebraic multigrid. Hypre

also provides commonly used Krylov-based iterative methods to be combined with its scalable preconditionners. This includes Conjugate Gradient and GMRES algorithms. Data structure in Hypre can be of different forms. Hypre provides data structures to represent and manipulate sparse matrices through interfaces. Each interface gives access to several solvers without the need to write new interface codes. These interfaces include stencil-based structured/semi-structured interfaces, finite-element based unstructured interface, and a linear algebra based interface.

Fig. 2.19 depicts the comparison of performance for the following Poisson solvers:

- our algorithm based on partial diagonalization and using the vectorized tridiagonal solves described in Section 2.4,
- iterative method based on multigrid preconditioned SOR solver (see Appendix A) with a straightforward implementation,
- Hypre routine for multigrid preconditioned SOR solver.

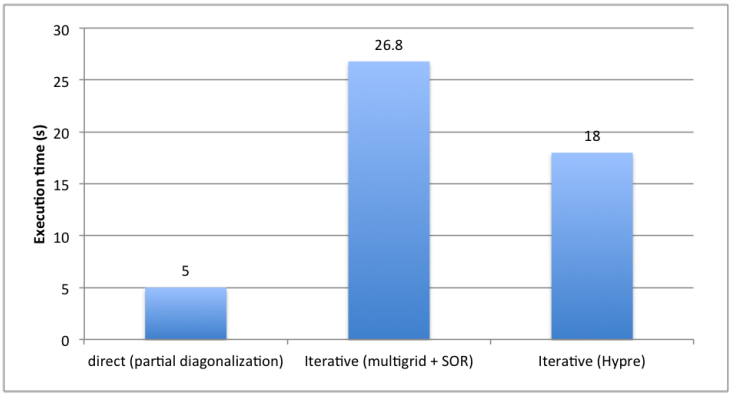


FIGURE 2.19: Performance comparison between different Poisson solvers.

These experiments were carried out using one node of the Stampede system described in Section 2.5.2 The problem that we tested is the 3D Taylor-Green vortex that will be detailed in Chapter 4. The domain is considered to be full fluid and thus with no obstacles. The mesh size is 128^3. We can use both direct method (partial diagonalization) and iterative method (multigrid + SOR) to solve the Poisson equation in the NS problem. We measured the average execution time for solving the Poisson equation using 2 MPI processes and 8 threads per process. We observe in Fig. 2.19 is that the direct method is more than 3 times faster than the iterative method based on the Hypre library. Note that the Hypre implementation outperforms our straightforward implementation for multigrid preconditioned SOR solver. However, as will be explained in Chapter 4, the iterative method enables us to address fluid flow simulations that include obstacles in the domain, which is not possible with the direct method.

2.6 Conclusion of Chapter 2

In this chapter, we have described the implementation of our Navier-Stokes solver for CPUs. Using domain decomposition method, the solver shows good performance and satisfying scalability on both shared and distributed memory architectures. We have optimized the tridiagonal solver using vectorization techniques and improved the speedup with a factor of three, when compared to the LAPACK routine DGTSV linked with MKL. For the MPI/OpenMP hybrid implementation, we obtain a speedup of 24 using 32 MPI processes and 16 threads per MPI process. The problem size is set to 256^3 which is commonly used in such Navier-Stokes solvers. We published some of these results in a recent publication [100].

We also presented the performance of the Poisson solver implemented in the Navier-Stokes solver. We observe that the iterative method for solving Poisson problems can be improved using the Hypre library while the direct method, for suitable problems, remains faster.

If we want to further accelerate the simulations and adapt our NS solver to the current architecture (namely the heterogeneous architectures), vectorization techniques will not be sufficient and the use of accelerators can not be avoided. In the next chapter, we will explain how to take advantage of Graphics Processing Units (GPUs) without modifying the main structure of our Navier-Stokes solver.

Chapter 3

Taking advantage of GPU in Navier-Stokes equations

Contents

In the domain of computational science, when considering high performance computing applications, we cannot neglect the use of accelerators which has become a major component of modern supercomputers. Originally designed for graphics processing, GPU (Graphics Processing Unit) can also enhance performance in scientific computing applications and in particular in computational fluid dynamics. In this chapter, we describe how GPU computing can be used in our Navier-Stokes solver. In the first section, we discuss the development of CPU/GPU computing. In the second section, we propose GPU algorithms and implementations to accelerate our Navier-Stokes solver. We also describe the GPU version of the Helmholtz and Poisson solvers. In the third section, performance results are presented, followed by a concluding section.

3.1 Introduction to GPU computing

Different from classical CPU processors, GPUs are mostly designed and used in visual processing. The world's first GPU is GeForce 256, product of Nvidia in 1999. The technical definition of this GPU is "a single-chip processor with integrated transform, lighting, triangle setup/clipping, and rendering engines that are capable of processing a minimum of 10 million polygons per second"[1].

FIGURE 3.1: Nvidia GeFroce 256.

As the name suggests, GPUs are developed for graphics rendering. During the first years of appearance, the main purpose of GPUs was related to graphics pipeline in visual processing. Later on, GPU has been developed into a strong programmable processor. An application programming interface is added and the compute capacity has much increased. This era is marked by GeForce 8800 GTX, that has a capacity of over 330 Gflops which is higher than a high-end CPU at that time (2006).

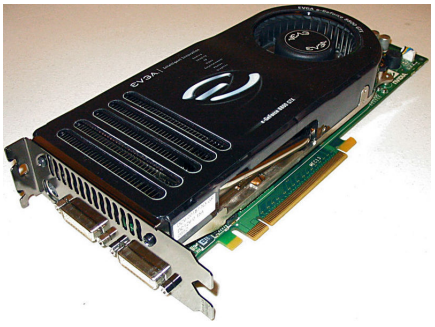


FIGURE 3.2: Nvidia GeFroce 8800 GTX.

Since then, GPU design has entered a new stage where it is not only used for calculations related to 3D computer graphics but also for other general purpose computations,

[1] www.nvidia.com/page/geforce256.html

leading to the term GPGPU (General Purpose Graphics Processing Unit). GPGPU is considered as a modified form of stream processor. This concept turns the massive computational power of a modern graphics accelerator's shader pipeline into a general-purpose computing power. In some applications requiring massive vector operations, this can yield several orders of magnitude higher performance than a conventional CPU.

GPGPUs are used for many types of parallel tasks. They are generally suited to high-throughput type computations that exhibit data-parallelism. Furthermore, GPU-based high performance computers are playing a significant role in large-scale modeling. In the list of Top500 [2], three of the 10 most powerful supercomputers in the world use GPU as accelerators.

Great progress have been achieved by using GPU in many applications. For example, we can find on the site of Nvidia[3], the software HMMER [26], which is used for searching sequence databases for homologs of protein sequences, and for making protein sequence alignments, can be accelerated by a factor of 100 using 3 Tesla C1060 GPUs instead of one CPU. In [30], we can find some experiments on Matlab code with CUDA extension. For example, the simulation of 2D elliptic vortex evolution (mesh size 256×256) is accelerated by a factor of 11 using a Quadro FX5600 GPU comparing to an Opteron 250 processor.

So far, there are only few applications of GPUs to solve the full non-stationary incompressible 3D Navier-Stokes equations with an Eulerian grid based approach. Krüger [61] was one of the first to publish results in this field with the target of real-time applications for fluid dynamics. His solver uses Chorin's projection approach [21] on a staggered grid, finite differences with forward and central differencing, velocity advection as proposed by Stam [88], a conjugate gradient solver for the pressure Poisson equation and vorticity confinement [89] to reduce the numerical diffusion introduced by Stam's method.

Furthermore, Thibault and Senocak [91] implemented a multi-GPU solver for the full incompressible Navier-Stokes equations. Instead of Stam's advection approach they use a first order explicit Euler scheme. The pressure Poisson equation is solved by a Jacobi iterative solver. To use multiple GPUs, they consider a shared-memory parallelization by Posix threads and a standard domain decomposition approach. Due to hardware limitations, they also compute in single precision. This way, they obtain a speedup factor of 33 on one GPU compared to a single CPU and a speedup factor of 100 on four GPUs.

[2] www.top500.org
[3] http://www.nvidia.com/object/bio_info_life_sciences.html

Cohen and Molemaker [23] implemented a double precision solver for the Navier-Stokes equations. They included temperature into their model via the Boussinesq approximation [38]. The discretization employs a second order finite volume approach on a staggered regular grid, the pressure projection method and a second order Adams-Bashfort time integration [44]. A multigrid solver handles the Poisson equation. This way, a maximum speedup of 8.5 is obtained on the latest available graphics hardware (NVIDIA C1060) compared to an eight-core multithreaded fluid solver.

A significant effort to implement three-dimensional finite difference methods on GPUs has been made by Micikevicius [70]. He introduced base patterns for the fast computation of high order finite difference stencils. Additionally, he presented a scalable and fast multi-node/multi-GPU parallelization using MPI.

Because our Navier-Stokes solver is already functional on multicore architectures, we do not want to modify its main structure for the sake of re-usability for physicists at LIMSI. The strategy of adding GPU computation to the original code is called the "minimal invasion" strategy [22]. The minimal invasion strategy means that we replace the codes to be optimized, by some GPU codes while keeping the same inputs and outputs. In this way, we may not reach the optimum GPU performance but it is simple to switch between CPU and CPU/GPU codes. The main purpose of our work is to develop a general Navier-Stokes solver supporting different architectures. In Chapter 2, we have shown that the user can configure the solver with MPI and OpenMP. Now if the user has access to some accelerators, he can also take advantage of the devices by adding the GPU computing in the solver.

The most common programming languages on GPU are CUDA ("Compute Unified Device Architecture", [73]) by Nvidia and OpenCL ("Open Computing Language", [95]) by the Khronos Group[4]. CUDA is specifically for NVIDIA GPUs whilst OpenCL is designed to work across a multitude of architectures including GPU, CPU and digital signal processor (DSP). These technologies allow specified functions from a C program to run on the GPU's stream processors. This enables C programs to take advantage of GPU's ability to operate on large matrices in parallel, while still making use of the CPU when it is appropriate.

We use CUDA in our work. To avoid using the CUDA Fortran compiler, we add a C interface (see Algorithm 4) using the `ISO_C_BINDING` module to correctly call any CUDA function.

[4]The Khronos Group was founded in January 2000 by a number of leading media-centric companies, including 3Dlabs, ATI, Discreet, Evans & Sutherland, Intel, NVIDIA, SGI and Sun Microsystems, dedicated to creating open standard APIs to enable the authoring and playback of rich media on a wide variety of platforms and devices.

Algorithm 4: Illustration of Fortran-C-CUDA interface.

In a Fortran file:

```
use ISO_C_BINDING
type(C_PTR) :: device_array
    !... declare a CUDA array of C-pointer type
integer(C_LONG) :: device_ad
    !... declare a variable to store the address of device array
Call GPU_alloc(device_array, device_ad, size)
    !... Allocates some GPU memories
call GPU_routine(device_array, device_ad, ...)
    !... Do some computes on GPU
call GPU_free(device_array, device_ad)
    !... Free GPU memories
```

In a C file:

```
GPU_alloc_(device_array, device_ad, size) {
      cudaMalloc(device_array, size); // CUDA memory allocator
      device_ad = (unsigned long) *device_array;}// store explicit address
GPU_routine_(device_array, device_ad) {
      device_array = (double*) device_ad;}// find the right memory to proceed
      some computes;
GPU_free_(device_array, device_ad) {
      device_array = (double*) device_ad;}
      cudaFree(device_array); // CUDA memory deallocator
```

3.2 Using GPU for solving Navier-Stokes equations

As described in Chapter 2, the Navier-Stokes solver is composed mainly of a Helmholtz-like solver and a Poisson solver. So it is necessary to improve the performance of these two solvers in order to achieve better performance of the Navier-stokes solver. We explain in the following how we can use GPU capabilities to attain such a goal.

3.2.1 A GPU Helmholtz-like solver

In this section, we present a Helmholtz solver using GPUs. The method for solving a Helmholtz-like problem is, as mentioned in Section 2.1, the ADI method. With the same domain configuration and spatial discretization, the ADI method consists in solving three tridiagonal systems. Then it is essential for developing the Helmholtz solver to have an efficient tridiagonal system solver on GPU.

Fig. 3.3 shows how the global computational time is distributed among these tasks when considering one iteration of the Navier-Stokes solver (CPU code using only MPI) for a Helmholtz-like problem (mesh size = 240^3) on a multicore system of two Intel E5645

6-core processors. We observe that solving the tridiagonal systems (solve) represents about 2/3 of the execution time. We also see in this figure that the calculations for convection and diffusion flux represent about 30% of the total execution time. So it is also an important aspect to take into account in the design of a GPU Helmholtz solver.

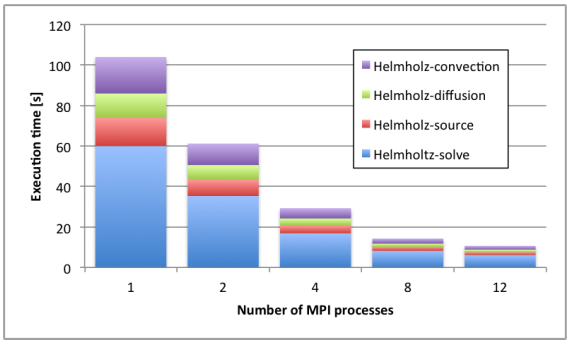


FIGURE 3.3: Time breakdown in Helmholtz equation (Intel Xeon E5645 2×6 cores 2.4 GHz.)

To solve the tridiagonal systems resulting from the ADI method, we implement the Thomas algorithm on GPU. If the domain has no solid obstacles, the tridiagonal system is considered as one tridiagonal matrix with multiple right-hand-side vectors. If we look at Fig. 2.6 and Eq. (2.4), when solving the block tridiagonal subsystem with B, each tridiagonal block in B is identical. Thus matrix B can be represented by only one of its tridiagonal block. With this "multiple RHS structure" of the system, we can exploit parallelism with GPU computing, as will be explained in the remainder.

One important aspect in GPU computing is the manipulation of GPU threads. As shown in Fig. 3.4, threads are gathered into blocks that form a grid. One thread block is executed on one streaming multiprocessor. Threads in the same block have faster access to the shared memory than to the global memory. If possible, it is better to store data on shared memory for faster I/O operations.

We describe the GPU version of the Thomas algorithm using one GPU. When there are multiple GPUs, or in other words multiple subdomains, we apply the Schur complement method described in Section 2.1, which divides the tridiagonal system into smaller ones and associates each subproblem to one GPU.

As illustrated by the upper part of Fig. 3.5, we store the tridiagonal matrix in the GPU shared memory so that it is accessible by all threads in the same thread block. The right-hand side matrix is divided into blocks according to the number of thread blocks which is a preset parameter. The size of the thread block, or in other words, the number of threads in one block, is defined by the ratio between the column number of the right-hand side matrix and the number of blocks. For a problem of size 256^3, if we want to use 32 thread blocks, then the block size should be $256^2/32 = 2048$. Once the

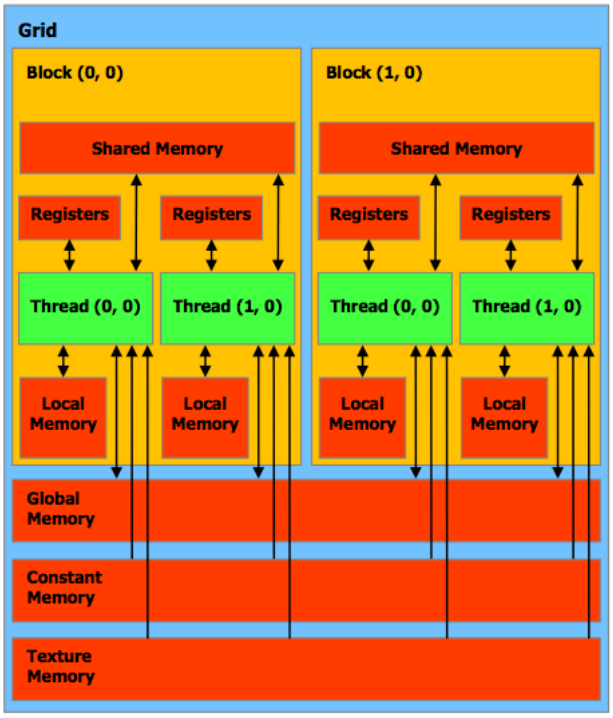


FIGURE 3.4: Illustration of GPU thread, block and grid.

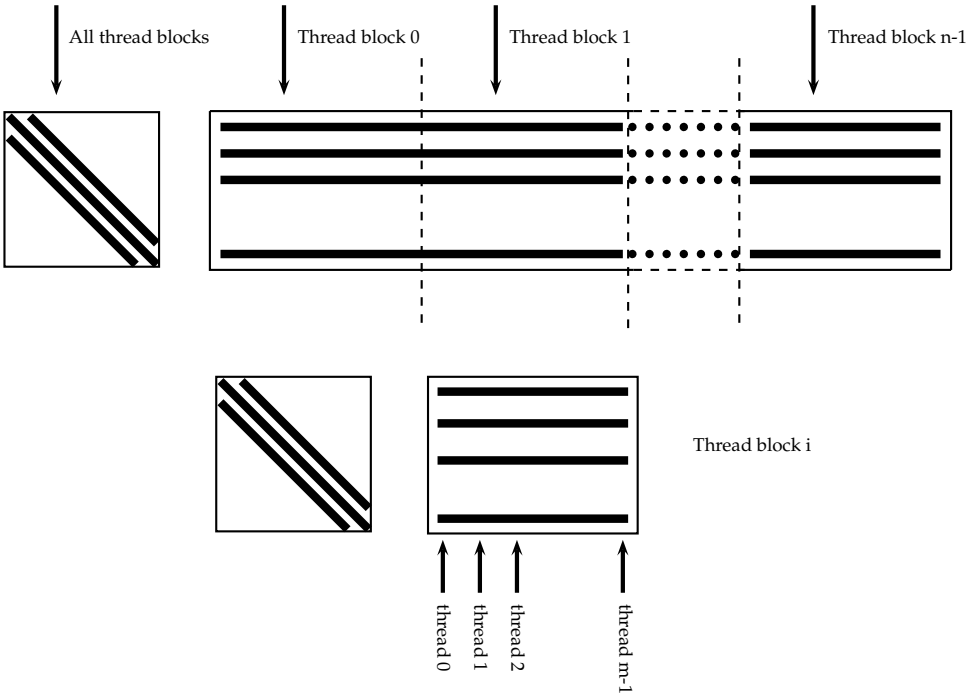


FIGURE 3.5: Assignment of tridiagonal matrix and RHS to thread blocks.

input data have been distributed, as shown in the lower part of Fig. 3.5, each thread block has access to the tridiagonal matrix and one part of the right-hand side matrix. However, the right-hand-side matrix cannot fit in the shared memory. For the problem with size 256^3, the size of the matrix is 4MB while the size of the shared memory is only 49152 bytes on a Kepler K20 GPU. So we keep the right-hand side matrix in the global memory and load the row that we want to update into the shared memory as explained in Algorithm 5.

Recall the Helmholtz-like equations resulting from the ADI method:

$$\begin{cases} \left(I - \dfrac{\alpha\Delta t}{\mathrm{Re}}\Delta_x\right) T_1 = S, & \text{(3.1a)} \\ \left(I - \dfrac{\alpha\Delta t}{\mathrm{Re}}\Delta_y\right) T_2 = T_1, & \text{(3.1b)} \\ \left(I - \dfrac{\alpha\Delta t}{\mathrm{Re}}\Delta_z\right) \delta_{\mathbf{u}}^{(n+1)} = T_2. & \text{(3.1c)} \end{cases}$$

As system (3.1) is solved successively, the solution of the first equation is used as the right-hand side of the second equation. However, the matrices ($I - \dfrac{\alpha\Delta t}{\mathrm{Re}}\Delta_x$, $I - \dfrac{\alpha\Delta t}{\mathrm{Re}}\Delta_y$ and $I - \dfrac{\alpha\Delta t}{\mathrm{Re}}\Delta_z$) in system (3.1) are tridiagonal only if the variables are ordered with respect to the solving direction (x, y, z, respectively). So we need a GPU kernel which deals with the reordering according to the solving direction.

Algorithm 6 is an example of reordering from $x \to y \to z$ to $y \to z \to x$, where n_1, n_2, n_3 are sizes of variables along the x, y, z directions, respectively.

Another method for solving Eq. (3.1) uses the explicit inverse of matrix B. We can find in [98] an expression for the inverse of a general non-singular tridiagonal matrix A, where each component of A^{-1} can be expressed as

$$A_{ij}^{-1} = \begin{cases} (-1)^{i+j} c_i c_{i+1} \dots c_{j-1} \theta_{i-1} \phi_{j+1} / \theta_n, & i < j, \\ \theta_{i-1}\phi_{i+1}/\theta_n, & i = j, \\ (-1)^{i+j} a_{j+1} a_{j+2} \dots a_i \theta_{j-1} \phi_{i+1} / \theta_n, & i > j, \end{cases} \tag{3.2}$$

where θ_i verify the recurrence relation,

$$\theta_i = b_i \theta_{i-1} - c_{i-1} a_i \theta_{i-2}, \quad \text{for } i = 2, ..., m,$$

with initial conditions $\theta_0 = 1$ and $\theta_1 = b_1$, and ϕ_i verify the recurrence relation,

$$\phi_i = b_i \phi_{i+1} - c_i a_{i+1} \phi_{i+2}, \quad \text{for } i = m-1, ..., 1,$$

Algorithm 5: GPU implementation of Thomas algorithm with multiple RHS.

Data: Copy of the tridiagonal matrix (ld, d, ud) and the RHS vectors b_d.
Result: Solution x_d (stored in b_d).

Declare two shared vector s1 and s2
Forward elimination:
Load row 1 of b_d into s1
for $i = 2, 4, 6, \ldots,$ **do**
    Load row i of b_d into s2
    Compute the elimination coefficient δ_i
    for *each thread* **do**
        Update s2 by s2 += δ_is1
    end
    Store s2 back to b_d
    Load row $i+1$ of b_d into s1
    Compute the elimination coefficient δ_{i+1}
    for *each thread* **do**
        Update s1 by s1 += δ_{i+1}s2
    end
    Store s1 back to b_d
end
Backward substitution:
Load row n of b_d into s1
for *each thread* **do**
    Update s1 using s1 /= d_n s1
end
Store s1 back to b_d
for $i = n-1, n-3, \ldots,$ **do**
    Load row i of b_d into s2
    for *each thread* **do**
        Update s2 using s2 = $\dfrac{s2 - ud_i s1}{d_i}$
    end
    Store s2 back to b_d
    Load row $i-1$ of b_d into s1
    for *each thread* **do**
        Update s1 using s1 = $\dfrac{s1 - ud_{i-1} s2}{d_{i-1}}$
    end
    Store s1 back to b_d
end

with initial conditions $\phi_{m+1} = 1$ and $\phi_m = b_m$. We also observe that $\theta_m = |A|$.

We use formulas (3.2) to compute the inverse of the three tridiagonal matrices of system (3.1) in the initialization step of the solver and store the inverse in GPU memory. The solution of system (3.1) is then computed by the matrix-matrix multiplications.

Algorithm 6: Reorder the solution array from $x \to y \to z$ to $y \to z \to x$.

int id_1, id_2, id_3;
for *each thread* i **do**
  $id_1 = i\%n_1$;
  $id_2 = (i\%(n_1 \times n_2))/n_1$;
  $id_3 = i/(n_1 \times n_2)$;
  $array_{new}[id_2 + id_3 \times n_2 + id_1 \times n_2 \times n_3] = array_{old}[i]$;
end

3.2.2 A GPU Poisson solver

In this section, we will discuss the construction of a Poisson GPU solver using the partial diagonalization method described in Chapter 1.

Fig. 3.6 represents the time breakdown for one iteration of a Poisson problem (mesh size $= 240^3$) using a CPU implementation on a multicore system with two Intel E5645 6-core processors and using only MPI parallelization. According to the partial diagonalization method mentioned in Section 1.5, the main tasks are base projections and tridiagonal solves. We observe in Fig. 3.6 that the most time-consuming part is the base projections, which correspond to matrix-matrix multiplications. In the remainder of this section, we explain how to improve this calculation using GPU accelerators.

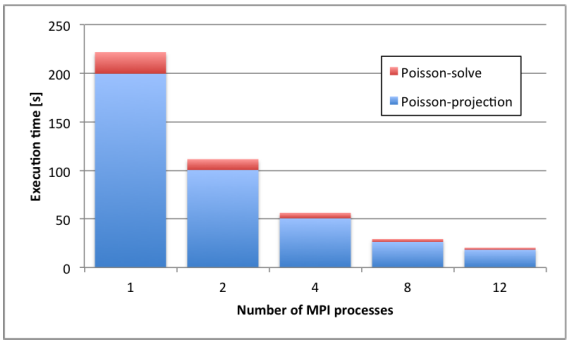


FIGURE 3.6: Time breakdown in Poisson equation (Intel Xeon E5645 2×6 cores 2.4 GHz.)

To reduce the execution time spent in matrix-matrix multiplication, we take advantage of GPU accelerators by calling the MAGMA [10, 93] routine `magmablas_dgemm`. MAGMA is a dense linear algebra library designed for multicore+GPU heterogeneous/hybrid systems. It contains most of the BLAS [1] and LAPACK routines modified to use accelerators like GPUs and more recently Intel Xeon Phi co-processors. One remark for performing the matrix-matrix multiplication is that we need to reorder the variables according to the considered direction. For example, to compute the new source term s', we have to reorder the source array s by the order $y \to z \to x$ to be able to use the `magmablas_dgemm` routine to compute $Q_y^{-1}s$. Next, we have to once again reorder

the result $Q_y^{-1}s$ in the order of $x \to y \to z$ to perform the second multiplication with Q_x^{-1}. For the same reason, the product $Q_x^{-1}Q_y^{-1}s$ must be ordered by $z \to x \to y$ to fit into the block tridiagonal structure, and the solution needs to be once again reordered according to the multiplication factor. The algorithm for the reordering is exactly the same as Algorithm 6 shown in Section 3.2.1.

Let us now study how to implement on GPU the matrix-matrix multiplication for the matrices Q_i on multiple subdomains. Suppose that the 3D domain is divided into p subdomains along x direction as shown in Fig. 3.7. According to the principles of domain decomposition, each subdomain is assigned to one multicore processor P_i and to one GPU G_i.

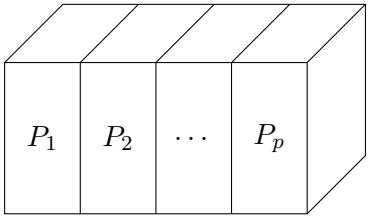


FIGURE 3.7: 3D domain decomposition along $i = 1$ direction.

Recall that the Poisson equation is solved via,

$$(\Lambda_x + \Lambda_y + \Delta_z)\tilde{\phi} = \tilde{s},$$

where

$$\begin{cases} \tilde{\phi} = Q_y^{-1}Q_x^{-1}\phi, \\ \tilde{s} = Q_y^{-1}Q_x^{-1}s. \end{cases}$$

Let us consider for instance the multiplication of Q_x^{-1} and s that are used to compute to projection of source term $\tilde{s}$. As the source array s is already distributed on the corresponding processor or accelerator, and its size is often important, we do not want to re-distribute s by column blocks to perform the usual parallel matrix-matrix multiplication. The redistribution of s can be very expensive especially when s is stored on the GPU memory. On the other hand, we distribute the matrix Q_x^{-1} by column blocks to the corresponding processor or accelerator as shown in Fig. 3.8.

With Q_x^{-1} distributed in blocks, on accelerator G_i, we multiply Q_{ji} and s_i with $j = 1, 2, ..., p$, where $Q_{ji} \in \mathbb{R}^{n_1 \times n_1}$ and $s_i \in \mathbb{R}^{n_1 \times (n_2 \times n_3)}$. Once all the p multiplications are performed, we send the results back to processor P_i and we call MPI routines `MPI_ALLTOALL` and `MPI_REDUCE` to distribute the block multiplication results and to obtain the final multiplication result as shown in Fig. 3.9.

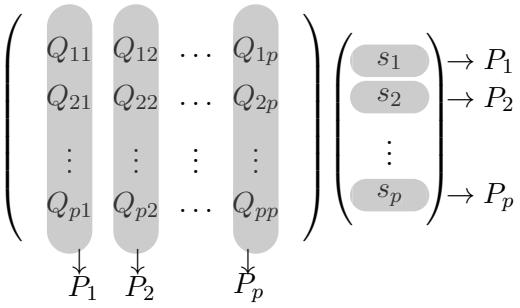


FIGURE 3.8: Distribution of $Q_1^{-1} = \{Q_{ij}\}_{i=1,\ldots,p;j=1,\ldots,p}$ and s on multiple processors.

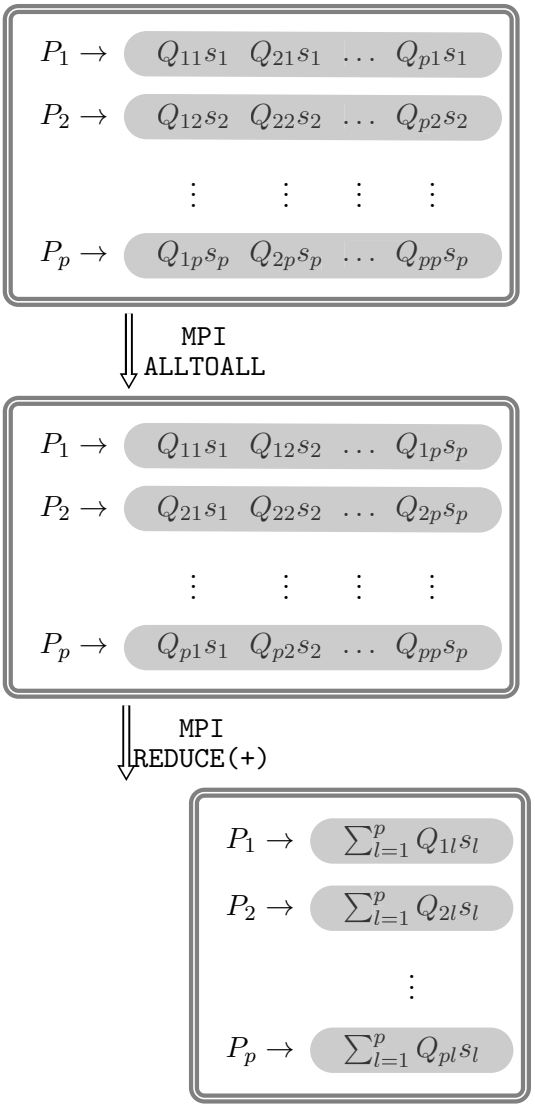


FIGURE 3.9: Matrix-matrix multiplication with multiple subdomains.

To summarize, we compute Q_x, Q_y, Q_x^{-1} and Q_y^{-1} in the initialization phase and copy the corresponding column blocks in the associated GPU. While doing the matrix-matrix multiplication, we first transfer (or not if it is already on the GPU) the source array s to the GPU and then call the MAGMA routine `magmablas_dgemm` to obtain the blocks $Q_{ij}s_j$ which are then sent back to the processor. Then we exchange the necessary information via MPI routines and send back to GPU the final solution in order to perform the next multiplication or the tridiagonal solve.

3.2.3 General structure of the GPU solver

In the previous sections, we described our GPU solvers for Helmholtz and Poisson problem separately. In this section, we will explain how GPU routines are integrated in the general NS solver.

We described in Chapter 2 a Navier-Stokes solver for shared and distributed memory architectures. Because of the fast development of GPUs, most of the modern computers have at least one GPU attached to the CPU host. So if we want our solver to be for general use, it should exploit the capabilities of GPU for the tasks that are able to take advantage of them, similarly to the approach described for instance in [11]. However, the solver should be able to work and thus the main structure of the code will remain unchanged and the use of GPU is considered as an alternative when performing simulations.

The strategy we applied in developing the code is called the "minimum invasion" method, meaning that the GPU code should not interfere much with the CPU code. A minimum interface will be established between these two programs for information exchanges.

Of course, GPU computing is not always the best choice when constructing the Navier-Stokes solver. As shown in Sections 3.2.1 and 3.2.2, we can identify the most time-consuming tasks in the Navier-Sokes solver. We choose to add GPU alternatives for routines dealing with tridiagonal solves and matrix-matrix multiplications. In this way, if the user chooses to use GPU resources, the solver will execute the GPU version of the routines. Otherwise, the solver keeps the CPU version and the GPU routines are simply ignored.

The code structure can be illustrated by Fig. 3.10 showing how GPU computing is integrated into the Navier-Stokes solver.

- Initialization
  - Data initialization: read the input file
  - Domain initialization: construct the subdomains
  - Fields initialization: allocate arrays
  - Operators initialization: construct the tridiagonal matrices for Helmholtz-like and Poisson problems
  - Compute projection matrices Q_x, Q_y, Q_x^{-1}, Q_y^{-1} if partial diagonalization is chosen
  - Memory transfer: Send the tridiagonal matrices and projection matrices to device
- Loop on time
  - Solve Helmholtz-like problem on device:
    - Send right-hand sides to device
    - Perform tridiagonal solve on device
    - Send solution to host
  - Solve Poisson problem
    - Send right-hand sides to device
    - Perform matrix-matrix multiplication on device and tridiagonal solve on device
    - Send solution to host
  - Velocity update
  - Store results
- Finalize

FIGURE 3.10: Solver structure with GPU computing.

3.3 Experimental results

3.3.1 Overview of computational resources

We perform experiments mainly on two machines. The first one, which is used for code development, is a local workstation in our laboratory. This machine is composed of 2 sockets of one Intel Xeon E5645 hexacore processor and a Tesla C2060 GPU.

The second machine is the Stampede system from Texas Advanced Computing Center (TACC) already described in Section 2.2.3. In particular we will use the NVIDIA K20 GPUs associated to some of the nodes (128) of Stampede.

3.3.2 Performance of the Helmholtz solver

The 3D Helmholtz test problem that we consider is defined as follows.

$$\begin{cases} \mathbf{V}(\mathbf{x}) - \alpha\Delta\mathbf{V}(\mathbf{x}) &= \mathbf{S}(\mathbf{x}), & \mathbf{x} \in \Omega = (0,1)^3, \\ \mathbf{V}(\mathbf{x}) &= 0, & \mathbf{x} \in \partial\Omega, \end{cases} \tag{3.3}$$

where $\mathbf{S} = (1 + 3\alpha\pi^2)\mathbf{V}$, $\mathbf{x} = (x_1, x_2, x_3)$ and $\alpha = 10^{-7}$. The exact solution is:

$$\mathbf{V}(\mathbf{x}) = \begin{pmatrix} sin(\pi x_1)sin(\pi x_2)sin(\pi x_3) \\ sin(\pi x_1)sin(\pi x_2)sin(\pi x_3) \\ sin(\pi x_1)sin(\pi x_2)sin(\pi x_3) \end{pmatrix}.$$

In Fig. 3.11 we compare the two methods described in Section 3.2.1 for solving the tridiagonal systems resulting from Eq. (3.3) for different mesh sizes (Thomas algorithm and explicit inverse). For both methods we compute the absolute error given by $||u-u_h||/\sqrt{\mathcal{N}}$ where u and u_h are respectively the exact and approximate solutions. We noticed that the error is the same for both methods. Regarding the performance, Fig. 3.11 represents the execution time for the two methods. For instance, for a mesh size of 256^3, using an explicit inverse enables us to gain a factor 4 over the Thomas algorithm. However, as computing the explicit inverse suits only for problems with same boundary conditions for each direction, the standard Thomas algorithm will be still useful in more general Helmholtz problems.

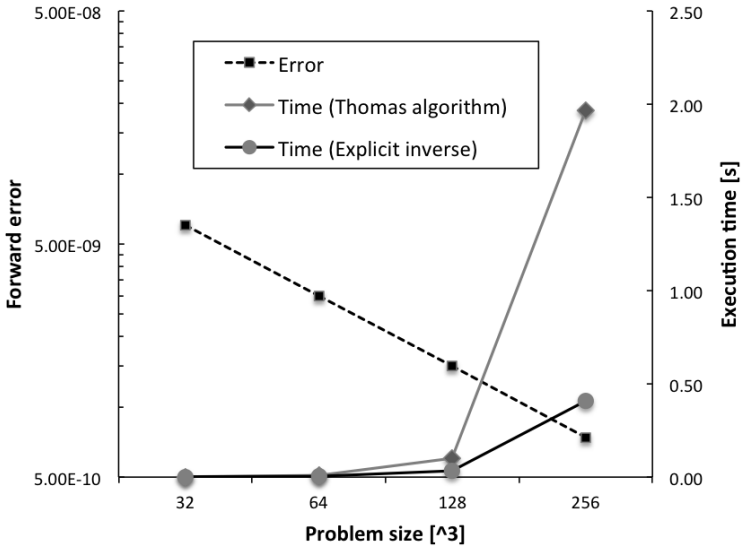


FIGURE 3.11: Performance of Helmholtz solver using Thomas algorithm and explicit inverse.

3.3.3 Performance of the Poisson solver

Let us now consider the following Poisson problem.

$$\begin{cases} \Delta\phi(\mathbf{x}) &= s(\mathbf{x}), & \mathbf{x} \in \Omega = (0,1)^3, \\ \dfrac{\partial\phi}{\partial\mathbf{n}}(\mathbf{x}) &= 0, & \mathbf{x} \in \partial\Omega, \end{cases} \tag{3.4}$$

where $s = -\pi^2\phi$ and the exact solution is

$$\phi(\mathbf{x}) = cos(\pi x_1)cos(\pi x_2)cos(\pi x_3).$$

The performance of the Poisson solver is presented in Fig. 3.12. We observe that, when the problem size grows, the error (same definition as in Section 3.3.2) decreases and the execution time increases.

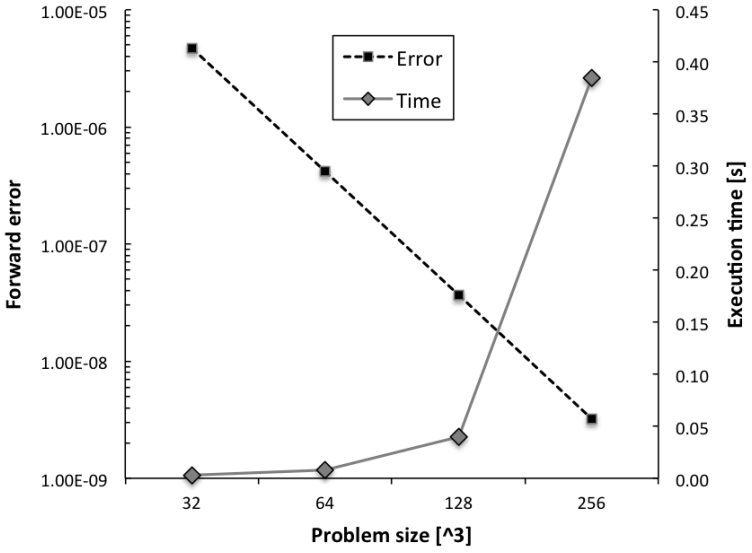


FIGURE 3.12: Performance of Poisson solver.

Tab. 3.1 lists the time breakdown for one iteration of the Helmholtz and Poisson solvers and compares the performance of the GPU solvers to that of CPU solvers. We observe that the GPU implementations enable us to accelerate the calculations with roughly a factor 8 and 5 for the Helmholtz and Poisson solvers respectively when compared to the CPU solvers using MPI or multithreading. The data transfer from CPU to GPU for the Helmholtz solver includes three RHS vectors (size 240^3) and three inverted matrices B^{-1} (240^2). For the Poisson solver, the amount is three diagonals of size 240^3, one RHS vector of size 240^3, and the matrices $Q_1, Q_1^{-1}, Q_2, Q_2^{-1}$ of size 240^2. Consequently, the

data movements for the Poisson solver require 1/3 more time than for the Helmholtz solver, which is confirmed in Tab. 3.1.

	Helmholtz (with B^{-1})	Poisson
Transfers CPU-GPU (only once)	85	109
Matrix multiplication	216	96
Solution reordering	126	84
Tridiagonal system solve	-	165
Total CPU solver (12 MPI procs)	2700	1460
Total CPU solver (12 threads)	2750	1760
Total GPU solver	342	345

TABLE 3.1: Time (ms) distribution for Poisson and Helmholtz GPU solvers (mesh size $= 240^3$).

3.3.4 Performance of the hybrid CPU/GPU Navier-Stokes solver

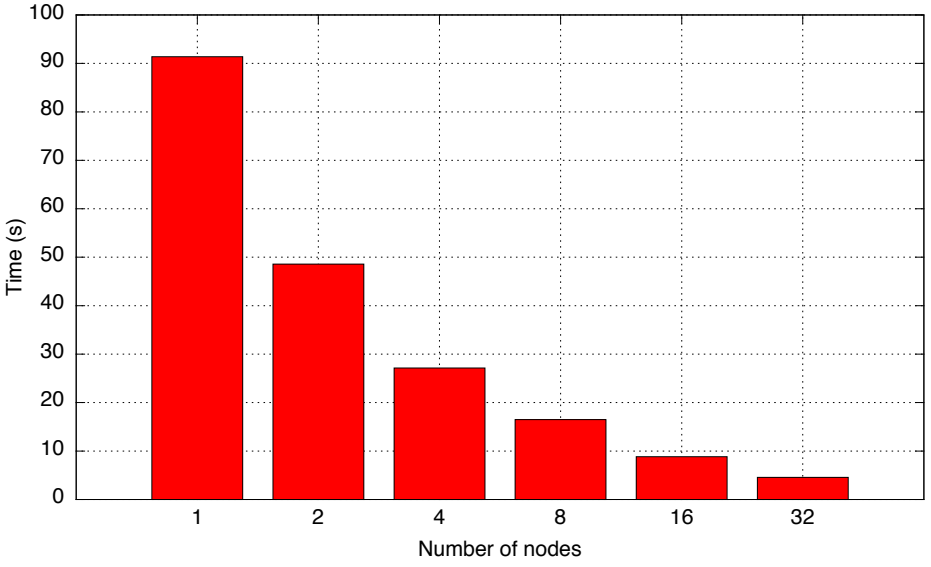


FIGURE 3.13: Time for solving Navier-Stokes equations using CPU/GPU system (Stampede).

Fig. 3.13 depicts the performance of the Navier-Stokes solver by measuring the execution time of 10 time iterations for a problem of size 256^3. On the x-axis, one node includes 2 Intel Xeon E5 processors and one Kepler K20 GPU. We observe in Fig. 3.13 that the execution time decreases with the increasing number of computing nodes. We compare this result to the performance result presented by Fig. 2.17(a) in Section 2.5.2 by computing the acceleration ratio $1 - \frac{\text{Time(GPU)}}{\text{Time(CPU)}}$. We obtain an acceleration of 44.8% with one node and 34.5% with 32 nodes. The acceleration ratio decreases when we increase the nodes. This is because in the strong scaling test, the computational workload per node decreases when we increase the number of nodes. GPU routines are more efficient

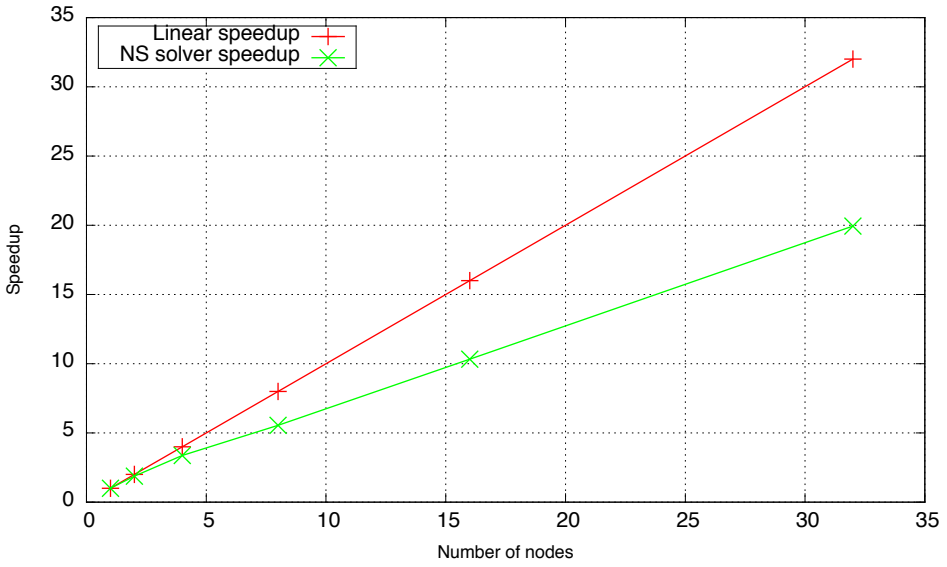


FIGURE 3.14: Parallel speedup for CPU/GPU Navier-Stokes solver (Stampede).

when they deal with large amount of data. Thus the acceleration ratio drops with multiple nodes, illustrating that GPU are less efficient in this case. In Fig. 3.14 we plot the speedup of the solver (strong scaling). We obtain a speedup of 20 with 32 computing nodes.

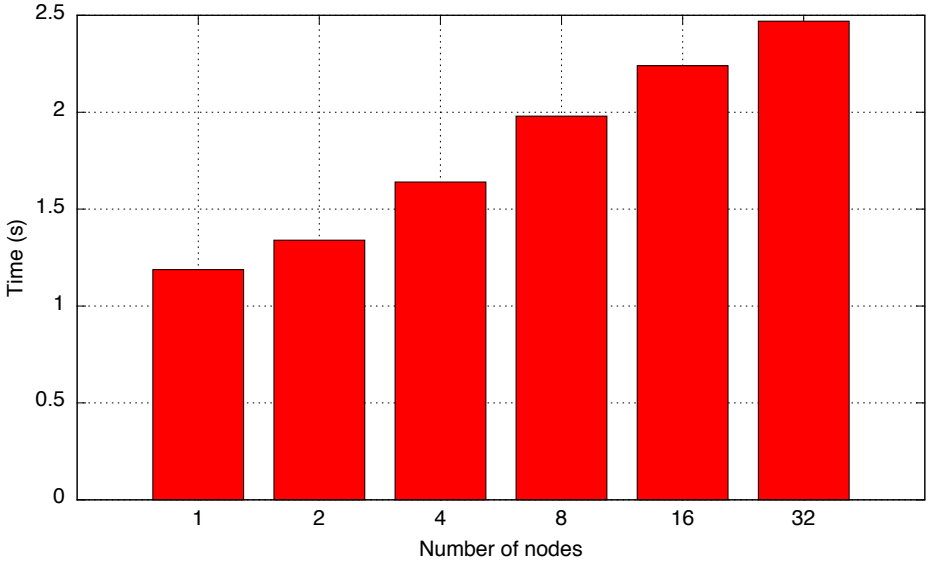


FIGURE 3.15: "Weak" scaling performance for Navier-Stokes solver (Stampede).

We also study the "weak" scalability of the solver, with a mesh size per node equal to 64^3. In this experiment, we increase the number of nodes and measure the execution time for 10 iterations. We observe in Fig. 3.15 that the execution time increases slightly with the number of nodes. This behavior is similar to that of Fig. 2.18 in Section 2.5.2. This is because the amount of information exchanged between nodes is the same in these

two tests. Only the execution time for the same number of nodes is different because of the existence of GPU computing in Fig. 3.15.

3.4 Conclusion of Chapter 3

In this chapter, we explained our motivation for using GPU accelerators in our Navier-Stokes solver. We designed new algorithms and implementations for the Helmholtz and Poisson solvers using GPUs and we obtained satisfactory speedups. Then we modified our Navier-Stokes solver to integrate the new GPU versions for some routines. The resulting hybrid CPU/GPU Navier-Stokes solver is operational and can be used on different types of architectures. The experiments using the CPU/GPU solver on a cluster with up to 32 computational nodes (512 cores and 32 K20 GPUs) show a good scalability and a gain up to 45% of the execution time when compared to the CPU-only implementation. Our performance results also show that our GPU solver is more efficient in large-scale problem simulations. We published some of these results in a recent publication [101].

Chapter 4

Simulations of Physical Problems

Contents

In the field of computational fluid dynamics, comprehensive benchmarks are very important yet numerically challenging. Numerical benchmark cases give people frameworks to quantitatively explore limits of the computational tools and to validate them. In this Chapter, we use two benchmark problems for the purpose of validating our numerical solver and assessing its computational efficiency. The first benchmark concerns the simulation of the evolution of three dimensional Taylor-Green vortices in a 3-periodic cube. This setting allows us to use GPUs acceleration for the tridiagonal solves of the linear systems resulting from the ADI (see Section 1.4) and partial diagonalization (see Section 1.5) methods for the Helmholtz and Poisson problems. The second benchmark problem concerns the three-dimensional flow around a square cylinder. The presence of an obstacle (the cylinder) inside the computational domain renders the problem non-separable. The solver then continues to rely on the ADI method for solving the Helmholtz

systems, but no partial diagonalization is made for the Poisson equations. Instead, an iterative method (SOR with multigrid, see Appendix A) is employed.

The numerical simulations presented in the Chapter were all carried out on the Stampede system described in Section 2.2.3. The visualization of the flow fields are performed using ParaView [71].

4.1 Three dimensional Taylor-Green vortices

The three dimensional Taylor-Green vortices benchmark is selected as it gives rise to a separable problem. It allows for the use of the direct methods designed in the thesis for the solution of the Helmholtz and Poisson equations. The benchmark is then ideal to measure the performance of our solver on an heterogeneous architecture.

4.1.1 Benchmark settings

The Taylor-Green (TG) flow is three-dimensional and periodic in all spatial directions. The dimensionless initial condition of the TG flow is given by

$$\begin{cases} u = sin(x)cos(y)cos(z), & \text{(4.1a)} \\ v = cos(x)sin(y)cos(z), & \text{(4.1b)} \\ w = 0, & \text{(4.1c)} \\ p = p_0 + \frac{\rho_0}{16}(cos(2x) + cos(2y))(cos(2z) + 2), & \text{(4.1d)} \end{cases}$$

for $(x, y, z) \in [-\pi, \pi]^3$. The TG flow is governed by the 3D incompressible Navier-Stokes equations. The Reynolds number of the flow is here defined as $\mathrm{Re} = \frac{\rho_0 V_0 L}{\mu}$ where V_0 and L are reference velocity and length. In our tests, we considered $\mathrm{Re} = 400$ and 800, while simulations are carried out from the initial condition at $t = 0$ to the final time $t_{final} = 20\, t_c$, where $t_c = \frac{L}{V_0}$ is the characteristic convective time.

Regarding the domain decomposition, we used 8 subdomains, 2 along each spatial dimension. Each subdomain is assigned to one computational node; thus, the simulation uses at total 8 nodes. Computations then use 3 mesh sizes 64^3, 128^3, and 256^3

Fig. 4.1 shows the initial condition of the flow, where the magnitude of the dimensionless velocity is plotted. The plot shows the initially regular arrangement of the vortices in the periodic domain. From this initial condition, the vortices subsequently evolve in a non-linear fashion, break-up into smaller vortices with the emergence of complicated small scale structures.

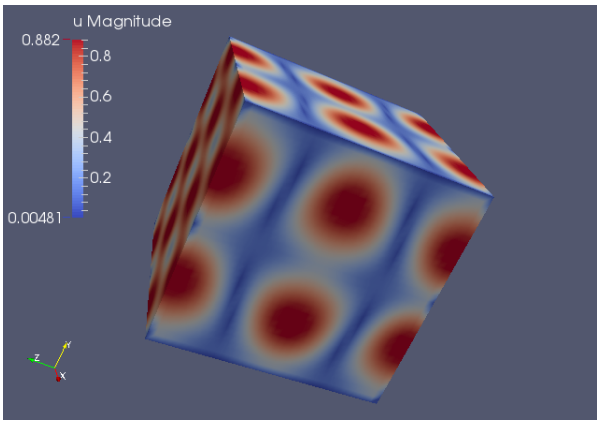


FIGURE 4.1: Initial status of Taylor-Green vortices problem.

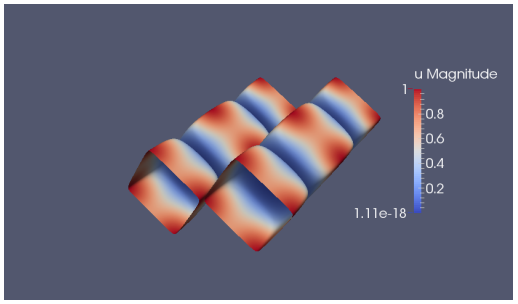
4.1.2 Validation

For the visualization of the flow and the evolution of the vortices, we plot at different times iso-surfaces of the so-called Q-criterion [49], a quantity classically used in CFD to extract and visualize coherent eddy structures in turbulent flows [50]. The Q-value is defined as the second invariant of $\nabla \mathbf{u}$. For an incompressible flow, we have

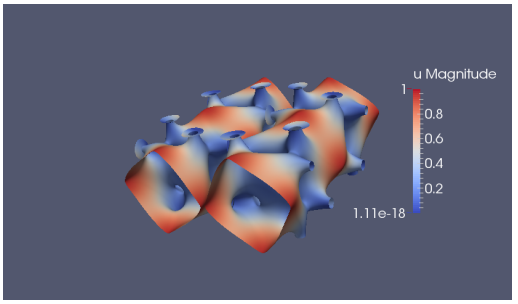
$$Q = \frac{1}{2}(\|\mathbf{\Omega}\|^2 - \|\mathbf{S}\|^2).$$

where $\mathbf{S}$ and $\mathbf{\Omega}$ the symmetric and antisymmetric parts of $\nabla \mathbf{u}$ respectively, and $\|\cdot\|$ is the Euclidean matrix norm. Coherent eddy-structures are then visualized by plotting the iso-surfaces of Q. In our results, we plot the iso-surface of $Q = 0.01$.

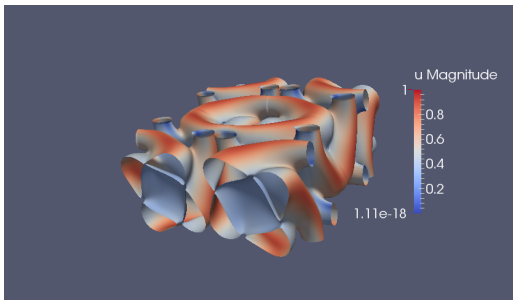
Fig. 4.2 shows iso-surfaces of the Q-criterion at different times and for Re $= 800$ with mesh size 256^3. For clarity, only the upper half of the domain is shown. In the plots, the iso-surfaces are colored by the magnitude of the velocity. We observe that from the very simple coherent structures at $t = 0$ (see Fig. 4.2(a)), the flow quickly evolves to a more complicated pattern of coherent structures at $t = 5$ convective times (see Fig. 4.2(d)), and eventually yields coherent structures at smaller and smaller scales, as time further increases, through vortices reconnection and break-up. The emergence of small scale structures also highlights the need for a fine mesh to correctly approximate the dynamics of such small features of the flow during the simulation. In addition, the complexity of the iso-surface relates to high velocity gradients where viscous dissipation takes place. The dissipation can be effect of the viscous dissipation can also be appreciated by the global decay of the velocity magnitude as time advances. Note however the presence at early time of higher velocity regions, in particular at $t = 5$ (see dark red areas in Fig. 4.2(d)), due to the vortex stretching.



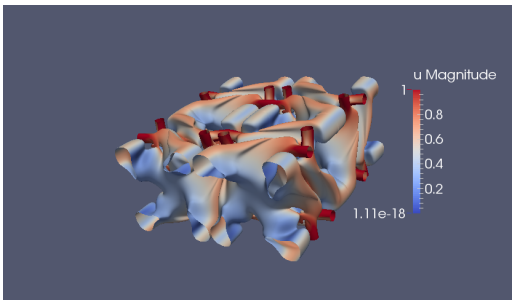
(a) Iso-surface of $Q = 0.01$ at $t = 0$.



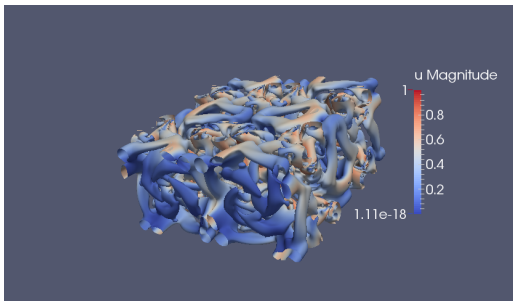
(b) Iso-surface of $Q = 0.01$ at $t = 1$.



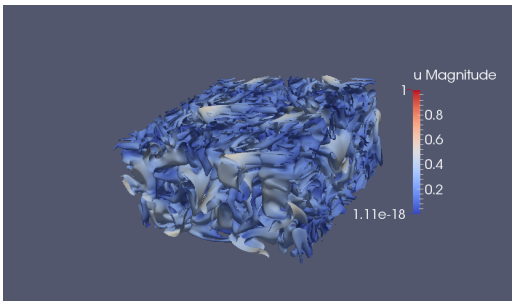
(c) Iso-surface of $Q = 0.01$ at $t = 3$.



(d) Iso-surface of $Q = 0.01$ at $t = 5$.



(e) Iso-surface of $Q = 0.01$ at $t = 10$.



(f) Iso-surface of $Q = 0.01$ at $t = 20$.

FIGURE 4.2: Iso-surface of $Q = 0.01$ of Taylor-Green vortices at different times.

The representation of the coherent structures provides of qualitative validation of the solver. For a quantitative validation we need a better criteria allowing for a comparison with the results published in the literature. To this end, we shall consider the time evolution of the dissipation rate [79]. The dissipation rate, $\langle \epsilon(t) \rangle$ is defined as the (negative) instantaneous variation of the averaged kinetic energy, that is

$$\langle \epsilon(t) \rangle \doteq -\frac{d \langle k(t) \rangle}{dt}, \quad \langle k \rangle = \frac{1}{2} \left[\langle u^2 \rangle + \langle v^2 \rangle + \langle w^2 \rangle \right].$$

Fig. 4.3 and 4.4 report the time evolutions of the dissipation rates from our simulations,

at Re = 400 and 800 respectively. For the two Reynolds numbers, computations using two meshes are contrasted to appreciate the convergence of the simulations. Focusing first on the case of Re = 400, in Fig. 4.3, we observe that the computed dissipation rates are the same for the 64^3 and 128^3 meshes up to $t \approx 5$. For latter times differences appear denoting the lack of resolution for the coarser mesh. In particular, the plateau in the maximum dissipation rate for $t \in [6, 9]$ is not well captured for the coarser mesh. In the case of the larger Reynolds number tested, Re = 800, the simulations for the two meshes agree better, although the 128^3 mesh seems to slightly over estimate the dissipation rate after the maximum. Again, insufficient discretization is to be blamed.

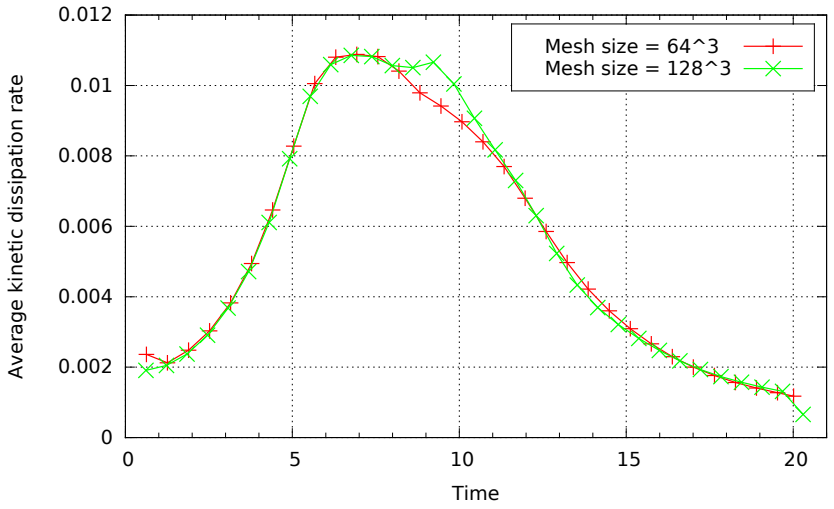


FIGURE 4.3: Dissipation rate $\langle \epsilon(t) \rangle$ at Re = 400, using 64^3 and 128^3 meshes.

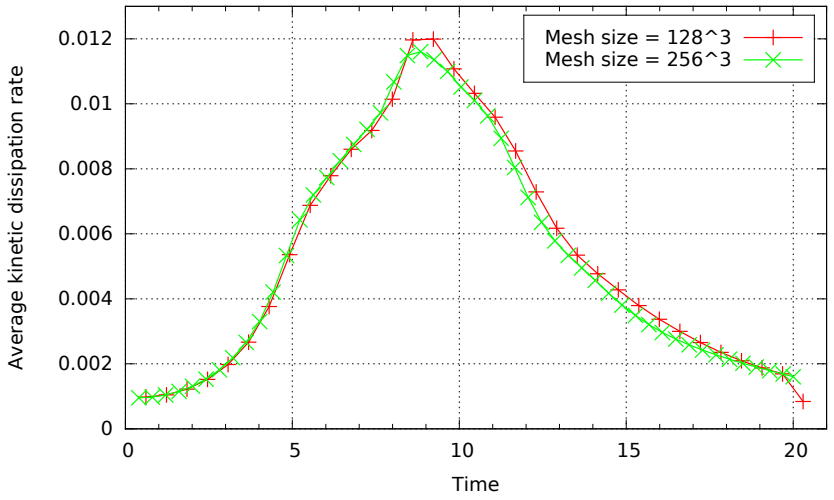


FIGURE 4.4: Dissipation rate $\langle \epsilon(t) \rangle$ at Re = 800, using 128^3 and 256^3 meshes.

To complete the validation of the solver on the TG flow benchmark, we compared in Fig. 4.5 our computation of the dissipation rate $\langle \epsilon(t) \rangle$ at Re = 800, using the 256^3 global mesh, with computations reported in the literature [15, 35, 76]. We observe that our

computation agrees well with the reference results. In [76], the mesh sizes are the same as in our tests for different Re numbers. The authors of [15] have chosen a mesh size of 256^3 for all tests with different Re numbers. As for [35], authors have used a high-order discontinuous Galerkin discretization for the simulation and the result we plotted as reference was obtained by using 64^3 degrees of freedom.

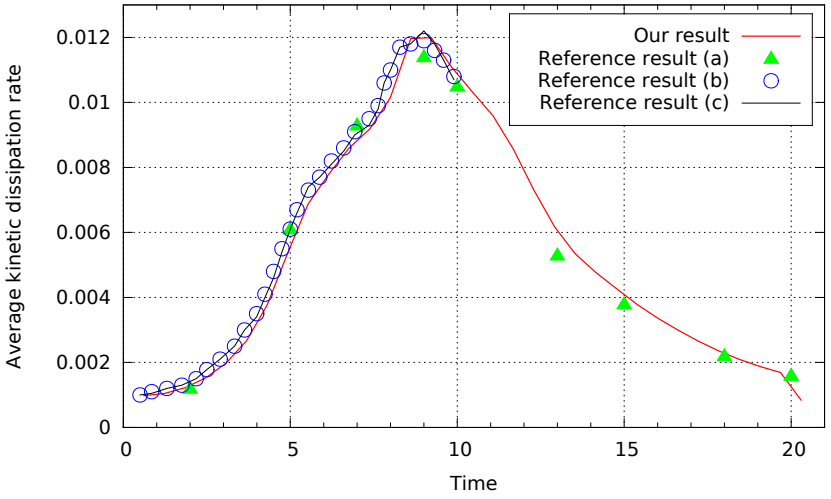


FIGURE 4.5: Comparison of our computation for the dissipation rate at Re = 800 with the computations of (a) Ouzzine [76], (b) Brachet *et al* [15] and (c) Gassner and Beck [35].

4.1.3 Performance analysis

To quantify the improvements brought by the GPU acceleration developed in the thesis, we compare in Tab. 4.1 the execution times for different types of parallelization of the 3D Navier-Stokes solver:

- **MPI+OpenMP+CUDA**: CPU/GPU hybrid parallelization using MPI, OpenMP and CUDA,
- **MPI+OpenMP**: parallelization using MPI combined with OpenMP,

The tests were carried out using 8 computational nodes of Stampede (total 128 cores), for meshes with total sizes, 64^6, 128^3 and 256^3, and a Reynolds number Re = 400. For the time measurement, we use 100 time iterations, without storing the solution. The initialization and set-up times are also excluded from the time measurements. In this way, the reported execution times truly represent the computational times spent in solving NS equations.

Table 4.1 reports the computational times (in second) of one iteration, for the three meshes and the different parallelizations. We first observe that, as expected, the GPU acceleration is able to significantly reduce the computational time, for all the mesh sizes considered. The improvement is quantified by means of the acceleration percentage defined as

$$\text{Acceleration percentage} = 1 - \frac{\text{Time(GPU)}}{\text{Time(CPU)}}, \tag{4.2}$$

where Time(GPU) represents the execution time using the CPU/GPU three-level parallelization and Time(CPU) is the execution time using MPI with OpenMP. The reported values for the acceleration percentage indicate that the GPU parallelization is becoming more efficient as the mesh size increases. Specifically, the acceleration relative to the MPI+OpenMP parallelization goes from $\approx 22\%$ to 36% using meshes 64^3 and 128^3. This positive trend can be explained by the fact that the GPU better exploits the parallelism when dealing with large amounts of data. This effect is also visible from the numbers in red shown in Tab. 4.1, which are the factors in the increase of execution time, with respect to the 128^3 mesh and for each of the parallelizations: while going to the 128^3 and 256^3 meshes we expect theoretically an increase of computational loads by factors of 8 and 64 respectively, it is seen that the execution times for the MPI+OpenMP+CUDA parallelization increase only by factors ≈ 7 and ≈ 58 respectively. This finding confirms the huge potential of using GPUs in large scale CFD simulations.

Mesh size	64^3	128^3	256^3
MPI+OpenMP+CUDA	0.028	0.2 (×7.14)	1.62 (×57.85)
MPI+OpenMP	0.036	0.27 (×7.5)	2.53 (×70.3)
GPU Acceleration from MPI+OpenMP	22.2%	25.9%	36%

TABLE 4.1: Time (s) per iteration of NS solver for Taylor-Green vortices (Re = 400) on Stampede (128 cores).

4.2 Flow around a square cylinder

The second benchmark considered in the thesis concerns the flow around a square cylinder. This benchmark, along with many others, has been defined within a DFG High-Priority Research Program by Schäfer and Turek [86], and since then has been investigated by many researchers. We can find in [55] and [14] results on this benchmark. In these articles, Re = 20 and this low Reynolds number leads to a steady flow. For higher Reynolds number, as the flow becomes unsteady, there is no precisely determined results [13]. However, we can find simulation results in many publications such as [56, 60, 81]. In our test, we are interested in the unsteady state of the flow around a square cylinder.

As the unsteadiness appears when Re $>$ 45 [96], we choose to study the case where Re $= 50, 100, 150$.

4.2.1 Benchmark settings

The configuration consists in a fixed cylinder with square section, having edge length D and axis in the z-direction. The cylinder is placed in the center of a channel made of two parallel walls (with normal in the y-direction) separated by a distance $H = 10\,D$. The distance from the cylinder centerline to the channel entrance is equal to $10\,D$. The inflow conditions at the channel entrance are set with a bulk velocity U_m in the x-direction with a parabolic velocity profile:

$$\begin{cases} u(0,y,z) = 6U_m y/H(1-y/H), & \text{(4.3a)} \\ v(0,y,z) = 0, & \text{(4.3b)} \\ w(0,y,z) = 0. & \text{(4.3c)} \end{cases}$$

The boundary conditions on the walls at $y = 0, y = 10\,D$ are homogeneous Dirichlet conditions $\mathbf{u} = 0$. On the lateral plans, at $z = 0$ and $z = 8$, we apply periodic boundary conditions. On the outlet plan of the channel, at $x = 40\,D$, we apply the Neumann condition $\dfrac{\partial \mathbf{u}}{\partial n} = 0$. We also have the flow rate preservation condition to ensure that the inlet volume equals to the outlet volume. The configuration of the flow domain is illustrated in Fig. 4.6.

Finally, the discretization of the domain is performed using orthogonal grids matching the cylinder boundaries. The numerical simulations are carried out using 4 nodes of Stampede, with 2 subdomains along both x and z directions. No domain decomposition is used along the y direction, because when sharing the obstacle between subdomains the iteration method used to solve Poisson equation converges slowly.

4.2.2 Validation

Based on the characteristic velocity U_m, length D and fluid viscosity ν, the problem is entirely defined from the Reynolds number $\mathrm{Re} = \dfrac{U_m D}{\nu}$, which is set to 100. For this value of the Reynolds number, the flow around the cylinder is unstable and develops a pattern of alternated vortices in the wake, the so-called Von-Karman street.

Fig. 4.7 shows fields of the longitudinal component of the velocity at different t, normalized by the convective time $t_c = D/U_m$. The plots show the results in a (x, y)-plan of the domain. We can see that from the inlet with parabolic profile, the flow develops instabilities behind the cylinder to form the Von-Karman street. The instability grows

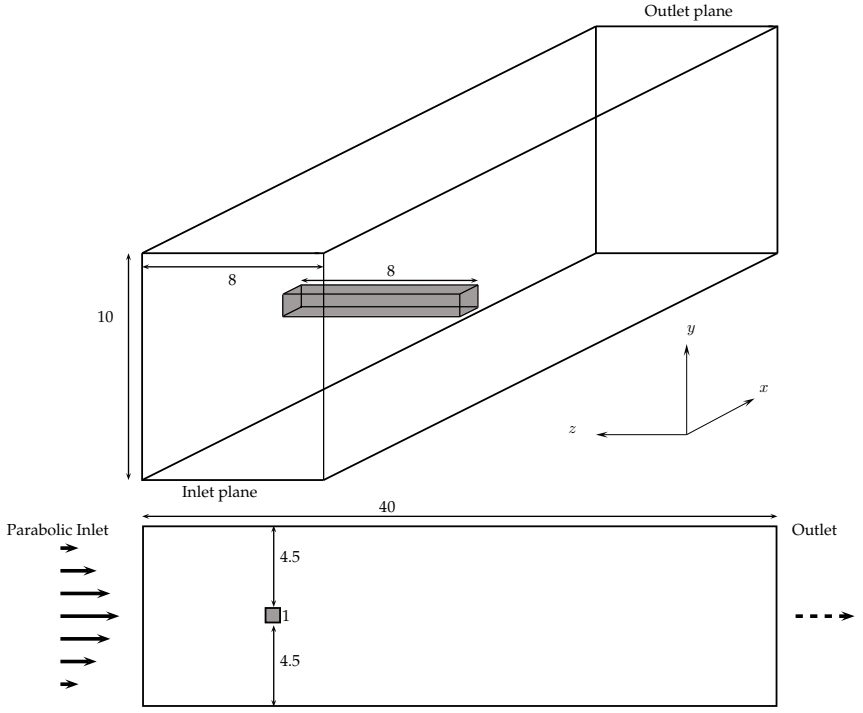


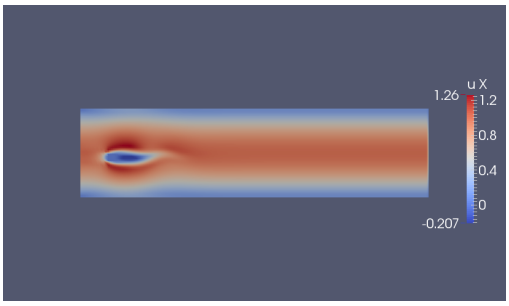
FIGURE 4.6: The geometry of the 3D laminar flow problem.

as time goes on, to eventually reach a periodic dynamics as expected for the Reynolds number considered here. Further analysis of the results (not shown) reveals that the flow is actually two-dimensional (w-component of the velocity is zero), although the simulation is three-dimensional. This finding demonstrates the correct behavior of the parallel solver which does not break the z-invariance of the flow by introducing spurious numerical instabilities.

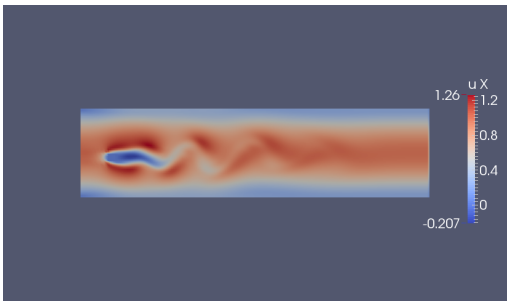
The Von-Karman street pattern is closely related to the Reynolds number, and we expect to observe vortex shedding with different wave-length and amplitudes for different Reynolds numbers. In Fig. 4.8, we plot simulation results for Reynolds numbers of 50, 100 and 150. We see that the shedding is weaker for flows at lower Reynolds numbers: in Fig. 4.8(a) the shedding is hardly formed while in Fig. 4.8(c) the shedding is very intense.

The vortex shedding process and the Von-Karman street can be better appreciated in Fig. 4.9, where plotted is the transverse component of the vorticity field.

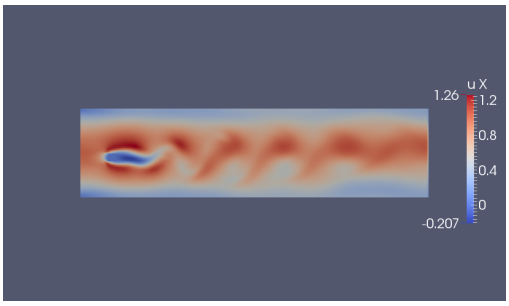
For the validation of the numerical simulations we focus on the reduced frequency of the vortex shredding. To this end, we record the temporal velocity field at two observation points P_1 and P_2 in the domain. We set P_1 and P_2 on the center line of the domain at a distance of D and $2\,D$ respectively from the cylinder in the downstream direction, and monitor the magnitude of the transverse velocity at P_1 and P_2. The signals are reported



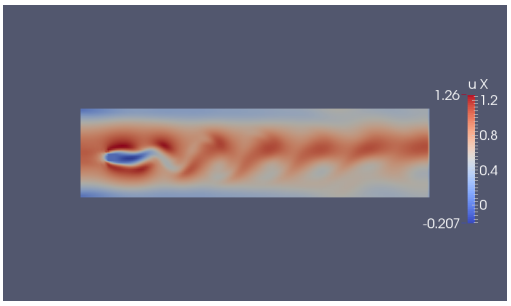
(a) $t = 10$.



(b) $t = 30$.

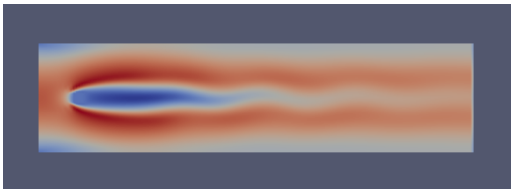


(c) $time = 50$.

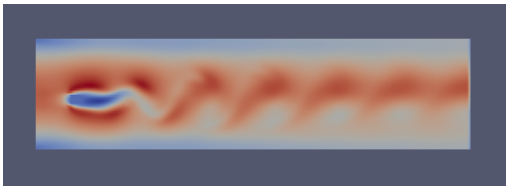


(d) $time = 70$.

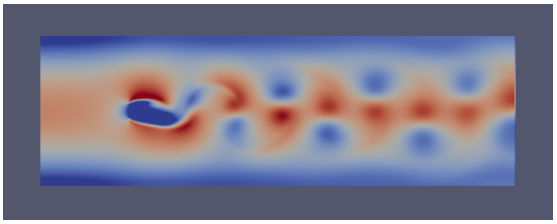
FIGURE 4.7: Longitudinal velocity field at different times. Simulation for Re = 100 with total mesh size = $320 \times 240 \times 32$.



(a) Re = 50.



(b) Re = 100.



(c) Re = 150.

FIGURE 4.8: Snapshots of the longitudinal velocity field illustrating the structure of the Von-Karman street at different Reynolds numbers.

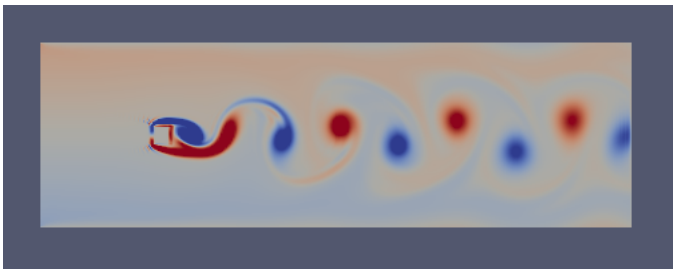


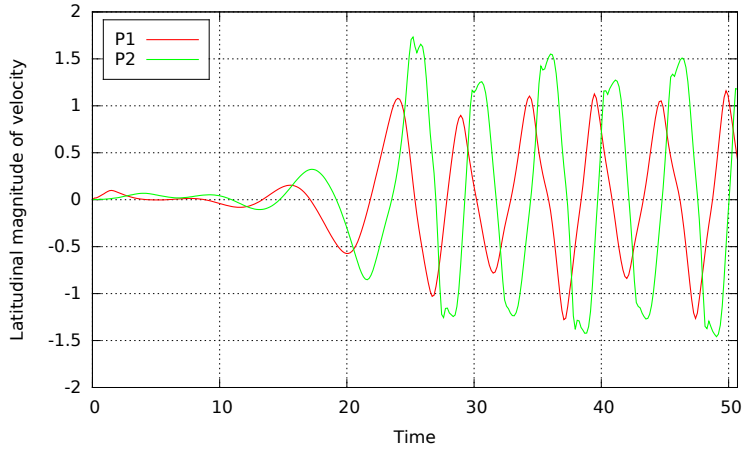
FIGURE 4.9: Transverse component of the vorticity for the flow at Reynolds number 150.

for Re = 150 in Fig. 4.10. The first plot, in Fig. 4.10(a), shows the signals during the transient time when the wake instability develops; the second plot, in Fig. 4.10(b), shows the signals after the periodic state has been reached. The spectra of these signal can be computed to extract their dimensionless fundamental frequency, or Strouhal number [104], defined by $St = \frac{fD}{U_m}$ where f denotes the shedding frequency. For our computation with Re = 150 we found $St = 0.144$, a value that agrees favorably with results reported in the literature [6, 36], therefore validating our simulations.

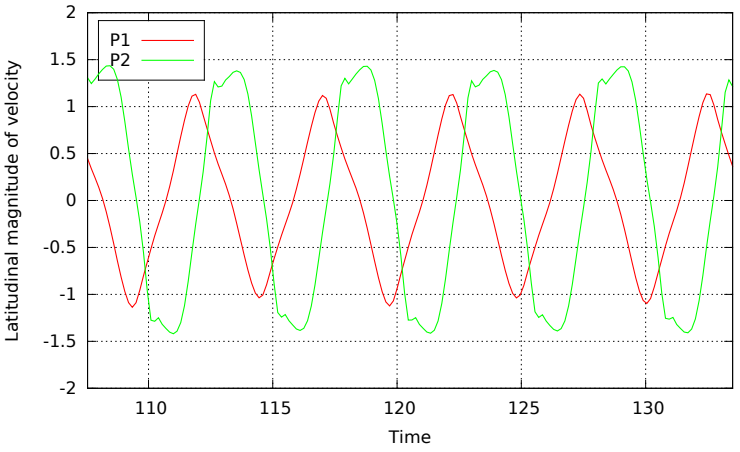
4.2.3 Performance analysis

Next, we look at the performance of the solver on this benchmark. As for the first benchmark, we compare the performance of two types of parallelization of the 3D Navier-Stokes solver. However, because of the obstacle and for the sake of accuracy, we can not perform the simulation for a too coarse mesh, so that only meshes are used to assess the performances. In addition, the configuration of the domain decomposition can significantly impact the number of iterations for the convergence of the iterative method, making difficult a fair comparison of the computational times when changing the configuration of the domain decomposition. As a result, we choose to report the performances of three parallelization types for a fixed domain decomposition configuration, and we provide no complete scalability results for this benchmark.

Table 4.2 reports the execution times (in second per iteration) for the two different parallelizations of the NS solver. As previously, we also report the acceleration percentages defined by Eq. 4.2. It is seen that the use of GPU does not decrease the execution time significantly. This is due to the fact that most of the execution time ($\sim 70\%$) is dedicated to solve the Poisson problem, a task completely performed on CPU. In these tests, only the tridiagonal solves (for the Helmholtz problem using ADI) are performed on GPUs, and these calculations do not represent a large workload. This percentage is not constant because for the second mesh, locally coarser, the number of unknowns per process in smaller than that for the first one. The fork-join operation of threads being



(a) Oscillations begin to form for P_1 and P_2.



(b) Stable oscillations for P_1 and P_2.

FIGURE 4.10: Transverse component of the velocity for observation points P_1 and P_2.

a constant overhead in both settings, the computational work shared by the threads is heavier and the final acceleration percentage is higher.

4.3 Conclusion of Chapter 4

In this chapter, the 3D Navier-Stokes solver developed in the thesis has been tested on two classical benchmarks in CFD, with the osbjective of validating the simulation code and assessing the performance of the parallelizations.

First, the separable problem of Taylor-Green vortices has been considered. This problem was solved using direct solvers for the Helmholtz and Poisson problem. GPU accelerators

Number of nodes	1	9
(number of subdomains along each direction)	$(1, 1, 1)$	$(3, 3, 1)$
(mesh size per node)	$(320 \times 240 \times 32)$	$(160 \times 64 \times 32)$
(total mesh size)	$(320 \times 240 \times 32)$	$(480 \times 192 \times 32)$
MPI+OpenMP+CUDA	0.98	0.21
MPI+OpenMP	1.07	0.225
GPU acceleration from MPI+OpenMP	8.4%	6.7%

TABLE 4.2: Time (s) for one NS iteration.

described in Chapter 3 could be used in this case, not only for tridiagonal solves but also for solving the Poisson equations. We observed several criteria in order to validate the numerical solutions. We also performed some computations for different types of parallelization and different mesh sizes. The results confirmed the benefit of using GPUs for large scale numerical simulations in the domain of fluid dynamics. Specifically, we obtained an acceleration of up to 36% by integrating GPUs in the Navier-Stokes solver.

Second, the flow around a square cylinder was considered. In this case, the domain includes one obstacle, making the problem non-separable and requiring an iterative method to solve the Poisson equation. A combination of SOR and multigrid methods was used to solve the Poisson problem (while the Helmholtz problem was solved using the ADI method on GPU). The simulations were first validated by comparing the Strouhal number of the flow with values reported in the literature. Finally, the investigation of the parallelization performance has shown that we cannot achieve significant improvement by using GPU for this problem, because most of the computational time is spent on CPU by solving the Poisson problem with the iterative method. Future work will investigate the use of GPUs in iterative methods.

Conclusion

This PhD manuscript described a parallel 3D Navier-Stokes solver that runs efficiently on different types of architectures and uses GPU accelerators. The solver has been enhanced thanks to the use of several levels of parallelism based on MPI, OpenMP and GPU programming. The improvement concerns the solution of the Helmholtz and Poisson problems that represent the main computational cost of the Navier-Stokes solver, including also the solution of tridiagonal systems using SIMD vectorization.

Our solver can be used with or without GPU accelerators, depending on the targeted architecture. It has been tested on various parallel architectures and has shown satisfactory scalability results. For instance on the Stampede system, we obtained a speedup of 24 using 32 multicore nodes (total 512 cores), and a speedup of 20 with 32 compute nodes when using also GPUs. We have also developed independent GPU solvers for Helmholtz and Poisson problems. Benchmarks on real applications enabled us to validate numerically the computed solutions by comparing them to reference results. We plan to integrate this solver in a future library for fluid dynamics simulations.

There are still some research directions that deserve further investigations.

This PhD thesis illustrated how it is possible to solve incompressible NS equations by taking advantage of heterogeneous parallel architectures. However the memory transfers between the CPU host and the GPU device can still be reduced in a future implementation. These data movements between CPU and GPU occur for instance in the following phases: First for every time step, when solving a tridiagonal system, we need to transfer the right-hand side vectors (that are different for each time step) to the GPU while the tridiagonal matrix is transferred at once in the initialization phase because the matrix is constant. The solution of tridiagonal systems are then sent back to the CPU for the next time iteration. Second, in the Poisson equation, before calling "dgemm" from MAGMA, we have to send the source matrix (matrix S in Section 3.2.2) to the GPU and receive the solution matrix ϕ from the GPU after the MAGMA call. In a next version, it would be preferable to overlap these memory transfers by computation (e.g., the computation of the temperature when the temperature is a variable). Another possibility is to use task

streaming which would allow us to transfer data by blocks and to start the computation before having the whole amount of data.

Moreover, the Navier-Stokes solver should be also adapted for the Intel Xeon Phi co-processors in order to take advantage of the large-size register for better performance in vectorizations. This coprocessor can also be considered as an accelerator by using the offload execution mode.

The solver could also benefit from closer links with external libraries such as Hypre that provides many iterative solvers. Indeed, when there are obstacles inside fluid flows (see Section 4.2), our solver is not applicable and the Hypre library can provide good performance on CPU architectures (see Section 2.5.3) in the solution of the Poisson equation (also in the Helmholtz-like equation as a replacement method for ADI). Our solver could also benefit from some external libraries for which there has been recent development on GPUs, such as the Paralution [5] library.

Finally our solver, which is developed for simulations of incompressible fluid flows, can also be adapted to address dilatable flows and low mach number flows. It can also address the case where Navier-Stokes equations are coupled with transport equations. Different numerical methods are used in such situations and they are not addressed by this PhD thesis.

Appendix A

Iterative Methods for Linear Systems

A.1 Iterative Methods

Direct methods for solving linear systems theoretically give the exact solution in a finite number of operations. Unfortunately, this is not always true in applications because of round-off errors. Contrary to direct methods, iterative methods consist in constructing a series of approximate solutions such that it converges to the exact solution of the system. The main advantage of an iterative method is that it is self-correcting. In this appendix, we present some iterative methods for solving a general linear system

$$Ax = b. \tag{A.1}$$

where $A \in \mathbb{R}^{n\times n}$ and $x, b \in \mathbb{R}^n$.

A.1.1 Bases of iterative methods

An iterative method for solving the linear system $Ax = b$ constructs a series of approximations x_i, $i = 0, 1, 2, \ldots$, which under certain conditions will converge to the exact solution x_e of the system $Ax_e = b$. To do so, it is necessary to choose a starting point x_0 and a rule that is iteratively applied to compute x_{i+1} from x_i.

A starting point x_0 is usually chosen as an approximation of x_e. Next, given $x_i, i \in \mathbb{N}$, the next element of the series is computed using a rule of the form

$$x_{i+1} = B_i x_i + C_i b\ , \quad i = 0, 1, 2, \ldots, \tag{A.2}$$

where $B_i, C_i \in \mathbb{R}^{n\times n}, i \in \mathbb{N}$. Different choices of B_i and C_i define different iterative methods.

To guarantee the convergence of an iterative method, several conditions must be satisfied. First of all, it has to satisfy that

$$B_i + C_i A = I_n,$$

for all $i \in \mathbb{N}$, or equivalently,

$$x_e = B_i x_e + C_i A x_e \ , \qquad i \in \mathbb{N}.$$

In other words, the exact solution x_e is a fixed point of the rule and the method can not diverge from the exact solution. Secondly, given a starting point $x_0 \neq x_e$, the rule must ensure that approximate solution x_i converges to x_e as i increases.

To satisfy the second condition, we must have

$$\lim_{i\to\infty} B_i B_{i-1} \dots B_0 = 0.$$

If we choose the stationary iterative method, which means that $B_i = B$ for all i, we must have

$$\rho(B) < 1, \tag{A.3}$$

where $\rho(B)$ is the spectral radius of B and $\rho(B) = \max_{i=1,\dots,n} |\lambda_i|$ where the λ are the eigenvalues of B.

We note that the convergence condition $\rho(B) < 1$ holds, for example, if $\|B\| < 1$ in any matrix norm. Moreover, the condition (A.3) guarantees the self-correcting property of iterative methods since convergence takes place independently of the choice of starting point x_0. Thus, if a round-off errors affect x_i during the $i-$th iteration, x_i can always be considered as a new starting point and the iterative method will further converge. As a result, the iterative methods are in general more robust than the direct methods.

Of course, an iterative procedure should be kept going until $x_i = x_e$. This is impractical and usually unnecessary. Therefore, a stopping (or convergence) criteria is used to stop the iterative procedure when a pre-specified condition is met. One of the most used stopping criteria is based on the change of the solution or residual vector along the iterations. Specifically, given a small $\varepsilon > 0$, the iterative procedure is stopped after the $i-$th iteration when $\|x_i - x_{i-1}\| \leq \varepsilon$, $\|r_i - r_{i-1}\| \leq \varepsilon$, or $\|r_i\| \leq \varepsilon$, where $r_i = Ax_i - b$ is the residual vector. A maximum number of iterations is also usually specified. If an iterative methods does not meet the stopping criteria before reaching the maximum

iteration number, it is considered to be inefficient. In this case, other methods should be considered or a preconditioner should be applied.

From the above general principles of iterative methods, several variants can be derived by choosing different matrices B and C (for stationary iterative methods).

A.1.2 Jacobi method

The Jacobi method is supported by the following observation. Suppose that A have nonzero diagonal elements. Then the diagonal part D of A is nonsingular and Eq. (A.1) can be rewritten as $Dx + (L + U)x = b$ where U and L denote the upper and lower triangular parts of A respectively. As a result,

$$x = D^{-1}[(-L - U)x + b]. \tag{A.4}$$

Replacing x on the left-hand side by x_{i+1} and x on the right-hand side by x_i leads to the Jacobi iterations:

$$x_{i+1} = -D^{-1}(L + U)x_i + D^{-1}b. \tag{A.5}$$

The intuition of the Jacobi method is very simple: given an approximation x^{old} of the solution, let us express the $k-$th component x_k of x as a function of the other components from the $k-$th equation and compute x_k^{new} given x^{old}:

$$x_k^{\text{new}} = \frac{1}{A_{kk}} \left(b_k - \sum_{j=1}^{n} A_{kj} x_j^{\text{old}} \right), \quad k = 1, \ldots, n. \tag{A.6}$$

The convergence condition for Jacobi method is $\rho(D^{-1}(L + U)) < 1$. This condition is satisfied for a relatively large class of matrices, including diagonally dominant matrices[1] and symmetric matrices[2] such that D, $L+D+U$, and $-L+D-U$ are all positive definite[3]. Although there are many variants of the basic principle of the Jacobi method to improve the convergence rate, the advantages of this method is a simple and fast implementation: the computation of a component of the new iterate, x_k^{new} is independent from its other component, making the Jacobi method embarrassingly parallel.

[1] Matrices A such that $\sum_{j=1, j\neq i}^{n} |A_{ij}| \leq |A_{ii}|$ for $i = 1, \ldots, n$.

[2] Matrices A such that $A_{ij} = A_{ji}$.

[3] A symmetric real matrix A is said to be positive definite if $z^T Az$ is positive for every non-zero column vector z.

A.1.3 Gauss-Seidel method

From the decomposition $A = D + L + U$, similar to the Jacobi method, the Gauss-Seidel method expresses Eq. (A.1) as

$$(L + D)x + Ux = b, \tag{A.7}$$

which implies

$$x = (L + D)^{-1} [-Ux + b] . \tag{A.8}$$

This gives us the iteration formula of the Gauss-Seidel method:

$$x_{i+1} = -(L + D)^{-1} U x_i + (L + D)^{-1} b, \tag{A.9}$$

or componentwise

$$x_k^{\text{new}} = \frac{1}{A_{kk}} \left(b_k - \sum_{j=1}^{k-1} A_{kj} x_j^{\text{new}} - \sum_{j=k}^{n} A_{kj} x_j^{\text{old}} \right), \quad k = 1, \ldots, n. \tag{A.10}$$

The main difference between the Gauss-Seidel and Jacobi methods lies in a more efficient use of Eq. (A.6). When computing the $k-$th component of the new approximate solution x_k^{new}, the first $k-1$ elements $x_1^{\text{new}}, \ldots, x_{k-1}^{\text{new}}$ are already known and are assumed to be more accurate than $x_1^{\text{old}}, \ldots, x_{k-1}^{\text{old}}$. Thus, it is possible to use these new values instead of the old ones and increase the converge rate. Moreover, using this strategy, the newly computed elements of x^{new} can directly overwrite the respective elements of x^{old}, with memory saving as a result. The Gauss-Seidel method's convergence condition is that $\rho((L+D)^{-1}U) < 1$. This condition stands for diagonally dominant and definite matrices.

A.1.4 Successive over-relaxation method

The successive over-relaxation (SOR) method is a further refinement of the Gauss-Seidel method. By adding and subtracting Dx_i in the Gauss-Seidel formula (A.9), we obtain

$$(D + L)x_{i+1} = b - (U + D)x_i + Dx_i, \tag{A.11}$$

or

$$x_{i+1} = x_i - D^{-1} [Lx_{i+1} + (D + U)x_i - b] = x_i - \Delta_i, \tag{A.12}$$

which expresses the next approximate solution with a correction Δ_i from x_i to x_{i+1}. It is then natural to question whether the iterations can converge faster if we "overly" correct x_i at each iteration (x_i is corrected by a multiple ω of Δ_i in each iteration). This

idea leads to the SOR formula:

$$x_{i+1} = x_i - \omega D^{-1} \left[L x_{i+1} + (D + U) x_i - b \right] \tag{A.13}$$

or in component form

$$x_k^{\text{new}} = \omega \frac{1}{A_{kk}} \left(b_k - \sum_{j=1}^{k-1} A_{kj} x_j^{\text{new}} - \sum_{j=k}^{n} A_{kj} x_j^{\text{old}} \right) + (1 - \omega) x_k^{\text{old}}. \tag{A.14}$$

The parameter $\omega > 0$ is called the (over)relaxation parameter and it can be shown that SOR can only converge for $\omega \in (0, 2)$ [57].

A well-selected ω can accelerate the convergence, as measured by the spectral radius of the corresponding iteration matrix B [42] (a lower spectral radius $\rho(B)$ means faster convergence).

One important result regarding the value of ω can be found in [105]. Let the matrix A be two-cyclic consistently ordered[4]. Then, if the Gauss-Seidel iteration matrix $B = -(L + D)^{-1} U$ has a spectral radius $\rho(B) < 1$, the optimal relaxation parameter ω in SOR is given by

$$\omega_{\text{opt}} = \frac{2}{1 + \sqrt{1 - \rho(B)}}, \tag{A.15}$$

and for this optimal value it holds $\rho(B; \omega_{\text{opt}}) = \omega_{\text{opt}} - 1$.

Using SOR with the optimal relaxation parameter significantly increases the rate of convergence compared to the Gauss-Seidel method. If ω_{opt} cannot be computed exactly, it is better to take ω slightly larger than its optimal value, rather than smaller.

A.2 Multigrid methods

Iterative solvers (Jacobi, Gauss-Seidel, SOR) when applied to the resolution of discretized partial differential equations, *e.g.* Helmholtz and Poisson equations, have a tendency to "stall", *i.e.* to fail in effectively reducing the residual after several iterations. The problem usually becomes more important when the spatial mesh is refined. In fact, standard solvers behave much better on coarse meshes. A close inspection of this behavior reveals that the convergence rate is a function of the error frequency, *i.e.* the fluctuation of the error from a grid point to another. The error distributed in high frequency modes has fast convergence rate and is quickly smoothed-out. However, the

[4] A matrix A is said to be two-cyclic consistently ordered if the eigenvalues of the matrix $M(\alpha) = \alpha D^{-1} L + \alpha^{-1} D^{-1} U$, $\alpha \neq 0$, are independent of α.

remaining error with low frequency fluctuations has a much lower convergence rate. As a result, most iterative methods exhibit a number of iterations to reach a converged solution that is linearly proportional to number of nodes (in one direction). This behavior can be rooted out to the fact that during the iterative process, the information travels over one grid point per iteration, while the convergence requires the information to travel back and forth through the mesh several times.

To remedy this issue, multigrid methods [16] have been proposed. These methods are based on the idea of using coarser grids, on which a low frequency error will be seen as a high frequency one, to improve the convergence rate of iterative methods. Multigrid methods aim at accelerating the convergence of a basic iterative methods, introducing a global correction which is accomplished by solving a coarse problem. This principle involves interpolation between coarser and finer grids [97]. There are many variants of multigrid algorithms, but the common features (of geometric ones) are a hierarchy (levels) of discretization (grids) [103], the restriction and prolongation operators to interpolate the residual and solutions between levels, and finally the smoothing procedure. In summary, the procedure of a multigrid method can be defined in three elementary steps:

- Smoothing: reducing high frequency errors by solving a residual equation. An iterative method is usually used for that purpose.
- Restriction: casting the residual error from a fine grid to a coarser one. For example, we keep the point-wise residual defined on the points shared by two successive grid levels.
- Prolongation: interpolating a correction computed on a coarse grid into a finer grid. A weighted mean is often used for the interpolation. For example, on the finer grid, we keep the value on the shared points and send half of the value to the neighbor points. In this way, the value on the points which are not defined on the finer grid is computed by the means of its two neighbors.

A typical multigrid approach is illustrated in Fig. A.1, for the case of a two dimensional domain. From the finer grid, having 8×8 cells, a hierarchy of 3 successive coarser grids is constructed by halving the number of cells in each spatial direction when going to a level to the next.

Different multigrid algorithms can be constructed from for a hierarchy of grids and multigrid operators. The most common multigrid algorithms are the V-cycle, W-cycle and F-cycle. In this thesis, we only consider V-cycle which is also illustrated in Fig. A.1.

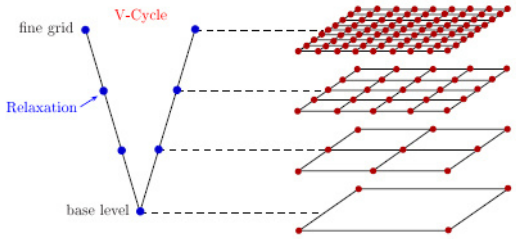


FIGURE A.1: V-cycle multigrid scheme

A V-cycle starts on the finer grid, supporting the current approximation of the solution, where several smoothing iterations are performed to reduce the high frequency components of the residual. The remaining (low frequency) residual is restricted to the next coarse grid, where it forms the right-hand side for the problem discretized on the current grid. Several smoothing iterations are performed to approximate the solution at the current level, before restricting the remaining residual to the next coarser grid. The sequence of restriction / smoothing steps is repeated till the coarser grid of the hierarchy is reached, terminating the first leg of the V-cycle. The second leg of the V-cycle proceeds from the coarser grid to the finer one. First, the solution obtained on the coarse grid is prolongated and added to the solution at the next grid level. The solution at the next level being updated, several smoothing iterations are performed to further reduce the residual, before before being prolongated and added to the next finer level solution. This sequence is repeated till the finer grid is reached and the V-cycle is completed. The convergence criteria is finally checked and if it is not satisfied a new V-cycle is performed. Clearly, an important parameter of the multigrid algorithm is the number of smoothing iterations performed at each level. The number of iterations must be large enough to effectively reduce the high-frequency components of the residual, but not too large to avoid ineffective iterations when the stagnation occurs.

Bibliography

[1] Basic Linear Algebra Subprograms Technical Forum Standard. *Int. J. of High Performance Computing Applications*, 16(1), 2002.

[2] Boost.Thread Web page, 2007. http://www.boost.org/doc/libs/release/libs/thread/.

[3] Ada reference manual. http://www.ada-auth.org/standards/12rm/html/RM-TTL.html, 2012.

[4] OpenMP application program interface. version 4.0. http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf, 2013.

[5] PARALUTION Web page. http://www.paralution.com, 2014.

[6] A. Agrawal, L. Djenidi, and R. A. Antonia. Investigation of flow around a pair of side-by-side square cylinders using the lattice boltzmann method. *Computers and Fluids*, 35:1093–1107, 2006.

[7] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. SIAM, 1999. Third edition.

[8] J. P. D. Angeli, A. M. P. Valli, N. C. J. Reis, and A. F. D. Souza. Finite difference simulations of the Navier-Stokes equations using parallel distributed computing. In *15th Symposium on Computer Architecture and High Performance Computing*, SBAC-PAD'03, pages 149–156, Washington, DC, USA, 2003. IEEE Computer Society.

[9] U. M. Ascher and L. R. Petzold. *Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1st edition, 1998.

[10] M. Baboulin, J. Dongarra, and S. Tomov. Some issues in dense linear algebra for multicore and special purpose architectures. In *9th International Workshop*

on State-of-the-Art in Scientific and Parallel Computing, volume 6126-6127 of *PARA'08*. Springer-Verlag, 2008.

[11] M. Baboulin, S. Donfack, J. Dongarra, L. Grigori, A. Rémy, and S. Tomov. A class of communication-avoiding algorithms for solving general dense linear systems on cpu/gpu parallel machines. In *International Conference on Computational Science (ICCS 2012)*, volume 9 of *Procedia Computer Science*, pages 17–26. Elsevier, 2012.

[12] G. K. Batchelor. *An Introduction to Fluid Dynamics*. Cambridge University Press, 2000.

[13] E. Bayraktar, O. Mierka, and S. Turek. Benchmark computations of 3d laminar flow around a cylinder with cfx, openfoam and featflow. *International Journal of Computational Science and Engineering*, 7(3):253–266, 2012.

[14] M. Braack and T. Richter. Solutions of 3D Navier-Stokes benchmark problems with adaptive finite elements. *Computers & Fluids*, 35(4):372–392, 2006.

[15] M. Brachet, D. I. Meiron, S. A. Orszag, B. G. Nickel, R. H. Morf, and U. Frisch. Small-scale structure of the taylor-green vortex. *Journal of Fluid Mechanics*, 130: 411–452, 1983.

[16] A. Brandt. Multi-level adaptive solutions to boundary-value problems. *Mathematics of Computation*, 31(138):333–390, 1977.

[17] D. L. Brown, R. Cortez, and M. L. Minion. Accurate projection methods for the incompressible Navier-Stokes equations. *Journal of Computational Physics*, 168: 464–499, 2001.

[18] D. R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley, 1997.

[19] C. Canuto, M. Hussaini, A. Quateroni, and T. Zang. *Spectral Methods, Fundamentals in Single Domains*. Scientific Computing. Springer, 2006.

[20] A. J. Chorin. A numerical method for solving incompressible viscous flow problems. *Journal of Computational Physics*, 2:12–26, 1967.

[21] A. J. Chorin. Numerical solution of the Navier-Stokes equations. *Mathematics of Computation*, 22(104):745–762, 1968.

[22] J. Cohen and M. Garland. Novel architectures: Solving computational problems with gpu computing. *Computing in Science Engineering*, 11(5):58–63, 2009.

[23] J. M. Cohen and M. J. Molemaker. A fast double precision CFD code using CUDA. In *21st International Conference on Parallel Computational Fluid Dynamics (ParCFD2009)*, 2009.

[24] G.-H. Cottet and P. Koumoutsakos. *Vortex Methods: Theory and Practice*. Cambridge University Press, 2000.

[25] V. Delean, P. Jolivet, and F. Nataf. An introduction to domain decomposition methods: algorithms, theory and parallel implementation. Lecture note, 2015.

[26] S. R. Eddy. Accelerated profile HMM searches. *PLoS Computational Biology*, 7 (10):e1002195, 2011.

[27] D. F. Elger, B. C. Williams, C. T. Crowe, and J. A. Roberson. *Engineering Fluid Mechanics*. Wiley, 10th edition, 2012.

[28] A. Ern and J.-L. Guermond. *Theory and Practice of Finite Elements*. Number 159 in Applied Mathematical Sciences. Springers, 2004.

[29] P. Estérie, M. Gaunard, J. Falcou, J.-T. Lapresté, and B. Rozoy. Boost.SIMD generic programming for portable SIMDization. In *21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 431–439, New York, NY, USA, 2012. ACM.

[30] M. Fatica and W.-K. Jeong. Accelerating MATLAB with CUDA. *the High Performance Embedded Computing (HPEC) Workshop*, 2007.

[31] J. H. Ferziger and M. Perić. *Computational Methods for Fluid Dynamics*. Springer, 3rd edition, 2002.

[32] M. P. I. Forum. *MPI : A Message-Passing Interface Standard*. Int. J. Supercomputer Applications and High Performance Computing, 1994.

[33] Y. Fraigneau. Principes de base des méthodes numériques utilisées dans le code SUNFLUIDH pour la simulation des écoulements incompressibles et à faible nombre de mach. Tech. Rep. 2013-09, LIMSI, 2013.

[34] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, 2004.

[35] G. J. Gassner and A. D. Beck. On the accuracy of high-order discretizations for underresolved turbulence simulations. *Theoretical & Computational Fluid Dynamics*, 27(3/4):221–237, 2013.

[36] B. Gera, K. S. Pavan, and R. K. Singh. CFD analysis of 2D unsteady flow around a square cylinder. *International Journal of applied engineering research*, 1(3): 602–610, 2010.

[37] V. Girault and P.-A. Raviart. Finite element approximation of the Navier-Stokes equations. *Lecture Notes in Mathematics, Berlin Springer Verlag*, 749, 1979.

[38] M. Griebel, T. Dornseifer, and T. Neunhoeffer. *Numerical Simulation in Fluid Dynamics : A Practical Introduction*. Society for Industrial and Applied Mathematics, 1998.

[39] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, 1999.

[40] J.-L. Guermond. Remarques sur les méthodes de projection pour l'approximation des équations de Navier-Stokes. *Numerische Mathematik*, 67(4):465–473, 1994.

[41] M. D. Gunzburger. *Finite Element Methods For Viscous Incompressible Flows: A Guide To Theory, Practice, And Algorithms*. Academy Press, 1989.

[42] A. Hadjidimos. Successive overrelaxation (SOR) and related methods. *Journal of Computational and Applied Mathematics*, 123(1-2):177–199, 2000.

[43] G. Hager and G. Wellein. *Introduction to High Performance Computing for Scientists and Engineers*. CRC Press, 2011.

[44] E. Hairer, S. P. Nørsett, and G. Wanner. *Solving ordinary differential equations I: Nonstiff problems*. Springer Verlag, 2nd edition, 1993.

[45] G. J. Haltiner. *Numerical Weather Prediction*. John Wiley & Sons, 1971.

[46] F. H. Harlow and J. E. Welch. Numerical calculation of time-dependent viscous incompressible flow of fluid with free surface. *Physics of Fluids*, 8(12):2182–2189, 1965.

[47] J. S. Hesthaven and T. Warburton. *Nodal Discontinuous Galerkin Methods: Algorithms, Analysis and Applications*. Number 54 in Texts in Applied Mathematics. Springers, 2008.

[48] W. Hundsdorfer. Partially implicit BDF2 blends for convectioni dominated flows. *SIAM Journal on Numerical Analysis*, 38(6):1763–1783, 2001.

[49] J. C. R. Hunt, A. A. Wray, and P. Moin. Eddie, Stream, and Convergence Zones in Turbulent Flows. Technical Report CTR-S88, Center for Turbulence Research, 1988.

[50] A. K. M. F. Hussain. Coherent structures and turbulence. *Journal of Fluid Mechanics*, 173(303-356), 1986.

[51] Intel. Math Kernel Library (MKL), 2014. https://software.intel.com/en-us/intel-mkl.

[52] R. I. Issa. Solution of the implicitly discretised fluid flow equations by operator-splitting. *Journal of Computational Physics*, 62:40–65, 1986.

[53] A. Jameson. Computational aerodynamics for aircraft design. *Science*, 245(361-371), 1989.

[54] J. Jim Douglas. Alternating direction methods for three space variables. *Numerische Mathematik*, 4(1):41–63, 1962.

[55] V. John. Higher order finite element methods and multigrid solvers ina benchmark problem for the 3D Navier-Stokes equations. *International Journal for Numerical Methods in Fluids*, 40(6):775–798, 2006.

[56] V. John. On the efficiency of linearization schemes and coupled multigrid methods in the simulation of a 3d flow around a cylinder. *International Journal for Numerical Methods in Fluids*, 50:845–862, 2006.

[57] W. Kahan. *Gauss-Seidel methods of solving large systems of linear equations*. PhD thesis, University of Toronto, Canada, 1958.

[58] J. Kim and P. Moin. Application of a fractional-step method to incompressible Navier-Stokes equations. *Journal of Computational Physics*, 59(2):308–323, 1985.

[59] Z. Kopal. *Tables of supersonic flow around cones*. Massachusetts Institute of Technology, 1947.

[60] I. M. Kozlov, K. V. Dobergo, and N. N. Gnesdilov. Application Of RES methods for computation of hydrodynamic flows by an example of 2D flow past a circular cylinder for Re=5-200. *International Journal of Heat and Mass Transfer*, 54:887–893, 2011.

[61] J. H. Krüger. *A GPU Framework for Interactive Simulation and Rendering of Fluid Effects*. Dissertation, Technische Universität München, München, 2006.

[62] H. Lamb. *Hydrodynamics*. Cambridge University Press, 1895.

[63] *HYPRE Reference Manual version 2.9.0b*. Lawrence Livermore National Laboratory, 2012.

[64] R. J. LeVeque. *Numerical Methods for Conservation Laws*. Lectures in Mathematics. ETH Zurich. Birkhäuser, 2nd edition, 1992.

[65] S. Li and W. K. Liu. Meshfree and particle methods and their applications. *Applied Mechanics Review*, 55:1–34, 2002.

[66] P.-L. Lions. On the Schwarz alternating method. I. In R. Glowinski, G. H. Golub, G. A. Meurant, and J. Périaux, editors, *First International Symposium on Domain Decomposition Methods for Partial Differential Equations*, pages 1–42. SIAM, 1988.

[67] P. Lynch. *The Emergence of Numerical Weather Prediction*. Cambridge University Press, 2006.

[68] B. Massey. *Mechanics of Fluids*. Taylor & Francis, 8 edition, 2006.

[69] *MPI: A Message-Passing Interface Standard Version 3.0*. Message Passing Interface Forum, 2012.

[70] P. Micikevicius. 3D finite difference computation on GPUs using CUDA. In *2nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, pages 79–84. ACM, 2009.

[71] K. Moreland. The ParaView tutorial, version 4.1. Technical Report SAND 2013-6883P, Sandia National Laboratories, 2013.

[72] C. A. d. Moura and C. S. Kubrusly. *The Courant-Friedrichs-Lewy (CFL) Condition: 80 Years After Its Discovery*. Birkhäuser Basel, 2012.

[73] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable Parallel Programming with CUDA. *ACM Queue*, 6(2):40–53, 2008.

[74] *NAG Parallel Library Manual*. The Numerical Algorithms Group (NAG), Oxford, United Kingdom, 2000.

[75] J. Oliger and A. Sundström. Theoretical and practical aspects of some initial boundary value problems in fluid dynamics. *Appl. Math.*, 35(3):419–446, 1978.

[76] K. Ouzzine. Étude d'une classe de schémas numériques de haute précision et mise en œuvre sur le cas du tourbillon de Taylor-Green. Master's thesis, Université Pierre & Marie Curie, 2013.

[77] R. L. Panton. *Incompressible Flow*. John Wiley & Sons, 4th edition, 2013.

[78] R. Peyret. *Spectral Methods for Incompressible Viscous Flow*. Springer Science & Bussiness Media, 2002.

[79] S. B. Pope. *Turbulent Flows*. Cambridge University Press, 2011.

[80] A. Quarteroni and A. Valli. *Domain Decomposition ethods for Partial Differential Equations*. Oxford University Press, 1999.

[81] B. N. Rajani, A. Kandasamy, and S. Majumdar. Numerical simulation of laminar flow past a circular cylinder. *Applied Mathematics Modeling*, 33:1228–1247, 2009.

[82] R. Rannacher. Finite element methods for the incompressible Navier-Stokes equations. In G. P. Galdi, J. G. Heywood, and R. Rannacher, editors, *Fundamental directions in mathematical fluid mechanics*, pages 191–293. Birkhäuser, 2000.

[83] S. M. Richardson. *Fluid Mechanics*. New York : Hemisphere Pub. Corp., 1989.

[84] B. Riviere. *Discontinuous Galerkin Methods For Solving Elliptic And Parabolic Equations: Theory and Implementation*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2008.

[85] Y. Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, 2nd edition, 2003.

[86] M. Schäfer and S. Turek. Benchmark computations of laminar flow around a cylinder. In E. Hirschel, editor, *Flow Simulation with High-Performance Computers II. DFG priority research program results 1993-1995*, number 52, pages 547–566. Vieweg, 1996.

[87] H. A. Schwarz. Über einen Grenzübergang durch alternierendes Verfahren. *Vierteljahrsschrift der Naturforschenden Gesellschaft in Zürich*, 18:272–286, 1870.

[88] J. Stam. Stable fluids. In *26th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH'99, pages 121–128, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co.

[89] J. Steinhoff and D. Underhill. Modification of the Euler equation for "vortivity confinement" - Application to the computation of interacting vortex rings. *Physics of Fluids*, 6(8):2738–2744, 1994.

[90] C. Taylor and P. Hood. A numerical solution of the Navier-Stokes equations using the finite element technique. *Computers & Fluids*, 1(1):73 – 100, 1973.

[91] J. C. 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*, 2009.

[92] L. H. Thomas. Elliptic problems in linear differential equations over a network. Technical report, Columbia University, 1949.

[93] S. Tomov, J. Dongarra, and M. Baboulin. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Computing*, 36(5/6):232–240, 2010.

[94] A. Toselli and O. Widlund. *Domain Decomposition Methodes - Algorithms and Theory*. Springer, 2005.

[95] N. Trevett. *OpenCL introduction*. Khronos Group, 2013.

[96] D. J. Tritton. *Physical fluid dynamics*. Oxford University Press, 2nd edition, 1987.

[97] U. Trottenberg, C. W. Oosterlee, and A. Schüller. *Multigrid*. Academic Press, 2001.

[98] R. A. Usmani. Inversion of a tridiagonal Jacobi matrix. *Linear Algebra and its Applications*, 212-213:413–414, 1994.

[99] H. von Helmholtz. Über integrale der hydrodynamischen gleichungen, welcher der wirbelbewegungen entsprechen. *Journal für die reine und angewandte Mathematik*, 55:25–55, 1858.

[100] Y. Wang, M. Baboulin, J. Dongarra, J. Falcou, Y. Fraigneau, and O. L. Maître. A parallel solver for incompressible fluid flows. *Procedia Computer Science*, 18: 439–448, 2013.

[101] Y. Wang, M. Baboulin, K. Rupp, O. Le Maître, and Y. Fraigneau. Solving 3D Incompressible Navier-Stokes Equations on Hybrid CPU/GPU Systems. In *Proceedings of the High Performance Computing Symposium*, HPC'14, pages 12:1–12:8, San Diego, CA, USA, 2014. Society for Computer Simulation International.

[102] J. F. Wendt, editor. *Computational Fluid Dynamics: An Introduction*. Springer, 3rd edition, 2010.

[103] P. Wesseling. *An introduction to Multigrid methods*. John Wiley & Sons, 1992.

[104] F. W. White. *Fluid Mechanics*. McGraw Hill, 4th edition, 1999.

[105] D. Young. Iterative methods for solving partial differential equations of elliptic type. *Transactions of the American Mathematical Society*, 76(1):92–111, 1954.

[106] D. F. Young, B. R. Munson, T. H. Okiishi, and W. W. Huebsch. *A brief introduction to fluid mechanics*. John Wiley & Sons, 5th edition, 2010.

[107] F. Zhang. *The Schur Complement and its Applications*. Springer, 2006.