POLITECNICO DI MILANO

SCUOLA DI INGEGNERIA INDUSTRIALE
Corso di Laurea Magistrale in Ingegneria Aeronautica

Scaling Performance of a DNS solver written in CPL

Candidato:
Mirco Meazzo
873477

Relatore:
Prof. Maurizio Quadrio

Anno Accademico 2018-2019

Dedicata alla mia famiglia,
a chi mi ha sempre sostenuto,
a chi è presente oggi
e a chi non c'è più

"This is Ground Control to Major Tom
You've really made the grade..."
David Bowie

Abstract

A numerical method for the direct numerical simulation of the incompressible Navier–Stokes equations in rectangular geometries is presented. The method implement the MPI Standard [14] to the engine developed by M. Quadrio and P. Luchini described in [42].

The method is based on Fourier expansions in the homogeneous directions and fourth-order accurate, compact finite-difference schemes over a variable-spacing mesh in the wall-normal direction.

Two different versions of the solver have been developed, based on the domain decomposition. In the first the domain is decomposed through 1D (*Slab*), while in the second version 2D (*Pencil*) decomposition is used. The performances of these versions, in terms of speedups and efficiencies, have been compared against each other at number of cores variation, processors architecture variation and mesh dimensions variation, highlighting the scalability benefits which derives from the latter decomposition method as soon as the mesh dimensions becomes important. The principal drawback of the code has been highlighted, together with the possible solution to improve the global efficiency of the code.

To manage the decomposition we rely on the APIs present in *fftMPI*, developed by Steve Plimpton at Sandia National Laboratories [55].

Key words: Navier–Stokes equations, direct numerical simulation, parallel computing, turbulence, 2D decomposition, pencil decomposition

Sommario

Nel presente lavoro viene presentato un metodo numerico per la simulazione numerica diretta delle equazioni di Navier-Stokes in geometria rettangolare. Tale solutore è il frutto dell'applicazione dello Standard MPI [14] ad un precedente lavoro dei professori M. Quadrio e P. Luchini, descritto in [42].

Il metodo impiega espansioni di Fourier nelle direzioni omogenee, mentre nella direzione normale alla parete la discretizzazione è affidata alle differenze finite compatte, le quali garantiscono un'accuratezza al quarto ordine.

Sono state realizzate due versioni distinte del solutore. Nella prima viene adottata una decomposizione del dominio 1D (*Slab*), mentre nella seconda si fa affidamento alla moderna soluzione della decomposizione 2D (*Pencil*). E' stato effettuato un confronto tra questi due solutori, in termini di speedups ed efficienze, al variare del numero di cores impiegati nella soluzione, dell'architettura del processore e della dimensione della mesh. Nel documento sono stati sottolineati i benefici derivanti dal secondo metodo di decomposizione al crescere della dimensione della mesh e le problematiche associate ad una parallelizzazione che fa affidamento al solo paradigma MPI, assieme ad una possibile soluzione.

La decomposizione degli array è affidata alla libreria esterna *fftMPI*, sviluppata da Steve Plimpton presso i Sandia National Laboratories [55].

Key words: equazioni di Navier–Stokes, simulazione numerica diretta, calcolo parallelo, turbolenza, decomposizione 2D, decomposizione pencil

Ringraziamenti

Ringrazio il Politecnico di Milano ed in particolare la figura del Professor Maurizio Quadrio, il quale mi ha dato questa grande opportunità di crescita. Le sarò sempre grato.

Dedico questo tesi alla persone che mi sono state accanto e mi hanno accompagnato durante questo percorso.

Ringrazio Giulia, Ludovica, Peppe, Eugenio, Michel, Francesco, Santiago ed Hermes per aver reso il "viaggio" divertente, e non solo istruttivo. A loro vanno i miei più sinceri auguri per il loro futuro, e spero vivamente di incontrarli ancora durante il mio percorso.

Ringrazio Darli, Renato, Orse ed Elisa per il loro sostegno in tutti questi anni, fatti di momenti bellissimi e difficoltà, nei quali loro non si sono mai tirati indietro, facendomi sempre sentire il loro supporto.

Ringrazio mia sorella Valeria per avermi sopportato e supportato, avere un fratello ingegnere deve essere complicato...

Ringrazio i miei genitori che con il loro lavoro ed impegno non mi hanno mai fatto mancare niente ed hanno reso possibile tutto questo, li ringrazio per non aver mai dubitato di me ed aver sempre assecondato le mie richieste, specie nei momenti più complicati.

Ringrazio la mia ragazza, Susanna, per essermi sempre stata accanto durante la stesura di questo elaborato ed avermi aiutato a superare i momenti più bui.

Infine ringrazio i miei nonni, i quali fin da piccolo, mi sono sempre stati accanto, dandomi tutto il loro amore e la loro fiducia. Ringrazio in particolare le mie nonne, Maddalena e Franca, per avermi insegnato più di chiunque altro a credere in me.

Estratto della tesi in lingua italiana

La turbolenza ha una grande importanza in molti processi fisici che coinvolgono i fluidi. Partendo dai fluidi interstellari, passando per i fenomeni atmosferici, la corrente attorno ad un aereo, il moto di un liquido in un tubo, lo strato limite e la scia attorno a corpi, fino a giungere alla corrente sanguigna all'interno del corpo umano, sono tutti esempi di moti caratterizzati dalla presenza di turbolenza.

Tuttavia, benché questo fenomeno sia così esteso, la sua comprensione è ancora ad oggi un mistero della fisica classica. L'assenza di una rigorosa definizione per tale fenomeno fornisce un chiaro campanello d'allarme sul livello del nostro sapere.

Lo studio della turbolenza è un ramo della fluidodinamica che ebbe inizio all'incirca centotrenta anni or sono, con l'esperimento di Osborne Reynolds. Tuttavia, l'impossibilità di trovare una soluzione analitica al problema, unita alle scarse competenze tecnologiche delle epoche precedenti, limitarono sensibilmente i margini di progresso.

L'odierno avvento dei calcolatori ha portato in dote la possibilità di risolvere numericamente, e in tempi ragionevoli, le equazioni. E' così nata la simulazione numerica diretta delle equazioni, abbreviata in DNS.

Questa tecnica risulta essere molto dispendiosa in termini computazionali, pertanto è necessario provvedere alla così detta parallelizzazione del codice. La stesura di un codice parallelizzato consente ad un programma di essere eseguito sotto forma di diverse istanze su differenti CPU, collegate tra loro attraverso una network, le quali, lavorando su una sezione del problema ciascuna, restituiscono la soluzione del problema completo.

Questa tesi mostra i processi che sono stati attuati al fine di modificare un precedente solutore DNS per renderlo parallelo. Tale realizzazione impiega il paradigma MPI, Message Passing Interface, uno standard mondiale per quanto concerne la realizzazione di algoritmi studiati per le odierne architetture a memoria distribuita, presenti nei supercomputer odierni. Benché i risultati, che verrano mostrati in seguito, siano di buon auspicio, sono necessarie ulteriori iterazioni sulla struttura del codice per poter ottenere un solutore all'apice tecnologico. La mancanza di una parallelizzazione intranodale lascia infatti lacune e margini di miglioramento. Questo lavoro pertanto deve essere visto come una solida base di partenza per un futuro solutore ad alta scalabilità, piuttosto che come un punto di arrivo.

La tesi si apre con una parte introduttiva sui flussi turbolenti volta a fornire i concetti generali di tale fenomeno. Tale capitolo inoltre fornisce al lettore le motivazioni dietro alla necessità di eseguire le simulazioni DNS e cenni di storia della stessa.

Nel secondo capitolo si fornisce una definizione analitica del dominio di interesse e delle equazioni che lo governano. In particolare viene mostrato come è possibile ridurre il problema da un sistema di tre equazioni differenziali alle derivate parziali ad un sistema di sole due equazioni analoghe attraverso l'uso delle equazioni della componente normale alla parete della vorticità e della velocità. In seguito viene dato ampio spazio alla formulazione discreta nel dominio di Fourier delle stesse, con particolare attenzione alla descrizione della

struttura delle “compact finite difference scheme”. Tale capitolo si conclude mostrando la struttura del codice implementato.

Il successivo capitolo descrive brevemente le librerie al quale ci siamo affidati per effettuare l'I/O e la trasposizione degli array. In questo capitolo è presente la descrizione dei cluster su cui abbiamo lavorato e testato il codice. Viene introdotta la struttura del codice di benchmark, mentre i risultati di tali benchmarks sono riportati nel capitolo successivo.

Tale capitolo indaga in modo approfondito il comportamento del nostro solutore, in termini di speedup ed efficienza, al variare di diversi parametri quali il numero di cores, l'architettura del processore e la strategia di decomposizione degli array, sfruttando una mesh di dimensioni costanti. Lo studio è ripetuto al variare della mesh per quattro diverse dimensioni del problema.

Nel quinto capitolo vengono mostrati i risultati di due simulazioni al variare del Re_τ. Le statistiche ottenute vengono commentate e confrontate con quelle di database del passato, sottolineando le caratteristiche del moto del fluido.

La tesi si conclude descrivendo possibili sviluppi futuri, volti a rendere più efficiente il solutore.

Contents

1 Introduction

In this chapter we present the main features of a turbulent flow, providing an historical background of the direct numerical simulation field applied to the channel flow problem.

1.1 Turbulent flows

Every smoker can observe the nature of turbulence one inch away from their nose. However a proper definition of turbulence is not yet given, due to the complexity of turbulence behaviour.
To use Prandtl words, who began an important lecture as follows:

"What I am about to say on the phenomena of turbulent flows is still far from conclusive. It concerns, rather, the first steps in a new path which I hope will be followed by many others. The researches on the problem of turbulence which have been carried on at Göttingen for about five years have unfortunately left the hope of thorough understanding of turbulent flow very small. The photographs and kinetographic pictures have shown us only how hopelessly complicated this flow is."

Nowadays we can entrust to computational units that allows us to be no longer *"hopeless"* as Prandtl was, although we are still far from having a solution, or at least a unique definition, of what the turbulence is. At the present time we define the turbulence as a flow regime, characterized by high Reynolds numbers and the presence of high level of diffusivity and irregularity, dissipation and three dimensional chaotic fluctuations in space and time[32].

1.2 General concepts

An elderly definition of turbulence was provided by Hinze [47], in 1959, and say:

"Turbulent fluid motion is an irregular condition of flow in which the various quantities show a random variation with time and space coordinates, so that statistically distinct average values can be discerned".

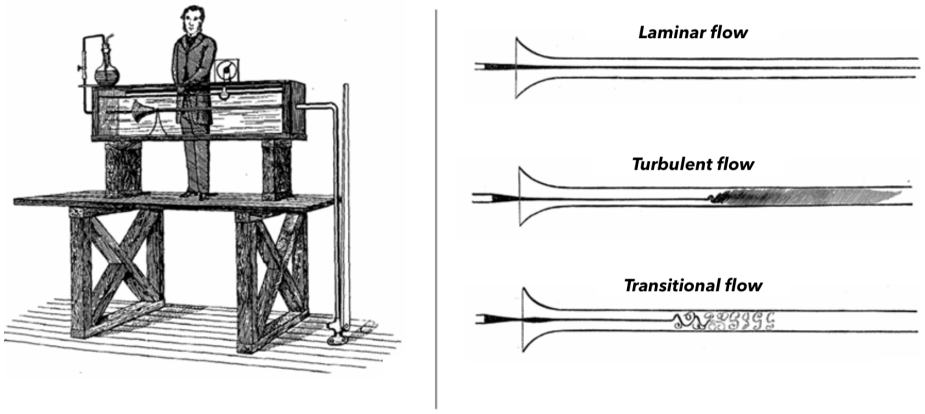

Figure 1.1: Sketch of the Reynolds experiment (*left*) and flow patterns (*right*)

The concept of *average* is the keyword that humanity has used to start digging into the turbulence mysteries. This kind of process, with its high sensitivity to the boundaries and initial conditions, can be defined as chaotic, so it can not be treated with a deterministic approach, therefore such randomness can be handled only by using a statistical approach. In fact turbulence recovers its deterministic side inside statistical analysis: the detailed properties of the signal show a non predictable behaviour, but its statistical properties are consistent [20]. At statistical level, turbulent phenomena become reproducible and subject to systematic study, providing a basis for theoretical description. Therefore, the three-dimensional time-dependent Navier-Stokes equations can be solved and then the solution is averaged in order to obtain the statistics [13]. Note, however, that irregular motion and chaotic advection do not guarantee turbulence. Small point vortices can advect themselves in a chaotic manner or particles can follow complex trajectories, yet this is not turbulence. The definition, in fact, requires diffusivity. If the flow pattern looks random but does not exhibit high mix of momentum, mass and heat, it is surely not turbulent. Diffusivity is the single most important feature of turbulence, as highlighted by the experiment of Osborne Reynolds [60], in 1883.
In its famous work Reynolds has defined a ratio among inertia forces and viscous forces:

$$Re = \frac{ul}{\nu} \tag{1.1}$$

with u that is the characteristic velocity of the fluid, l is the reference length of the scale and ν is the kinematic viscosity; able to predict the presence, or not, of the turbulence. He saw that when the inertia forces are huge the flow become unstable and the ink of its experiment started mixing with the surrounding water, as shown in the sketch of figure 1.1.
This first work has correlated the presence of different states of the flow, laminar, transitional and turbulent, and their relationship with the couple viscous term-nonlinear inertia term. Further observations revealed the presence

of three-dimensional eddies. Although we are still unable to determine their shapes, we have understood that they play a key role in the turbulence sustenance. Under the assumptions of incompressible flow, not subjected to external forces of volume or surface, the vorticity equation states

$$\frac{D\boldsymbol{\omega}}{Dt} = (\boldsymbol{\omega} \cdot \boldsymbol{\nabla})\boldsymbol{u} + \nu \boldsymbol{\nabla}^2 \boldsymbol{u}. \tag{1.2}$$

The central term of the equation, $(\boldsymbol{\omega} \cdot \boldsymbol{\nabla})\boldsymbol{u}$, is known as vortex-stretching term. The vortex stretching is at the core of the description of the turbulent energy cascade, from the large scales to the smallest scale, determined, as we will see, by the turbulence itself. For incompressible flow, due to volume conservation of fluid elements, the lengthening implies thinning of the fluid elements in the directions perpendicular to the stretching direction. This reduces the length scale of the associated vorticity. Finally, at the smallest scales the turbulence kinetic energy is dissipated into heat through the action of molecular viscosity [10].

1.3 The history of the direct numerical simulation

The direct numerical simulation (DNS) of the Navier-Stokes equations is a mathematical tool used to analyze turbulent flows since it allows to have an inner viewpoint in the transition and turbulence phenomena processes. It is part of the so called Computation Fluid Dynamics, or CFD, research field. Given the high computational cost of these simulations, DNS is not used to reproduce real-life flows, but as a research tool for flows with simple boundaries[48]. Despite of such kind of simulations, due to their limits, could seem useless, they assume relevant importance in the study of the turbulence, who, dominating the small scales, affect the behavior of the large scales, determining the raise of phenomenas such as flow separations, drag increases or losses of lift. These simulations rely on high accuracy computational methods and they do not employ turbulence models, hence they require an ever-increasing computational power, as we move towards engineering relevant Reynolds numbers. However we can identify an ultimate threshold for direct numerical simulation of the wall bounded flows, which, thanks to its scale separation, give the possibility to model the turbulence phenomena once for all. In [31] professor Jiménez set such threshold around $Re_\tau = 10000$.

The DNS history is recent, with the first milestone work carried out by Kim, Moin and Moser [36] in the 1987, using a $192 \times 129 \times 160$ grid of points distributed in a channel flow domain, in which they studied the homogenous isotropic turbulence using spectral modes. Follow this seminal work other authors proposed their simulations. Accurate DNS calculations of the turbulent channel flow, using spectral methods, have been carried out by Lyons *et al.* [43], Antonia *et al.* [4], Kasagi *et al.* [33], Rutledge and Sleicher [61] in the first nineties. In the 1999 Moser *et al.* [49] proposed their $Re_\tau = 590$ simulation, while to see the first channel flow simulation using finite differences we have to wait Abe *et al.* [1] in 2001.

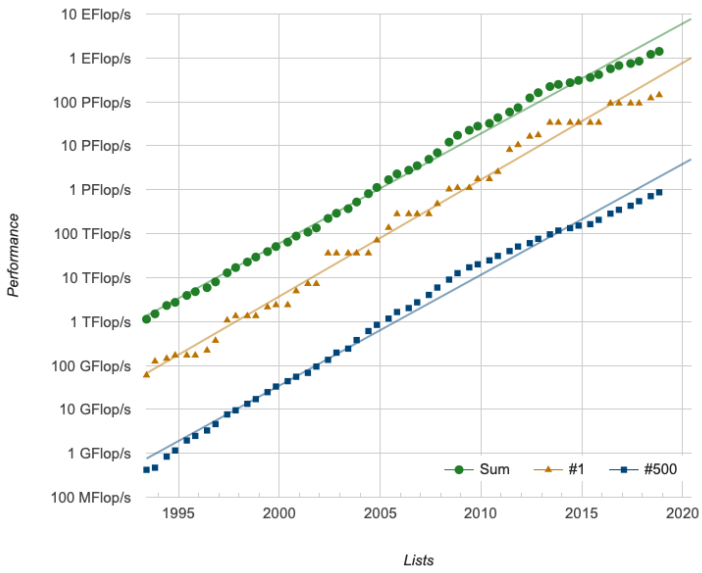

Figure 1.2: Supercomputers grown trend, courtesy of TOP500.org

Other works, from the first years of the twenty-first century, are for example Iwamoto *et al.* [28] ones, Del Alamo and Jiménez [2] and, the first simulation with Re_τ over a thousand, Del Alamo *et al.* [12] work, in 2004.
Between 2004 and 2007 were presented different works, alternating the finite differences techniques with the more established spectral methods approach. Tanahashi *et al.* [62], Iwamoto *et al.* [27], Hoyas and Jiménez [23], Hu *et al.* [24]are some examples.
More recent simulation have been carried out by Lozano-Duràn *et al.* [40], Lozano-Duràn and Jiménez [41], Vreman and Kuerten [68], Bernardini *et al.* [5] and the actually biggest simulation ever, with $Re_\tau = 5200$, by Lee and Moser [37].
The grows in Re_τ number is correlated with the grown in supercomputing performances on those years, as could be understood by looking at figures 1.2 and 1.3. However the Re_τ growth is not proportional with the computational power, as we can clearly see.
We are not far from the possibility to solve the biggest useful simulation, in [31] is reported that a theoretical 500 Pflops supercomputer could carry out the $Re_\tau = 10000$ simulation in a reasonable time. The cited document has been published in 2003, but, unfortunately, starting from 2013, the rate of growth of the supercomputers speed has halved with respect to the past decade, and this made the forecast embedded in such document wrong. However, with the current rate of growth, it is likely that we could carry out such simulation within the next three years.

In this thesis our goal was to provide an highly parallelized DNS solver, based on pseudo-spectral approach, able to carry out simulation on a wide number of processors in an efficient way.

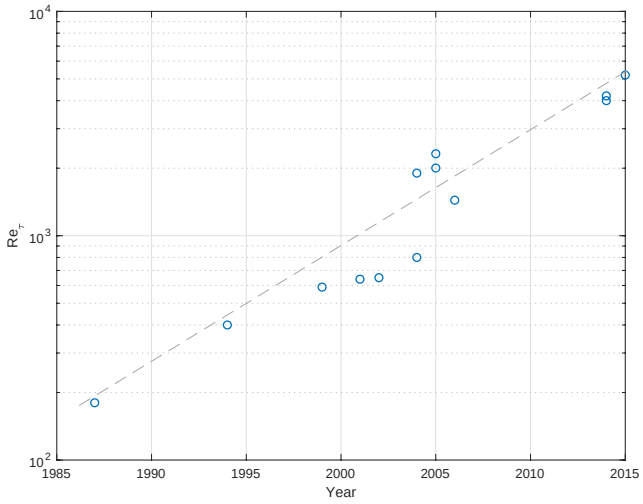

Figure 1.3: Re_τ growth trend

We were particularly interested in the possibility to generate flow statistics at high Re_τ values, in reasonable time, maintaining code efficiency above the 40%.

2

Parallel DNS of a Turbulent Channel Flow

This chapter introduce the governing equations of the turbulent channel flow. It will be shown how it is possible to reduce the computational cost, passing from three PDEs to 2 PDEs plus one linear system. Will be introduced our discretization, in terms of time advancement and spatial resolution.
At the end of the chapter we will provide some useful principles about the code parallelization.

2.1 Problem definition

Before moving to what has been done in this thesis I wish to briefly discuss the setup of our channel flow and the equations used to solve the problem.

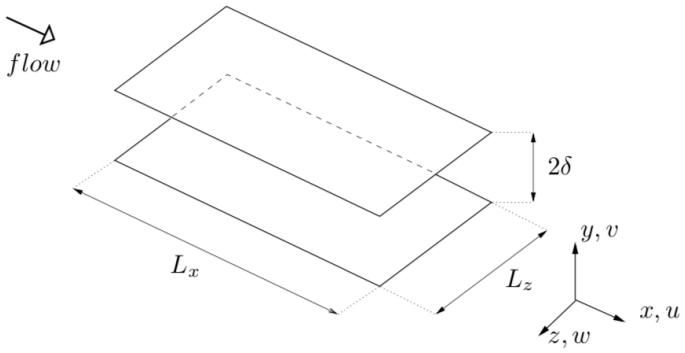

Figure 2.1: Domain of interest

We have the domain sketched in figure 2.1 where the x and z coordinates denote the streamwise and spanwise directions of the flow, while the y coordinate is the wall normal ones. Along these three dimensions we have u, v and w components of velocity.
The flow is assumed to be periodic in the streamwise and spanwise directions.
The lower wall is at position y_l and the upper wall at position y_u. The reference

length δ is taken to be one half of the channel height. Once an appropriate reference velocity is chosen, we can define the Reynolds number as:

$$Re = \frac{U\delta}{\nu}$$

where ν is the kinematic viscosity of the fluid.
According to our geometry and the assumption of incompressible flow, we can express the behavior of the flow through the mass conservation law:

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z} = 0 \tag{2.1}$$

and the Navier-Stokes equations, which in a dimensionless form states:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} + w\frac{\partial u}{\partial z} = -\frac{\partial p}{\partial x} + \frac{1}{Re}\nabla^2 u \tag{2.2a}$$

$$\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} + w\frac{\partial v}{\partial z} = -\frac{\partial p}{\partial y} + \frac{1}{Re}\nabla^2 v \tag{2.2b}$$

$$\frac{\partial w}{\partial t} + u\frac{\partial w}{\partial x} + v\frac{\partial w}{\partial y} + w\frac{\partial w}{\partial z} = -\frac{\partial p}{\partial z} + \frac{1}{Re}\nabla^2 w \tag{2.2c}$$

The differential problem is closed when initial conditions for all the fluid variables are specified, and suitable boundary conditions are chosen. At the walls the no-slip condition are imposed.

2.2 Governing equations

The numerical method does not rely on the equations (2.2), instead it solves the wall-normal velocity and the wall-normal vorticity equations, recovering, at the end of the the solution, the three velocity components.

2.2.1 Wall normal vorticity equation

Defining the wall-normal component of the vorticity vector as:

$$\eta = \frac{\partial u}{\partial z} - \frac{\partial w}{\partial x}$$

which, in Fourier space, holds:

$$\hat{\eta} = i\beta\hat{u} - i\alpha\hat{w}$$

with the hat indicating Fourier-transformed quantities, i the imaginary part, α and β the streamwise and spanwise wave numbers; allows to write a one-dimensional second-order evolutive equation for $\hat{\eta}$ which does not involve pressure, as proposed in [36].
Taking the y-component of the curl of the momentum equation we obtain:

$$\frac{\partial \hat{\eta}}{\partial t} = \frac{1}{Re}\big(D_2(\hat{\eta}) - k^2\hat{\eta}\big) + i\beta\widehat{HU} - i\alpha\widehat{HW} \tag{2.3}$$

where D_2 is the second derivative in the wall-normal direction, k^2 is the sum of α and β, and the nonlinear terms are defined as:

$$\widehat{HU} = i\alpha\widehat{uu} + D_1\widehat{uv} + i\beta\widehat{uw} \tag{2.4a}$$

$$\widehat{HV} = i\alpha\widehat{uv} + D_1\widehat{vv} + i\beta\widehat{vw} \tag{2.4b}$$

$$\widehat{HW} = i\alpha\widehat{uw} + D_1\widehat{vw} + i\beta\widehat{ww}. \tag{2.4c}$$

To solve the equation (2.3) we must set suitable initial conditions for $\hat{\eta}$. Such initial conditions are computed using the initial velocity field and the definition of η itself. Turning such conditions into frequency domain is straightforward and satisfy the periodic boundary conditions. Finally, the *no-slip* condition for velocity vector enforce the condition at the walls, which, simply, translate in $\hat{\eta} = 0$ at $y = y_l$ and $y = y_u$.

2.2.2 Wall normal velocity equation

An equation for the wall-normal velocity component $\hat{v}$, which does not involve pressure, is derived in [36], by summing the equation (2.2a) derived two times w.r.t. x and y, and (2.2c) derived two times w.r.t. y and z, then subtracting (2.2b) derived w.r.t. x and x and substracting once again after derivation w.r.t. z and z. Further simplifications are invoked through the equation (2.1), which lead to the following fourth-order evolutive equation for $\hat{v}$, which is the so called wall-normal velocity equation:

$$\frac{\partial}{\partial t}\big(D_2(\hat{v}) - k^2\hat{v}\big) = \frac{1}{Re}\big(D_4(\hat{v}) - 2k^2 D_2(\hat{v}) + k^4\hat{v}\big) - k^2\widehat{HV} - D_1\big(i\alpha\widehat{HU} + i\beta\widehat{HW}\big). \tag{2.5}$$

To solve such equation we have to enforce initial conditions on $\hat{v}$.
According to Fourier expansions, the periodic boundary conditions in the homogeneous directions are automatically satisfied, whereas the no-slip condition for the velocity vector immediately translates in $\hat{v} = 0$ to be imposed at the two walls.
The two remaining conditions for the fourth-order equation (2.5) comes from the continuity equation (2.1), written at the vertical edges of the domain, $y = y_l$ and $y = y_u$.

2.2.3 Velocity components in the homogeneous directions and mean flow

As reported before, once the two preceding equations are solved, we can use them to recover the velocity components in the homogeneous directions. Assuming the non-linear terms (2.4) known, as is the case when such terms are treated explicitly in the time discretization, the equations (2.3) and (2.5) become uncoupled and, after proper time discretization, can be solved for advancing the solution by one time step, provided the nonlinear terms (2.4) and their spatial derivatives can be calculated. To this aim, one needs to know how to compute $\hat{u}$ and $\hat{w}$ at a given time starting with the knowledge of $\hat{v}$ and $\hat{\eta}$. By using the definition (2.3) of $\hat{\eta}$ and the continuity equation (2.1) written in Fourier space, a 2×2 algebraic system can be written for the unknowns $\hat{u}$ and $\hat{w}$; its analytical solution read:

$$\begin{cases} \hat{u} &= \dfrac{1}{k^2}(i\alpha D_1(\hat{v}) - i\beta\hat{\eta}) \\ \\ \hat{w} &= \dfrac{1}{k^2}(i\alpha\hat{\eta} + i\beta D_1(\hat{v})) \end{cases} \tag{2.6}$$

For $k^2 = 0$ the system of equation (2.6) is singular.
The present method therefore enjoys its highest computational efficiency only when Fourier discretization is used in the homogeneous directions.
Since the previous system of equations (2.6) has been obtained starting from equations (2.3) and (2.5) the solutions are sensible to homogeneous spatial derivatives through the wave numbers.
Let us introduce a plane-average operator defined as:

$$\tilde{f} = \frac{1}{L_x}\frac{1}{L_z}\int_0^{L_x}\int_0^{L_z} f\,dx\,dz. \tag{2.7}$$

If we apply such operator to our velocity components vector $\mathbf{V}$, it turns out that $\mathbf{V}(x, y, z, t) = \widetilde{\mathbf{V}}(y, t)$. According to this, our velocity components are function of time and wall-normal coordinate only. In Fourier domain this behavior is denoted by the absence of the wave numbers, so for $k^2 = 0$.
In agreement with our reference system, where the x axis is aligned with the mean flow, the temporal average of $\tilde{u}$ will denote the mean velocity profile, whereas the temporal average of $\tilde{w} = 0$ throughout the channel.
Anyway $\tilde{w}$ can be different from zero at different time and distance from the wall.
Finally, applying the plane-average operator to the components of the momentum equation let us compute the $\tilde{u}$ and $\tilde{w}$:

$$\frac{\partial\tilde{u}}{\partial t} = \frac{1}{Re}D_2(\tilde{u}) - D_1(\widetilde{uv}) + f_x \tag{2.8a}$$

$$\frac{\partial\tilde{w}}{\partial t} = \frac{1}{Re}D_2(\tilde{w}) - D_1(\widetilde{vw}) + f_z \tag{2.8b}$$

In these expressions, f_x and f_z are the forcing terms needed to force the flow through the channel against the viscous resistence of the fluid. For the streamwise direction, f_x can be an imposed mean pressure gradient, and in the simulation the flow rate through the channel will oscillate in time around its mean value. f_x can also be a time-dependent spatially uniform pressure gradient, that has to be chosen in such a way that the flow rate remains constant in time. The same distinction applies to the spanwise forcing term f_z: in this case however the imposed mean pressure gradient or the imposed mean flow rate is zero, while the other quantity will be zero only after time average.
What has been shown in this chapter is intended just to be a brief discussion. Further informations are available in [42, pp. 1–3].

2.3 Spatial discretization along homogeneous directions

Our solver is based on a Fourier approach. Among the advantages of such approach we face the possibility to expansion of the unknown functions in terms

of truncated Fourier series in the homogeneous directions. For example the wall-normal component v of the velocity vector is represented as:

$$v(x,y,z,t) = \sum_{h=-nx/2}^{+nx/2} \sum_{l=-nz/2}^{+nz/2} \hat{v}_{hl}(y,t) e^{i\alpha x} e^{i\beta z} \tag{2.9}$$

where:

$$\alpha = \frac{2\pi h}{L_x} = \alpha_0 h; \quad \beta = \frac{2\pi l}{L_z} = \beta_0 l \tag{2.10}$$

h and l are integer indexes corresponding to the streamwise and spanwise direction respectively, and α_0 and β_0 are the fundamental wavenumbers in these directions, defined in terms of the streamwise and spanwise lengths $L_x = 2\pi/\alpha_0$ and $L_z = 2\pi/\beta_0$ of the computational domain. The computational parameters given by the streamwise and spanwise lenght of the computational domain, L_x and L_z , and the truncation of the series, nx and nz, must be chosen so as to miminize computational errors. For further details regarding the proper choice of a value of L_x see [59].
The convolutions required to solve the equations 2.3 and 2.5 are computationally expensive if carried out in the frequency domain. The same evaluation can be performed efficiently by first transforming the three Fourier components of velocity back in physical space, multiplying them in all six possible pair combinations and eventually retransforming the results into the Fourier space. Fast Fourier Transform algorithms are used to move from Fourier to physical space and viceversa. The aliasing error is removed by expanding the number of modes by a factor of at least 3/2 before the inverse Fourier transforms, to avoid the introduction of spurious energy from the high-frequency into the low-frequency modes during the calculation [42].

2.4 Finite difference scheme

The discretization of the wall-normal derivatives D_1, D_2 and D_4, required for the numerical solution of the present problem, is performed through finite difference (FD) compact schemes [38] with fourth-order accuracy over a computational molecule composed by five arbitrarily spaced grid points. We indicate here with $d_1^j(i), i = -2, \ldots, 2$ the five coefficients discretizing the exact operator D_1 over five adjacent grid points centered at y_j :

$$D_1\big(f(y)\big)|_{y=y_j} = \sum_{i=-2}^{2} d_1^j(i) f(y_{j+i}) \tag{2.11}$$

The basic idea of compact schemes can be most easily understood by thinking of a standard FD formula in Fourier space as a polynomial interpolation of a trascendent function, with the degree of the polynomial corresponding to the formal order of accuracy of the FD formula. Compact schemes improve the interpolation by replacing the polynomial with a ratio of two polynomials, i.e. with a rational function. This obviously increases the number of available coefficients, and moreover gives control over the behavior at infinity (in frequency space) of the interpolant, whereas a polynomial necessarily diverges. This allows a compact FD formula to approximate a differential operator in a

wider frequency range, thus achieving resolution properties similar to those of spectral schemes [38]. Compact schemes are also known as implicit finite-differences schemes, because they typically require the inversion of a linear system for the actual calculation of a derivative [44][38]. Here we are able to use compact, fourth-order accurate schemes at the cost of explicit schemes, owing to the absence of the third-derivative operator from the equations of motion. Thanks to this property, it is possible to find rational function approximations for the required three FD operators, where the denominator of the function is always the same, as highlighted first in the original Gauss-Jackson-Noumerov compact formulation exploited in his seminal work by Thomas [66], concerning the numerical solution of the Orr-Sommerfeld equations.
To illustrate Thomas' method, let us consider an 4th-order one-dimensional ordinary differential equation, linear for simplicity, in the form:

$$D_4(a_4 f) + D_2(a_2 f) + D_1(a_1 f) + a_0 f = g, \tag{2.12}$$

Where the coefficients $a_i = a_i(y)$ are arbitrary functions of the independent variable y, and $g = g(y)$ is a known RHS. Let us moreover suppose that a differential operator, for example D_4, is approximated in frequency space as the ratio of two polynomials, say $\mathcal{D}_4$ and $\mathcal{D}_0$. Polynomials like $\mathcal{D}_4$ and $\mathcal{D}_0$ have their counterpart in physical space, and d_4 and d_0 are the corresponding FD operators. The key point is to impose that all the differential operators appearing in the example equation 2.12 admit a representation such as the preceding one, in which the polynomial $\mathcal{D}_0$ at the denominator remains the same.
Equation 2.12 can thus be recast in the new, discretized form:

$$d_4(a_4 f) + d_2(a_3 f) + d_1(a_1 f) + d_0(a_0 f) = d_0 g, \tag{2.13}$$

and this allows us to use explicit FD formulas, provided the operator d_0 is applied to the right-hand-side of our equations. The overhead related to the use of implicit finite difference schemes disappears, while the advantage of using high-accuracy compact schemes is retained [42].

2.4.1 Compute of the finite difference coefficients

The actual computation of the coefficients d_0, d_1, d_2 and d_4 to obtain a formal accuracy of the 4^{th} order descends from the requirement that the error of the discrete operator d_4/d_0 decreases with the step size according to a power law with the desired exponent -4. In practice, following a standard procedure in the theory of Padé approximants [57], this can be enforced by choosing a set t_m of polynomials of y of increasing degree:

$$t_m(y) = 1, y, y^2, \ldots, y^m, \tag{2.14}$$

by analytically calculating their derivatives $D_4(t_m)$, and by imposing that the discrete equation:

$$d_4(t_m) - d_0(D_4(t_m)) = 0 \tag{2.15}$$

is verified for the nine polynomials from $m = 0$ up to $m = 8$. Our computational stencil contains 5 grid points, so that the unknown coefficients d_0

and d_4 are 10. There is however a normalization condition, and we can write the equations in a form where for example:

$$\sum_{i=-2}^{2} d_0(i) = 1. \tag{2.16}$$

The other 9 conditions are given by equation 2.15 evaluated for $m = 0, 1, \ldots 8$. We thus can set up, for each distance from the wall, a 10×10 linear system which can be easily solved for the unknown coefficients. The coefficients of the derivatives of lesser degree are derived from analogous relations, leading to two 5×5 linear systems once the d_0's are known. An additional further simplification is possible. Since the polynomials 2.14 have vanishing D_4 for $m<4$, thanks to the normalization condition 2.16 the 10×10 system can be split into two 5×5 subsystems, separately yielding the coefficients d_0 and d_4. Due to the turbulence anisotropy, the use of a mesh with variable size in the wall-normal direction is advantageous. The procedure outlined above must then be performed numerically at each y_j station, but only at the very beginning of the computations. The computer-based solution of the systems requires a negligible computing time.
We end up with FD operators which are altogether fourth-order accurate; the sole operator D_4 is discretized at sixth-order accuracy. As suggested in [36] and [44], the use of all the degrees of freedom for achieving the highest formal accuracy might not always be the optimal choice. In [42] Quadrio and Luchini attempted to discretize D_4 at fourth-order accuracy only, spending the remaining degree of freedom to improve the spectral characteristics of all the FD operators at the same time. Their search has shown however that no significant advantage can be achieved: the maximum of the errors can be reduced only very slightly, and, more important, this reduction does not carry over to the entire frequency range.
The boundaries obviously require non-standard schemes to be designed to properly compute derivatives at the wall. For the boundary points we use non-centered schemes, whose coefficients are computed following the same approach as the interior points, thus preserving by construction the formal accuracy of the method. Nevertheless the numerical error contributed by the boundary presumably carries a higher weight than interior points, albeit mitigated by the non-uniform discretization [42].

2.5 Time discretization

Time integration of the equations is performed by a partially-implicit method. The use of a partially-implicit scheme is a common approach in DNS [36]: the explicit part of the equations can benefit from a higher-accuracy scheme, while the stability-limiting viscous part is subjected to an implicit time advancement, thus relieving the stability constraint on the time-step size Δt.
Our preferred choice, following [42][36] is to use an explicit third-order, low-storage Runge-Kutta method for the integration of the explicit part of the equations, and an implicit second-order Crank-Nicolson scheme is used for the implicit part. This scheme has been anyway embedded in a modular coding implementation that allows us to change the time-advancement scheme very

easily without otherwise affecting the structure of the code. In fact, we have a few other time-advancement schemes built into the code for testing purposes.
Here we present the time-discretized version of equations 2.3 and 2.5 for a generic wavenumber pair and a generic two-levels scheme for the explicitly-integrated part coupled with the implicit Crank-Nicolson scheme:

$$\frac{\lambda}{\Delta t}\hat{\eta}_{hl}^{n+1} - \frac{1}{Re}\left[D_2(\hat{\eta}_{hl}^{n+1}) - k^2\hat{\eta}_{hl}^{n+1}\right] = \frac{\lambda}{\Delta t}\hat{\eta}_{hl}^{n} + \frac{1}{Re}\left[D_2(\hat{\eta}_{hl}^{n}) - k^2\hat{\eta}_{hl}^{n}\right] + \theta\left(i\beta_0 l\widehat{HU}_{hl} - i\alpha_0 h\widehat{HW}_{hl}\right)^n + \xi\left(i\beta_0 l\widehat{HU}_{hl} - i\alpha_0 h\widehat{HW}_{hl}\right)^{n-1} \quad (2.17)$$

$$\frac{\lambda}{\Delta t}\left(D_2(\hat{v}_{hl}^{n+1}) - k^2\hat{v}_{hl}^{n+1}\right) - \frac{1}{Re}\left[D_4(\hat{v}_{hl}^{n+1}) - 2k^2D_2(\hat{v}_{hl}^{n+1}) + k^4\hat{v}_{hl}^{n+1}\right] = \frac{\lambda}{\Delta t}\left(D_2(\hat{v}_{hl}^{n}) - k^2\hat{v}_{hl}^{n}\right) + \frac{1}{Re}\left[D_4(\hat{v}_{hl}^{n}) - 2k^2D_2(\hat{v}_{hl}^{n}) + k^4\hat{v}_{hl}^{n}\right] + \theta\left(-k^2\widehat{HV}_{hl} - D_1\left(i\alpha_0 h\widehat{HU}_{hl} + i\beta_0 l\widehat{HW}_{hl}\right)\right)^n + \xi\left(-k^2\widehat{HV}_{hl} - D_1\left(i\alpha_0 h\widehat{HU}_{hl} + i\beta_0 l\widehat{HW}_{hl}\right)\right)^{n-1} \quad (2.18)$$

The three coefficients λ, θ and ξ define a particular time-advancement scheme. For the simplest case of a 2^{nd}-order Adams-Bashfort, for example, we have $\lambda = 2$, $\theta = 3$ and $\xi = -1$.
The procedure to solve these discrete equations is made by two distinct steps. In the first step, the RHSs corresponding to the explicitly-integrated part have to be assembled. In the representation 2.9, at a given time the Fourier coefficients of the variables are represented at different y positions; hence the velocity products can be computed through inverse/direct FFT in wall-parallel planes. Their spatial derivatives are then computed: spectral accuracy can be achieved for wall-parallel derivatives, whereas the finite-differences compact schemes described in 2.4 are used in the wall-normal direction. These spatial derivatives are eventually combined with values of the RHS at previous time levels. The whole y range from one wall to the other must be considered.
The second step involves, for each α, β pair, the solution of a set of two ODEs, derived from the implicitly integrated viscous terms, for which the RHS is now known. A finite-differences discretization of the wall-normal differential operators produces two real banded matrices, in particular pentadiagonal matrices when a 5-point stencil is used. The solution of the resulting two linear systems gives $\hat{\eta}_{hl}^{n+1}$ and $\hat{v}_{hl}^{n+1}$, and then the planar velocity components $\hat{u}_{hl}^{n+1}$ and $\hat{w}_{hl}^{n+1}$ can be computed by solving system 2.6 for each wavenumber pair. For each α, β pair, the solution of the two ODEs requires the simultaneous knowledge of their RHS in all y positions. The whole α, β space must be considered. In the $\alpha - \beta - y$ space the first step of this procedure proceeds per wall-parallel planes, while the second one proceeds per wall-normal lines [42].

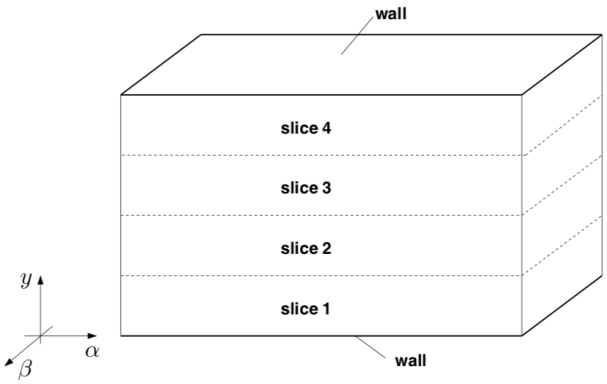

Figure 2.2: Original domain decomposition in case of 4 processors

2.6 Domain decompositions

The engine of Quadrio and Luchini described into [42] works per *y-slabs*, as shown in figure 2.2, allowing to perform convolutions and Fourier transformations locally on each processor, avoiding the cost of non-local transposition for the velocity array and the non-linear terms ones. Such implementation, denominated pipelined-linear-system (PLS), lead to a minimum in communications, in fact this approach require to send and receive only the values stored in the two upper and lower boundary cells of the slice, in order to provide and gather the data required by the fourth-order finite difference scheme, used along the y-direction.
Using the PLS approach the bytes exchanged in a three step Range-Kutta method are:

$$D_t = 3 \times 8 \times (p-1)\frac{3}{2} \times \frac{nx}{p}\frac{nz}{p} \times ny \times 18 \tag{2.19}$$

where:

3 takes into account the number of time steps;

8 for counting the bytes;

$(\mathbf{p-1})$ is the number of nodes across which the exchange take place;

$\frac{\mathbf{3}}{\mathbf{2}}$ corresponds to the expansion in horizontal modes required by the dealiasing process;

$\frac{\mathbf{nx}}{\mathbf{p}} \times \frac{\mathbf{nz}}{\mathbf{p}}$ is the grid portion, for each plane, to exchange with the others nodes;

18 due to the 3 velocities plus the 6 products to be exchanged twice, before and after the FFT;

ny takes into account the number of planes to be exchanged.

Further details about the PLS communications are available in [42, chapter 4.2]. Although efficient for small processors grid, the performances of this approach falls quickly whether the processors number becomes comparable with *ny*. Furthermore the communications cost limit the number of parallel process to be just a fraction of the *ny* extension.

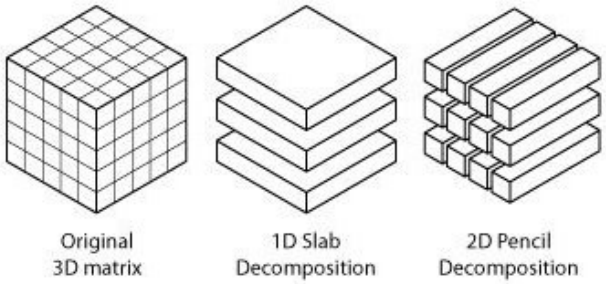

Figure 2.3: Kinds of decomposition

To avoid such limitations and increase the number of parallel processes we decided to move from PLS approach to something different. We have identified two possible solutions:

employ **slab** decomposition along x or z axis;

employ **pencil** decomposition.

Both implementations require extensive use of the MPI paradigm and, possibly, a library to handle such decompositions.
We opted to employ OpenMPI [64] for what concern the MPI paradigm, in particular we entrusted to the well established MPI standard 3.1 [17], using OpenMPI release 3.1.3. The ideas behind the choice of such library rely on the fact that OpenMPI is released behind BSD license, it is designed to group different MPI implementation, avoiding fragmentation and forking problem [65] and, although less optimized on proprietary fabric such as Intel Omni-Path fabric [50][26], it is likely the most widespread message passing interface package.
For what concern the decomposition we entrusted in a new library, released by Steven Plimpton from the Sandia National Laboratories, called *fftMPI* [55]. Such library leans on the MPI implementation, providing the proper cartesian communicator needed to perform the transposition of the arrays. Next to the communicator, this brand new library provide some useful tweak like permutations or the possibility to select the desired communication mode, which, compared with the FFTW-MPI Transpose [18][19] features, provide a wider personalization and reduce the work needed for the subsequent operations, such as the FFTs.
Finally is important to highlight that *fftMPI* can handle both 1D and 2D decomposition, unlike FFTW-MPI. Others similar libraries are P3DFFT [53] from professor Dmitry Pekurovsky (UCSD), PFFT from professor Michael Pippig [54] (Technische Universitat Chemnitz) and 2DECOMP&FFT [39] by Ning Li.

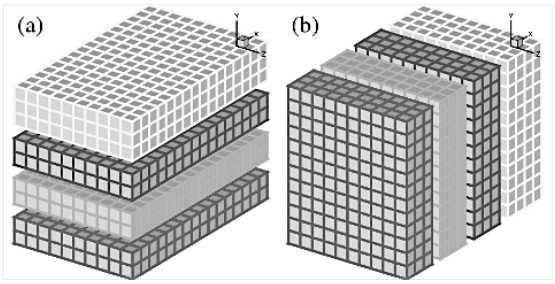

Figure 2.4: Example of 1D decomposition: a) *PLS*, b) *x-slabs*

2.6.1 1D decomposition

Objective of the decomposition is spread the computational cost across as much as tasks possible. This criteria goes against our needs to have all the dimensions local on chip, in order to carry out the 2D FFT, along xz plane, and to resolve the finite difference scheme, along y direction. A reasonable tradeoff to these requirements is to employ a slab decomposition.
Let us focus firstly on the possible approaches to solve the 2D FFTs on xz plane. Given an $N_x \times N_y \times N_z$ grid of points to be distributed over p processors two mainly strategies arise in order to carry out the task and arrange the data as needed.
The first approach is to implement a sequence of 2 one-dimensional FFTs, interleaved with data exchange among processors. The other approach, which requires less overall communications, is to make the data local for the dimensions to be transformed (PLS approach [42]).
Since the code already employed a PLS approach, we moved to the fore ones. In detail we exploit the N_y planes independence, to loop among them. Every plane is originally decomposed in $\frac{N_x}{p} \times N_z$. The code perform the z-dimensionals Fourier transformations and an MPI-transposition take place, leading to the new domain decomposition $N_x \times \frac{N_z}{p}$. In this arrangement the x-dimensionals FFTs and subsequent convolutions are performed. Once completed the opposite process takes place.
At the end of the process we have the velocity convolutions, in Fourier domain, stowed in yz planes. Every plane belong to an independent processor, which, since owns all the y values, can build the RHS term and solve the system.
For a tiny cluster or problems with limited dimensions, this kind of approach could be a good choice, since the decomposition limit varies with the lowest decomposed dimension, in this case, N_x or N_z. However the speedup is limited. Such limitations about domain decomposition and, consequently, speedups, raise their importance in modern HPC architectures, that counts many thousands cores. This limits can be avoided implementing a 2D decomposition.

2.6.2 2D decomposition

In 2D approach the domain is decomposed through a grid of processors, in order to form pencils instead of slabs. The initial pencil dimensions are $\frac{N_x}{p_1} \times \frac{N_y}{p_2} \times N_z$ where the total amount of task is $p = p_1 \times p_2$.

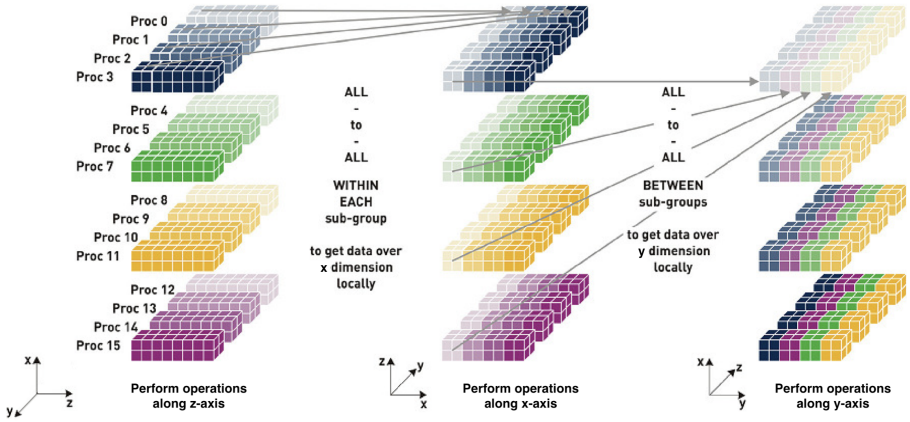

Figure 2.5: 2D decomposition of a 3D array

Starting from such decomposition, we carry out the one-dimensional FFTs along z, then we transpose the array in order to have x-pencils, so the local dimensions will be $N_x \times \frac{N_y}{p_1} \times \frac{N_z}{p_2}$. Once here FFTs and convolutions are performed, then we come back to the original decomposition. The final step, needed to guarantee the availability of the values along the y dimension, is to execute a final remap sequence, from the z-pencils to y-pencils, which will end up with $\frac{N_x}{p_2} \times N_y \times \frac{N_z}{p_1}$ pencils.
The sequence of operations required to remap a 3D array along the three dimensions is illustrated in figure 2.5.

The 2D approach raise the limit of decomposition to be $N \times N$, allowing to scale to an higher number of processes with respect to the 1D method, thus achieve an higher speedup and efficiency when the dimension of the problem become significant.

2.7 Parallel I/O

Input-output could be a serious bottleneck if we do not pay the right attention. In particular, when dealing with supercomputers, two major problems arise:

As first thing let us introduce I/O.

In computer architecture, the combination of the CPU and main memory, to which the CPU can read or write directly using individual instructions, is considered the brain of a computer. Any transfer of information to or from the CPU/memory combo, for example by reading data from a disk drive, is considered I/O [51]. When dealing with cluster the transit of data from disk to CPU is not so straightforward. The presence of multiple CPU require the adoption of one of the following strategies.

The most basic strategy is the master-slaves setup, depicted in figure 2.6.

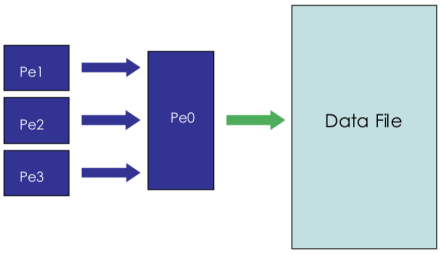

Figure 2.6: Master-Slaves I/O setup

- needing of parallel I/O,
- avoid endianness problem related.

In this kind of strategy a single node of the grid have access to the storage, therefore no scalability is provided. The slave nodes must send/receive data from the master, therefore we face strong slowdown related to the huge workload required to perform I/O by the single node and the following communications among nodes.
A second approach is shown here beside and consist in performing distributed I/O on local files. Such kind of implementation is scalable, ensure data consistency and avoid communication during I/O phase. However, since every processor writes data on its own hard storage, it require a great deal of post processing work to glue data among each others, which increase linearly with the number of processes. For this reason we can not consider it affordable.
The last kind of I/O setup, which is the most updated and optimized, is the so called coordinated controlled accesses. Illustrated in figure 2.7, scalability reaches its peak with this kind of implementation, which takes care of possible communications needing by its own. In this approach every CPU can access to the single storage memory in which the dataset is hosted and, in concomitancy with the other processes, writes the data. As can be understood, the reading/writing operation is intrinsically fragile, since guarantee data consistency can be hard. To avoid consistency lacks, the MPI-IO has been introduced with the deployment of MPI-2 standard [15].

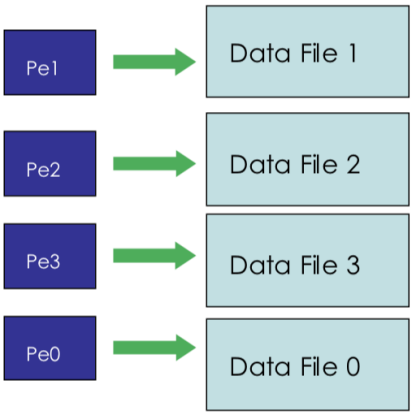

Figure 2.7: Distributed I/O on local files

On top of MPI-IO several high level I/O libraries arose, two well established examples are parallel netCDF and parallel HDF5.
At exception of the master-slave approach, every presented strategy require the adoption of a parallel file system. In computing, a file system or filesystem controls how data is stored and retrieved. Without a file system, information placed in a storage medium would be one large body of data with no way to tell where one piece of information stops and the next begins. By separating the data into pieces and giving each piece a name, the information is easily isolated and identified. We can briefly define the file system as the structure and logic rules used to manage the groups of information and their names. In the same

fashion a parallel file system maintains logical space and provides efficient access to data for distributed memory configurations.

Let us establish the concept of endianness. Intel introduces their white paper with the following sentence: "Endianness describes how multi-byte data is represented by a computer system and is dictated by the CPU architecture of the system. Unfortunately not all computer systems are designed with the same endian-architecture. The difference in endian-architecture is an issue when software or data is shared between computer systems" [6]. Since our binary database has been built on Marconi, at Cineca, but the post-processing analysis take place on our personal computers, we need to guarantee results portability with a reliable method to store the data. Unfortunately MPI-IO can not set a bit ordering different from the machine's natives ones, and we can not assure portability in this way. To do so we have to move from MPI-IO to a library capable to satisfy our requirements. Employ the well established parallel HDF5 library is the natural choice.

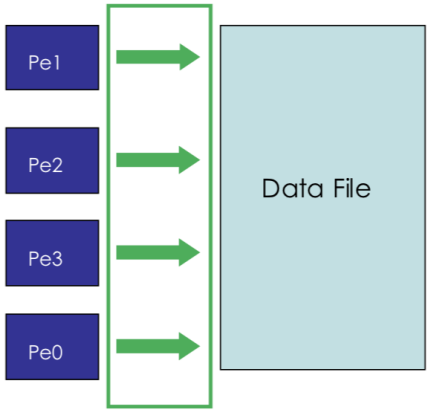

Figure 2.8: Coordinated controlled accesses

3 Code Structure

The present chapter opens with a description of the code implemented, with a focus on the libraries employed for the realization.
The testing environment is present alongside with the performance measurement description employed.

3.1 Code parallelization

The code inherited the data structures from the original solver described in [42]. The code structure is shared with its parent, although the pencil parallelization process required extensive reworking, in particular on the loops and the variables dimensions, which no longer have two entire dimensions of the problem locally, but rather just one, handling a fraction of the domain in the other two directions.

```
LOOP FOR ix=ilo TO ihi
    LOOP FOR iz=klo TO khi
        LOOP FOR ALL iy
            ...
        RETURN
    RETURN
RETURN
```

Listing 3.1: The new FOR loop design. ilo and klo indicates the smallest local mode, while ihi and khi indicates the biggest local mode on the processor

Since the code exploit the finite differences along the y direction we have designed our solver on top of a y-pencil domain decomposition, but this clashes with the need to locally possess the x and z directions in order to carry out the Fourier transformation, so that it is possible to compute the convolutions in the time domain. This required an array transposition library and for this purpose we have entrusted ourself on the *fftMPI* package.
The convolutions routine is de facto the only one whom dynamically switch among the three different decompositions. It starts owing z-pencils, perform the associated Fourier transformation and then switch to x-pencils, where a successive transformation take place. Once the 2D backward Fourier

transformation is completed the convolutions are computed and collected. The process take place in the opposite direction, computing the 2D forward Fourier transformation of the six convolutions. At the end of the process nine values, constituted by the three velocity plus the six convolution values for each grid node, are reordered to y-pencil. During the aforementioned process the anti-aliasing and de-aliasing routine are called locally along each direction. At the end of each timestep a routine moves the data from a y-pencil to a z-pencil order and the process restart.

```
   POINTER TO ARRAY(0..localdim_x-1) OF REAL u
   var = u[ (jhi-jlo+1)*(khi-klo+1)*ix+(jhi-jlo+1)*iz+iy ]
```

Listing 3.2: Example of variable declaration and access to a particular value of the array. The 3D pencil is flattened to a 1D vector with y as fast-varying index, z as mid-varying index and x as slow-varying index. This is mandatory to work with MPI

To minimize the memory requirements and maximize the performance all computations are carried out in-place, overwriting the no longer useful data. To the same purpose the data are initialized using dynamic memory allocations and accessed through pointers. This allowed to minimize the memory requirements, furthermore an allocation in the stack memory revealed to be unreliable in case of wide simulations on a small number of processors.
The I/O has been written from scratch, implementing the MPI I/O standard relying on the high level constructs of the pHDF5 package. This allows very fast operations since all the processors are involved into the process of reading and writing.
Also the optional feature of the live post-processing has been developed from scratch. This feature is particularly useful in case of wide simulations, because avoiding the write on disk process reduce the amount of time of the simulation. It is completely equivalent to its offloaded counterpart, but it is designed in a slightly different manner. At runtime it produce the statistics associated to each time, that will be merged in a second moment, at the end of the simulation. The drawback of this approach is the impossibility to recover the images of the velocity field at each time.

3.2 fftMPI

fftMPI is the package we choose to implement to take care of the decomposition. It is installed through the classic set of commands

```
./configure
make
make install
```

and produce a couple of different libraries, once for 1D and once for 2D decomposition.
These libraries are accessed through four headers, based on the user needing. In our case we were interested at the remap APIs for 3D arrays, so we called

```
remap3d.h .
```

Once the header was included we compiled the source code with

```
-lfft3dmpi
```

flag to produce the executable. To link the code is mandatory to use

```
mpicxx
```

or equivalent, since *fftMPI* is written in C++, and not in C.
Currently the remap works with floating point data only. Anyway, each datum in a distributed 3D grid can be one or more floating point values. In our case, we employed the method with the parameter

```
nqty = 2
```

so that the method select two adjacent floating point values per grid point, and treat them as a single complex number.
A pseudo-code to perform array remapping using such package expect the following structure:

```
#include "remap3d_wrap.h"
remap3d_create(MPI_COMM_WORLD,&remap);

remap3d_set(remap,"collective",cflag);
remap3d_set(remap,"pack",pflag);
remap3d_set(remap,"memory",mflag);

remap3d_setup(local_input_tiles_coords, local_output_tiles_coords,
            nqty,permute,memoryflag,&sendsize,&recvsize);

FFT_SCALAR *ARR = (FFT_SCALAR *) malloc(remapsize*sizeof(FFT_SCALAR));
FFT_SCALAR *sendbuf = (FFT_SCALAR *) malloc(sendsize*sizeof(FFT_SCALAR));
FFT_SCALAR *recvbuf = (FFT_SCALAR *) malloc(recvsize*sizeof(FFT_SCALAR));

/* Fill the array ARR with data */
ARR= ...

remap3d_remap(remap,ARR,ARR,sendbuf,recvbuf);

remap3d_destroy(remap);
```

As we can see the process is made up of:

- create the remap container;
- set options for the remap;
- define the tiles dimensions and commit the container;
- execute the remap;
- destroy the remap container.

The options allow users to set the method, used by MPI, to pass the messages. For example, it is possible to decide whether to use collective calls instead of

point-to-point communications, or it is possible to select how to pack the messages before sending them.
For further informations, and the source code, refer to [55].

3.3 Parallel HDF5 Library

Hierarchical Data Format 5, or HDF5, is widespread scientific data format used by many application to deal with large sets of data [22]. Parallel Hierarchical Data Format 5, or pHDF5, is the parallelized version for clusters.
Designed to store and organize large amounts of data, HDF5 has been originally developed at the National Center for Supercomputing Applications. The crucial feature is its ability to store binary dataset in a processor independent fashion, guaranteeing the portability of the data. This allows extremely compact file dimensions, since we are dealing with binary data, and in the same time we can exploit the advantages of ASCII encoding.
We could think at the parallel I/O as a bridge between the application and the data, which are stored on memory. In this parallelism our high-level library can be thought as the deck of the bridge. Such deck is built on top of pylons, which is the middleware, the MPI-IO. Our library provide an high level of abstractions, allowing to instruct and tune the middleware to perform the requested operations. On its own MPI-IO deals with organizing access to disk by many processes. Everything is anchored to the ground through foundations, which is our parallel file system.
The hierarchical file ordering allows to define a Linux like environment made up of root folder, subfolders, tables, figure, attributes, links and others useful things to maintain the database as readable as possible.
The price for the high flexibility of such data format is the impossibility to have a plug and play VTK reader. In fact we have to rely on third party software to instruct the VTK tool, in our case Paraview, to inspect the database.
We decide to use XDMF3 [29], acronym of eXtensible Data Model and Format 3, to generate an XML file to put beside of our HDF5 file, so that Paraview, or any other VTK software, use it to read the database. Such XML file is automatically generated when the post processor is run. It contains the environmental conditions, such as domain geometry, cell placement, timestep and some instruction to join the velocities data of a cell into a vector. It is used to tell to the VTK reader where and what read from the HDF5 file.

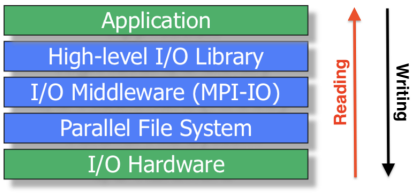

Figure 3.1: Ideal structure of a parallel I/O implementation

3.4 Testing environment

The following chapter will show the performances obtained after the MPI integration in our code. We have benchmarked four different problem sizes:

- small dimension problem $128 \times 128 \times 128$;
- medium dimension problem $256 \times 256 \times 512$;
- large dimension problem $512 \times 512 \times 1024$;
- very large dimension problem $4096 \times 512 \times 512$.

All problems have been tested using 1D and 2D decomposition, so that it is possible to have a comparison among this two methods.
In addition to these tests we have performed a single core benchmark against the original code, at mesh size varying, and a couple of comparisons, selecting a mid-sized grid of points, switching the compiler and the processor architecture.
We want to highlight that all these problems exploit the Hermitian symmetry along the streetwise axis.

Our test were conducted at CINECA[9], an Italian academic research center, which host the 19^{th} most powerful supercomputers of the TOP500 [67] of November 2018 list, Marconi, and GALILeO.
We worked on Marconi[45] supercomputer, in particular on Marconi-A2 partition. Marconi structure fuse different partition to reach the peak performance of 20 PFlop/s; in particular our partition is characterized by 3600 nodes, connected through Intel OmniPath[26] high performance network. Each node host a 68-cores Intel Xeon Phi 7250, code name Knight Landings, and about 100GB of ram.
The machine runs on CentOS 7.2, a Linux distribution, and our benchmark code has been compiled using GNU GCC 7.3 [58] with OpenMPI 3.0.0 [64][16].
We have chosen to compile our benchmark code using the GNU Compiler Collection, instead of using proprietary and optimized Intel compilers, to ensure the possibility to carry out code run time comparisons across CPU from different vendors, with different architectures.
We have tested different flags during the compilation phase, trying to enable different levels of optimization and code vectorization, the most important feature of the Xeon Phi. The best results, using the GCC compiler, have been achieved using:

```
-O2 -fpic -march=native -std=c99
```

This behavior was expected since our code does not include the OpenMP[52] features at present time, so the MIC[30] (Many Integrated Cores) architecture can not exploit such fundamental feature. Furthermore the compiler flags used are general purpose, and does not providing the desired tuning for the Intel Knights Landing processors, thus resulting in lack of performances and efficiency.

Further tests have been carried out, selecting the mid-sized mesh, changing the compiler and the processor's architecture.

The first comparison have taken place on Xeon Phi processors, switching from GCC compiler to Intel C Compiler 18.0. Such tests have revealed that the latter provide more than 2x faster code.
The second tests have taken place at Cineca, using GALILeO [21] instead of Marconi. Such benchmarks let us move from the MIC architecture of the aforementioned processor, towards a more traditional ones. GALILeO is a supercomputer based on IBM NeXtScale cluster, offers 400 general purpose compute nodes, each ones equipped with two 18-core Intel Xeon E5-2697 v4, running at 2.30GHz. Every node count 128 GB of RAM, which means 8 GB per core, and the network communications rely on Infiniband technology.

3.5 Performances measurement description

In this section we illustrate the procedures used to compute the performance indexes.
The measures were carried out using the same code, compiled once for all the simulations, tested at *tasks* variations.
Attention should be posed on the concepts of *tasks*, *cores* and *processors* or *nodes*. The first identify the number of parallel processes of the simulation and is obtained as the product between the other two values:

$$tasks = n\ cores \times processors$$

As we will see shortly the code exhibit sensible performances variation depending on the number of nodes involved in the simulation and the cores per nodes configuration.

We start providing the definition of speedup:

$$S = \frac{T_0}{T_p}$$

In this simple equation T_0 identify the single core runtime, while T_p is the runtime associated with the execution of the code on p tasks.
The efficiency is calculated starting from the speedup as

$$E = \frac{S}{p}.$$

Every simulation of our benchmarks started with the single core run, in order to establish the baseline.
The tool designed to catch the runtime is simple and lightweight. It wraps the simulation, save the runtime for every processor and retrive the highest value only. The following pseudo-code should give an idea of the working principle.

```
   begin = clock();
LOOP forward WHILE time < t_max-deltat/2
   ...
   \*Perform simulation*\
   ...
REPEAT forward
```

```
end = clock();
sim_time = (end - begin) / CLOCKS_PER_SEC;
max_sim_time=0.0;
MPI_Allreduce(&sim_time,
    &max_sim_time,1,MPI_DOUBLE,MPI_MAX,MPI_COMM_WORLD);
FILE *ft = fopen("time_out","w"); fprintf(ft, "Simulation performed
    in %f s", max_sim_time);  fclose(ft);
```

4 Code Benchmarks

Our fourth chapter shows the results of our benchmarks. We will present a single core comparison facing the original code against the new pencil decomposition approach. Following we will present the performances obtained on four different meshes sizes, varying the number of cores employed during the computation and comparing the 1D decomposition against the 2D solution. Towards the end of the chapter we will present two subsections containing the performances obtained using a different compiler and the performances obtained employing a different cluster, which face a different processor architecture. The chapter closes comparing our parameters with the ones obtained in another simulation, showing our speedup trend and the effects of the Intel proprietary technology called Hyper threading on our curves.

4.1 Single core comparison

Before seeing the multi-cores performance of our code we have performed a benchmark against the original implementation of Quadrio and Luchini [42], which can be considered as a state of the art solver, and our *pencil decomposed* solver. We are particularly interested in comparing the timings for a *pencil approach* against a slab ones, as the PLS is. For this reason we do not carry out comparison against PLS and our *slab decomposed* algorithm, since they use exactly the same approach to solve the problem, and it is likely to perform in the same time. The simulations have taken place in Debian, a Linux environment, employing an Intel i5 running at 3.1 GHz.
Keeping in mind the two algorithm structures, we expect that CPL perform faster. Despite we already know which code is faster, this test is important to understand how far our code is moving away from the single core optimum.
Our benchmark suggested that on a 64^3 simulation, the CPL perform a single time step in 0.29 s, while our code employ 0.36s. On wider mesh of 128^3 points, the CPL is still faster, with a time step completed in 2.94s, while the other code employ 3.11s.
Our results confirm the predictions and highlighted that, although on little mesh we performed about 24% slower than the optimum, on a larger grid our gap reduced to be just the 5%.

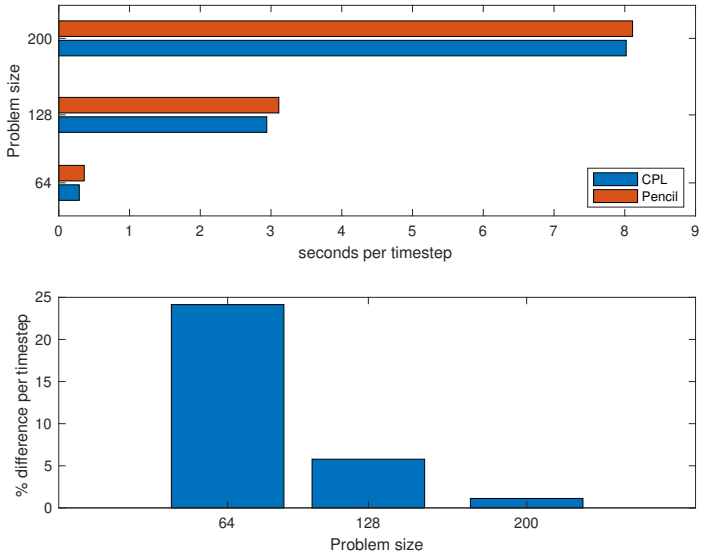

Figure 4.1: Single core timing comparison

Increasing again the mesh size to 200^3 points lead to further reduction in time difference, with the CPL that still perform faster, with 8.02s per time step, against the 8.11s per step of our code. This means just 1% of difference between the two codes per time step. On figure 4.1 are reported the percentage differences and costs per time step of the three tests.

4.2 Scaling performance of 128^3 problem

We proceed now showing the performances achieved by our code for the small problem. This is likely the most critical benchmark for our code, since implement a distributed parallel approach to a problem with tiny dimensions could lead to lack of efficiency quickly.
In this kind of problem the arrays size fits the cache dimension of the Intel Xeon Phi processor; in fact, we achieve greater speedup by using less nodes as possible at cores equality. For this reason in this test we used 64 cores per node. The results of figure 4.2 shows that, although on single core the structure of the slab decomposed algorithm is faster, suddenly the benefits of pencil decomposition overcome the cost due to the poorer array storage. To understand the latter sentence we should recall the figure of page 16. Keeping in mind figure 2.3, is possibile to understand why the slab decomposition is faster on single core, and typically also for a tiny cores number. Since the slab algorithm work *per plan* we can, wisely, allocate and work just on a small dataset. This affect the communication phase, which will be faster if compared with the ones of the pencil decomposition that, instead, require to allocate all the data at once.

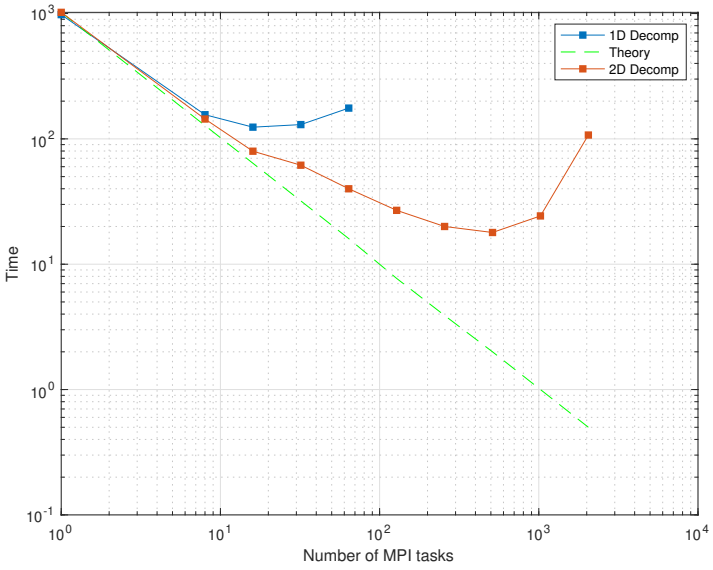

Figure 4.2: Scaling performance of 128^3 simulation

In figure 4.2 is possible to look at the time needed to perform the DNS, at varying of the cores number and algorithm. The green dashed line represent the theoretical limit and has been obtained as the ratio between the single core time and the number of cores of the simulation.
Despite of the results could seems poor, the qualitative comparison of figure 4.3 against a 3-dimensional FFT, using P3DFFT, reported in [8, p. 43], suggest that our results are on the average, or better, until the communications cost overcome the benefits of such parallel distributed approach.
The table 4.1 summarizes all data related to the 128^3 simulation.
According to this table the figure 4.4 shows the speedup at the varying of the cores number.
At the speedup peak the code runs 57 times faster than serial ones. Such peak is obtained using 512 cores. However, as expectable, the efficiency of this implementation is quite poor. In fact, if we use more than 16 cores we drop immediately to performances around 50% or lower. For completeness the efficiency behavior is reported in figure 4.5.

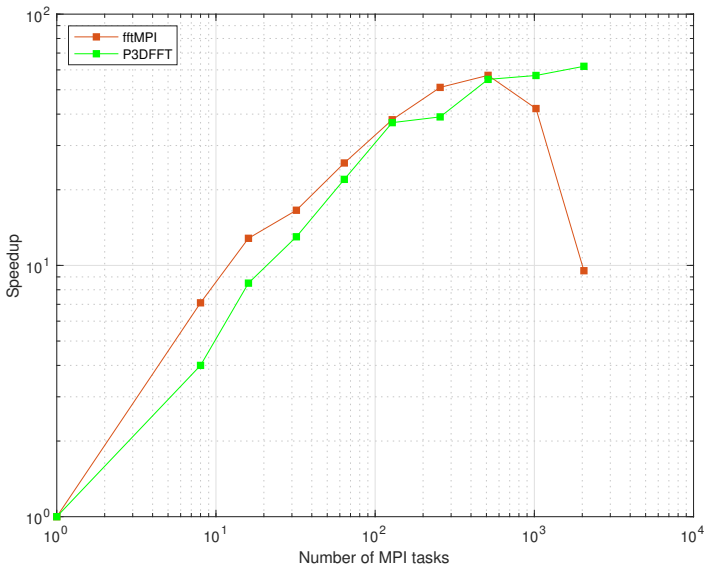

Figure 4.3: Comparison of a 3D FFT against DNS with fftMPI

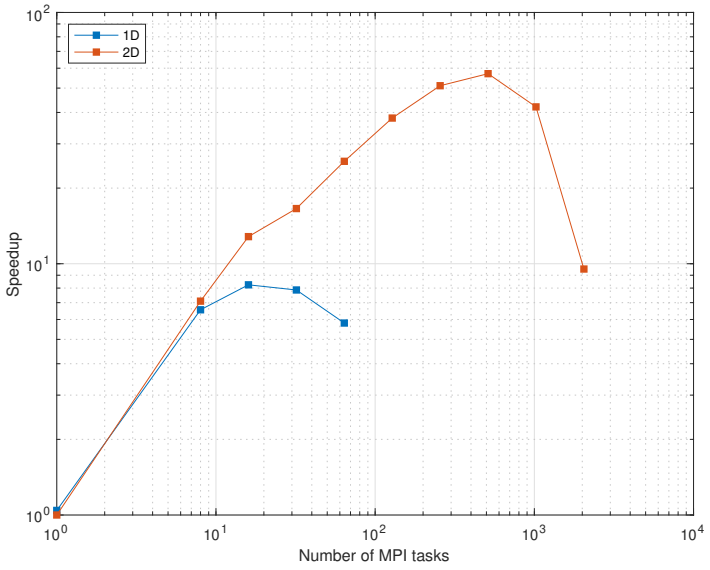

Figure 4.4: Speedup performance factor of 128^3 simulation

Table 4.1: Data from 128^3 simulation

#Processes	Time [s]	Speedup	Efficiency [%]	Decomp
1	980	1.05	100	1D
	1022.1	1	100	2D
8	155.9	6.56	79	1D
	144	7.1	89	2D
16	124.2	8.24	49	1D
	79.7	12.82	80	2D
32	130.4	7.86	24	1D
	61.7	16.6	52	2D
64	176.1	5.81	9	1D
	40	25.6	40	2D
128	26.9	38	30	2D
256	20	51.11	20	2D
512	17.9	57.07	11	2D
1024	24.3	42.1	4	2D
2048	107.3	9.52	0	2D

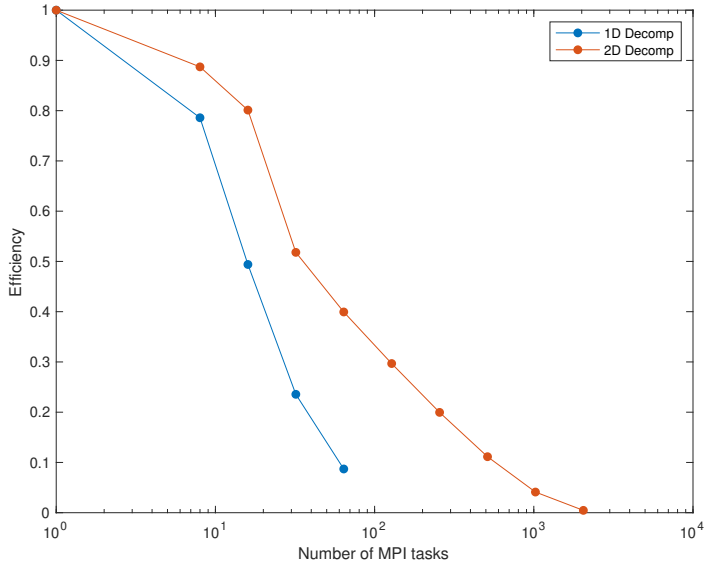

Figure 4.5: Efficiency factor of 128^3 simulation

4.3 Scaling performance of $256 \times 256 \times 512$ problem

Many authors in past have highlighted how bigger problems provide better scaling capabilities and our $256 \times 256 \times 512$ simulation fulfill such trend. The medium sized problem shows better scaling performances compared to the small ones.
A slab decomposed algorithm provide gains of $\mathcal{O}(10)$ in terms of execution times, less than a pencil decomposed algorithm, but with better results for small processors grid. In fact, as depicted in figure 4.6 the 1D decomposition curve achieve lower execution times than the 2D ones, until 32 cores.
Passed 32 cores the pencil decomposition prevails, reaching speedup factors above 120, with time savings in the order of magnitude of $\mathcal{O}(100)$ with respect to the single core runtime. In the figure 4.7 is possible to see the efficiencies achieved by the two methods, running on 64 threads per processor. It is important to denote the behavior of the pencil decomposed algorithm, which, until 8 cores are used, exhibits a very high scaling efficiency.
Comparing image 4.6 with its counterpart for the 128^3 problem, figure 4.2, we can see that the curves are quite similar. Both exhibits a very good fitting with the theoretical ones until 16 parallel processes take place. Once passed this threshold, the bigger problem maintains a better scaling efficiency, as we could see by comparing figure 4.7 and 4.3, for both decomposition methods.
The better efficiency allows to reach higher speedup factors at number of processes equality, and the larger dimensions move the performances peak towards higher number of threads, as is possible to see by looking at figure 4.8. The combination of this two factors doubles the last speedup factor, passing from 57, for the 128^3 problem, to 122 for the $256 \times 256 \times 512$.
A comparison of the performances of 1D decomposition against the 2D for the present problem dimension is presented in table 4.2.
Passed 8 cores, to recover high efficiency we must decrease the number of threads per processor. We have executed a detailed analysis varying the threads per processor number, seeking the optimization for both the decomposition methods.
For what concern the slab decomposition the results, reported in table 4.3, shows that, although slightly improvements have been achieved, the 1D decomposed algorithm is quite insensitive to cores per processor variations, showing constant speedups, efficiencies and timing.
Through figure 4.9, by looking at the single core curves, it is interesting to denote the presence of a knee, when 32 simultaneous processes take place, which origin a performances decrease. Such loss of linearity is present also using different cores per processor combinations, although on single core it appears to be more evident. Thus lead us to think that the intrinsic scaling limit, which depends on geometry and is caused by the raise in interprocessor communications time, has been reached. This limit clearly tear down the efficiency curve, as depicted in figure 4.10.
Far more interesting is the pencil decomposed algorithm behavior. The data of such simulation, reported in table 4.4, shows relevant improvements by varying the number of cores per processor. Another not yet cited, but always present tuning using 2D decomposition, is related to processor grid balancing. Our code

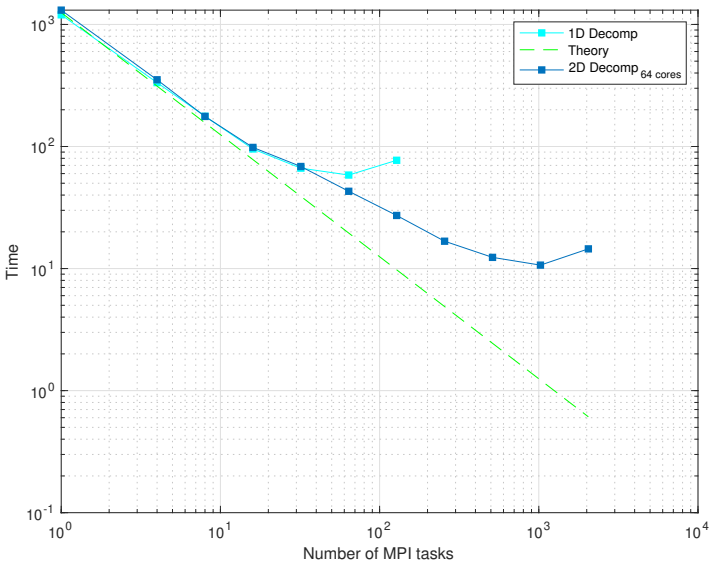

Figure 4.6: Scaling performance of $256 \times 256 \times 512$ simulation

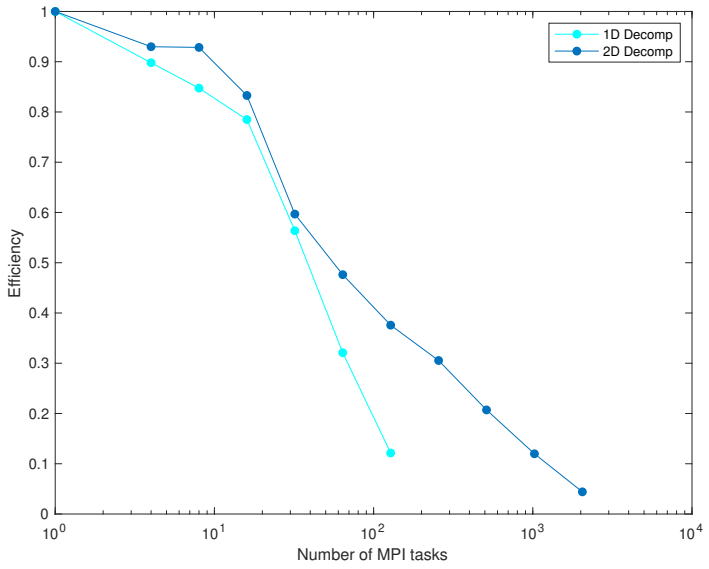

Figure 4.7: Efficiency factor of $256 \times 256 \times 512$ simulation

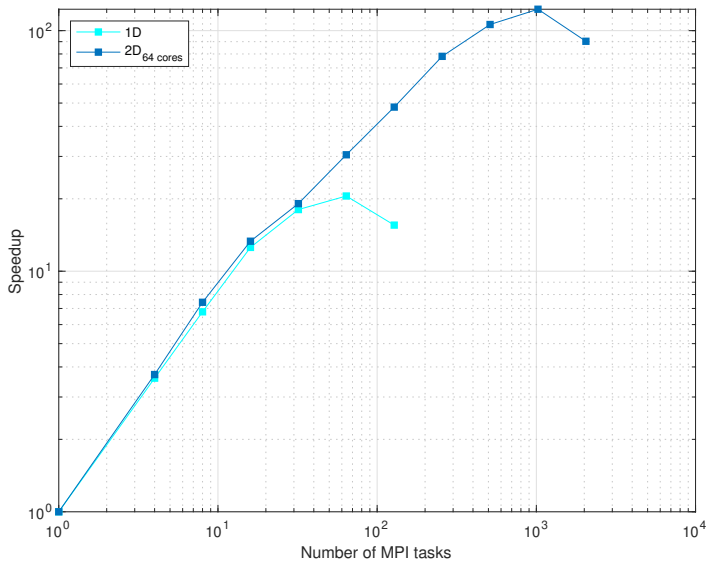

Figure 4.8: Speedup performance factor of $256 \times 256 \times 512$ simulation

Table 4.2: Data from $256 \times 256 \times 512$ simulation

#Processes	Time [s]	Speedup	Efficiency [%]	Decomp
1	1198.8	1	100	1D
	1309.7	14.28	89	2D
4	333.7	3.59	90	1D
	352.1	3.72	93	2D
8	176.8	6.78	85	1D
	176.3	7.43	93	2D
16	95.5	12.56	78	1D
	98.3	13.33	83	2D
32	66.5	18.04	56.3	1D
	68.6	19.1	60	2D
64	58.4	20.54	32	1D
	43	38.48	48	2D
128	77.1	15.55	12	1D
	27.2	48.1	36	2D
256	16.8	78.19	31	2D
512	12.4	106.1	21	2D
1024	10.7	122.7	12	2D
2048	14.5	90.33	4	2D

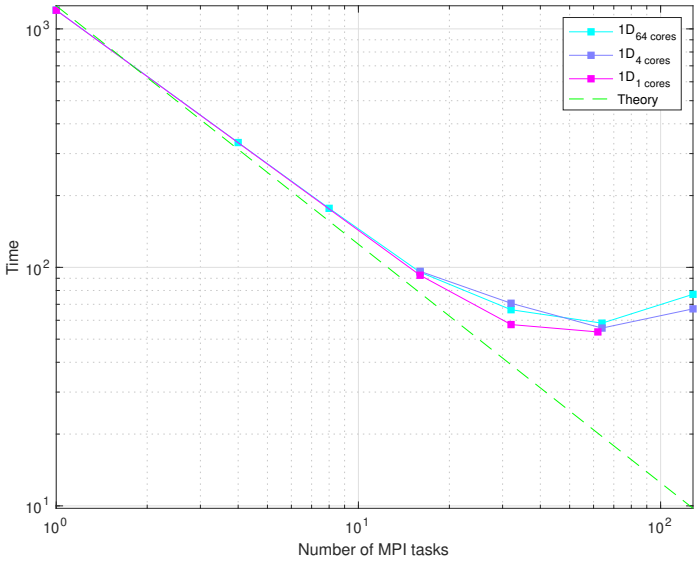

Figure 4.9: Time scaling comparison using 1D decomposition for $256 \times 256 \times 512$ simulation

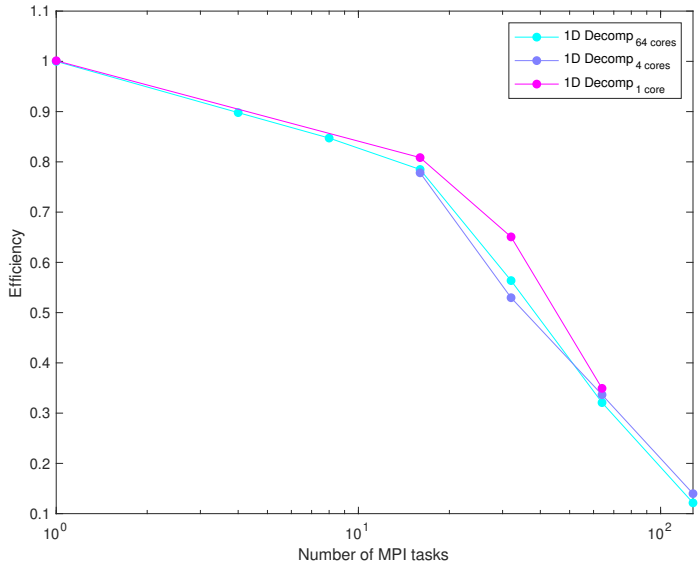

Figure 4.10: Efficiency comparison using 1D decomposition for $256 \times 256 \times 512$ simulation

Table 4.3: Data from $256 \times 256 \times 512$ simulation, 1D decomposition

#Processes	Time [s]	Speedup	Efficiency [%]	cores
1	1198.75	1	100	64
4	333.7	3.59	90	64
8	176.8	6.78	85	64
16	92.7	12.94	81	1
	96.3	12.45	78	4
	95.5	12.56	79	64
32	57.6	20.82	65	1
	70.7	16.95	53	4
	66.5	18.04	56	64
16	53.62	22.36	35	1
	55.7	21.51	34	4
	58.4	20.54	32	64

Table 4.4: Data from $256 \times 256 \times 512$ simulation, 2D decomposition

#Processes	Time [s]	Speedup	Efficiency [%]	cores
1	1309.7	1	100	64
4	352.1	3.72	93	64
8	176.3	7.43	93	64
16	98.3	13.33	83	64
32	49.9	26.26	82	4
	68.6	19.1	60	64
64	29.3	44.73	70	4
	43	30.48	48	64
128	21.4	61.17	48	4
	21.5	61.03	48	8
	23.2	56.58	44	32
	27.2	48.1	38	64
256	11.5	113.9	44	4
	11.8	110.9	43	8
	13.5	97.23	38	32
	16.75	78.19	31	64
512	9.4	140.1	27	4
	9.6	136.7	27	8
	11.8	110.7	22	32
	12.4	106.1	21	64
1024	7.3	178.9	17	8
	9.9	132.7	13	32
	10.7	122.7	12	64
2048	14.5	90.33	4	64

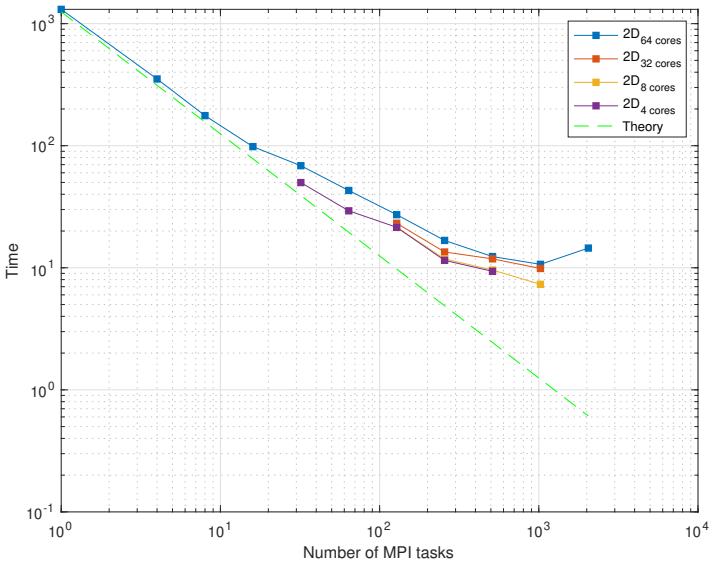

Figure 4.11: Time scaling comparison using 2D decomposition for $256 \times 256 \times 512$ simulation

provide optimal results when the processors grid have the same amount of MPI tasks among the two dimensions. Such behavior has been already described in deep in [8, p. 39]. However it is not always possible to have the same number of threads in the two dimensions. In this case, after proper benchmark, we found that better results where achieved whether the number of tasks which decomposed the streamwise direction was lower than the other.

As already said, the gains that we can obtain by reducing the number of threads per processor are high. Thus is due to the MPI standard that, although able to handle SMP processors [63], lacks in efficiency as the number of cores becomes higher. This is well highlighted by figure 4.11, in which we can see that a 4 cores approach can provide similar, or better, results than running the same code on 64 cores using twice the number of processes. The same reasoning holds also for an 8 versus 64 cores code execution.

This provide a boost in terms of execution time and, consequently, in terms of speedup factor, as can be recovered by looking at the results in table 4.4 where, at the peak, our speedup pass from 122.7 to 178.9.

The figure 4.11 allow to see how, reducing the number of cores per processor, influence also the efficiency. Indeed we can see that running the code on 32 parallel processes using just 4 threads per nodes tends to realign the curve to the theoretical ones.

The efficiency curves can be seen in figure 4.13. Although they exhibit a macroscopically linear behavior and a tendency to moves rightward as the number of cores per processor decreases, it is possibile to identify the arise of steps. The steps tends to become more prominent as the number of tasks per processor decrease, suggesting that better efficiencies are achieved when a

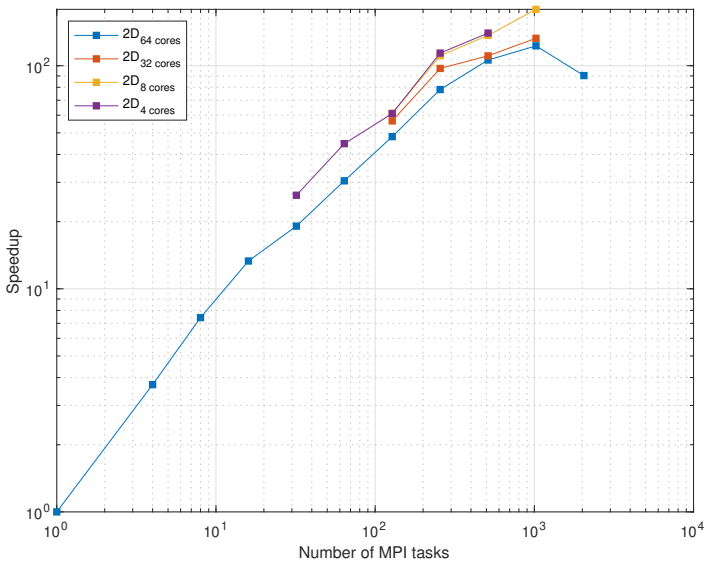

Figure 4.12: Speedup comparison using 2D decomposition for $256 \times 256 \times 512$ simulation

balanced task decomposition is used, as pointed in [8] . For completeness the speedup factor curves has been reported, in figure 4.12.

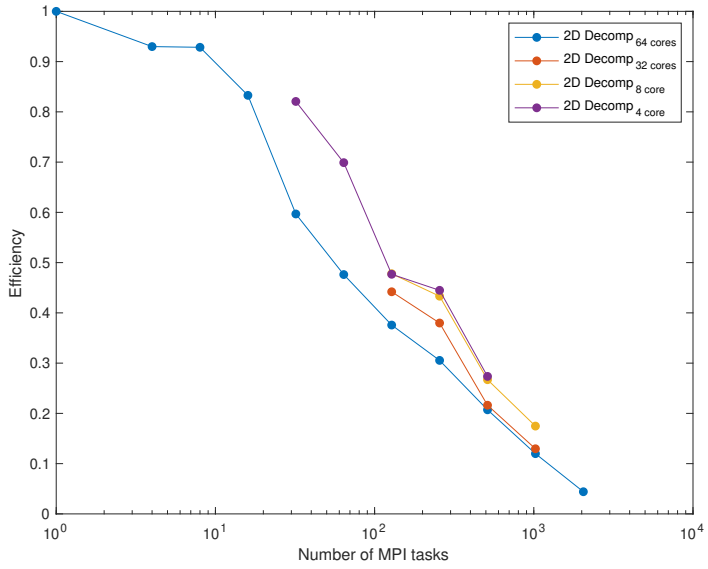

Figure 4.13: Efficiency comparison using 2D decomposition for $256 \times 256 \times 512$ simulation

4.4 Scaling performance of $512 \times 512 \times 1024$ problem

Let us proceed now showing the performances achieved by our code in a large sized problem. The present simulation shown a better scaling effectiveness and efficiency for both methods with respect to the previous problem.
The best results are reached using 8 cores per processor, indicating that the efficiency lack cost overcome the message passing price. In theory we would rather to use a pure MPI approach instead of a heavily threaded ones, because the speedup achieved by the first method are significantly faster than the latter ones in our implementation. However, for costing reasons, a tradeoff between the two solutions is preferred.
The speedup peak is remarkable, with a factor above 430 on 2048 cores using pencil decomposition, while stops around 50 using 128 cores and slab decomposition.
As we can see from figure 4.14, where is compared the slab decomposition against the pencil ones running on 64 cores per processor, the speedup factor of the latter algorithm increase approximately linearly until 512 cores. The raise in performance continue at lower factor until 2048 cores are reached, where performances start decreasing. For what concern the first algorithm we face a sub-optimal linear increase in performances until 64 cores are reached.
We refer to figure 4.15 to have an idea of the efficiency of the 1D decomposed algorithm. The figure shows that the efficiency remains near the 80% until 32 cores are used. However, once passed such limit, the behavior is still linear and less sloping with respect to the ones shown in figure 4.7.

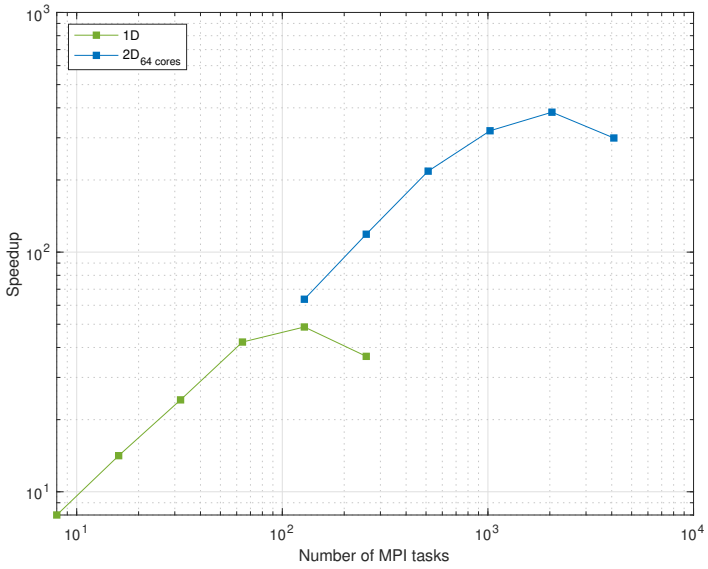

Figure 4.14: Speedup factor of $512 \times 512 \times 1024$ simulation

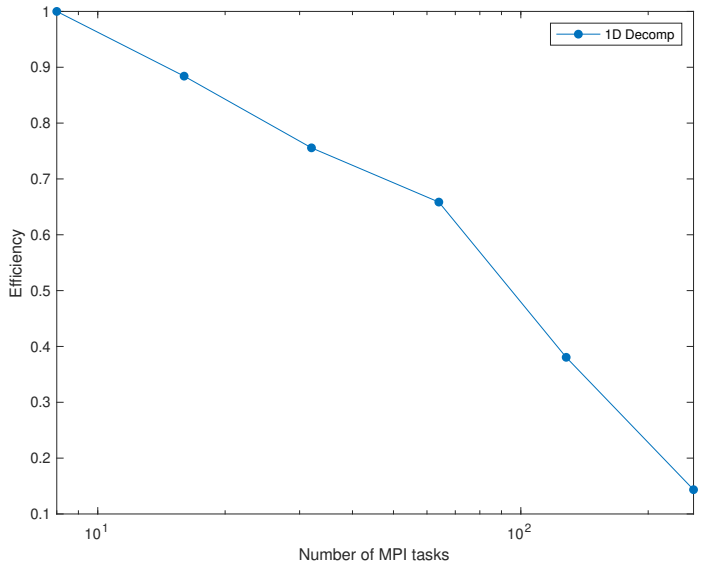

Figure 4.15: Efficiency factor of $512 \times 512 \times 1024$ simulation using 1D decomposition

Table 4.5: Data from $512 \times 512 \times 1024$ simulation, 1D decomposition

#Processes	**Time [s]**	**Speedup**	**Efficiency [%]**
8	3615.3	8	100
16	2044	14.15	89
32	1196.4	24.18	76
64	686.2	42.15	66
128	593.7	48.71	38
256	787.3	36.73	15

It is interesting to denote how advantageous the 2D decomposed algorithm is; it is clear by comparing the tables 4.5 and 4.6. The time saving could be in the order of magnitude of $\mathcal{O}(10)$ if compared with the 1D decomposed ones.
It is evident that the pencil decomposition outclass the slab ones. However we could improve our results by selecting the proper number of threads per processor.
Since the preceding simulation highlighted that the slab decomposition is less sensitive to the cores per nodes optimization, we decided to skip it and directly pass to analyze the pencil ones.
The problem dimensions are relevant, in fact a single processor analysis can not be carried out because we face an out of memory error. To decide what is the best solution in terms of 2D decomposition we must evaluate the performances in an homogeneous way. By looking at the values in table 4.6 and the depicted counter part, in figures 4.16, it is evident that the single core performance fits the theoretical limit, or overcome it. The difference between such implementation and an heavily threaded ones rely on the library used. In fact, has already been reported in the previous section, although can handle multicores processes, OpenMPI cannot hold the intra-node communications such efficiently as the extra-node ones. This behavior is highlighted in figure 4.17 where the efficiencies comparison for the pencil decomposed algorithm are reported.
Since a single core run is not possible to be carried out due to memory limitations, we decided to take the runtime of the 16 parallel processes on single core as reference, computing the speedups and efficiency basing on the performances achieved by such run. Since the behavior of this run shows an efficiency equals or above the 100% until 64 parallel processes are used, we are confident that our speedups would be correct, or under-estimated.
As the results of the previous problems have highlighted, the reduction of tasks per processor lead to consistent gains in terms of timing execution, which turns out to provide remarkable gains in speedups, as can be seen in figure 4.16 that shows the speedup factor variation depending on the thread number. In particular passing from 64 threads per processor to 8 threads per processor allows the code to improve the execution time of the 20%, and there is still room for further improvements, as could be understood by looking at the difference, in terms of speedup, between the single and the 8 cores run at 64 parallel processes. The single core provide 1.5× faster performances compared to the 8 cores run.

Table 4.6: Data from $512 \times 512 \times 1024$ simulation, 2D decomposition

#Processes	**Time [s]**	**Speedup**	**Efficiency [%]**	**cores**
16	2008	16	100	1
	2249.8	14.28	89	8
32	941.3	34.14	107	1
	1137.1	28.26	88	8
	1264.9	25.41	79	16
64	494.8	64.95	102	1
	693.9	46.31	72	16
	750.5	42.82	67	32
128	294.1	109.3	85	1
	387.4	82.95	65	16
	408.4	78.8	61	32
	505.4	63.59	50	64
256	209.1	153.7	60	8
	215.2	149.3	58	16
	225.2	142.7	56	32
	270.6	118.8	46	64
512	122.5	262.3	51	8
	147.6	217.7	43	64
1024	82.37	390.1	38	8
	87	369.4	36	16
	89.9	357.5	35	32
	100.2	320.9	31	64
2048	74.45	431.6	21	16
	78.3	410.4	20	32
	83.8	383.5	19	64
4096	91.1	352.7	9	32
	107.3	299.3	7	64

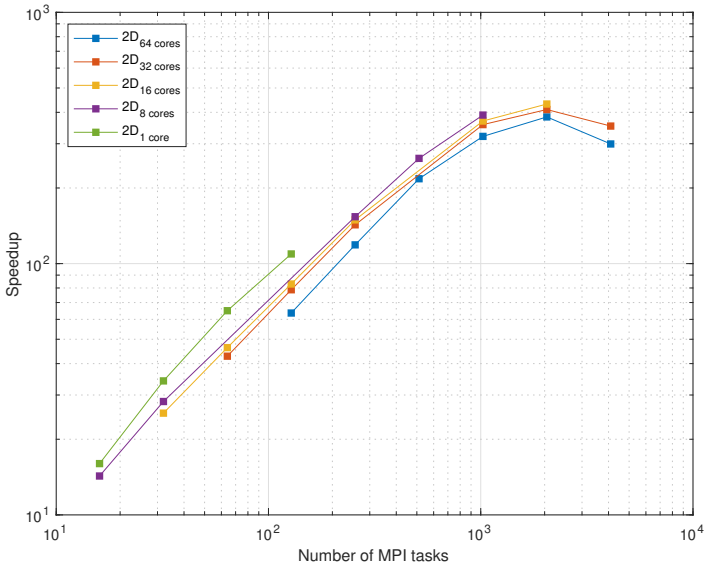

Figure 4.16: Speedup factor comparison for 512 × 512 × 1024 simulation

To sum up, by looking at figure 4.16 and figure 4.17, it is possible to generalize that decreasing the number of threads per processor moves the speedup curves upwards, leading to better performances, and yielding to flatter efficiency curves. Such efficiency curves tend to slide rightward, achieving better results, respecting the requirement of at least two processors minimum.

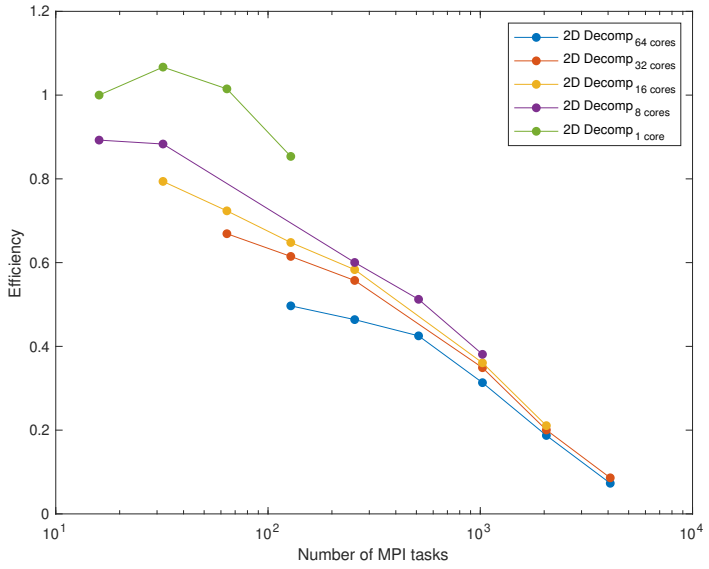

Figure 4.17: Efficiency comparison for $512 \times 512 \times 1024$ simulation

4.5 Scaling performance of $4096 \times 512 \times 512$ problem

The last benchmark deal with very large problems. Like for the previous large problem, the dimensions are so huge to require the adoption of multiple processors to run, otherwise we will face an out of memory error. On a Intel Xeon Phi [25] the minimum requirements are to employ at least 2 processors and use 32 cores, or less, per processor.

As the previous problem have highlighted, the less cores are used and the better results are scored, so our impossibility to go further than 32 cores per processor, as pointed some rows before, would not be a big deal. We may suppose that the poorest results will be achieved by 32 cores runs, instead of the 64 ones.

Let us start showing figure 4.18 in which is reported the time scaling of our code.

As could be seen, the 2D decomposition using single core achieved the lowest timing execution. It is interesting to denote how this combination fits the theoretical behavior perfectly.

All other 2D decomposed combinations exploit a worse behavior with respect to the single core run, with a marked trend, where the increase in cores per processor number leads to poorer performances.

Such behavior is aligned with our predictions.

For what concern about the 1D decomposed algorithm, which, since the code structure is slightly different, can run also on 64 cores per processor, it achieve the worst performances among all possible solutions, highlighting once again the benefits of using a pencil decomposed approach.

The speedups achieved by this kind of domain decomposition can be seen in

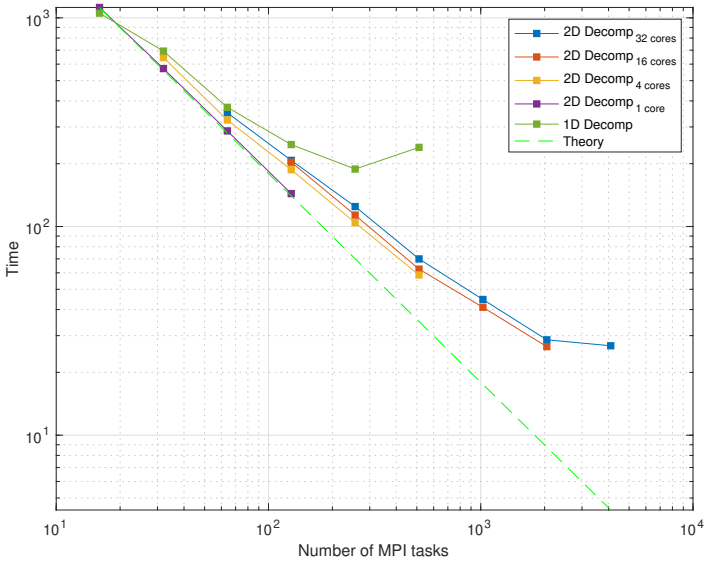

Figure 4.18: Time scaling comparison for $4096 \times 512 \times 512$ simulation

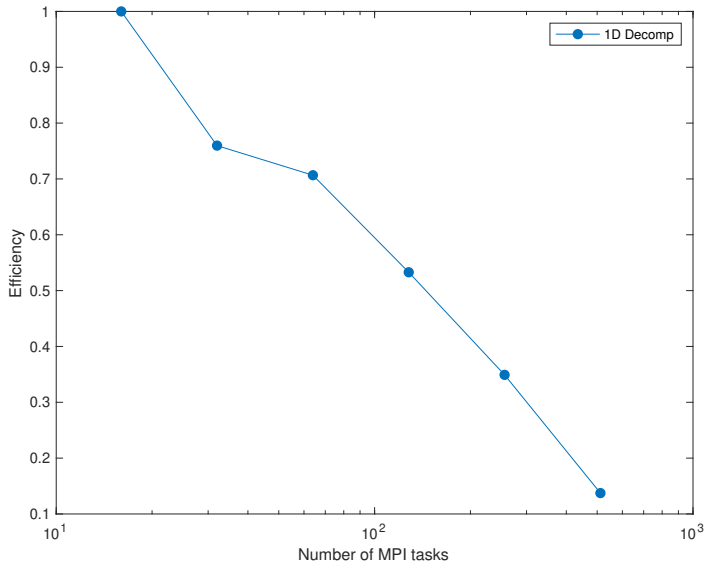

Figure 4.19: Efficiency factor of $4096 \times 512 \times 512$ simulation using 1D decomposition

Table 4.7: Data from $4096 \times 512 \times 512$ simulation, 1D decomposition

#Processes	**Time** [s]	**Speedup**	**Efficiency** [%]
16	1052.9	16	100
32	693	24.31	76
64	372.5	45.23	71
128	247	68.2	53
256	188.5	89.37	35
512	239.5	70.34	14

table 4.7 while the efficiency graph, which shows a poor behavior, can be seen in figure 4.19.
Far more interesting, are the data in table 4.8, which report the speedups, efficiency and timing achieved by the algorithm with 2D decomposition. The data, and the graphical counterpart which can be seen in figure 4.21 and 4.20, report a very high efficiency using single core, while, although smaller, a still high efficiency is preserved by using 4 cores per processor.
Increasing the counter of threads per processor leads to constant losses, as expectable. However, such losses between adjacent stations are lower than the ones of the previous simulations, furthermore we can see wider gaps among the efficiency curves, symptom that there is wider room for improvements.
Moreover, at very high number of cores, the efficiency curves slope is minor than before, preserving efficiency and allowing us to perform faster computations with greater speedups.
To talk about speedups is useful to introduce figure 4.21, in which these are reported.
The graph, on page 49, shows the results achieved by the 1D and 2D domain decomposed algorithm, with emphasis on the effects of the variation of cores per processor quantity, for the 2D algorithm only.
As has been done in the previous section, the high efficiency shown by the single core run, combined with the physical impossibility to use less cores due to memory limitations, has lead us to made the assumption of speedup equal to 16 for a 16 parallel processes run in a single core environment. All other efficiencies and speedups have been derived using such data as reference.
As can be seen in table 4.8 the best result is achieved using 16 cores per processor and 2048 parallel processes. Unfortunately, due to some policy limitations, we can not push forward our resources request, although the graph clearly shows margins of improvements for such configuration.
Differently from all the previous simulations, the processor grid for the pencil decomposition results balanced when the streamwise stencil has half the modes with respect to the other direction. Such configuration has been suggested by benchmarks.
In conclusion we may say that, although the modes distribution results unbalanced in this simulation, the speedup trend still remain aligned with the ones exhibited in the other simulations.

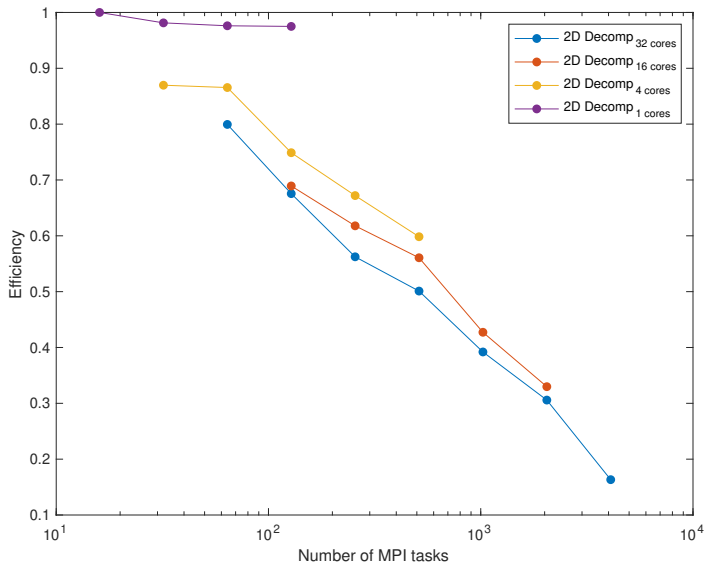

Figure 4.20: Efficiency factor of 4096 × 512 × 512 simulation using 2D decomposition

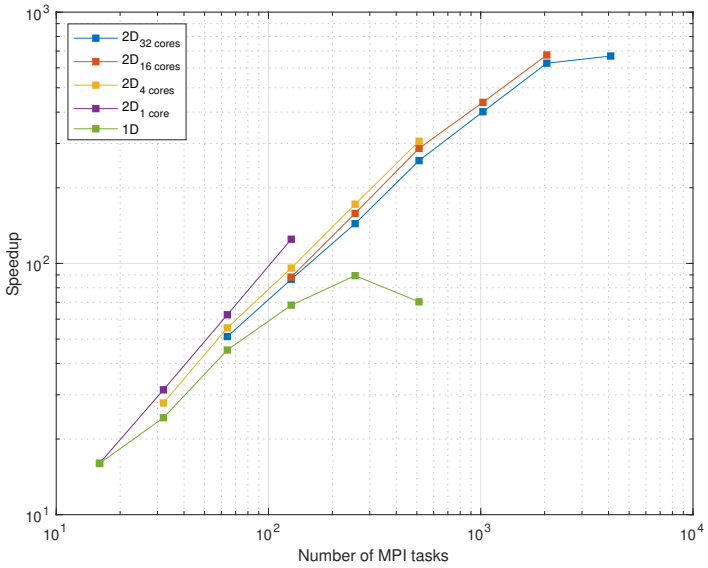

Figure 4.21: Speedup factor of 4096 × 512 × 512 simulation

Table 4.8: Data from $4096 \times 512 \times 512$ simulation, 2D decomposition

#Processes	**Time [s]**	**Speedup**	**Efficiency [%]**	**cores**
16	1121.5	16	100	1
32	571.4	31.4	98	1
	644.8	27.83	87	4
64	287.2	62.47	98	1
	323.9	55.39	87	4
	350.7	51.17	80	16
128	143.8	124.8	98	1
	187.2	95.85	75	4
	203.4	88.23	69	16
	207.5	86.47	68	32
256	104.3	172	67	4
	113.4	158.2	62	16
	124.6	144	56	32
512	58.6	306.4	60	4
	62.5	287.1	56	16
	70	256.5	50	32
1024	41	437.5	43	16
	44.7	401.4	39	32
2048	26.6	675.5	33	16
	28.7	626.3	31	32
4096	26.8	668.8	16	32

4.6 Further tests

4.6.1 Intel compiled code performances

Not happy of the previous results, we decided to move from GCC to Intel proprietary compiler, as advised also by the Cineca authorities. The Intel C++ Compiler 18 is designed to take care of the MIC architecture, using dedicated flags during the compilation process.
Using

```
-AVX512 -parallel
```

flags is possible to instruct the compiler to generate vectorized code autonomously. Although there are no guarantees that all loops will be vectorized, leading this solution to be less efficient than an OpenMP implementation, the usage of these flags speeded up our code roughly of a factor two, providing also doubled efficiency with respect to the previous GCC solution, as can be seen comparing figure 4.22 with 4.12 and figure 4.23 with 4.13.

As before, the performance peak remains at 1024 simultaneous tasks, however the maximum speedup moved from 178.9, of the 8 cores run using GCC, to 347.1, using the same number of cores with Intel compiler compiled code, with an efficiency not far from the 40% threshold. The efficiency gap between the 64 cores runs and the 4 cores ones remains around the 10%, but the double efficiency with respect to the GCC solution enables the possibility to use 64 cores during production.

Next to the already cited flags we used others to refine the auto-vectorization process, in particular the distribution of the MPI tasks among the tiles of the processors, that in this configuration tends to fill adjacent cores with adjacent arrays values, the prefetching level and the mapping of the High Bandwidth Memory, that in this configuration is available as L3 cache memory. We also moved to a deeper level of optimization and we disabled fractions in favor to reciprocal multiplications.

We experienced the impossibility to run on less then 4 cores using Intel compiled program. For such issue, we decided to set the reference time using 4 processor, running in single core mode.

4.6.2 Performances on GALILeO

Since the Xeon Phi architecture is markedly different from the traditional ones, we have decided to carry out a benchmark also using Intel Xeon E family processors. The code used on these 18 cores processors has been compiled using Intel Compiler 18, as in the previous chapter, but without trying to optimize the performance. Indeed just the flags

```
-O2 -xCORE-AVX2
```

have been used.

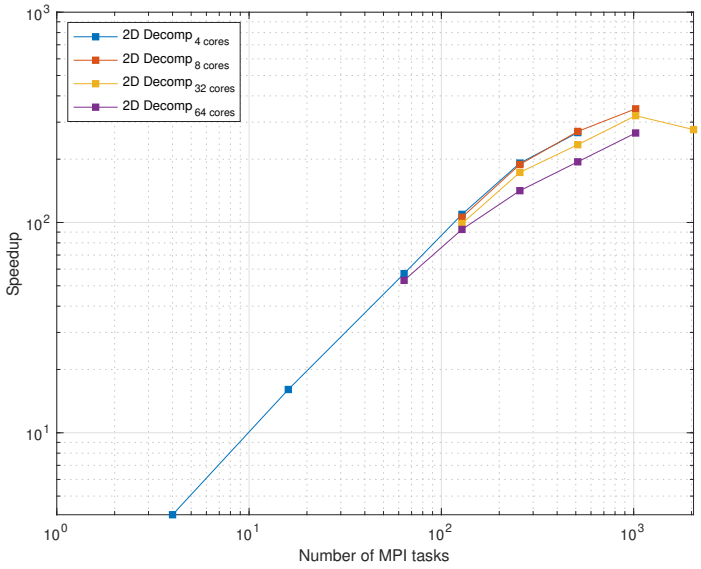

Figure 4.22: Speedup using 2D decomposition for $256 \times 256 \times 512$ simulation with Intel Compiler 18

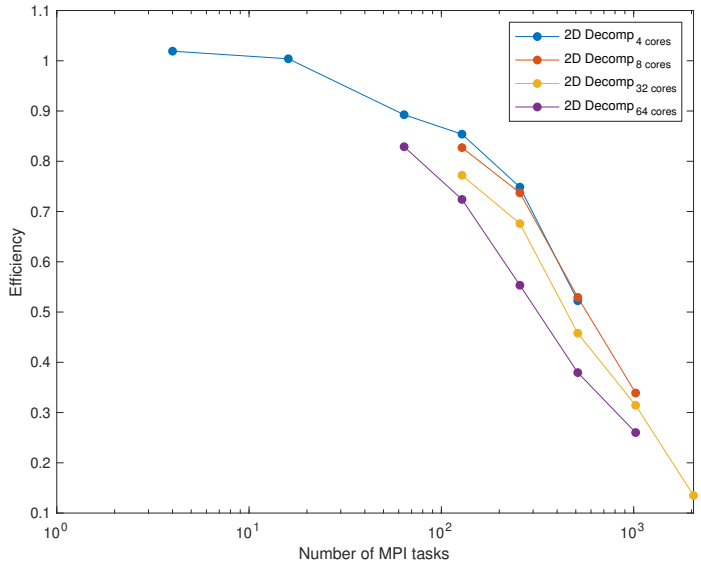

Figure 4.23: Efficiency using 2D decomposition for $256 \times 256 \times 512$ simulation with Intel Compiler 18

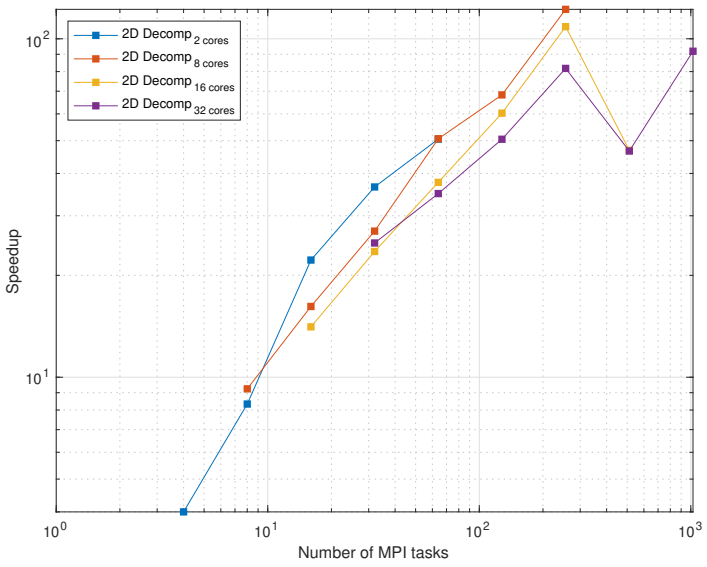

Figure 4.24: Speedup using 2D decomposition for $256 \times 256 \times 512$ simulation on GALILeO

The results, shown on figure 4.24 and 4.25, are not far from the baseline, suggesting that the code perform similarly. Although this more traditional architecture provide slightly better performances at cores equality, with a 5% gain in terms of efficiency, the highest clock frequency reduce the scaling range, moving the peak from 1024 cores towards 256. However, since we imposed the restriction of efficiency above the 40%, we can affirm that a less heavily threaded architecture provide better results. Furthermore, the absence of optimization leave rooms for further improvements in both terms of efficiency and speedup.

As in the previous chapter the Intel compiled program is unable to run on less than 4 cores, indeed we set reference time running the program using 4, single core, processors.

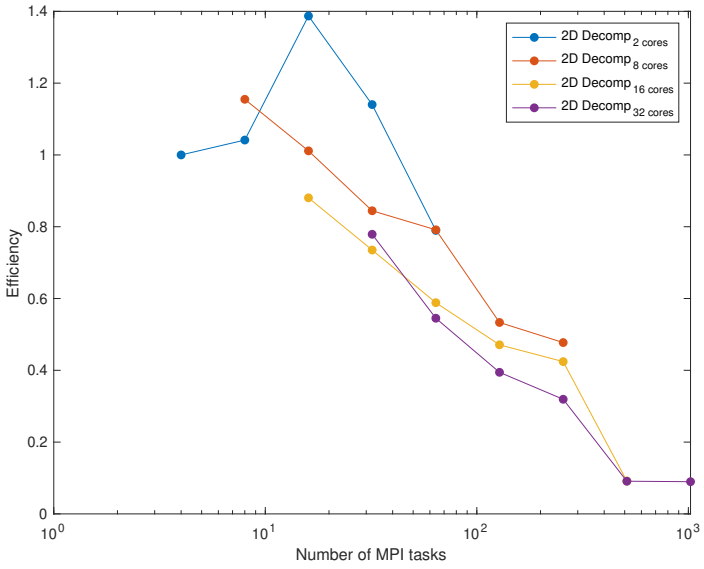

Figure 4.25: Efficiency using 2D decomposition for $256 \times 256 \times 512$ simulation on GALILeO

4.7 Benchmarks conclusions

The benchmark series has highlighted common trends present in our simulations. By looking at the curves, we can see that the performances envelope is bordered by the 64 cores run and the single core ones. We can catch from the graphs that the code tends to perform faster using as less threads per processor as possible. This behavior is reasonable, since our code and the library in which we rely on to perform the MPI transposition, does not implement the OpenMP technology at the moment, so, although we could carry on the simulation basing our communications on MPI, we will experience efficiency lacks when dealing with intra-node messagings. In particular, when dealing with many cores per processor, we face a speedup tendency to pass from 2 to $2^{2/3}$, as the number of MPI tasks gets doubled, as highlighted in [35].

Unfortunately, the Intel KNL's architecture [25] is designed to exploit code vectorization as much as possible and OpenMP plays a key role in this kind of implementations.
We suggest to refine the flags during the compilation phase in order to exploit the automatic code vectorization process, as highlighted in chapter 4.6.1, alternatively move to traditional processors architectures, instead of using MIC, to experience lower losses, as shown in chapter 4.6.2.
Our simulations in fact has highlighted that, although MPI can not guarantee high efficiency in heavily threaded applications, the lacks using 16 cores, or less, per processor can be acceptable.

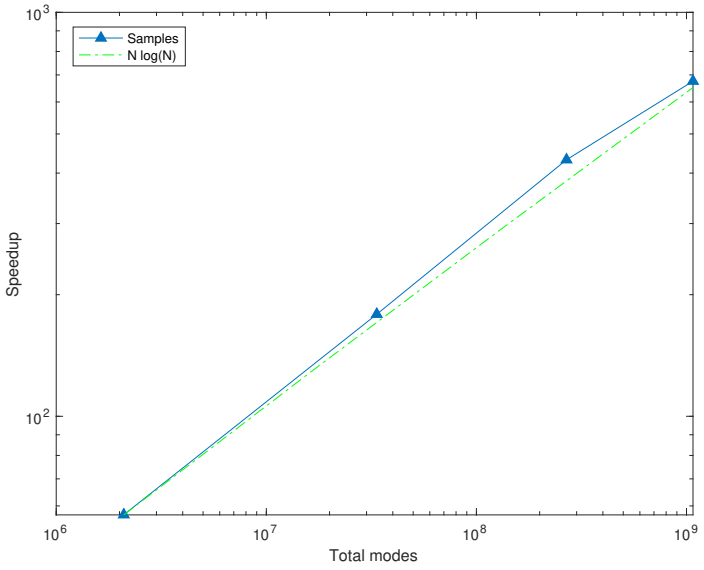

Figure 4.26: Speedup factors growth

As the problem size grows, the optimal number of MPI tasks, to achieve the best speedup, grows. In fact, we passed from 512 parallel processes for a 128^3 simulation to 2048 for a $512 \times 512 \times 1024$ simulation.
On the other hand, the speedup factor increase its peak in a fashion which lies on $N \log(N)$ curve, like testify by figure 4.26 in which our samples are plotted against such behavior.

Let us introduce now the speedup comparison with hyper threading turned on. Hyper threading is a technology developed by Intel [46] that virtually doubles the cores on the CPU, making the CPU run faster and more efficient by scheduling the workload between the cores. On modern Xeon Phi we can quadruplicate the number of cores, obtaining until 272 threads per processor. However, as our benchmark shows, this technology does not provide a boost in terms of speedup, on the contrary it penalizes our results in evident fashion. In figure 4.27 are compared the original results of a $512 \times 512 \times 1024$ simulation, running on different cores per processor, against two curves which exploit the hyper threading technology.

In conclusion we would like to show the cost for a single degree of freedom (DOF) in terms of CPU time. Such cost has been obtained considering the simulation time, the number of time steps required by the simulation and the total number of degrees of freedom, so it has to be intended as a mean value. The cost is defined as:

$$Time/DOF = \frac{SimTime}{timestep} \times \frac{1}{nx \times ny \times nz} \times tasks \times n_{cores} \qquad (4.1)$$

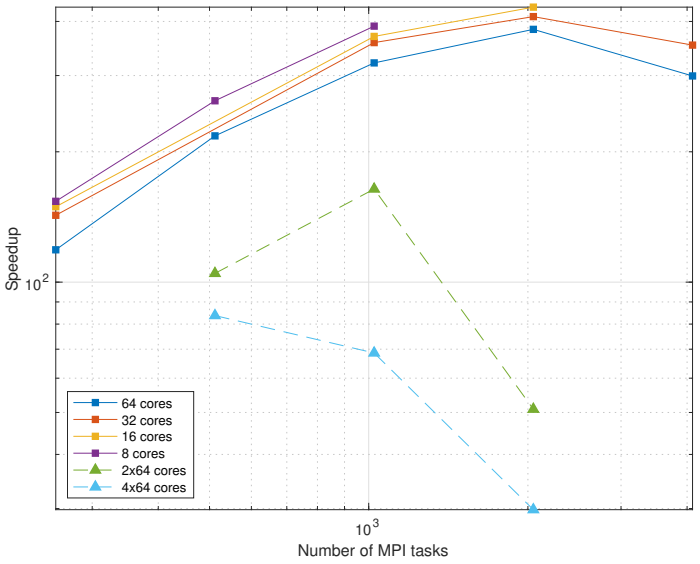

Figure 4.27: Hyper threading benchmark

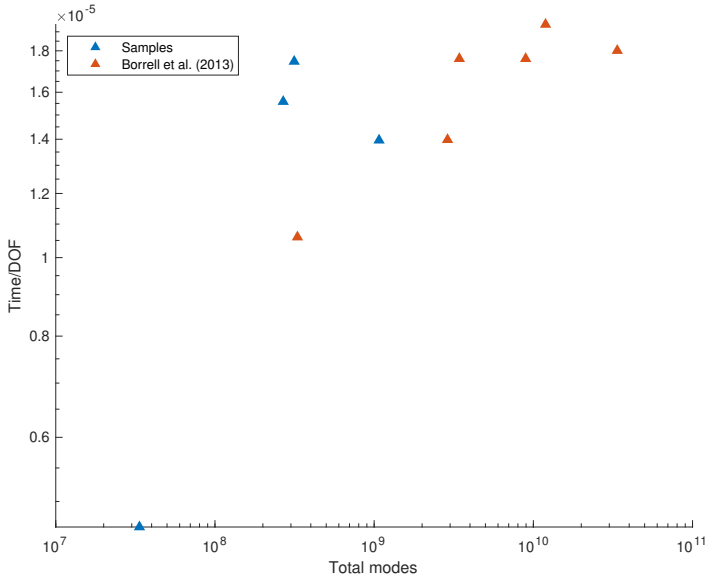

Figure 4.28: Qualitative comparison of Time-DOF ratio

Our results have been compared with the ones achieved in [7], where a boundary layer simulation is carried out on a flat plate, through direct numerical simulation, on JUGENE, a Blue Gene P architecture [34][3] installed at Juelich Forschungszentrum, with its 32,768 cores.
Since the problems compared are just similar we can only obtain qualitative informations from that. However, the results of figure 4.28 shows good fittings among the data.

5 Simulations Results

The present section present the statistics gathered from our simulations. In the first section we will compare our results with the ones of the famous KMM87, the founder research in this field. To follow we will show the results of a wider simulation, comparing the results against an analogue simulation of Moiser and Lee. The chapter concludes with an investigation of the Reynolds number effects on the statistics.

5.1 $Re_\tau = 180$ simulation

The $Re_\tau = 180$ simulation has been used to validate our results against the [36] ones.
The channel length is $4\pi\delta$, while the width is $2\pi\delta$, with δ indicating the half channel height.
Since our code employ spectral decomposition along the xy plane, we define the streamwise and spanwise periods as $L_x = 2\pi/\alpha_0$ and $L_z = 2\pi/\beta_0$, with α_0 and β_0 fundamental wavenumbers.
The simulation take place using constant pressure gradient, imposed along x direction, therefore $meanpx = 1$.
The mesh along y direction is non-uniform, discretized as

$$y(i) = y_{min} + \frac{1}{2}(y_{max} - y_{min})\frac{\tanh(a(2i/ny - 1))}{\tanh(a) + 0.5(y_{max} - y_{min})}$$

with $a = 1.6$.
The timestep is constant, with $dt = 0.0001$ and the simulation time is $T = 50$ non dimensional units. The grid employ 96 modes along x dimension, which, thanks to the Hermitian symmetry, behave as 192 points. The y is discretized using 128 points, while, along the z direction, we have 160 points. The total mesh size reach approximately 2 millions of grid points, decomposed across 512 cores, employed during the simulation.
The domain and the details about the simulation are summarized in table 5.1.

Table 5.1: Simulation data for Re_τ=180

L_x	L_z	δ	nx	nz	ny	α_0	β_0	Δx^+	Δz^+	px	dt	T
4π	2π	1	192	160	128	0.5	1	11.8	7	1	0.0001	50

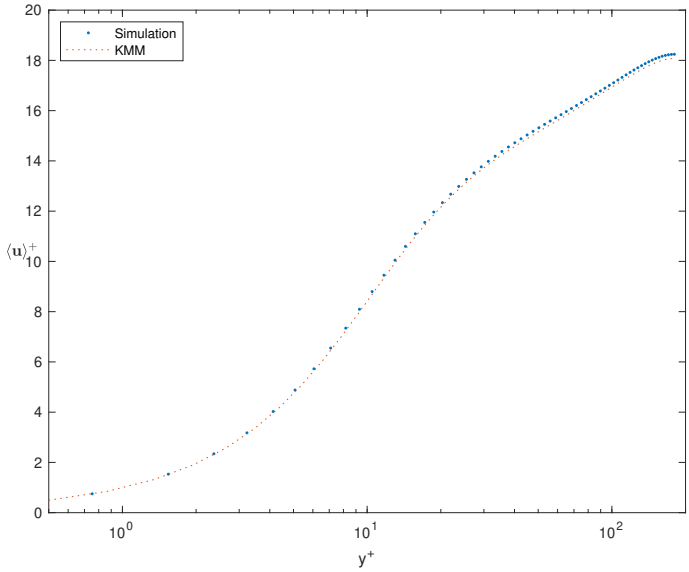

Figure 5.1: $\bar{u}^+$ in the near wall region for a $Re_\tau = 180$ simulation

The bulk mean velocity, defined as

$$U_m = \int_0^1 \bar{u} d(\frac{y}{\delta}) \tag{5.1}$$

normalized by the wall-shear velocity, is 15.66, which gives the Reynolds number based on the bulk mean velocity and the full channel width, $Re_b \approx 5600$.

The graphs 5.1 and 5.2 show the behavior of the mean velocity and the roots means squares in the near wall region, with $\bar{u}$ indicating the mean velocity profile.
In both figures we reported the *Kim et al.* results, using dotted line, for comparison.

Our statistics have been registered using a simulation time of 50 non dimensional units, sampling data every 0.1 steps. In total 500 fields have been used to perform the ensemble average.
The data fitting is good, despite being perfect. The divergences among our database and the [36] ones are possibly due to the fluctuations, rounding errors and differences in averaging times.

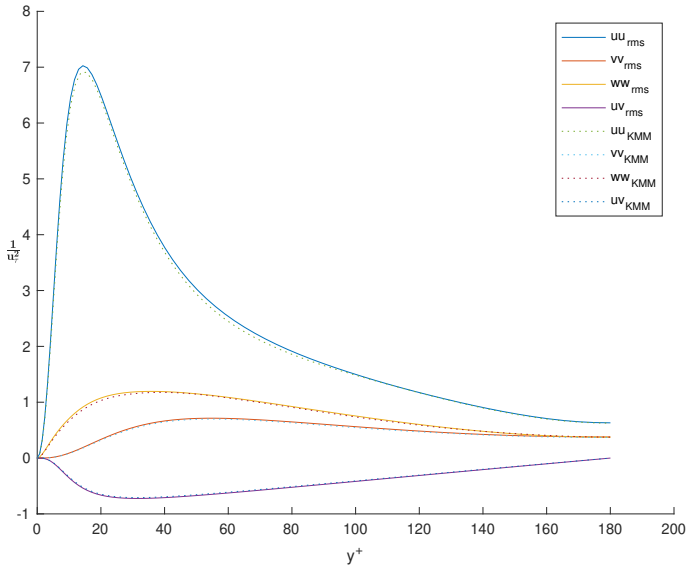

Figure 5.2: *rms* terms for a $Re_\tau = 180$ simulation

Nevertheless the results are in agreement with the typical curves behavior, in particular, by looking at the root mean square curves, we can clearly see that $\langle uu \rangle$ and $\langle ww \rangle$ depart from 0 as y^2, while $\langle uv \rangle$ and $\langle vv \rangle$ increase more slowly, as y^3 and y^4, in agreement with [56, p. 284]. All these information testify that, close to the walls, there is a *two component flow*, with $v = 0$ whereas u and w are non-zero. The resulting motion corresponds to flow in planes parallel to the wall.

The figure 5.1 report the $\bar{u}$ behavior near the wall. From 0 up to 5 y^+ units we can see the typical $\bar{u} = y^+$ behavior, which characterize the viscous sublayer. Once $y^+ > 30$ we see the arise of the logarithmic law of the wall, characterized by the equation

$$\bar{u}^+ = \frac{1}{\kappa} \ln y^+ + C^+,$$

where the constants $\kappa = 0.41$ and $C^+ \approx 5.2$, like smooth wall experiments, made by Von Karman, evidenced.
This velocity profile is typically denoted as *the law of the wall*, and has been postulated by Prandtl in 1925.
Far away from the wall, the implications of the previous law originate the so called *velocity defect*. Our experimental data fits the theory, as figure 5.3 suggest.

The turbulence intensity trend reported on figure 5.4 shows the comparison between the *rms*, normalized by u_τ, obtained by *Kim et al.* and our results,

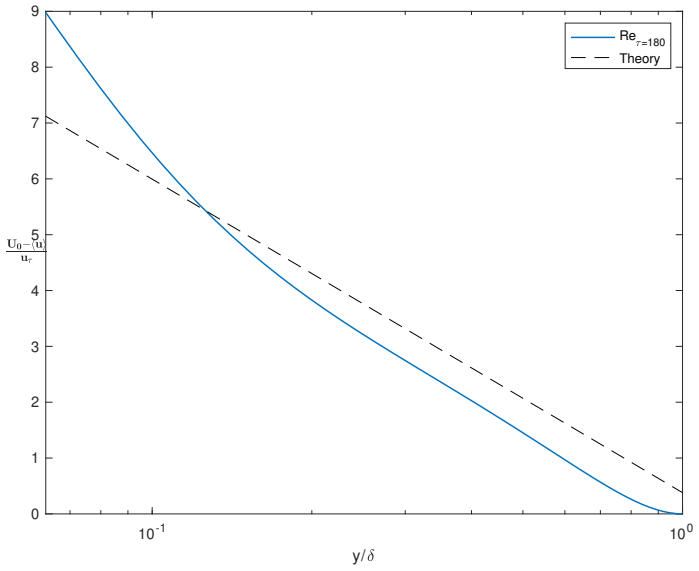

Figure 5.3: Velocity defect for a $Re_\tau = 180$ simulation

plotted against the wall-normal distance y^+. The fitting is good, particularly by approaching the centerline.
The maxima are located between the outer region of the buffer layer and the beginning of the log-law region: the streamwise u'/u_τ peak is positioned at $y^+ \approx 14$ with a value of $u'/u_\tau \approx 2.65$. The other two components show a smoother and less prominent behavior, with a $w'/u_\tau \approx 1.08$ around $y^+ \approx 38$ and $v'/u_\tau \approx 0.84$ for $y^+ \approx 50$.

A deeper knowledge of what happen close to the wall can be obtained by looking at figure 5.5. Such picture shows the behavior of the three *rms* components, normalized by the wall coordinate, for the first 9 wall units. In dashed line it is possible to see the data from *Kim et al.*
Once again the fitting between data is good, with both curves that follow the same trends. It is interesting to show that for the first wall units, in the viscous sublayer, the ratio u'/y^+ remains constant, indicating constant turbulence generation. It is quite flat also the w'/y^+ behavior, while the v'/y^{2+} exhibit a more slope trend.
The last curve is not in scale, the graph, indeed, shows a 10x magnified value, just for plotting purpose.

On page 64 is possible to look at the plot of the turbulent kinetic energy with the *rms* terms, while figure 5.7 shows the *production term*, defined as $P = -\langle u'v' \rangle \partial\bar{u}/\partial y$.
The two images highlight that the energy associated with the turbulence tends to develop close to the walls, and lose effectiveness once departing from there.
The *production term* is part of the so called turbulent kinetic energy budgets

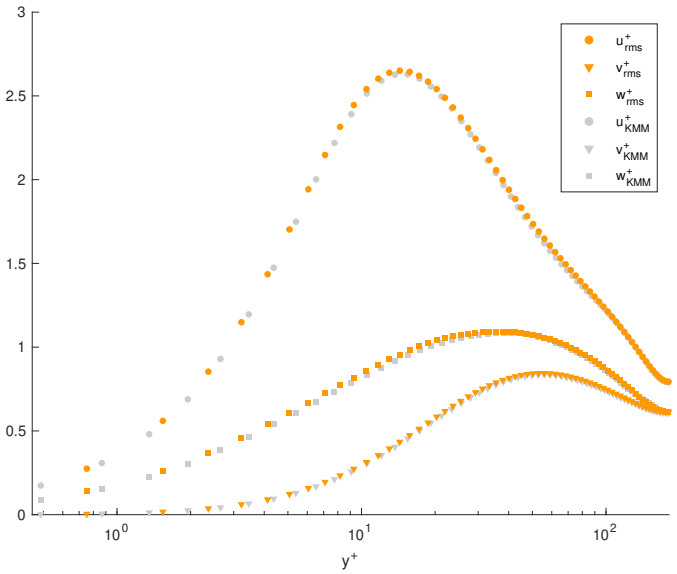

Figure 5.4: *rms* behavior on a $Re_\tau = 180$ simulation

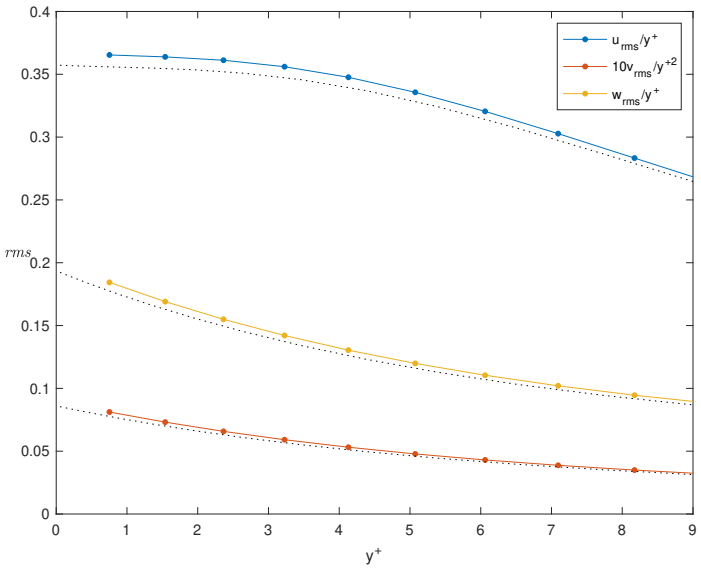

Figure 5.5: Normalized *rms* close to the wall for a $Re_\tau = 180$ simulation

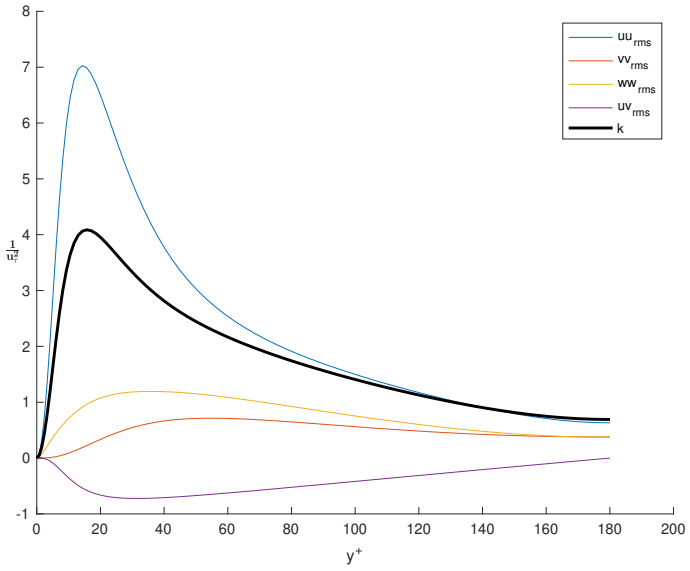

Figure 5.6: TKE and *rms* terms for a $Re_\tau = 180$ simulation

and plays a key role in the interaction of the mean energy equation and TKE. The action of the mean velocity gradients working against the Reynolds stresses removes kinetic energy from the mean flow and transfers it to the fluctuating velocity field.
By looking at figure 5.7 we can clearly see that its contribution concentrated near the walls, with the peak around $y^+ \approx 12$, and tends to become zero approaching the half channel height.

The *production* peak occurs where the Reynolds stresses becomes equal to the viscous ones. On figure 5.8 we reported the plot of the normalized total shear stress, with its contributions, in which we can see evidence of curves overlapping for $y \approx 12$.

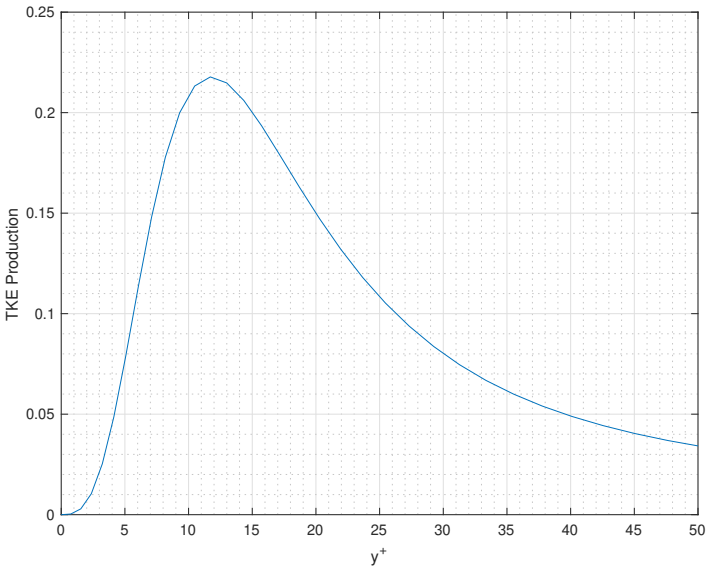

Figure 5.7: Production term of the TKE eq. normalized by $Re_\tau = 180$

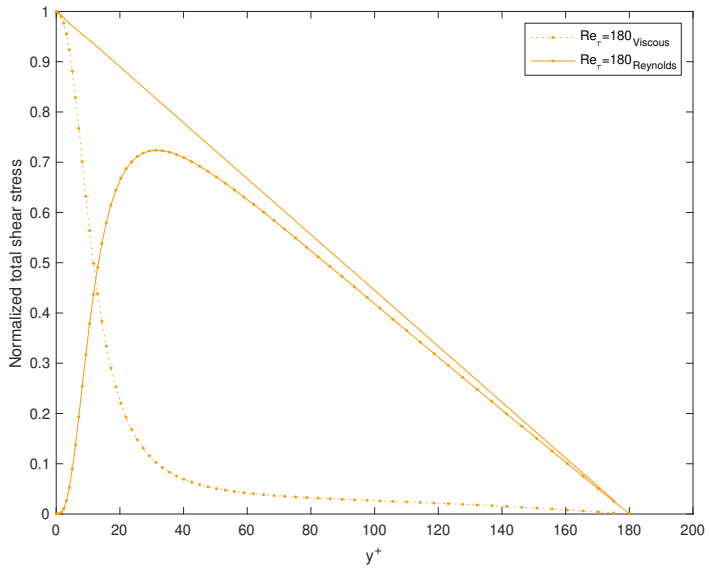

Figure 5.8: Normalized total shear stress for a $Re_\tau = 180$ simulation

5.2 $Re_\tau = 1000$ simulation

The second simulation performed is carried out at $Re_\tau = 1000$, which in terms of channel width and bulk velocity is equivalent to $Re_b \approx 40000$.
The bulk velocity, obtained as shown in 5.1, is 19.99, while α_0 and β_0 are respectively 0.5 and 1, in order to reproduce the correct dimensions of the channel, as shown in chapter 5.1.
Since the high computational cost needed to obtain these results we were unable to carry out a complete simulation. Thus the results reported show the statistics associated with a flow in a transitory state.

The simulation employed a variable timestep, determined through the Courant-Friedrichs-Lewy condition. Since its high computational cost, due to the needed transpositions and Fourier transformations, we decided to set the CFL limit below the stability threshold and calculate it once every n steps. In this way the gain in terms of performance is significant, combining the flexibility of the Courant-Friedrichs-Lewy condition with a reduced cost to compute it.

The grid employed in this simulation face 500 points in the wall-normal direction, 2048 in the spanwise direction and 2048 points along the streamwise dimension, direction in which we exploit the Hermitian symmetry. According to this configuration, the grid size reach the billion of points.
Table 5.2 report a summary of the simulation configuration for the $Re_\tau = 1000$ case.

Since are required approximately 50GB of disk space per each field we decided to avoid to save them on disk, instead we calculated the statistics runtime, merging the files at the end of the simulation, reducing the required space to few KB.

The results that we will present shortly are obtained through the ensemble average of 100 fields, and cover 0.1 non dimensional time units. In this lapse of time the mean Re_τ is approximately 991 units and it is slowly moving towards the nominal value of the simulation.
Let us focus now on the statistics gained from the simulation.

Figure 5.9 report the law of the wall. As we can see from the plot, made in semi-logarithmic scale using the wall units, our data fits the theoretical curve throughout the logarithmic region, while, towards the centerline, a residual sensitivity to the initial conditions is still present and lead to few differences with the results obtained by Moser & Lee in [37].
Far from the wall, the velocity defect law shows acceptable agreement with our

Table 5.2: Simulation data for Re_τ=1000

L_x	L_z	δ	nx	nz	ny	α_0	β_0	Δx^+	Δz^+	px	CFL
4π	2π	1	2048	2048	500	0.5	1	6.1	3	1	1.6

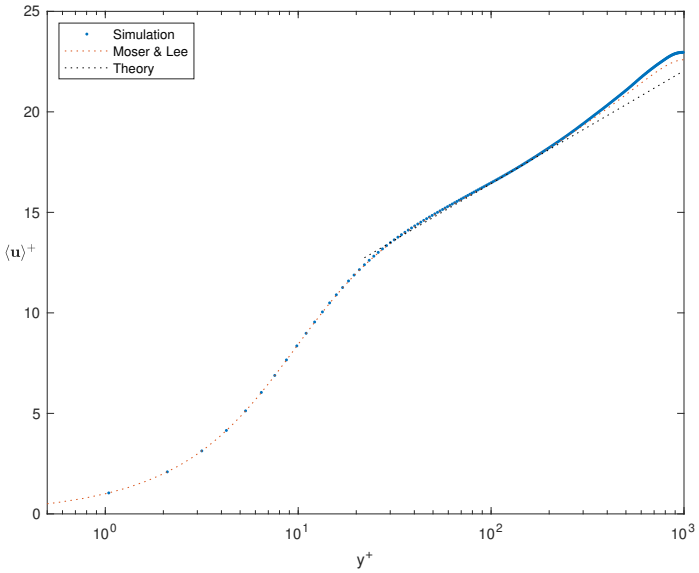

Figure 5.9: $\bar{u}^+$ in the near wall region for a $Re_\tau = 1000$ simulation

results, as figure 5.10 exhibit.

In figure 5.11 we reported the *rms* fluctuations, normalized by the u_τ^2, jointed with the TKE distribution. The first differences that we can immediately face by comparing our curves with the ones in figure 5.6 are the peak values. These values tends to increase with respect to the counterpart of the $Re_\tau = 180$ simulation, highlighting how this simulation contains more energy than the previous ones.
Although there is an higher content of energy, the curves shape remains aligned with the ones seen in the previous chapter.
The near-wall behavior present a two-components turbulent flow. In fact, as evidenced by the magnification, the wall-normal fluctuations are absent for the first few units.

The finer mesh and the higher Reynolds evidenced the appearance of a new turbulence peak, detached from the wall-cycle, identified through knees in the curves of figure 5.12.
In such figure our results are compared with the ones of Moser & Lee, which are the expected values for a complete $Re_\tau = 1000$ simulation.
The results shows accordance among the expected values and the ones obtained through the simulation. The u'/u_τ curve fits well the expect value, despite the little Re_τ difference among the two datasets. The v'/u_τ and w'/u_τ curves are in good accordance with the values of Moser & Lee in the inner region, while towards the centerline of the channel flow we face the raise of differences, possibly due to the transitory nature of the simulation and the dependency on the initial conditions.

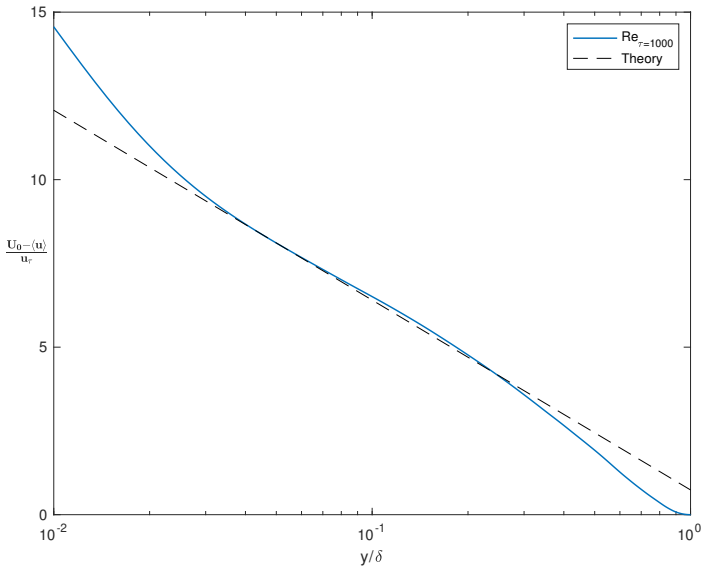

Figure 5.10: Velocity defect for a $Re_\tau = 1000$ simulation

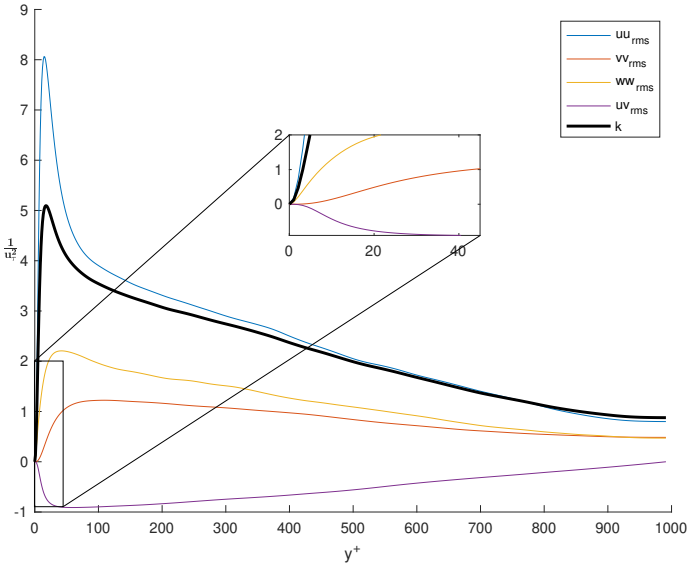

Figure 5.11: *rms* terms for a $Re_\tau = 1000$ simulation

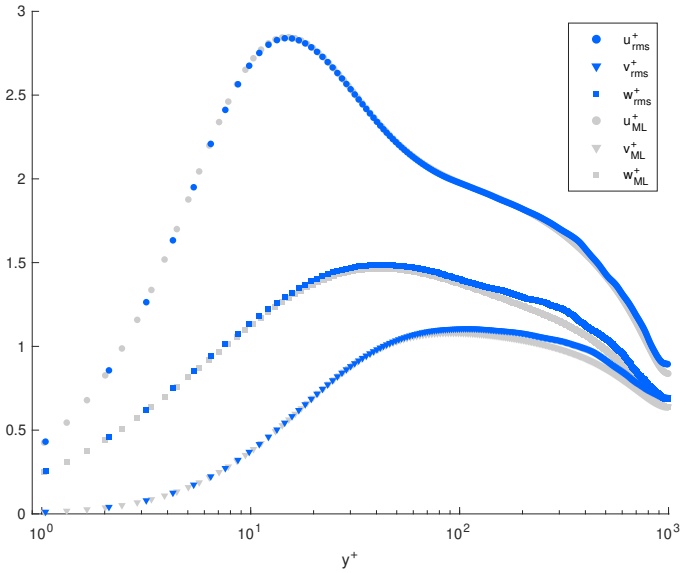

Figure 5.12: *rms* behavior on a $Re_\tau = 1000$ simulation

Comparing the results of this simulation with the ones of the $Re_\tau = 180$ we can clearly see a generalized upward shift of the *rms* fluctuations. Such trend is present also near the wall. Indeed the curves does not exhibit marked changes in shape with respect to their counterpart in the previous simulation, as figure 5.13 testify, just an upward shift, with the streamwise and spanwise components that depart from zero as y^+, while the wall-normal components leave the wall as y^{2+}.

Similar reasoning applies also for the graphs of the *production*, reported in figure 5.14, that reach a slightly higher peak of $P/Re_\tau = 0.24$, without showing significant changes of the curve shape. The peak is still located nearby $y \approx 12$.

As theory affirms, the wall coordinate of the peak of production corresponds to that in which the stress components become equivalent. This aspect will be investigated further, comparing the results of the two simulations together. At the present time we limit to illustrate the behavior of the stress components, which are reported in figure 5.15. As we can see, the normalized total shear stress is quasi-straight, suggesting that we are still in a transitory phase. The driving-train of this unsteadiness has to be searched in the excess of Reynolds stresses, which are forcing our flow towards higher Reynolds values.

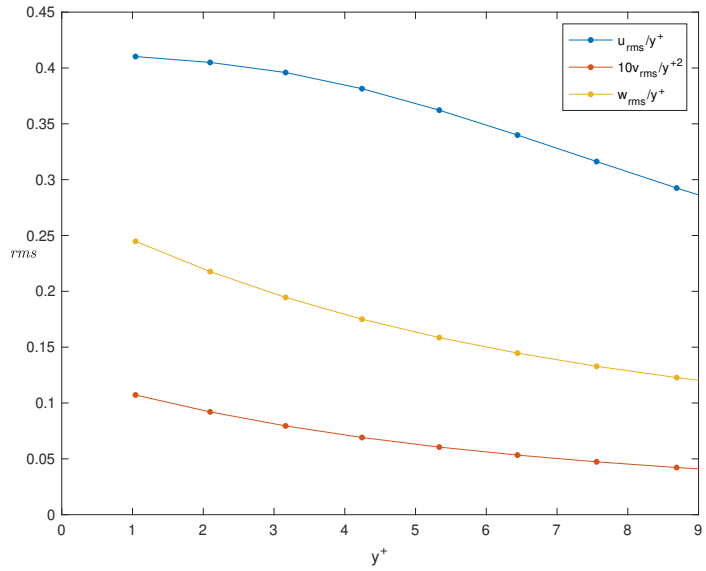

Figure 5.13: Normalized *rms* close to the wall for a $Re_\tau = 1000$ simulation

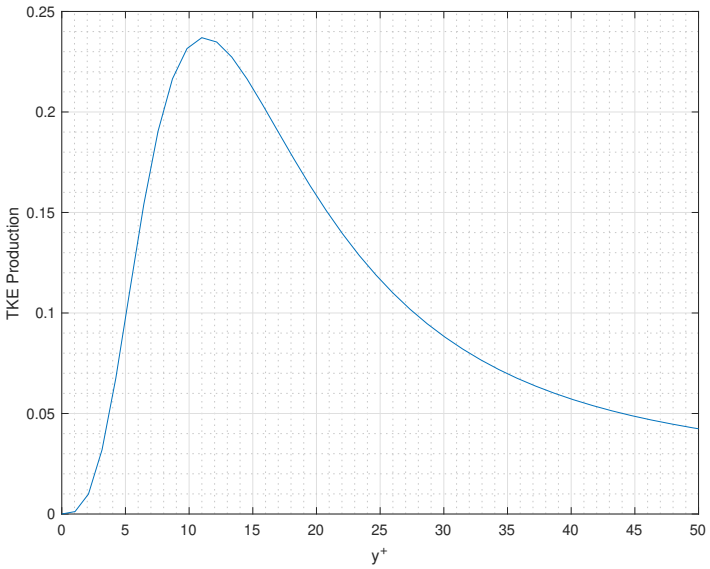

Figure 5.14: Production term of the TKE eq. for a $Re_\tau = 1000$ simulation

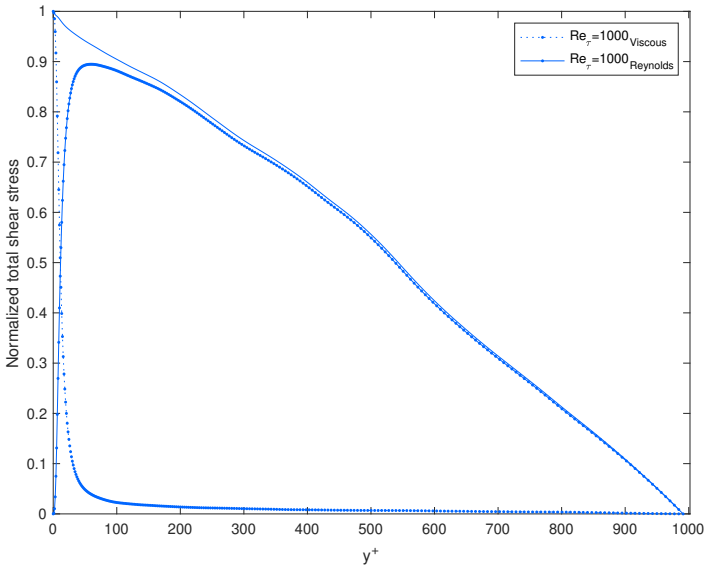

Figure 5.15: Normalized total shear stress for a $Re_\tau = 1000$ simulation

5.3 Reynolds effects

We are now interested to catch the effects of the Reynolds on our statistics through the comparison of our two simulations.

The first macroscopical effect that we face is a shift, towards higher values, of the mean velocity profile. The shadowed area of the graph 5.16 depict such situation. According to this figure, we can see a narrowing of the region subjected to high viscous stress, as the Reynolds number becomes larger. Under the mean velocity profile graph we reported the same quantity, but presented in semi-logaritmic scale, using both inner and outer scaling. The first plot of figure 5.17 correlates the results of the simulations and the theoretical behavior expectations. In particular we can see that, despite the Reynolds number, all the simulations present the linear $\bar{u} = y^+$ behavior expected in the viscous sublayer, and the logarithmic profile in the homonym region, with $k = 0.41$ and $B = 5.2$.

Figure 5.18 shows how the shear stress components modify as the Reynolds number increase.
Focusing on the normalized Reynolds stress curve we can clearly see that, as the Re_τ increase, the region subjected to this kind of stress becomes larger, with the peak moving towards the wall. Such kind of stress is associated with the fluid turbulent motions, therefore it was expectable a raise of this components as we move towards a more turbulent flow.
On the other hand the contribute of the viscous stress, associate to $\partial\bar{u}/\partial t$, is maximum at the wall and tends to become negligible as we move towards the

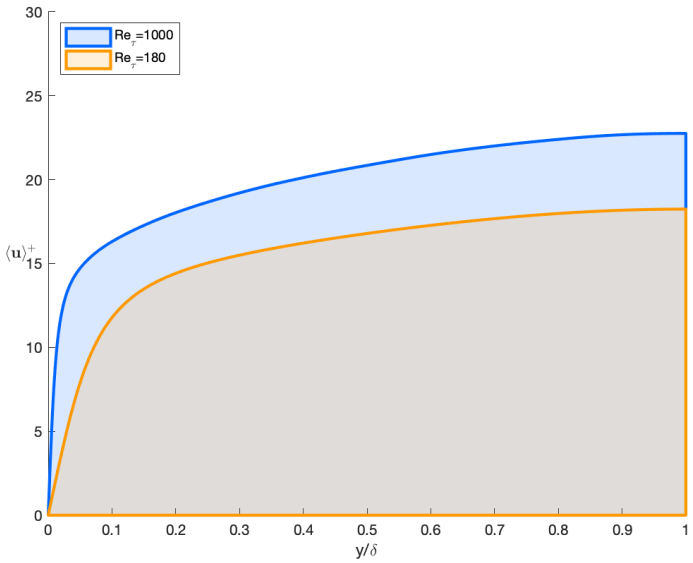

Figure 5.16: Mean velocity profile at Re_τ variation

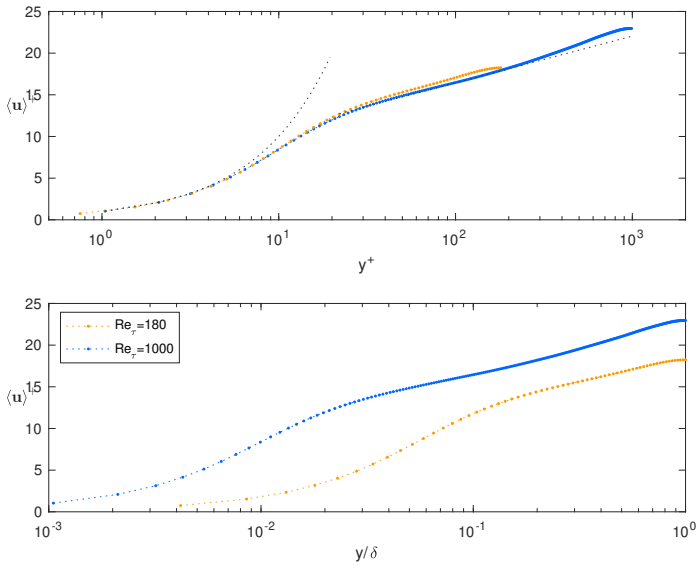

Figure 5.17: The law of the wall, in inner and outer scaling, at Re_τ variation

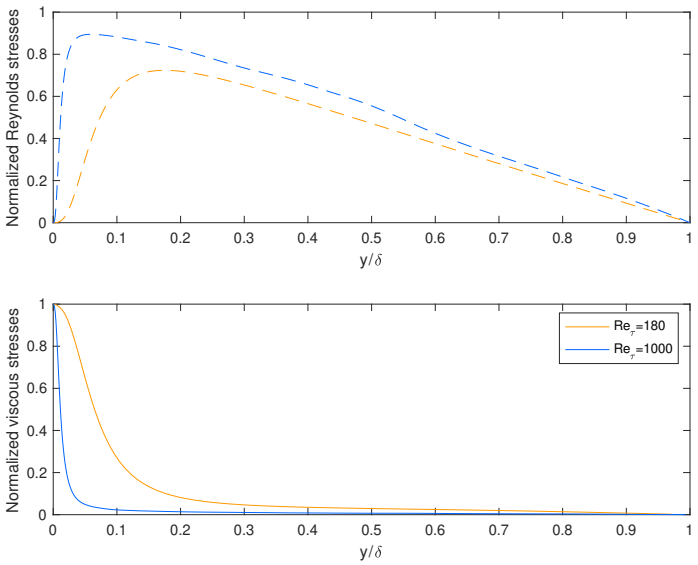

Figure 5.18: Normalized shear profiles at Re_τ variation

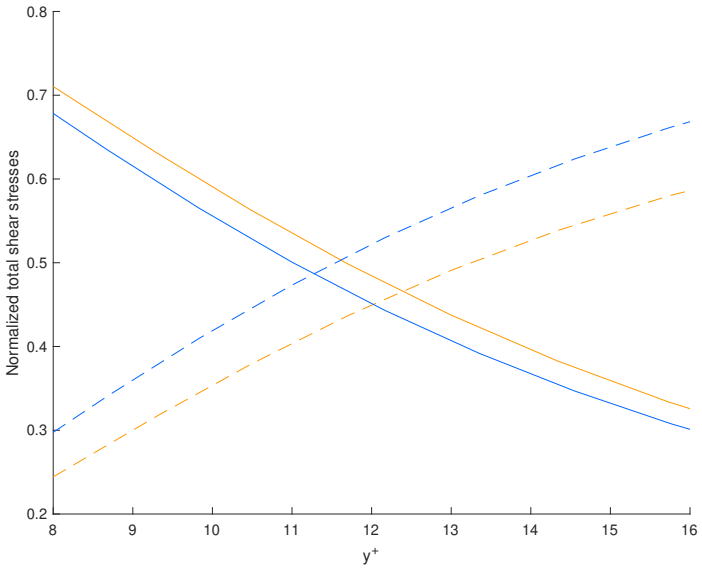

Figure 5.19: Particular of the shear stress, at Re_τ variation

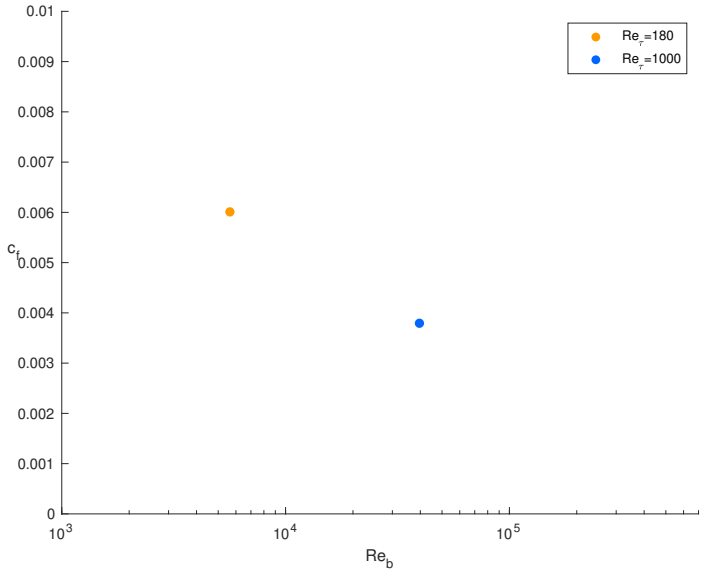

Figure 5.20: Dependance of the c_f from Re_b

centerline.
As testified by figure 5.17, the higher is the Reynolds and the wider is the area subjected to logarithmic profile, hence smaller is the area subjected to strong variation of the mean velocity profile with the wall-normal coordinate. This fact reflects on our viscous shear component reducing its range of effectiveness to few units, close to the wall, as the Re grows.
Although, in terms of outer scaling, the stress components are subjected to strong variations, in inner scaling we can see, in figure 5.19, that the point at which the two components cross themself remains quite constant, with $y^+ \approx 12$ wall-units.

One of the most important flow property for wall bounded flows is the friction coefficient. The c_f has been studied in detail by many famous authors of the past: Nikuradse, Prandtl, Blasius just to cite some of them.
In our simulations we computed the skin friction coefficient through the definition provided by [56, p. 279], which is based on centerline velocity (U_0) and the Re_b of the channel. The quantities have been defined as

$$c_f = 2(\frac{u_\tau}{U_0})^2 \qquad\qquad Re_b = \frac{2U_b\delta}{\nu},$$

and the results have been reported on figure 5.20.
Our c_f shows good fitting with the results of the experimental campaign of Dean reported in [11].

Let introduce now the comparison among the *rms* terms.

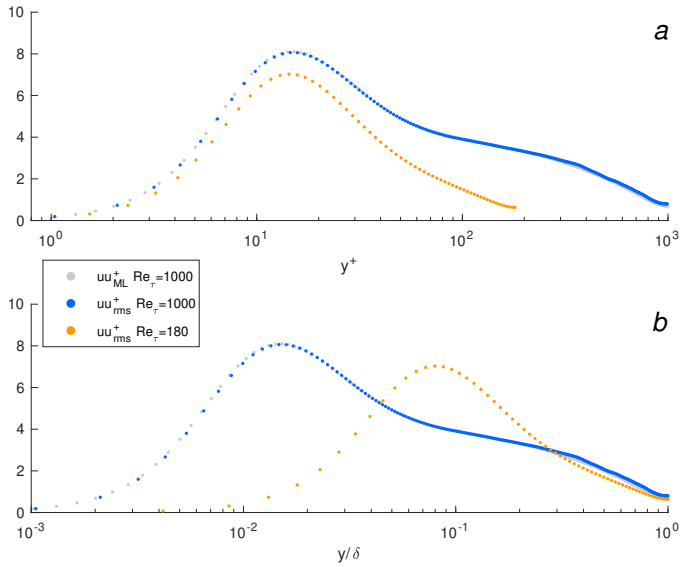

Figure 5.21: Streamwise fluctuations as function of the distance from the wall, plotted at Re_τ variation

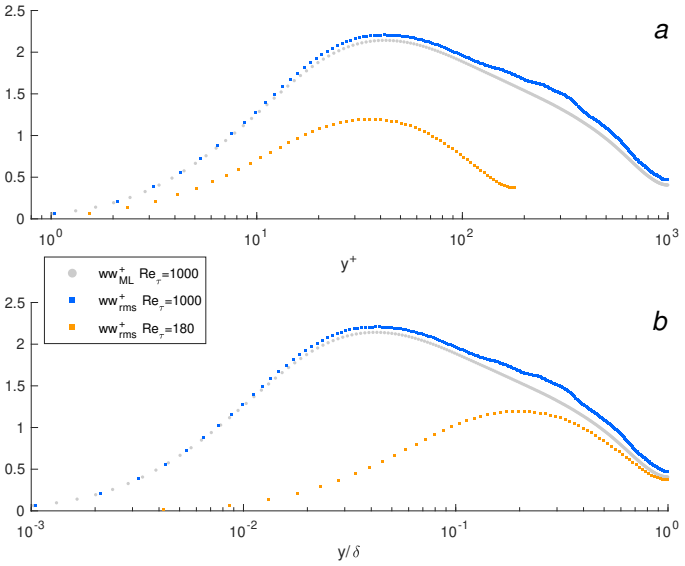

Figure 5.22: Spanwise fluctuations as function of the distance from the wall, plotted at Re_τ variation

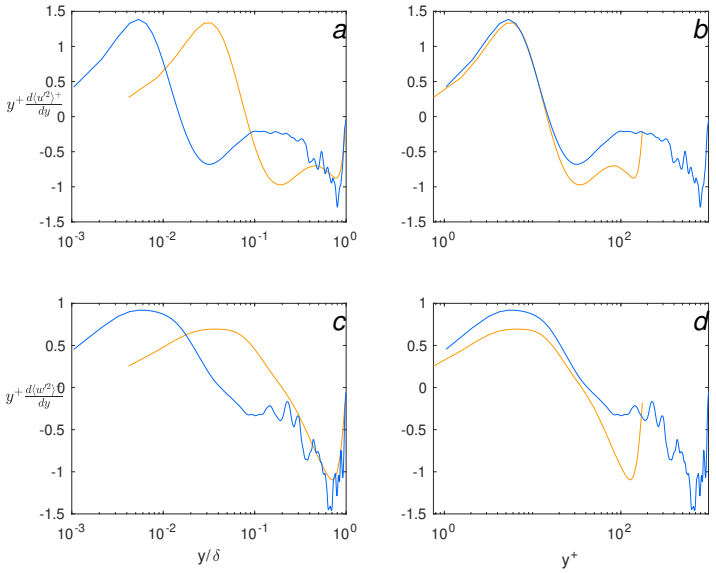

Figure 5.23: Spanwise and streamwise logarithmic region predictor. For legend refer to 5.21

Figure 5.22 shows the fluctuations of the spanwise term. As we can catch from the first plot, the raise of the Reynolds number, aside of the shift towards higher values, have a crucial effect on the trailing profile of the curve. Indeed we can appreciate the born of a logarithmic region for w'^2 term. In order to identify such region, we have used an estimator function defined as $y\partial_y\langle w'^2\rangle$, suggested in [37]. This function has the peculiarity to exhibit a flat profile whereas a logarithmic behavior is present.
In figure 5.23d we can see that the blue profile, which indicates the $Re_\tau = 1000$ simulation exhibit a flat path around $y^+ \approx 100$, in agreement to what is shown in figure 5.22 and predicted in [37]. The analysis for $Re_\tau = 180$ simulation does not highlight similar behaviors.

For what concern the streamwise fluctuations, reported in figure 5.23a and b, the indicator function is never flat, therefore we do not expect logarithmic regions. However, passed the 100 wall-units, the u'^2 profile seems to approach such behavior. It is likely that an higher Reynolds simulation could find evidence of logarithmic region.

The same indicator function can be applied also to the mean profile for the same purpose, in this case it is a common practice to indicate the function with β. In figure 5.24 we reported the comparison against our two simulation and the simulation at $Re_\tau = 1000$ made by Moser and Lee, in inner and outer scaling. Aside the outer region in which our $Re_\tau = 1000$ simulation exhibit strong fluctuations, the fitting of the two results is good.
Comparing the behavior of the *rms* in the homogeneous directions,

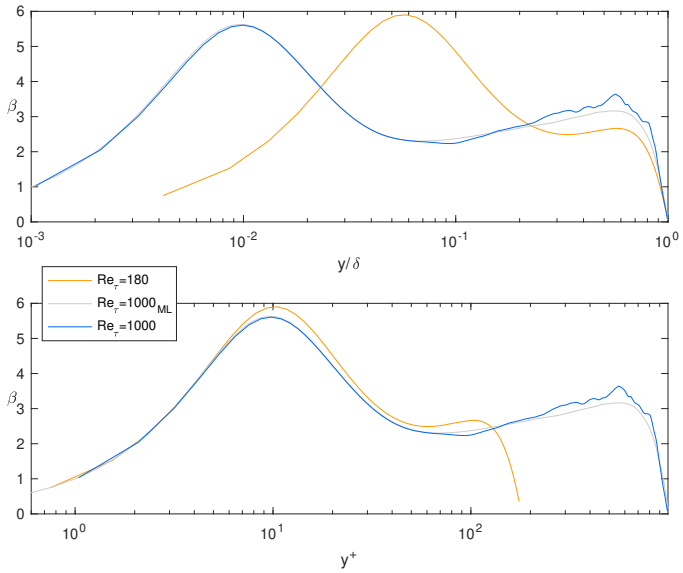

Figure 5.24: Logarithmic indicator function applied to mean profile $\langle u \rangle$

observing 5.21b and 5.22b, we can affirm that, while u'^2 exhibit a little upward shift in its values and a rigid shift towards the wall, w'^2 grows in all its aspects, with the peak that becomes more prominent and moves closer to the wall. The trailing part of the curve however is still higher than the peak of the $Re_\tau = 180$ simulation, enclosing it. To sum up we can affirm that the raise in Reynolds number seems to be more effective on the spanwise fluctuating term, instead of the streamwise ones.

The graph 5.25 shows the wall-normal fluctuations, expressed as always in function of y/δ and y^+.
As we can catch from the plot the blue curve exhibit higher values of fluctuations, distributed across a wider range of length scales than the orange ones. This expected behavior is associated to the higher turbulence content at which the flow is subjected.
A remarkable difference against figure 5.22a, 5.21a and figure 5.25a is the shift of the $Re_\tau = 1000$ peak towards higher values of y^+.

A similar peak trend can be recovered also in figure 5.26. Such figure report the product of the fluctuations among the streamwise and spanwise directions, which is directly involved in the processes of *production* and stress generation. The curves tend to raise their peak as the Reynold number becomes larger, thus increasing the already cited processes. The net movement of the peak towards the wall, shown in graph 5.26b, is responsible for the leftward movement of the normalized Reynolds stress of figure 5.18. Once again there are no evidence of logarithmic regions.

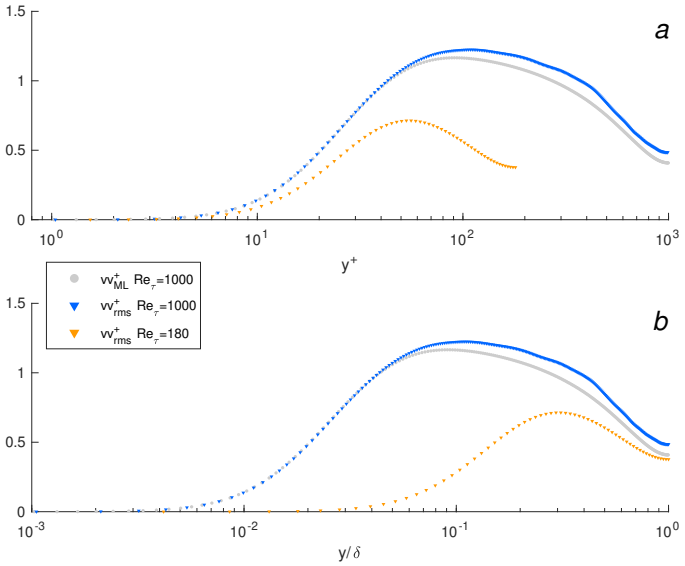

Figure 5.25: Wall-normal fluctuations as function of the distance from the wall, plotted at Re_τ variation

We summarized few quantities at Re_τ variation in table 5.3.

Table 5.3: Significant quantities at Re_τ variation

$\mathbf{Re_\tau}$	$\mathbf{Re_b}$	$\mathbf{U_b}$	$\mathbf{U_c}$	$\mathbf{u'^2_{peak}}$	$\mathbf{w'^2_{peak}}$	$\mathbf{v'^2_{peak}}$	$\mathbf{-u'v'_{peak}}$
180	5600	15.66	18.25	7.02	1.19	0.71	0.72
1000	40000	19.99	22.75	8.06	2.21	1.22	0.91

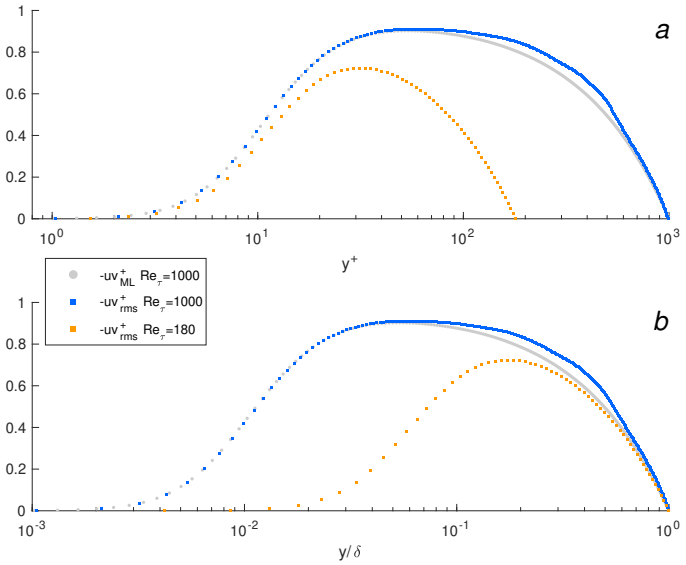

Figure 5.26: $-u'v'$ as function of the distance from the wall, plotted at Re_τ variation

6

Conclusions & Further Works

I am glad to say that we have reached our original goal to provide a scalable DNS solver through the usage of the MPI technology.
At present time the code lacks in an intra-nodal effectively parallelization, therefore its performances are limited by the MPI local communication performance. Despite of this the code revealed to be robust, being capable of working both with small datasets then extra large ones, exhibiting a linear gain in terms of productivity. The possibility to perform live post-processing of the data, instead of writing them, allows to save terabytes of memory, allowing the code to run also on networks of commodity hardware.

The fundamental restriction imposed by the original code about the number of parallel tasks has been removed, bringing the theoretical number of parallel processes to be limited by the product of $nx \times nz$ modes.

The engine developed is flexible and since is not affected by the geometry of the problem could be adapted quickly to carry out boundary layers simulations, just by imposing different boundary conditions, or can be used to solve pipe flows simulations.

The intent of this work was just to provide a study of feasibility for a solver based on pencil decomposition approach relying on inter-nodal parallelization, therefore we are satisfied by the results obtained. It should be denoted that, at present time, we cannot consider this as a "completed" solver. The lack of a dedicated shared memory parallelization reduce the efficiency significantly. However, we would like to highlight that a dedicated shared memory parallelization could be carried out just by changing few rows, without the needing of a significant reworking of the code. With today tendency of the HPC processors to increase the number of threads, instead of the number of physical CPU, this evolution, towards the so called *hybrid-programming*, seems mandatory.

Bibliography

[1] H. Abe, H. Kawamura, and Y. Matsuo. "Direct Numerical Simulation of a Fully Developed Turbulent Channel Flow With Respect to the Reynolds Number Dependence". In: *Journal of Fluid Engineering* 123 (June 2001).

[2] J. C. del Álamo and J. Jiménez. "Spectra of the very large anisotropic scales in turbulent channels". In: *Physics of Fluids* 15.6 (2003), pp. L41–L44. URL: https://aip.scitation.org/doi/abs/10.1063/.

[3] Gheorghe Almasi et al. "Overview of the IBM Blue Gene/P project". In: *IBM JOURNAL OF RESEARCH AND DEVELOPMENT* 52.1-2 (JAN-MAR 2008), 199–220. ISSN: 0018-8646.

[4] R. A. Antonia et al. "Low-Reynolds-number effects in a fully developed turbulent channel flow". In: *Journal of Fluid Mechanics* 236 (1992), pp. 579–605.

[5] M. Bernardini, S. Pirozzoli, and P. Orlandi. "Velocity statistics in turbulent channel flow up to $Re_\tau = 4000$". In: *Journal of Fluid Mechanics* 742 (2014), pp. 171–191.

[6] B. Blanc and B. Maaraoui. *Endianness or Where is Byte 0?* Tech. rep. 3B Consultancy, Dec. 2005. URL: http://3bc.bertrand-blanc.com/endianness05.pdf.

[7] G. Borrell, J. A. Sillero, and J. Jiménez. "A code for direct numerical simulation of turbulent boundary layers at high Reynolds numbers in BG/P supercomputers". In: *Computers & Fluids* 80 (2013). Selected contributions of the 23rd International Conference on Parallel Fluid Dynamics ParCFD2011, pp. 37–43. ISSN: 0045-7930. URL: http://www.sciencedirect.com/science/article/pii/S004579301200254X.

[8] E. Brachos. "Parallel FFT Libraries". University of Edinburgh, 2011.

[9] *CINECA*. URL: https://www.cineca.it/it/content/cineca.

[10] G. Comte-Bellot. "An introduction to turbulent flow". eng. In: *Journal of Turbulence* 2.1 (2001), pp. 83–84.

[11] R. B. Dean. "Reynolds Number Dependence of Skin Friction and Other Bulk Flow Variables in Two-Dimensional Rectangular Duct Flow". In: *Journal of Fluids Engineering* 100.2 (June 1978), pp. 215–223. URL: http://dx.doi.org/10.1115/1.3448633.

[12] J. C. Del Álamo et al. "Scaling of the energy spectra of turbulent channels". In: *Journal of Fluid Mechanics* 500 (2004), pp. 135–144.

[13] P. A. Durbin and B. A. Petterson-Reif. *Statistical Theory and Modeling for Turbulent Flows*. eng. Cambridge: Wiley, 2001.

[14] Message P Forum. *MPI: A Message-Passing Interface Standard*. Tech. rep. University of Tennessee Knoxville, TN, USA, 1994.

[15] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 2.2*. Sept. 209. URL: https://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf.

[16] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 3*. Sept. 2012. URL: https://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf.

[17] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 3.1*. June 2015. URL: https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf.

[18] M. Frigo and S. G. Johnson. "The Design and Implementation of FFTW3". In: *Proceedings of the IEEE* 93.2 (2005). Special issue on "Program Generation, Optimization, and Platform Adaptation", pp. 216–231.

[19] Matteo Frigo and Steven G. Johnson. *FFTW User Manual*. MIT. URL: http://fftw.org/doc/Advanced-distributed_002dtranspose-interface.html#Advanced-distributed_002dtranspose-interface.

[20] U. Frisch. *Turbulence: The Legacy of A. N. Kolmogorov*. Cambridge University Press, 1995.

[21] *Galielo: Cineca's Tier-1 system*. URL: http://www.hpc.cineca.it/hardware/galileo-0.

[22] The HDF Group. *Hierarchical Data Format, version 5*. http://www.hdfgroup.org/HDF5/. 1997-2019.

[23] S. Hoyas and J. Jiménez. "Scaling of the velocity fluctuations in turbulent channels up to $Re_\tau = 2003$". In: *Physics of Fluids* 18.1 (2006), p. 011702. URL: https://doi.org/10.1063/1.2162185.

[24] Z. Hu, C. L. Morfey, and N. D. Sandham. "Wall Pressure and Shear Stress Spectra from Direct Simulations of Channel Flow". In: *AIAA Journal* 44.7 (2006), pp. 1541–1549. URL: https://doi.org/10.2514/1.17638.

[25] Intel. *Intel Xeon Phi Product Spec*. Apr. 2016. URL: https://ark.intel.com/content/www/us/en/ark/products/94035/intel-xeon-phi-processor-7250-16gb-1-40-ghz-68-core.html.

[26] Intel. *Intel® Omni-Path Architecture Performance Tested for HPC*. July 2018. URL: https://www.intel.com/content/www/us/en/high-performance-computing-fabrics/omni-path-architecture-performance-overview.html.

[27] K. Iwamoto, N. Kasagi, and Y. Suzuki. "Direct Numerical Simulation of Turbulent Channel Flow at $Re_\tau = 2320$". In: *Proc. 6th Symp. Smart Control of Turbulence*. Tokyo, Japan, Mar. 2005.

[28] K. Iwamoto, Y. Suzuki, and N. Kasagi". "Reynolds number effect on wall turbulence: toward effective feedback control". In: *International Journal of Heat and Fluid Flow* 23.5 (2002), pp. 678–689. ISSN: 0142-727X.

[29] l. J. Clarke and E. Mark. "Enhancements to the eXtensible Data Model and Format (XDMF)". In: *2007 DoD High Performance Computing Modernization Program Users Group Conference*. June 2007, pp. 322–327.

[30] J. Jeffers. *Intel® Many Integrated Core Architecture: An Overview and Programming Models*. Mar. 2012. URL: `https://www.olcf.ornl.gov/wp-content/training/electronic-structure-2012/ORNL_Elec_Struct_WS_02062012.pdf`.

[31] J. Jiménez. "Computing high-Reynolds-number turbulence: will simulations ever replace experiments?" In: *Journal of Turbulence* 4 (2003), N22. URL: `https://www.tandfonline.com/doi/abs/10.1088/1468-5248/4/1/022`.

[32] A. Johansson and A D. Burden. "An Introduction to Turbulence Modelling". In: vol. 6. Jan. 1999, pp. 159–242.

[33] N. Kasagi, Y. Tomita, and A. Kuroda. "Direct Numerical Simulation of Passive Scalar Field in a Turbulent Channel Flow". In: *Journal of Heat Transfer* 114.3 (Aug. 1992), pp. 598–606.

[34] D. J. Kerbyson and A. Hoisie. "Performance Modeling of the Blue Gene Architecture". In: *IEEE John Vincent Atanasoff 2006 International Symposium on Modern Computing (JVA'06)*. Oct. 2006, pp. 252–259.

[35] Ali Khajeh-Saeed and J. Blair Perot. "Direct numerical simulation of turbulence using GPU accelerated supercomputers". In: *Journal of Computational Physics* 235 (2013), pp. 241–257. ISSN: 0021-9991. URL: `http://www.sciencedirect.com/science/article/pii/S0021999112006547`.

[36] J. Kim, P. Moin, and R. Moser. "Turbulence statistics in fully developed channel flow at low Reynolds number". In: *Journal of Fluid Mechanics* 177 (1987), pp. 133–166.

[37] M. Lee and R. D. Moser. "Direct numerical simulation of turbulent channel flow up to $Re_\tau \approx 5200$". In: *Journal of Fluid Mechanics* 774 (2015), pp. 395–415.

[38] S. K. Lele. "Compact finite difference schemes with spectral-like resolution". In: *Journal of Computational Physics* 103.1 (1992), pp. 16–42. ISSN: 0021-9991. URL: `http://www.sciencedirect.com/science/article/pii/002199919290324R`.

[39] N. Li and S. Laizet. "2DECOMP&FFT – A highly scalable 2D decomposition library and FFT interface". In: *Cray User Group 2010 conference*. Edinburgh, 2010.

[40] A. Lozano-Durán, O. Flores, and J. Jiménez. "The three-dimensional structure of momentum transfer in turbulent channels". In: *Journal of Fluid Mechanics* 694 (2012), pp. 100–130.

[41] A. Lozano-Durán and J. Jiménez. "Effect of the computational domain on direct simulations of turbulent channels up to $Re_\tau = 4200$". In: *Physics of Fluids* 26.1 (2014), p. 011702. URL: https://doi.org/10.1063/1.4862918.

[42] P. Luchini and M. Quadrio. "A low-cost parallel implementation of direct numerical simulation of wall turbulence". In: *Journal of Computational Physics* (2005).

[43] S. L. Lyons, T. J. Hanratty, and J. B. McLaughlin. "Large-scale computer simulation of fully developed turbulent channel flow with heat transfer". In: *International Journal for Numerical Methods in Fluids* 13.8 (1991), pp. 999–1028. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/fld.1650130805.

[44] K. Mahesh. "A Family of High Order Finite Difference Schemes with Good Spectral Resolution". In: *Journal of Computational Physics* 145.1 (1998), pp. 332–358. ISSN: 0021-9991. URL: http://www.sciencedirect.com/science/article/pii/S0021999198960223.

[45] *Marconi: Cineca's Tier-0 system*. URL: http://www.hpc.cineca.it/hardware/marconi.

[46] D. T. et al Marr. "Hyper-Threading Technology Architecture and Microarchitecture". In: *Intel Technology Journal* (Feb. 2002). URL: http://www.cs.virginia.edu/~mc2zk/cs451/vol6iss1_art01.pdf.

[47] J. J. McKetta. "Turbulence: An introduction to it's mechanism and theory (Hinze, J. O.)" In: *Journal of Chemical Education* 37.9 (1960), A556. URL: https://doi.org/10.1021/ed037pA556.

[48] P. Moin. "Direct numerical simulation: A tool in turbulence research". eng. In: *Annual Review of Fluid Mechanics* 30 (1998). ISSN: 00664189. URL: http://search.proquest.com/docview/220777817/.

[49] R. D. Moser, J. Kim, and N. N. Mansour. "Direct numerical simulation of turbulent channel flow up to $Re_\tau = 590$". In: *Physics of Fluids* 11.4 (1999), pp. 943–945.

[50] L. Q. Nguyen. *How to Install the Intel® Omni-Path Architecture Software*. Tech. rep. https://software.intel.com/en-us/articles/using-intel-omni-path-architecture: Intel, Mar. 2017.

[51] L. Null and J. Lobur. *The Essentials of Computer Organization and Architecture*. Jones and Bartlett Publishers, 2006. ISBN: 9780763737696. URL: https://books.google.com.au/books?id=QGPHAl9GE-IC.

[52] *OpenMP Application Programming Interface, version 5.0*. OpenMP Architecture Review Board. Nov. 2018. URL: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf.

[53] D. Pekurovsky. "P3DFFT: a framework for parallel computations of Fourier transforms in three dimensions". In: *SIAM Journal on Scientific Computing* 34.4 (2012), pp. C192–C209.

[54] M. Pippig. *PFFT User Manual*. Dec. 2018. URL: https://www-user.tu-chemnitz.de/~potts/workgroup/pippig/paper/PFFT_manual.pdf.

[55] S. Plimpton. *fftMPI*. Oct. 2018. URL: https://fftmpi.sandia.gov.

[56] S. B. Pope. *Turbulent flows*. Cambridge: Cambridge Univ. Press, 2011.

[57] A. Pozzi. *Application of Padé's Approximation Theory in Fluid Dynamics*. Advances in Mathematics for Applied Sciences. World Scientific, 1994.

[58] GCC Project. *GNU GCC*. Jan. 2018. URL: https://gcc.gnu.org.

[59] M. Quadrio and P. Luchini. "Integral space–time scales in turbulent wall flows". In: *Physics of Fluids* 15.8 (2003), pp. 2219–2227. ISSN: 1070-6631.

[60] O. Reynolds. "An Experimental Investigation of the Circumstances Which Determine Whether the Motion of Water Shall Be Direct or Sinuous, and of the Law of Resistance in Parallel Channels. [Abstract]". In: *Proceedings of the Royal Society of London* 35 (1883), pp. 84–99. ISSN: 03701662. URL: http://www.jstor.org/stable/114354.

[61] J. Rutledge and C. A. Sleicher. "Direct simulation of turbulent flow and heat transfer in a channel. Part I: Smooth walls". In: *International Journal for Numerical Methods in Fluids* 16.12 (1993), pp. 1051–1078. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/fld.1650161203.

[62] M. Tanahashi et al. "Scaling law of fine scale eddies in turbulent channel flows up to Re_τ=800". In: *International Journal of Heat and Fluid Flow* 25.3 (2004), pp. 331–340. URL: http://www.sciencedirect.com/science/article/pii/S0142727X04000074.

[63] M. Tanveer, I. M. Aqeel, and F. Azam. "Using Symmetric Multiprocessor Architectures for High Performance Computing Environments". In: *International Journal of Computer Applications* 27.9 (Aug. 2011). ISSN: 0975 – 8887. URL: https://pdfs.semanticscholar.org/3ba2/556d7f9f35edee8759f068fc96e05689460b.pdf.

[64] OpenMPI Team. *OpenMPI*. Oct. 2018. URL: https://www.open-mpi.org/doc/.

[65] OpenMPI Team. *OpenMPI FAQ*. Oct. 2018. URL: https://www.open-mpi.org/faq/?category=general.

[66] L. H. Thomas. "The Stability of Plane Poiseuille Flow". In: *Phys. Rev.* 91 (4 Aug. 1953), pp. 780–783. URL: https://link.aps.org/doi/10.1103/PhysRev.91.780.

[67] *TOP500 List*. URL: https://www.top500.org/list/2018/.

[68] A. W. Vreman and J. G. M. Kuerten. "Statistics of spatial derivatives of velocity and pressure in turbulent channel flow". In: *Physics of Fluids* 26.8 (2014), p. 085103. URL: https://doi.org/10.1063/1.4891624.