

Eigendecomposition-based Poisson Solvers for Fluid Flow Simulations in Non-uniform Grids

Duarte Palancha

Eigendecomposition-based Poisson Solvers for Fluid Flow Simulations in Non-uniform Grids

by

Duarte Seixas Carlos Palancha

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Monday March 9, 2026 at 10:00.

Student number: 5917980
Project duration: April 7, 2025 – March 9, 2026
Thesis committee: Dr. Pedro Costa, TU Delft, supervisor
Dr. Bernat Font, TU Delft, committee member
Dr. Ir. Mathieu Pourquoi, TU Delft, committee member

This thesis is confidential and cannot be made public until March 9, 2026.

Cover: Surface plot of a turbulent square duct flow calculated with the GEMM-based algorithm implemented in the present work.

An electronic version of this thesis is available at <http://repository.tudelft.nl/>.

Abstract

Fundamental research on fluid dynamics allows us to progress in our technological endeavours, ensuring safety in our daily lives. Understanding the underlying mechanics of turbulence is a challenge due to its unsteady, chaotic, three-dimensional structure. Mathematically, this is portrayed in the continuity and Navier-Stokes equations.

In incompressible flows, the velocity field must satisfy a global, instantaneous divergence-free constraint. The fractional-step method imposes this by solving a Poisson equation for the pressure field. The present work aims at formulating, implementing and evaluating an eigendecomposition-based framework for solving the discretized pressure Poisson equation in direct numerical simulation for non-uniform meshes.

The eigendecomposition-based approach was implemented into *CaNS* [1], a CPU- and GPU-based code which solves the discretized pressure Poisson equation in a massively-parallel fashion through an FFT-based approach on a finite-difference scheme. It was validated against lid-driven cavity and turbulent square duct flows, showing good agreement with the data of Gavrilakis [2].

The CPU-based computational performance of the framework was assessed through strong and weak scaling tests, and compared against results of *SNaC* [3], an implementation of an iterative solver for the pressure Poisson equation through geometric multigrid methods. It is shown that performance becomes communication-bound at 4096 cores, with wall-clock times higher than the FFT-based approach and lower than the multigrid approach. This bound has been hypothesized to be a consequence of the small computational domain tested (1024^3) and of possible memory fragmentation. In every test performed, regardless of scale, the eigendecomposition-based implementation is shown to have lower wall-clock times than the geometric multigrid implementation, despite its higher operation count.

Finally, the environmental and financial cost of developing the eigendecomposition-based approach was assessed. 525 thousand core hours were spent, representing ≈ 3000 kWh of energy, close to 1 ton of CO_2 emitted, $\approx 400\text{€}$ spent and over 160 hours of elapsed compute time. Deploying this approach in a large-scale environment for solving a turbulent square duct would require 300 to 400 thousand core hours.

All in all, the eigendecomposition-based approach here developed and implemented is deemed suitable for performing large-scale simulations on a many-CPU framework, with lower wall-clock times than comparable geometric multigrid methods. Unfortunately, in cases where a uniform mesh can discretize the problem, the FFT-based approach still demonstrates comparatively lower wall-clock times, meaning it should be the preferred approach where suitable. The present framework will be made available in a separate release of *CaNS* at github.com/CaNS-World.

Resumo

A investigação fundamental em mecânica dos fluidos permite-nos progredir nos nossos esforços tecnológicos, garantindo segurança nas nossas vidas diárias. É um desafio entender os mecanismos subjacentes da turbulência devido à sua estrutura transiente, caótica e tri-dimensional. Matematicamente, isto é retratado nas equações de continuidade e de Navier-Stokes.

Em escoamentos incompressíveis, o campo de velocidades tem de satisfazer a restrição de divergência zero de forma global e instantânea. O método de passo fracionário (*fractional-step method*) impõe esta restrição ao resolver uma equação de Poisson para o campo de pressão. O presente trabalho pretende formular, implementar e avaliar um método de resolução da equação discretizada de Poisson para a pressão em simulação numérica direta para malhas não uniformes com base na decomposição da matriz de discretização em valores e vetores próprios.

Esta abordagem foi implementada no *CaNS* [1], um código para CPUs e GPUs que resolve a equação discretizada de Poisson para a pressão em ambientes massivamente paralelos a partir de uma abordagem baseada em FFTs num esquema de diferenças finitas. Foi validada em escoamentos de uma cavidade com tampa móvel e de uma conduta quadrada com escoamento turbulento, concordando com os dados de Gavrilakis [2].

A performance computacional da implementação em CPU foi avaliada a partir de testes de escalabilidade forte e fraca e comparada com resultados do *SNaC* [3], uma implementação de um *solver* iterativo da equação de Poisson de pressão a partir de métodos geométricos multimalha (*multigrid*). Observou-se que a performance torna-se limitada pela comunicação entre processos a partir de 4096 núcleos de CPU, com tempos de execução superiores à abordagem baseada em FFTs e inferiores à abordagem baseada em métodos geométricos multimalha. Esta limitação pode ser consequência do pequeno domínio computacional testado (1024^3) e de possível fragmentação de memória. Em todos os testes realizados, independentemente da escala, a implementação baseada na decomposição em valores e vetores próprios aparenta ter tempos de execução inferiores à implementação de métodos geométricos multimalha, apesar de ter um número de operações superior.

Finalmente, o custo ambiental e económico de desenvolver a abordagem baseada na decomposição em valores e vetores próprios foi avaliada. 525 mil horas computacionais foram utilizadas, representando ≈ 3000 kWh de energia, perto de 1 tonelada de CO_2 emitido, $\approx 400\text{€}$ gastos e mais de 160 horas de tempo de execução. Esta abordagem exigiria entre 300 a 400 mil horas computacionais num ambiente de larga escala para resolver uma conduta quadrada de escoamento turbulento.

Em suma, a abordagem baseada na decomposição em valores e vetores próprios aqui desenvolvida e implementada é dada como adequada para realizar simulações em larga escala num ambiente de muitos CPUs, com tempos de execução inferiores aos métodos geométricos multimalha comparáveis. Infelizmente, em casos onde uma malha uniforme pode discretizar o problema, a abordagem baseada em FFTs demonstra tempos de execução inferiores, e assim deve ser escolhida sempre que possível. A abordagem aqui desenvolvida vai estar disponível num lançamento independente do *CaNS* em github.com/CaNS-World.

Acknowledgements

Although I am the singular author of this document, many were the people involved making sure I had all the resources required to make my best work.

Dr. Pedro Costa was instrumental throughout the development of this thesis. It was a pleasure working with someone who is clearly an expert in the topic, and that tries his best to ensure his knowledge is understood. Sharing our mother tongue allowed us to bond in unexpected ways in the most challenging of times, and I found comfort in his relentless positivity when things were not going as intended. I must also thank Giandomenico Lupo and Sanath Kotturshettar for their insights on getting myself accustomed to working with *CaNS* in HPC environments in the beginning of the project, many were the hours saved by their remarks. This work would not have been made possible without the computational time at the Dutch national supercomputer *Snellius*, made available by NWO Grant no. 2024/ENW/01704792, and for that I am thankful.

From a personal side, I must devote this thesis to Mafalda¹, who is someone truly special in every meaning of the word. She drove my heart in the moments where it needed the most, and calmed to my senses when the world seemed too much to handle. In the past five years I have had a partner, but most of all I have also had a best friend.

I must admit that sometimes my parents and sister were neglected throughout my coming to the Netherlands. Although I am saddened by this, it was with their support that I handled the most difficult challenges that making this document brought. They have made me realise that being a parent is not about discipline, its about strength and courage. Although we often fight about our conflicting opinions, it is always my mother's compassion and my father's bravery that keep our family tightly knit, tolerating, understanding and encouraging I follow my own path.

To my extended family, a special note must be given to my grandmother Albertina. Without her, my education would not have been made possible and being an engineer would not have been a reality. Despite this, what I admire the most in her is her ability to every single day display acts of friendship and comradeship. To the rest of my family, your presence has always been (and will always be) invaluable in my daily life, so I thank you for the sense of community you have instilled in me.

To my friends, words cannot describe how they have helped me throughout these years. In Delft, I have made friendships that I will carry with me for the rest of my life, with Doruk, Orpheas, Amedeo and Bernardo being those closest to my heart. From Portugal, many were the ones rooting for me in this process, emphasizing Rodolfo and Rodrigo Amorim, Pedro Cordeiro, Miguel André and Eduardo Couto.

Finally, I must recall two important members of my family. Dr. José Bioucas-Dias was someone who left an undeniable legacy. To this day, his work inspires many members in our family to chase the unknown and to explain the unexplainable, driving my own curiosity to be as passionate as he was. Complementing him, my grandfather João was someone who watched me grow up and become the man I am today. Too many are the memories I have of him smiling and cheering me, and that by itself is worth more than any description I could give of him.

Throughout making this thesis, many were the moments where everything felt a bit overwhelming. Some days, the anxiety provoked by feelings of imposter syndrome did not allow me to work in the way I wanted, with the joy that working on this topic usually gave me. It was the efforts of the people mentioned that really made this document a reality, and whose relationships I now value more than any scientific result.

*Duarte Seixas Carlos Palancha
Delft, March 2026*

¹Housemates might also call her "boo".

Nomenclature

Abbreviations

Abbreviation	Definition
AI	Artificial Intelligence
BC	Boundary Condition
CFL	Courant–Friedrichs–Lewy
CG	Conjugate Gradient (solver)
CPU	Central Processing Unit
CR	Cyclic Reduction
CFD	Computational Fluid Dynamics
DFT	Discrete Fourier Transform
DNS	Direct Numerical Simulation
DST/DCT	Discrete Sine/Cosine Transform
DT	Discrete Transform (operator)
FA	Fourier Analysis
FDM	Finite Difference Method
FEM	Finite Element Method
FFT	Fast Fourier Transform
FT	Fourier Transform
FVM	Finite Volume Method
GEMM	General matrix–matrix multiplication
GPU	Graphics Processing Unit
HPC	High-Performance Computing
JHTDB	Johns Hopkins Turbulence Database
LES	Large Eddy Simulation
LLM	Large Language Model
MAC	Marker-and-Cell (grid)
MDA	Matrix Decomposition Algorithm
MG	Multigrid (method)
MPI	Message Passing Interface
NS	Navier–Stokes (equations)
OpenACC	Open Accelerators (programming standard)
PDE	Partial Differential Equation
RANS	Reynolds-Averaged Navier–Stokes
RHS	Right-hand side
RK3	3rd-order Runge–Kutta time-stepping
SBU	Standard Billing Units
TDMA	Tridiagonal Matrix Algorithm (Thomas algorithm)

Symbols and operators

Symbol	Meaning
$\otimes$	Tensor (Kronecker) product
∇	Gradient operator
$\nabla \cdot ()$	Divergence operator
∇^2	Laplacian operator
$\frac{\partial}{\partial t}$	Time derivative
$\Delta()$	Finite difference / increment (context-dependent)
$\text{diag}()$	Diagonal operator
Ω	Physical domain
x, y, z	Spatial coordinates
X_k	Coordinate in direction k
ℓ_k	Domain length in direction k
i, j, k	Grid indices
$N_x, N_y, N_z/\text{ng}$	Number of grid cells in x, y, z /Computational representation
N	Total number of unknowns / equations ($N = N_x N_y N_z$)
$\Delta x, \Delta y, \Delta z$	Grid spacing
t	Time
Δt	Time step
n	Time-step index
$\mathbf{u}$	Velocity vector field
u, v, w	Velocity components
p	Pressure field
ϕ	Correction pressure (projection / Poisson solve)
$\mathbf{f}$	Body-force term (Navier–Stokes) / RHS forcing (context-dependent)
ρ	Fluid density
μ	Dynamic viscosity
ν	Kinematic viscosity
C	Courant number
F	Fourier number
AD^n	Advection–diffusion term at time level n
$\mathbf{u}^*$	Prediction (intermediate) velocity
Re	Reynolds number
Re_τ	Friction Reynolds number
u_τ	Friction velocity
U_b	Bulk velocity
U_L	Lid velocity (lid-driven cavity)
h	Characteristic length scale (problem-dependent)
y^+	Wall unit (dimensionless wall distance)
Δy_m	First-cell spacing near the wall
A	Discrete Laplace operator matrix (3D)
T	1D discrete Laplace operator (tridiagonal)
I	Identity matrix
Q	Eigenvector matrix
Λ	Diagonal matrix of eigenvalues
D	Similarity-transform diagonal matrix

Symbol	Meaning
δ_i	Entries of $D = \text{diag}(\delta_1, \dots, \delta_N)$
a_i, b_i, c_i	Sub-/main-/super-diagonal coefficients of a tridiagonal stencil
gr	Grid-stretching parameter
R	Ratio of maximum to minimum grid spacing
$dims$	Processor grid
np	Number of processes/cores

Contents

Abstract	iii
Resumo	v
Acknowledgements	vii
Nomenclature	ix
Abbreviations	ix
Symbols and operators	x
1 Introduction	1
1.1 Outline	2
2 Background	3
2.1 Time discretization	3
2.1.1 Application to governing equations	4
2.2 Spatial discretization	5
2.3 Linear solvers	5
2.3.1 Iterative methods	6
2.3.2 Direct methods	6
2.4 Parallelization	8
3 Objective and Scope of Thesis	11
4 Numerical Methods	13
4.1 Discretizing the pressure Poisson equation	13
4.2 Solving the linear system of equations	14
4.2.1 Designing an operator	14
4.3 Semi-implicit diffusion	17
4.4 Operation count and memory footprint	18
4.4.1 Summary	18
5 Implementation	21
5.1 Obtaining the eigendecomposition	22
5.2 Computational algorithm	24
5.2.1 CPU-based implementation	24
5.2.2 GPU-based implementation	26
5.3 Verification & Validation	26
5.3.1 Lid-driven cavity	27
5.3.2 Turbulent square duct	28
6 Results	31
6.1 CPU-based Computational Performance	31
6.1.1 Scaling Tests	31
6.2 Alternative Methods - Multigrid	36
6.2.1 Precision and tolerance effects	36
6.2.2 Single-core performance	37
6.2.3 Multi-core performance	38
6.3 Environmental and capital cost of simulations	40
7 Conclusion	43
8 Recommendations	45

References	57
A Similarity transformation of $\mathbf{T}_{x_j}$	67
B Eigendecomposition proofs	69
B.1 Eigendecomposition of a sum of tensor products	69
B.2 Linearity of eigendecomposition	70
C Describing non-uniform grids	71
D Calculating environmental and financial costs	73
D.1 Estimating the cost of a mock simulation	74

List of Figures

2.1	Modified wavenumbers of finite-difference schemes for different grid types.	5
5.1	Diagram of the <i>pencil</i> -like domain decomposition and transposes utilized in <i>CaNS</i>	22
5.2	Timings for obtaining the eigendecomposition of $\mathbf{T}$ numerically with respect to the number of grid cells.	22
5.3	Visualization of the simulation of the turbulent square duct.	27
5.4	Distributions of the velocity components along the cavity centrelines for the lid-driven case at $Re = 1000$	27
5.5	Time evolution of the Reynolds friction number for the turbulent square duct.	28
5.6	(a): Turbulent square duct mean flow represented by streamwise-velocity contours and in-plane velocity vectors. (b): Mean streamwise velocity profile sampled along the duct diagonal.	29
6.1	Strong scaling test of <i>CaNS</i> for all algorithmic configurations up to 16384 cores.	32
6.2	Timings of the solver sub-routine of <i>CaNS</i> for the cases of FFT + FFT and MD + MD.	33
6.3	Weak scaling test of <i>CaNS</i> for all algorithmic configurations with domain growth in x and y and up to 2048 cores.	34
6.4	Fitting of theoretical operation count of each operator to timings obtained in the weak scaling test of fig. 6.3.	35
6.5	Precision analysis of <i>SNaC</i> from trial simulations of a lid-driven cavity of domain size 64^3	37
6.6	Multi-core comparison for a lid-driven cavity with domain size of 64^3	39
6.7	Strong scaling test of direct and iterative methods up to 16384 cores.	39
6.8	Weak scaling test of direct and iterative methods up to 2048 cores.	40

List of Tables

6.1	Single-core comparison between direct (FFT and MD) and iterative (MG) methods for a lid-driven cavity.	37
6.2	Environmental and financial cost analysis of the tests performed for the present work.	41
6.3	Estimated environmental and financial cost of performing the simulation of turbulent square duct.	41
C.1	Example ratios of maximum to minimum cell spacing for differing values of n_g and g_r	71
C.2	Ratio of maximum to minimum grid cell spacing in the y direction or R_y for the weak scaling test of <i>SNaC</i> with respect to the ratio of maximum to minimum grid cell spacing of directions x and z or R_{xz} and the number of processes n_p allocated in that test.	72
D.1	Parameters used for calculating the average power consumption of the <i>Snellius</i> supercomputer and the workstation.	73

1

Introduction

Throughout history, fluid flow analysis has been ubiquitous, even if done in an unscientific or speculative manner. Often, solutions were found through trial and error, with little focus on the underlying reason or theory that made them work. An example of this is the Roman aqueducts [4] and dams [5] prevalent in many European countries in one way or another. With the evolution of modern analysis techniques for fluid flows, the number of applications that require information about flow behaviour has grown immensely. Examples of these include designing aircraft [6], cooling electronic devices [7], and predicting blood flow impediments in the brain [8].

The scientific method has enabled three distinct ways to obtain information in the aforementioned applications. Each method plays a role in developing knowledge of a fluidic system.

Analytical solutions define a ground truth from which theory can be extrapolated. By applying mathematical operations to conservation equations, they are often only valid for simple geometries and boundary conditions [9].

Experimental approaches, on the other hand, rely solely on the capabilities of the instruments and measurement methods used to measure the flow. Most often, these provide benchmark data for many computational methods by measuring standard (canonical) flows, especially when analytical solutions are not available [10]. Even so, some quantities and regions of specific flows are not easily measurable, even with the most advanced methods [11]. Computational efforts are essential to further analyse the intricacies of the flow in greater detail.

When analytical analysis is unfeasible and experimental methods are challenging or expensive, a computational approach is an attractive alternative. Indeed, most engineering applications are evaluated using computational methods for several reasons: the experimental setups required to assess the exact system of the engineering application are too expensive [12]; the diversity of computational methods currently available is immense [10], etc. It also offers an opportunity to test slightly out-of-scope scenarios, which provide a broader understanding of a system than simply testing the expected sequence of events. An example of this is in understanding extreme weather events [13].

Many engineering applications benefit from fundamental knowledge of fluid flow, and for this, accurate experiments and simulations are required. Direct numerical simulation (DNS) is a method that directly solves the governing equations of fluid flow (the continuity equation and the Navier-Stokes equations), resulting in solutions that represent reality without any modelling approaches are taken [10]. Due to the unsteady, inherently three-dimensional complex structures of turbulent flow, resolving all length and time scales in a system has proven to be costly, both in time and computational resources [14]. Because of this, many efficient numerical methods have been formulated to optimise calculations.

Specifically in the DNS space, second-order finite-difference schemes, in conjunction with the fractional-step method of Kim and Moin [15], have proven to be a hallmark in these computations, with modern codes [1, 16] leveraging them for extremely fast solvers. When using this method, the prediction velocity needs to be projected onto a divergence-free space through a correction pressure. This pressure is obtained through solving the pressure Poisson equation. This step has proven to be the most challenging in a fluid flow solver [3], thus it has received

much scientific interest over the past 70 years. This partial differential equation (PDE), when discretized with finite-differences, produces a sparse system of linear equations.¹ The most commonly used methods for solving these systems can be categorized as iterative or direct.

An iterative method is a procedure that uses an initial guess of the result to generate progressively more accurate approximations of the exact solution [10]. On the other hand, direct methods solve the problem by a finite sequence of operations, obtaining the solution up to machine precision in one iteration by definition [10].

Current implementations of iterative methods exploit multigrid (MG) algorithms with an iterative solver, such as the conjugate gradient (CG) algorithm [17]. They benefit from their comparatively low asymptotic computational complexity, which makes them rather efficient and performant [18].

Many are the direct methods available for the sparse system of linear equations produced by the discretization of the pressure Poisson equation. One option are cyclic reduction (CR) procedures to simplify the usually large, sparse matrix [19]. The more commonly used approach is through Fourier analysis (FA), where the matrix decomposition is calculated directly exploiting properties of this matrix [20]. Despite the higher computational complexity, in practice these methods outperform iterative methods due to the fact that modern computational frameworks are not powerful enough to exacerbate the small difference in complexity [3], ending up as the preferred methods of choice in many CFD solvers [1, 16, 21].

Unfortunately, Fourier analysis requires homogeneous boundary conditions (BCs) and more restrictively uniform grid spacing in the discretization [1]. Buzbee [22] suggests performing the matrix decomposition procedure with the use of numerical algorithms for calculating the eigendecomposition, showing that this is a method amenable for parallel computing.

In fact, MG methods, CR methods and FA-based methods have all been ported in one way or another to Graphics Processing Units (GPUs) and many-GPUs implementations, demonstrating great performance improvements in clusters that were erected for mainly artificial intelligence (AI) applications [23]. Despite this, to the best of our knowledge, *there has not been a thorough comparison of an eigendecomposition-based CFD solver for massively-parallel environments with the aforementioned methods in terms of accuracy, wall-clock time and scalability on non-uniform meshes.*

The present work aims at answering how such a solver behaves through its implementation on the already well-established *CaNS* [1] code, providing a framework for simulating test cases that benefit from non-uniform grid spacing along more than one direction, such as the turbulent square duct [2].

1.1. Outline

This thesis is organized as follows:

Chapter 2 introduces the governing equations of the flow, relevant computational methods required to understand computational fluid solvers, and a thorough chronological description of the scientific literature on DNS solvers and their parallelization.

Chapter 3 defines the primary research question and sub-questions that result from the research gap found in the literature.

Chapter 4 explains the numerical methods employed to solve the pressure Poisson equation, along with defining the matrix-based operator.

Chapter 5 interprets the mathematical framework established earlier and explains how it is implemented into the DNS tool *CaNS*.

Chapter 6 evaluates the computational performance of the implementation, compares it with established iterative methods, and assesses the environmental and capital costs of deploying the method developed.

¹ See Chapter 4.

2

Background

The computational analysis of incompressible, single-phase flow of a Newtonian fluid starts from the continuity equation and the Navier-Stokes equations:

$$\nabla \cdot \mathbf{u} = 0, \quad (2.1)$$

$$\rho \left(\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla) \mathbf{u} \right) = -\nabla p + \mu \nabla^2 \mathbf{u} + \mathbf{f}, \quad (2.2)$$

where $\nabla \cdot$ represents the divergence, $\mathbf{u}$ the velocity vector, p represents the local pressure, $\mathbf{f}$ the body forces acting on the continuum, ρ the fluid's density, and μ the fluid's dynamic viscosity.

One can observe that the continuity equation does not contain a time derivative in its formulation. Instead of describing the time evolution of the mass (or density) of the fluid, this equation imposes an instantaneous, global requirement that must be met at every instant and every point. This requirement is equivalent to simulating a medium with infinite speed of sound, where pressure waves travel instantly across the flow. The challenge arises when trying to couple this restriction to the Navier-Stokes equations [10]. It will be seen later that this is done by solving the pressure Poisson equation derived below.

In the present work, direct numerical simulation (DNS) will be carried out. DNS aims at solving equations 2.1 and 2.2 with no further modelling assumptions, meaning that no averaging or filtering is performed on the equations. Although it cannot simulate engineering applications, it establishes ground truths from which researchers can answer scientific questions regarding the fundamental structures and dynamics of fluid flow [24]. As these equations are in a continuous domain, they have to be discretized both spatially and temporally to be solved in a digital computer. In the present work, the staggered grid is considered, characterised by the placement of the velocity variables at the midpoints between cell centroids [10], which prevents artificial pressure oscillations generated by the pressure-velocity decoupling of the grid [25].

2.1. Time discretization

The numerical integration of the time-dependent terms $\frac{\partial \mathbf{u}}{\partial t}$ in the Navier-Stokes equations can be done in various ways, each with its own advantages. Most often, these schemes determine the stability of the solutions found as time advances through the simulation, depending on the treatment of the advection $(\mathbf{u} \cdot \nabla) \mathbf{u}$ and the diffusion $\nu \nabla^2 \mathbf{u}$ terms [10]. An explicit method calculates the solution at the next time level using already known time levels. On the other hand, an implicit method calculates the solution at the next time level using information from future time levels. In a similar vein, a semi-implicit method calculates the solution at the next time level using information from both the already known and future time levels [10]. When calculating the temporal advancement of advection terms explicitly, the Courant number $C = \frac{u \Delta t}{\Delta x}$ arises from the Courant-Friedrichs-Lewy (CFL) condition, which limits the distance in which information travels after time step Δt to a single cell spacing (in a 1D case). The stability of the temporal advancement of diffusion terms explicitly is determined by the Fourier number $F = \frac{\nu \Delta t}{\Delta x^2}$. The combination of these two non-dimensional numbers provides a sufficient criterion for stable temporal integration, given that both the Courant number and the Fourier number are prescribed *a priori*

[1, 26]:

$$\Delta t < \min \left(F_{\max}^{-1} \frac{\Delta x^2}{\nu}, C_{\max}^{-1} \frac{\Delta x}{u} \right). \quad (2.3)$$

The exact prescription of both non-dimensional quantities is dependent on the temporal integration scheme and spatial discretization. When the time stepping of the diffusive terms is done explicitly, the maximum allowed time step scales with the minimum grid spacing in a parabolic manner $\Delta t \sim \Delta x^2$. Often, this restricts the time step order to prohibitively small values [1]. Because of this, the diffusion term may be computed with implicit time stepping, while the advection term is typically computed with explicit time stepping. As such, most CFD codes implement a semi-implicit scheme in their temporal advancement [1, 16].

2.1.1. Application to governing equations

For simplicity, consider that both terms are calculated with explicit time stepping. Then, the semi-discrete Navier-Stokes becomes:

$$\frac{\mathbf{u}^{n+1} - \mathbf{u}^n}{\Delta t} = -\nabla P^{n+1} + \nu \nabla^2 \mathbf{u}^n - (\mathbf{u}^n \cdot \nabla) \mathbf{u}^n, \quad (2.4)$$

where the left-hand side represents the discretization of the time-dependent term $\frac{\partial \mathbf{u}}{\partial t}$ of the Navier-Stokes equations with a backward Euler scheme; $P = \frac{p}{\rho}$ is the kinematic pressure; and the superscript indicates the time level, with n being the current time level and $n+1$ the following one. Although this is not the time discretization scheme used in the present work, it provides a framework accurate enough for the basic derivations to be carried out.

The pressure term here is calculated with implicit time stepping for reasons that will become clear later. The advection and diffusion terms can be combined into a single term defined as the advection-diffusion term:

$$AD^n = \nu \nabla^2 \mathbf{u}^n - (\mathbf{u}^n \cdot \nabla) \mathbf{u}^n. \quad (2.5)$$

This allows for equation 2.4 to be written as:

$$\frac{\mathbf{u}^{n+1} - \mathbf{u}^n}{\Delta t} = -\nabla P^{n+1} + AD^n.$$

Letting $\mathbf{u}^* = \mathbf{u}^n + \Delta t AD^n$, one is able to write:

$$\mathbf{u}^{n+1} = \mathbf{u}^* - \Delta t \nabla P^{n+1}. \quad (2.6)$$

Applying the divergence to both sides of the equation:

$$\nabla \cdot \mathbf{u}^{n+1} = \nabla \cdot \mathbf{u}^* + \nabla \cdot (\Delta t \nabla P^{n+1}). \quad (2.7)$$

From the continuity equation, $\nabla \cdot \mathbf{u}^{n+1} = 0$. Knowing that $\nabla \cdot (\Delta t \nabla P^{n+1}) = \Delta t \nabla^2 P^{n+1}$, the pressure Poisson equation is derived:

$$\boxed{\nabla^2 P^{n+1} = \frac{\nabla \cdot \mathbf{u}^*}{\Delta t}}. \quad (2.8)$$

Were the diffusive terms calculated with implicit time stepping, the following would be observed:

$$\mathbf{u}^* = \mathbf{u}^n + \Delta t [\nu \nabla^2 \mathbf{u}^* - (\mathbf{u}^n \cdot \nabla) \mathbf{u}^n].$$

Now, it is clear that three Helmholtz equations must also be solved, one for each component of the velocity vector field:

$$\mathbf{u}^* - \Delta t \nabla^2 \mathbf{u}^* = \mathbf{u}^n - (\mathbf{u}^n \cdot \nabla) \mathbf{u}^n = \mathbf{RHS}, \quad (2.9)$$

where **RHS** means the right-hand side of the Helmholtz equation. As we have seen, when solving the Navier-Stokes equations using this approach, one combines the continuity equation and the momentum equations into a

Poisson equation that computes the (correction) pressure of the flow, given explicit time-stepping of the viscous terms in the Navier-Stokes equations. If the viscous terms are to be calculated with implicit time stepping, three Helmholtz equations, which calculate each component of the velocity vector field, are also solved [15].

With this, an algorithm for solving the Navier-Stokes equations can be formulated for explicit time stepping of the viscous terms:

1. Calculate the prediction velocity: $\mathbf{u}^* = \mathbf{u}^n + \Delta t \mathbf{A} \mathbf{D}^n$,
2. Solve the pressure Poisson equation: $\nabla^2 P^{n+1} = \frac{\nabla \cdot \mathbf{u}^*}{\Delta t}$,
3. Correct the prediction velocity: $\mathbf{u}^{n+1} = \mathbf{u}^* - \Delta t \nabla P^{n+1}$.

For a semi-implicit time stepping of the viscous terms, one may substitute step one in the algorithm aforementioned with:

1. Calculate $\mathbf{u}^*$ by solving $\mathbf{u}^* - \Delta t \nabla^2 \mathbf{u}^* = \mathbf{RHS}$.

This approach is known as the *fractional step method*. It now becomes clear why the pressure term in equation 2.4 is calculated with implicit time stepping: This way, one can perfectly force the zero velocity divergence constraint or, in other words, mass conservation. In the method above, spatial discretization is still necessary, as only temporal discretization has been done.

2.2. Spatial discretization

The simplest implementation of spatial discretization in partial differential equations (PDEs) is done through finite-difference methods (FDM). Due to their vast history, FDMs are stable and accurate under both high and low Reynolds numbers [27, 28], with higher-order schemes being frequently developed to ensure better accuracy [29, 30]. Unfortunately, these methods are always associated with a certain degree of inaccuracy due to their inability to represent higher wavenumbers accurately in the solution [31]. This will be the method used for spatial discretization in the present work mainly due to its accurate results and ease of implementation.

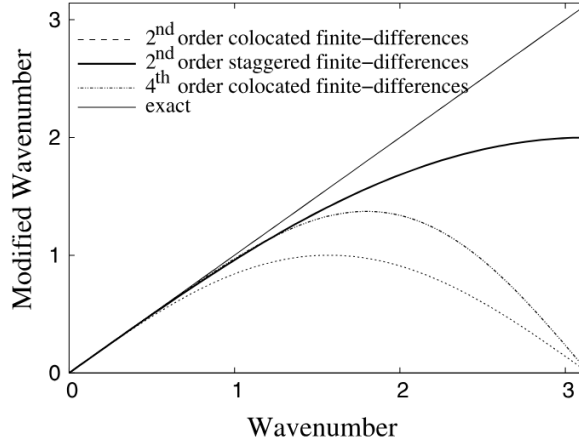


Figure 2.1: Modified wavenumbers of finite-difference schemes for different grid types. Figure from [31].

The fractional-step method of solving the Navier-Stokes equations introduced by Chorin in [32] and later matured by Kim and Moin in [15] has allowed for the FDMs to excel in terms of accuracy, performance and scalability as demonstrated by [1, 21, 16]. The pressure Poisson equation and the three Helmholtz equations, when discretized by finite-difference schemes, can be assembled into matrix equations, requiring a system of linear equations solver to obtain a solution [33, 34]. In this way, it is clear that these linear solvers are of great importance for a solver of incompressible flow.

2.3. Linear solvers

Historically, great interest has been attributed to developing fast and accurate linear solvers, due their demand in various physical problems [35]. These solvers are classified as iterative or direct.

2.3.1. Iterative methods

Iterative methods for solving elliptic PDEs, such as the Poisson or Helmholtz equations, are a field of research in their own right; thus, focus will be on the most commonly used methods. They are characterised by solutions with round-off errors and residuals within a certain accuracy, but this is traded off for theoretically shorter wall-clock times that these approaches achieve [10]. The Jacobi method is one of the simplest iterative methods for solving systems of linear equations, based on obtaining the following iteration exclusively with previous iterations [10]. Currently, the Gauss-Seidal method is a variation to the Jacobi method and a classical implementation of an iterative method for solving a system of linear equations [36] based on obtaining the following iteration with current and previous iterations. Most often, more optimised methods are used.

Currently, most CFD codes make use of a conjugate gradient (CG) solver [17, 18, 37] as it has been shown to be easy to implement while boasting good scalability [38]. CG methods are heavily dependent on the matrix's ratio between the maximum and minimum eigenvalue (its condition number) for maximum performance, so different methods or preconditioners are applied to ease this restriction [38]. The incomplete Cholesky factorization has proven to be a suitable preconditioner for Poisson and Helmholtz problems [39, 40].

As an alternative to the aforementioned approaches, multigrid (MG) methods have shown great potential due to their comparatively low asymptotical operation count of $O(N)$ for a 1D problem. They function by smoothing high-frequency error on a fine grid, then transferring the residual to a coarser grid to correct low-frequency error. This procedure is recursively cycled through a hierarchy of grids to eliminate errors at all scales in an optimal manner. It has seen use in various implementations of fluid flow solvers [17, 18, 37] and in unstructured stretched meshes [41]. Libraries that implement MG as a high-performance approach have reduced the complexity of implementing these methods [42], and their scalability [43] has enabled MG DNS to shape many numerical implementations today [44]. MG methods can also be combined with learning-based methods [45] in order to improve the performance of full MG methods, making these extremely efficient and versatile.

2.3.2. Direct methods

On the other hand, direct methods, by definition, solve the matrix equation in a single iteration to machine precision [1]. Despite the efficiency of iterative and especially MG methods, Hockney empirically shows that direct methods have the capability of being $O(10)$ faster than iterative methods [20]. These methods are varied in their scope and application, but it is essential to understand that the matrix generated by the finite-difference discretization is a tridiagonal, sparse matrix for 1D problems and block tridiagonal, sparse matrix for nD problems, $n > 1$ [35]. In these specific conditions, some methods are worth highlighting.

Cyclic reduction (CR) is a method that removes either the odd or the even rows of the matrix by substitution. This can be done once to effectively reduce the size of the Laplacian matrix in half. For a matrix of size $N \times N$ with $N = 2^k$, this process can be done k times for a single equation to be solved. The solution to these equations is then back-propagated through the reductions, until the initial system is fully solved [35]. This method was first introduced by Hockney in [20] and extrapolated to block-cyclic reduction by Buzbee et al. in [35]. Since this method relied on the fact that the matrix had to be of dimensions of powers of 2, Schumann and Sweet expanded the method to arbitrary-sized matrices in [46], broadening the domains (and applications) where this method could be implemented. Often, this method was unstable in its most rudimentary form [47, 48] and thus Buneman in [19] devised a computationally stable form of the algorithm. This method by itself has a computational complexity of $O(N \log_2 N)$ [49, 50]. The method is revisited in reviews by Dorr [47] and by Swarztrauber in [49]. In these reviews, Fourier analysis (FA) is mentioned as an accompanying method to cyclic reduction, and this was the manner in which FA was applied at the advent of these methods.

Currently, however, FA stands as a stand-alone method that is proven to be one of the most efficient and scalable [1]. This analysis is based on the method of eigenfunction expansion [1], where the solution is assumed to follow a linear combination of certain basis function (in this case Fourier modes). But for it, the Laplacian matrix has to be diagonalized. Spectral theory states that a matrix is diagonalizable by orthogonal matrices if it is a real symmetric matrix [51], which is the case. The method, in its essence, involves performing an eigendecomposition of the Laplacian matrix and performing a series of operations that lead to the final solution [35]. First introduced by Hockney in [20], it took advantage of the fact that the eigenvalues and eigenvectors of the Laplacian matrix for periodic boundary conditions were known analytically. The same work by Hockney [20] extended this method to commonly used BCs of a fluid problem, with discrete sine transforms (DST) and discrete cosine transforms (DCT), allowing for application of FA to a broad spectrum of fluid flow problems.

This method is extremely efficient in its calculations due to the fast Fourier transform (FFT) algorithm, which smartly only requires $O(N \log_2 N)$ operations for a 1D problem, the same complexity as CR. Swarztrauber in [49] shows that the constants in computational complexity for FA are smaller when compared to CR, but points out that this distinction could be negligible depending on the computational setup of the numerical solver.

In contemporary times, with the advent of these methods, often CR and FA were performed in conjunction to obtain a sequential algorithm that was performant and versatile. Since the number of reductions in CR part of the algorithm could change, a class of algorithms called FACR(n)¹, where n is the number of reductions by CR, emerged. Evidently, FACR(0) reduces to applying FA alone.

But it happens that FA is a specific case of a matrix decomposition algorithm (MDA) [52]. As mentioned, FA starts by performing an eigendecomposition and applying discrete transforms (DT) to efficiently calculate eigenvalues and eigenvectors. Since it relies on a DFT and DTs are performed in constant grid spacing, this method cannot solve non-uniform grid spacing for the discretization of the equations [52]. With this logic, an MDA can still be used without the use of DTs by simply numerically calculating this eigendecomposition. If the eigenvalues and eigenvectors are not known analytically *a priori*, this procedure has a computational complexity of $O(N^2)$ for 1D problems [52], proving to be asymptotically slower than FA or CR. For nD , the calculation of the eigendecomposition can be made with the use of tensor product (also called Kronecker product) properties in the case of separable elliptic equations [52]. This means that one can compute the eigendecomposition of the nD matrix resulting from the PDE by performing an eigendecomposition for each dimension of the domain. With these, by taking the tensor product of the vector of eigenvalues between dimensions and the matrix of eigenvectors between dimensions, one is able to describe the entire eigendecomposition of the original nD matrix [53].

As put by Bialecki et al. in [52] in their most recent review, *The early history of MDAs is quite convoluted*. When presenting his approach to combining FA with CR, Hockney advocates that MDAs were first published by Hyman in [54] in 1951. In contrast, the authors of this review advocate that the first instance of an MDA was published in 1960 in [55] and later refined by Osborne in [56]. The use of MDA with tensor product properties dates to Lynch et al. in [53, 57] but attributes the advent of MDAs to Egerváry in [58], showing little consensus on the original author for these methods. With Hockney popularising this method by applying DTs to efficiently obtain the eigendecomposition of elliptic PDEs in [20], many (but not all) of the works that follow it focus solely on MDAs with FA. Included Golub et al. in [59], Zhuang and Sun in [60] and more recently Qian et al. in [33] to analyse the thermal dissipation of a chip. Some reviews of MDA, CR, FA and FACR include Dorr in [47], Temperton in [61], Kadaljaboo et al. in [62] and Vajtersic's book on elliptic solvers [63].

These methods, especially CR and FA, are typically applied to uniformly spaced grid discretizations due to two criteria. First, as described, FA requires, by definition, constant grid spacing to apply DTs [52]. Secondly, the discretization by FDMs of the Poisson and Helmholtz equations yields a non-symmetric matrix for non-uniform spacing when assembled [52], which yields a much more computationally expensive eigendecomposition [64]. Both restrictions can be circumvented through varying approaches, and many efforts have been made in the literature to adapt these fast solvers to irregular domains. The first account of such an adaptation comes from George in [65] by partitioning the PDEs' matrix into segments that are responsible for the irregularities and their boundaries. In section 9 of Buzbee et al. [35], it is briefly mentioned how FACR(n) could be applied to non-rectangular domains, but little focus is given on the matter. One year later, Buzbee et al. apply many of the established algorithms at the time (essentially those surveyed by Dorr in [47]) to irregular domains in [66] in a centralized manner. Farnell in [67] utilizes MDAs to approach the irregularity of domains without FA. Babu and Korpela extend [66] to Neumann BCs maintaining focus on irregular domains in [68] and more recently Wang et al. in [69] develop a higher-order compact difference scheme specifically for the Laplace and Poisson equations on irregular grid systems. It is worth noting that these authors incorporate some sort of partitioning/slicing of the main matrix in their methods, which would later resemble multi-block approaches in more advanced implementations such as [3, 70].

Despite this, there are certain direct algorithms which are more generic in their scope. Most known of them is Gauss elimination, but this algorithm does not take advantage of the tridiagonal, sparse matrix that is originated from the PDEs' discretization. The tridiagonal matrix algorithm (TDMA) is an adaptation of the Gauss elimination algorithm where some steps are optimised to conform to the structure of the tridiagonal matrix. It sees use in some applications (see [1, 71]) which decompose the 3D domain and its matrix into tensor products to

¹FACR - Fourier Analysis Cyclic Reduction (algorithm).

calculate the solution of the system of linear equations. This effectively is an approach to parallelize the solver for multithreading or GPU implementation, but other approaches could also be taken.

2.4. Parallelization

In fact, the parallelization of these algorithms and their implementation is an active area of investigation, with strides being made from differing approaches, whether it is by improving the parallelization of iterative methods (especially MG algorithms and such pre-conditioners [42]), whether it is by enabling parallelization in direct methods that were previously thought to be out of reach (for example, by parallelizing FFT in solving the pressure Poisson equation [72]).

In terms of iterative methods, the CG algorithm is still studied thoroughly, with vector and parallel implementations shown in [73] applied to the Laplace equation. In conjunction with it, MG methods are used as in [74, 75], or other preconditioners as in [76, 39], such as incomplete Cholesky factorization, which was also used in the sequential versions of the algorithm. Xu presents a review of recent work on preconditioners for CG methods in [77], including a general framework for determining preconditioners (Fast Auxiliary Space Preconditioning, or FASP). With these, massively-parallel solvers for Poisson problems are made possible, with examples being Nottay and Napov in [78] for a purely CPU-based implementation and Zolfaghari and Obrist in [79] for a GPU-based implementation of a fast Poisson solver.

Direct methods also have research interest in their vectorization and parallelization since the sequential counterparts of these methods are around $O(10)$ faster than iterative algorithms [20]. Cyclic reduction algorithms were one of the first to be tackled for parallelization with Hockney refining his implementation of the FACR(n) algorithm in [80] by simultaneously defining the parameters in which the algorithm is optimised for. Before him, Stone evaluated the CR algorithm and Buneman's algorithm in terms of their parallelization capabilities by assessing the operation count in parallel operation in [81]. It finds that Buneman's algorithm has a higher operation count than CR in parallel operation, despite not approaching the stability of implementation of such algorithms, which motivated the creation of Buneman's algorithm. Different variations of the conventional CR algorithm are also devised to accommodate parallelization, as in [82], which utilizes MDAs for only the rows that are being reduced in every stage, achieving the same computational complexity as the sequential version of CR ($O(N \log_2 N)$) in Message Passing Interface (MPI) parallelization. GPU implementations of fast Poisson solvers can be seen in [83], which achieve $28\times$ speedup compared to the sequential counterpart implemented in Fortran's library LAPACK; and in [84], where emphasis is put on block CR, which allows for n D problems to be tackled.

FA-based algorithms have also been addressed for vectorization and parallelization. Hockney, by refining his approach to the FACR(n) algorithm [80], also undertakes some effort in parallelizing FA. For this analysis, domain decomposition becomes crucial for distributing the workload to each process. In [85], the 2D domain is decomposed into strips, and DTs are applied for each strip in the first direction, with the diagonalized system being solved using a method of choice. Recent approaches still take advantage of domain decomposition, with 2D strips being used in [86] and 3D pencil-like decompositions in [87, 88, 89] due to libraries such as [72, 90] that readily parallelize the DT process in a convenient manner. Extreme-scale parallelization in the form of GPU-based solvers and many-GPU-based solvers for solving Poisson problems becomes commonplace with open-source works such as [91, 92].

On the other hand, MDAs for solving Poisson-like problems or fluid flow problems have practically only been seen in use through the use of FA. If the domain needs to be discretized in a non-uniform fashion, FA is not suitable [52]. For this, Buzbee proposes a method that will be the foundation of the methods used in the present work in [22]. In it, he describes how MDA could be used to solve non-uniform discretizations for Poisson problems for three-dimensional problems by decoupling each dimension and utilizing MDAs for each 1D tridiagonal matrix. In fact, he proves how such an implementation of MDA in parallel benefits from an L -fold increase in efficiency compared to a single processor when calculated with L processors. Expectedly, since the eigenvalues and eigenvectors are not being calculated through the more efficient DTs, the computational complexity of this method is $O(N^3)$ for 2D problems, comparatively worse than $O(N^2 \log_2 N)$ that performing FA brings. Nevertheless, this method still proves to be 4 to 6 times faster than iterative methods [22], despite its worse asymptotic computational performance. In [3], one could explain this difference due to the scales of the problems considered. It is theorized that the size of the domains prevalent in fluid flow problems is not great enough to fully take advantage of the difference in computational complexity that modern multigrid methods have in their favour. Reviews of

the vectorization and parallelization of the aforementioned methods can be seen in Swarztrauber and Sweet [93], in Vajtersic's book [63], and more recently in Brown [94]. In the meantime, CFD codes are still investing in the FA approach due to their empirically superior performance.

With the rise in AI applications, high-performance computing has changed to a GPU-based architecture. With it, many CFD codes ported their many-CPU implementations to many-GPU implementations. These include [95] for a mixed-spectral finite difference scheme, [21] for a finite-difference FFT-based approach, *AFiD-GPU* [16] for a finite-difference approach for wall-bounded turbulent flows, [96] for a FVM approach suitable for arbitrary geometries and *CaNS* [97] for a finite-difference, fractional-step, FFT-based approach. Often, these codes provide the building blocks for more complex methods, such as two-phase turbulent flows in *CaNS-Fizzy* [98]. A slightly outdated review by Niemeyer and Sung in [23] summarises the progress and challenges of porting these legacy codes to GPU-based architectures.

To the best of our knowledge, an MDA-based solver for fluid flow problems computed in a massively parallel environment is not present in the literature. With this in mind, the problem statement is presented in the next chapter.

3

Objective and Scope of Thesis

As observed in the relevant work present in the literature, even modern implementations of CFD solvers can still be improved in many fronts. The present work aims to tackle the following research question:

What are the accuracy and performance trade-offs of an eigendecomposition-based CFD solver versus established methods on uniform and non-uniform meshes in a CPU-based implementation?

To further elaborate on this question, the following sub-questions are also answered:

1. Computational performance

- (a) *Time to solution*: How does wall-clock time per solution grow with problem size, and how does that compare to each algorithm's theoretical complexity?
- (b) *Scalability*: When increasing the number of GPUs/CPUs used in the calculation, how much speed-up does eigendecomposition gain, and where do returns start to diminish?
- (c) *Performance bottlenecks*: What are the main bottlenecks for an eigendecomposition-based CFD solver at different compute scales? How can they be minimised?

2. Cost of simulation

- (a) *Environmental cost*: How much energy and CO₂ emissions are required to simulate a mock large-scale simulation using an eigendecomposition-based CFD solver?
- (b) *Capital cost*: How much capital, in currency and in compute hours, is required to simulate a mock large-scale simulation using an eigendecomposition-based CFD solver?

The *scope* of this thesis is limited to single-phase incompressible, constant temperature, Newtonian flows.

4

Numerical Methods

By this point, enough background has been given to explain the need for solving the pressure Poisson equation, whose solution can project the velocity field into a divergence-free space. The present chapter explains how this equation has been solved, based on a general matrix-matrix multiplication (GEMM) of the eigendecomposition of the discrete Laplace operator.

4.1. Discretizing the pressure Poisson equation

The pressure Poisson equation is written as:

$$\nabla^2 \phi = f, \quad (4.1)$$

where the domain of the problem is $\Omega = \{0 \leq X_k \leq l_k, k = 1, 2, 3\}$, ∇^2 is the continuous Laplace operator, ϕ is the correction pressure, $f = \frac{\nabla \cdot \mathbf{u}^*}{\gamma_k \Delta t}$ is the right-hand side (RHS) of the Poisson equation, X_k is the position of the grid cell and l_k is the length of dimension x_k .

The pressure Poisson equation can be rewritten as the sum of the second derivatives of the correction pressure with regard to each direction of the problem:

$$\frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} + \frac{\partial^2 \phi}{\partial z^2} = f. \quad (4.2)$$

The domain is discretized on a Cartesian grid, with each direction x, y, z having N_x, N_y, N_z grid cells. Equation 4.2, when discretized, produces $N = N_x N_y N_z$ equations. Without a loss of generality, one may consider the second-order derivative for the 1D case of direction x :

$$\frac{\partial^2 \phi}{\partial x^2} = \frac{\partial}{\partial x} \left(\frac{\partial \phi}{\partial x} \right) \approx \frac{\frac{\partial \phi}{\partial x} \left(i + \frac{1}{2} \right) - \frac{\partial \phi}{\partial x} \left(i - \frac{1}{2} \right)}{X \left(i + \frac{1}{2} \right) - X \left(i - \frac{1}{2} \right)}, \quad (4.3)$$

where:

$$X \left(i + \frac{1}{2} \right) - X \left(i - \frac{1}{2} \right) = \frac{X(i+1) + X(i)}{2} - \frac{X(i) + X(i-1)}{2} = \frac{1}{2} \Delta_2 X(i+1). \quad (4.4)$$

Here $\Delta_j X(i) = X(i) - X(i-j)$. The first derivatives at the mid-points can be written as:

$$\frac{\partial \phi}{\partial x} \left(i + \frac{1}{2} \right) \approx \frac{\phi(i+1) - \phi(i)}{X(i+1) - X(i)} = \frac{\phi(i+1) - \phi(i)}{\Delta X(i+1)}, \quad (4.5)$$

$$\frac{\partial \phi}{\partial x} \left(i - \frac{1}{2} \right) \approx \frac{\phi(i) - \phi(i-1)}{\Delta X(i)}. \quad (4.6)$$

Substituting 4.4, 4.5 and 4.6 into 4.3 gives

$$\frac{\partial^2 \phi}{\partial x^2} \approx \frac{2}{\Delta_2 X(i+1)} \left[\frac{1}{\Delta X(i)} \phi(i-1) + \left(\frac{-1}{\Delta X(i)} + \frac{-1}{\Delta X(i+1)} \right) \phi(i) + \frac{1}{\Delta X(i+1)} \phi(i+1) \right]. \quad (4.7)$$

The discretized Poisson equation for a non-uniform mesh for a 3 dimensional problem is then written as:

$$\begin{aligned} & \frac{2}{\Delta_2 \mathbf{X}(i+1)} \left[\frac{1}{\Delta \mathbf{X}(i, j, k)} \phi(i-1) + \left(\frac{-1}{\Delta \mathbf{X}(i, j, k)} + \frac{-1}{\Delta \mathbf{X}(i+1)} \right) \phi(i, j, k) + \frac{1}{\Delta \mathbf{X}(i+1)} \phi(i+1) \right] + \\ & \frac{2}{\Delta_2 \mathbf{X}(j+1)} \left[\frac{1}{\Delta \mathbf{X}(i, j, k)} \phi(i-1) + \left(\frac{-1}{\Delta \mathbf{X}(i, j, k)} + \frac{-1}{\Delta \mathbf{X}(j+1)} \right) \phi(i, j, k) + \frac{1}{\Delta \mathbf{X}(j+1)} \phi(j+1) \right] + \\ & \frac{2}{\Delta_2 \mathbf{X}(k+1)} \left[\frac{1}{\Delta \mathbf{X}(i, j, k)} \phi(i-1) + \left(\frac{-1}{\Delta \mathbf{X}(i, j, k)} + \frac{-1}{\Delta \mathbf{X}(k+1)} \right) \phi(i, j, k) + \frac{1}{\Delta \mathbf{X}(k+1)} \phi(k+1) \right] + \\ & = f(i, j, k). \end{aligned} \quad (4.8)$$

$N = N_x N_y N_z$ equations are produced and assembled into a matrix equation, where $\mathbf{A}$ has dimensions $N \times N$:

$$\mathbf{A}\phi = \mathbf{f}. \quad (4.9)$$

The discretization of the 1D Laplace operator (here on out as $\mathbf{T}$) creates a tridiagonal matrix, where $[a \ b \ c]_i$ represent the subdiagonal, diagonal and supradiagonal elements of this discretization matrix. In this case, $a_i = \frac{2}{\Delta_2 \mathbf{X}(i+1)\Delta \mathbf{X}(i)}$, $c_i = \frac{2}{\Delta_2 \mathbf{X}(i+1)\Delta \mathbf{X}(i+1)}$ and $b_i = -(a_i + c_i)$. This matrix is not symmetric as the grid considered is not uniform, making $\Delta \mathbf{X}(i+1) \neq \Delta \mathbf{X}(i)$. From equation 4.8, one may see that the discretization of the Laplace operator creates a 7 diagonal matrix, with a structure that is to be exploited in the algorithm for solving the Poisson equation.

4.2. Solving the linear system of equations

The present work takes advantage of the fact that the matrix of the discrete Laplace operator can be written as a sum of tensor products, as the operator is separable:

$$\mathbf{A} = \mathbf{T}_x \otimes \mathbf{I}_y \otimes \mathbf{I}_z + \mathbf{I}_x \otimes \mathbf{T}_y \otimes \mathbf{I}_z + \mathbf{I}_x \otimes \mathbf{I}_y \otimes \mathbf{T}_z.$$

The approach taken in the present work to solve the discretized Poisson equation starts with the successive decomposition of $\mathbf{T}$ for the x and y directions, the isolation of the transformed correction pressure by solving the resulting tridiagonal system, and finally applying the inverse transformation to obtain the correction pressure. This series of operations could define, for the x and y directions, an operator that acts as a tool to separate the solving process from the solving method, as illustrated by the different ways in which the decomposition mentioned could be performed later. This ensures great modularity in the solving process.

The differentiator between the present work and other solvers comes from the construction of the operator that enables the transformation of the matrix of the discrete Laplace operator $\mathbf{A}$ into a matrix which makes isolating the transformed correction pressure trivial. In the next subsection, a generalized matrix-based operator is presented based on the work of Buzbee in [22], along with a short comparison with the established DT-based (Discrete Transform) operator from *CaNS* [1].

4.2.1. Designing an operator

The procedure described above allows for the choice of an operator based on the characteristics of the Poisson equation for each direction. This can depend on the boundary conditions, the discretization of the domain, and even on the platform on which the Poisson equation will be solved. The following is a method that allows for non-uniform meshes and non-homogeneous boundary conditions, the matrix-based operator.

Matrix-based operator

The spectral theorem for a real, symmetric matrix $\mathbf{A}$ states that an orthonormal diagonalization exists by the eigenvectors $\mathbf{v}$ of matrix $\mathbf{A}$, assembled column-wise in $\mathbf{Q}$, and the eigenvalues λ of $\mathbf{A}$, assembled in the diagonal matrix $\mathbf{\Lambda}$ [51]. This makes the calculation of the inverse of the matrix of eigenvectors trivial since $\mathbf{Q}^{-1} = \mathbf{Q}^T$.

It would be of interest to find such an eigendecomposition for the matrix $\mathbf{T} = \mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^{-1}$ where this matrix is symmetric. However, in the current form, the matrix $\mathbf{T}$ is not symmetric since it discretizes a direction with a non-uniform mesh through finite-differences. Such an eigendecomposition is not guaranteed to exist at the moment.

To circumvent this issue, a similarity transformation may be applied to calculate the eigendecomposition of $\mathbf{T}$ through a symmetric matrix. For a tridiagonal matrix, this similarity transformation is [99]:¹

$$\mathbf{T}^* = \mathbf{D}^{-1}\mathbf{T}\mathbf{D}, \quad (4.10)$$

where

$$\mathbf{D} = \text{diag}(\delta_i) \quad \text{where} \quad \delta_i = \begin{cases} 1 & \text{if } i = 1 \\ \sqrt{\prod_{j=1}^{i-1} \frac{a_{j+1}}{c_j}} & \text{if } i = 2, \dots, N_{x_k} \end{cases}, \quad (4.11)$$

as a_j and c_j represent the subdiagonal and supradiagonal elements of $\mathbf{T}$, respectively. Note how $\mathbf{D}$ is unique to each direction.

By construction, matrix $\mathbf{T}^*$ is symmetric and thus has an eigendecomposition $\mathbf{T}^* = \mathbf{Q}^*\mathbf{\Lambda}^*\mathbf{Q}^{*\text{T}}$ that can be computed numerically. The eigendecomposition of $\mathbf{T}$ can be obtained from the eigendecomposition of $\mathbf{T}^*$ by the following:²

$$\begin{cases} \mathbf{\Lambda} = \mathbf{\Lambda}^* \\ \mathbf{Q} = \mathbf{D}\mathbf{Q}^* \end{cases} \quad (4.12)$$

This proves that an eigendecomposition for $\mathbf{T}$ exists, regardless of the grid spacing used. Without a loss of generality, this eigendecomposition is applied over the x direction in equation 4.9, yielding:

$$[(\mathbf{Q}_x\mathbf{\Lambda}_x\mathbf{Q}_x^{-1} \otimes \mathbf{I}_y \otimes \mathbf{I}_z) + (\mathbf{I}_x \otimes \mathbf{T}_y \otimes \mathbf{I}_z) + (\mathbf{I}_x \otimes \mathbf{I}_y \otimes \mathbf{T}_z)]\phi = \mathbf{f} \quad (4.13)$$

where $\mathbf{I}_{x_k}$ is the identity matrix of direction x_k with dimensions $N_{x_k} \times N_{x_k}$.

This can be rewritten as:

$$[(\mathbf{Q}_x\mathbf{\Lambda}_x\mathbf{Q}_x^{-1} \otimes \mathbf{I}_y \otimes \mathbf{I}_z) + (\mathbf{Q}_x\mathbf{Q}_x^{-1} \otimes \mathbf{T}_y \otimes \mathbf{I}_z) + (\mathbf{Q}_x\mathbf{Q}_x^{-1} \otimes \mathbf{I}_y \otimes \mathbf{T}_z)]\phi = \mathbf{f}. \quad (4.14)$$

With the mixed-product property of the tensor product [100], one is able to write:

$$(\mathbf{Q}_x \otimes \mathbf{I}_y \otimes \mathbf{I}_z)[(\mathbf{\Lambda}_x \otimes \mathbf{I}_y \otimes \mathbf{I}_z) + (\mathbf{I}_x \otimes \mathbf{T}_y \otimes \mathbf{I}_z) + (\mathbf{I}_x \otimes \mathbf{I}_y \otimes \mathbf{T}_z)](\mathbf{Q}_x^{-1} \otimes \mathbf{I}_y \otimes \mathbf{I}_z)\phi = \mathbf{f}. \quad (4.15)$$

Note that $(\mathbf{Q}_x \otimes \mathbf{I}_y \otimes \mathbf{I}_z)^{-1} = (\mathbf{Q}_x^{-1} \otimes \mathbf{I}_y \otimes \mathbf{I}_z)$. Let:

$$\mathcal{F}^x(\phi) = (\mathbf{Q}_x^{-1} \otimes \mathbf{I}_y \otimes \mathbf{I}_z)\phi, \quad (4.16)$$

$$\mathcal{F}^x(\mathbf{f}) = (\mathbf{Q}_x^{-1} \otimes \mathbf{I}_y \otimes \mathbf{I}_z)\mathbf{f}. \quad (4.17)$$

At this moment, it is possible to solve the initial system in various manners. Firstly, it is possible to apply the same operator to all directions ³. Secondly, it is possible to apply a different or similar operator that takes advantage of the structure of $\mathbf{T}$. This is advantageous in cases where the Poisson equation is discretized with uniform meshes,

¹See Appendix A for a thorough derivation.

²See Appendix A for proof.

³See Appendix B for this method.

minimizing wall-clock time. Finally, it is possible to apply this or other operators to all directions except the final one, in this case the z direction. As this option is of great interest, the result is as follows:

$$[(\Lambda_x \otimes \mathbf{I}_y \otimes \mathbf{I}_z) + (\mathbf{I}_x \otimes \Lambda_y \otimes \mathbf{I}_z) + (\mathbf{I}_x \otimes \mathbf{I}_y \otimes \mathbf{T}_z)]\mathcal{F}^y(\mathcal{F}^x(\phi)) = \mathcal{F}^y(\mathcal{F}^x(\mathbf{f})), \quad (4.18)$$

By construction, the terms of the sum that are the result of the diagonalization of the respective directions are diagonal, meaning that $(\Lambda_x \otimes \mathbf{I}_y \otimes \mathbf{I}_z) = \text{diag}(\Lambda_x)$.⁴ One may also realise that the term $(\mathbf{I}_x \otimes \mathbf{I}_y \otimes \mathbf{T}_z)$ is tridiagonal, since the matrices $\mathbf{T}_{x_j}$, $j \in \{1, 2, 3\}$ are tridiagonal, and the tensor product of a tridiagonal matrix with an identity matrix is tridiagonal. Therefore, in equation 4.18 the term $[(\Lambda_x \otimes \mathbf{I}_y \otimes \mathbf{I}_z) + (\mathbf{I}_x \otimes \Lambda_y \otimes \mathbf{I}_z) + (\mathbf{I}_x \otimes \mathbf{I}_y \otimes \mathbf{T}_z)]$ is a tridiagonal matrix.

This creates an opportunity for solving equation 4.18 with TDMA (tridiagonal matrix algorithm), a variant of Gauss elimination that obtains its solution in $\mathcal{O}(N_{x_k})$ operations instead of the typical $\mathcal{O}(N_{x_k}^3)$ that Gauss elimination requires⁵.

Now that $\mathcal{F}^y(\mathcal{F}^x(\phi))$ has been found, ϕ is found through the application of the inverse operators $\mathcal{F}^{-1,x}(\mathcal{F}^{-1,y}(\square))$ of the x and y directions. Note how the inverse operator can be directly found from its definition as $\mathcal{F}^{-1,x}(\square) = (\mathbf{Q}_x \otimes \mathbf{I}_y \otimes \mathbf{I}_z)(\square)$. This can be written as:

$$\phi = \mathcal{F}^{-1,x}(\mathcal{F}^{-1,y}(\mathcal{F}^y(\mathcal{F}^x(\phi)))), \quad (4.19)$$

$$= (\mathbf{Q}_x \otimes \mathbf{I}_y \otimes \mathbf{I}_z)(\mathbf{I}_x \otimes \mathbf{Q}_y \otimes \mathbf{I}_z)\mathcal{F}^y(\mathcal{F}^x(\phi)). \quad (4.20)$$

The aforementioned method directly solves the Poisson equation given that the eigendecomposition of $\mathbf{T}^*$ is known, in particular for the x and y directions. For a non-uniform grid, this eigendecomposition must be computed numerically, since the non-uniform grid spacing alters the coefficients of $\mathbf{T}$ substantially. However, if the grid considered is uniform, this eigendecomposition has a known closed form that, in order to obtain it, one can use Fourier Transforms. The next sub-section explores the differences in applying the same operator-based algorithm with Fourier analysis (FA).

DT-based operator

If a certain direction is discretized using a uniform mesh, the resulting matrix $\mathbf{T}$ is a symmetric matrix, meaning that $a_{j+1} = c_j \wedge j \in [1, N_{x_k})$ where a_j and c_j are the subdiagonal and supradiagonal elements of the matrix, respectively. Due to this symmetry, and the fact that its coefficients are directly related to the discretization of the Laplace operator, its eigendecomposition is known analytically, at least for the simpler boundary conditions natively supported in *CaNS* (see [101]). Note how, in such a case, the similarity transformation from $\mathbf{T}_{x_k}$ to $\mathbf{T}_{x_k}^*$ is not necessary as calculating $\mathbf{Q}^{-1}$ is simple.

The closed form of these eigenvalues and eigenvectors, given by various discrete transforms (DTs) depending on the boundary conditions at play, is taken advantage of in two different steps of the solver. First, when computing the eigendecomposition directly by its closed form expression, effectively reducing any complexity needed from an algorithm which calculates an eigendecomposition numerically. Secondly, when applying the forward and backward operators in the solver. Since the structure of the matrix of eigenvectors is predictable, depending only on its dimensions (and thus the dimension of the direction in which the operator is being applied in), some of the calculations performed in a dense matrix multiplication are redundant and often repetitive, with these being omitted or reused from previous multiplications. Since these eigendecompositions are the result of known DTs, the gain in a reduction of the number of operations is analogous to the gains one experiences by calculating a Fourier transform (FT) with a Fast Fourier Transform (FFT) algorithm instead of calculating the FT using its definition. This is a special case of the matrix-based operator, as most of the rationale here is based on the methods employed in the matrix-based operator. Although mathematically they represent the same operator, they will be seen as distinct operations computationally, mostly due to their very different wall-clock times, as explored in Chapter 6.

Both of the aforementioned operators depend only on the discretization of the Poisson equation. Since the discretization is constant in time, the operators themselves do not vary in time. This means that the construction of the operators can and should only be done once, as a preparatory step to the actual solving algorithm. Although

⁴Here the x direction is considered, but the same applies to the term $(\mathbf{I}_x \otimes \Lambda_y \otimes \mathbf{I}_z)$.

⁵Operation count given for a 1-dimensional case.

the numerical (for the matrix-based operator) and analytical (for the DT-based operator) eigendecompositions have wall-clock times which are not comparable to the wall-clock time of solving the (pressure) Poisson equation⁶, this will inevitably result in a minimisation of the number of operations performed. Nevertheless, great effort has been in minimising the wall-clock time of the eigendecomposition (regardless of operator), in an attempt to future-proof the solver for non-constant discretization matrices, that inevitably will require calculating the eigendecomposition every time step.

Not all operators needed to solve the Navier-Stokes equations do not depend on time. So far, the implicit terms of the NS equations have been solved explicitly. As mentioned, solving these terms implicitly loosens restrictions regarding the minimum time step considered by creating the need for solving three Helmholtz equations, one for each velocity component of the flow. The following section explores how the operator based procedure can be applied to solve these equations.

4.3. Semi-implicit diffusion

As explained, the three Helmholtz equations originate from treating the diffusive terms of the Navier-Stokes equations semi-implicitly when calculating the prediction velocity:

$$\mathbf{u}^* = \mathbf{u}^n + \Delta t[\nu \nabla^2 \mathbf{u}^* - (\mathbf{u}^n \cdot \nabla) \mathbf{u}^n], \quad (4.21)$$

$$\mathbf{u}^* - \Delta t \nabla^2 \mathbf{u}^* = \mathbf{u}^n - (\mathbf{u}^n \cdot \nabla) \mathbf{u}^n = \mathbf{f}, \quad (4.22)$$

Where $\mathbf{f}$ means the right-hand side of the vector Helmholtz equation. The domain is the same as before Ω , but the discretization is different, as a staggered grid is considered. Each direction x_i is said to be discretized with $N_{x_i}^{u_{x_i}}$. So far, $N_{x_i}^\phi \equiv N_{x_i}$. Similarly, $N^{u_{x_i}} = \prod_{j=1}^n N_{x_j}^{u_{x_i}}$.

The discretization of these 3 equations with the finite-difference method is assembled into 3 matrix equations:

$$(\mathbf{I} - \Delta t \mathbf{A}_{x_k}) \mathbf{u}_{x_k}^* = \mathbf{f}_{x_k}, \quad \text{where } k \in \{1, 2, 3\}, \quad (4.23)$$

where $\mathbf{I}$ is the identity matrix of dimensions $N^{u_{x_i}} \times N^{u_{x_i}}$, $\mathbf{A}_{x_i}$ is the discretization matrix of Laplace operator for velocity component $\mathbf{u}_{x_i}^*$ and Δt is the time step of the temporal discretization.

From this point forward, this matrix equation is solved in the same procedure as the linear system of equations produced by the discretized Poisson equation. This is possible due to the fact that the discretized Helmholtz equation differs only from the discretized Poisson equation in that the diagonal of the matrix is the right-hand side of the equation, which is updated:

$$\left(-\frac{1}{\Delta t} \mathbf{I} + \mathbf{A}_{x_k}\right) \mathbf{u}_{x_k}^* = -\frac{1}{\Delta t} \mathbf{f}_{x_k}. \quad (4.24)$$

With this in mind, utilizing both the matrix-based and DT-based operators is a matter of updating the eigendecomposition of $(-\frac{1}{\Delta t} \mathbf{I} + \mathbf{A}_{x_k})$ each time step, particularly by subtracting only $-\frac{1}{\Delta t}$ from all eigenvalues.⁷ Noteworthy is the fact that the time step Δt is most often not constant throughout the advancement of time in the simulation, depending on the domain discretization but also on the velocity of the flow, meaning that each time this time step changes, so do the eigenvalues of $\mathbf{T}_{x_j}$. For non-uniform meshes, where a similarity transformation is required to ensure an eigendecomposition of a symmetric matrix $\mathbf{T}_{x_j}^*$, the procedure remains the same, as the similarity transformation is dependent only on the subdiagonal and supradiagonal elements of $\mathbf{T}_{x_j}$. No other changes are required for the operator-based procedure to solve the Helmholtz equations.

With the mathematical description of the operator-based procedure and the concrete operators finalized, a framework to support the implementation of the algorithm and its operators can be developed. The following section outlines the blueprint for implementation.

⁶See section 5.1 for details.

⁷See Appendix B for proof.

4.4. Operation count and memory footprint

With the appropriate procedures determined, there are multiple ways to execute it, each with differing computational complexities, memory footprints, and bottlenecks

When applying the matrix-based operator in one direction, a dense matrix multiplication is done with a vector. This matrix has dimensions $N_{x_k} \times N_{x_k}$, where $k \in \{1, 2, 3\}$, and the vector (either ϕ or $\mathbf{f}$) has dimensions $N \times 1$, meaning that this vector is sliced into strips along the direction x_k , each multiplied by $\mathbf{Q}_{x_k}$ (or $\mathbf{Q}_{x_k}^{-1}$). In practice, this transforms the matrix multiplication with a vector into a matrix-matrix multiplication. The matrix-vector multiplication results in $N_{x_k}^2$ multiplications and $(N_{x_k} - 1)N_{x_k}$ additions per strip. Considering only the x direction without loss of generality, there are $N_z N_y$ strips, the matrix-matrix multiplication results in an operator with $\mathcal{O}(N_z N_y N_x^2)$ operations. If one applies all operators except the operator for the z direction, the operation count is then of $\mathcal{O}(N_z(N_y N_x^2 + N_x N_y^2))$. [102] indicates that matrix-matrix multiplication kernels such as `dgemm` are mostly compute-bound, meaning that the aforementioned operation count will become highly relevant.

Still considering the x direction, in terms of memory footprint, this operator demands the storage of the eigenvalue matrix of each direction, requiring N_x of memory space per strip; and demands the storage of both the eigenvector matrix and its inverse, requiring $2N_x^2$ of memory space per strip. For the domain, the operator of each direction occupies $\mathcal{O}(N_z N_y N_x^2)$ in memory footprint. If one applies all operator except the operator for the z direction, the memory footprint required is then of $\mathcal{O}(N_z(N_y N_x^2 + N_x N_y^2))$.

For the DT-based operator, the gains are considerable in terms of operation count and memory footprint. When applying all operators except for the z direction, its operation count is $\mathcal{O}(N_z(N_y N_x \log N_x + N_x N_y \log N_y))$ and its required memory footprint is $\mathcal{O}(N_z(N_y N_x \log N_x + N_x N_y \log N_y))$. On the contrary, [102] mentions how FFT kernels, responsible for calculating the DTs, are I/O-bound, meaning that the difference in operation count compared to the matrix-based operator is significant enough to warrant a change in computational bound.

Finally, for the Tridiagonal Matrix Algorithm (TDMA) based on Gauss elimination, it can solve the tridiagonal system of equations with an operation count of $\mathcal{O}(N_z N_y N_x)$. There are no currently implemented operators with the same asymptotical computational complexity as TDMA.

With this established, all the necessary components to advance to a practical implementation of the numerical methods are gathered. As a way of aiding the development of the aforementioned implementation, a summary of the methods here conceived is presented below in the form of a list of steps required to solve both the discrete Poisson equation and the discrete Helmholtz equations.

4.4.1. Summary

Below is a systematic procedure that acts as the basis for the computational complexity analysis and the computational algorithm to be implemented in order to solve the discretized pressure Poisson equation:

0. Calculate the eigendecomposition of the discrete Laplace operator for directions x and y .
1. **Apply the forward operators** for directions x and y to the right hand side $\mathbf{f}$, computing $\mathcal{F}^y(\mathcal{F}^x(\mathbf{f}))$. – $\mathcal{O}(N_z(N_y N_x \log N_x + N_x N_y \log N_y))$ operations for the DT-based operator and $\mathcal{O}(N_z(N_y N_x^2 + N_x N_y^2))$ operations for the matrix-based operator.
2. **Solve the tridiagonal system** of equation 4.18 in the z direction with TDMA (Gauss elimination), computing $\mathcal{F}^y(\mathcal{F}^x(\phi)) - \mathcal{O}(N_z N_y N_x)$ operations
3. **Apply the backward operators** for directions x and y in reverse order of the forward passes to the correction pressure $\mathcal{F}^y(\mathcal{F}^x(\phi))$, thus computing $\phi = \mathcal{F}^{-1,x}(\mathcal{F}^{-1,y}(\mathcal{F}^y(\mathcal{F}^x(\phi))))$. – $\mathcal{O}(N_z(N_y N_x \log N_x + N_x N_y \log N_y))$ operations for the DT-based operator and $\mathcal{O}(N_z(N_y N_x^2 + N_x N_y^2))$ operations for the matrix-based operator.

If the diffusive terms of the Navier-Stokes equations are treated implicitly, solving one of the three Helmholtz equations would result in the following procedure:

0. Calculate the eigendecomposition of the discrete Laplace operator for directions x and y .
1. **Update the diagonal** of the eigenvalue matrix of x and y by subtracting from each eigenvalue the inverse of the time step, $\Lambda_{x_k} - \frac{1}{\Delta t}$.
2. **Perform steps 1, 2 and 3** of the procedure for solving the pressure Poisson equation.

In practice, neither the matrix-based operator nor the DT-based operator is implemented strictly according to the procedure described previously. Some adaptations are necessary to ensure that both memory and operations are preserved, thereby minimising the wall-clock time required to compute a solution to the Poisson equation. The following chapter describes in detail how these operators have been implemented into the DNS tool *CaNS*, from the libraries used to the functions called throughout the code.

5

Implementation

In the preceding chapter, a mathematical description of the procedure for solving the Poisson equation has been given. In the present chapter, a computational description of the adaptations to the aforementioned procedure is provided, with emphasis on obtaining the necessary eigendecomposition and implementing the matrix-based operator, whether in a CPU-based or GPU-based framework. Finally, these implementations are verified and validated against established works on canonical flows such as the lid-driven cavity and the turbulent square duct of Gavrilakis [2].

As reported earlier, the matrix-based operator has been implemented into the incompressible Navier-Stokes *CaNS* developed by Costa [1]. It is written in Fortran90, parallelized with a hybrid MPI–OpenMP (Message Passing Interface) approach to grant distributed-memory and shared-memory parallelization, respectively. The GPU-acceleration of *CaNS* is made using OpenACC, a programming standard which aims at simplifying the parallelization of code to GPU-accelerated systems by automating device memory management [103]. OpenACC is directive-based, meaning that the host code (in this case *CaNS*) is portable across systems. As the communication between host (CPU) and device (GPU) constitutes a severe bottleneck for general purpose GPU computation, great effort has been made at minimising this need by maintaining arrays in device for as long as possible. Although slightly outdated, [97] describes this and other details of the porting in more detail.

The domain is decomposed in a 2D *pencil*-like decomposition¹, often aligned with the z direction throughout the NS solver. However, when applying the matrix-based (and DT-based for that matter) operators, these pencils have to be aligned with the directions in which the operators are being applied, as it requires information along the entire direction to perform the matrix-vector multiplication for each strip (and the matrix-matrix multiplication for the entire pencil). This means that, after each application of the operator, the result is communicated (or transposed) to other processes in an MPI_ALL_TO_ALL operation, handled by the 2DECOMP&FFT library [72].

The code has been adapted in several aspects to accommodate non-uniform grids in the x and y directions prior to the implementation of the matrix-based operator by Dr. Pedro Costa. These include changes in arrays that handle most variables of the code (grid cell spacing, pressure and velocity fields, ...), changes to sub-routines that were previously hardcoded to have a uniform grid spacing as an input to their calculations, and a relevant change to the present work to the input file in the form of the inclusion of the variable `is_poisson_fft` in the `numerics` namelist. This variable is logical of dimension(2), which determines if the discretized Poisson equation is solved using the DT-based operator or the matrix-based operator. If `is_poisson_fft(:) = T, F`, it means that the decomposition of $\mathbf{T}_x$ is obtained with the DT-based operator and the decomposition of $\mathbf{T}_y$ is obtained with the matrix-based operator. As expected, it also indicates that the x direction is discretized with a uniform mesh, but does not indicate the uniformity of the mesh of the y direction, since the matrix-based operator in the implementations described below is able to handle uniform and non-uniform meshes.

With the aforementioned preliminary changes handled, the Poisson solver could be renovated using the matrix-based operator. But first, the matrix-based operator requires the numerical computation of the eigendecompositions of the discretization matrices for the second-order derivatives in the x and y directions. The following

¹Note how a pencil is different from the strip mentioned in 4.4, as the pencil is a collection of strips.

section describes how this numerical eigendecomposition is implemented in *CaNS*.

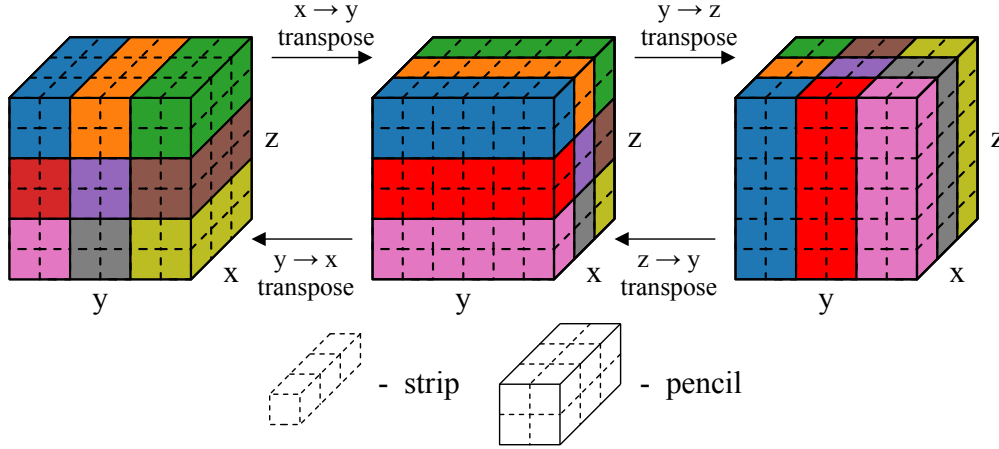


Figure 5.1: Diagram of the *pencil*-like domain decomposition and transposes utilized in *CaNS*. Each colour represents a different *pencil* attributed to an MPI rank. Pencils denote a collection of strips, and strips denote a collection of cells along the direction of alignment of the decomposition.

5.1. Obtaining the eigendecomposition

The eigendecompositions of $\mathbf{T}_x$ and $\mathbf{T}_y$ are computed based on the `is_poisson_fft` variable for the respective direction. For the present work, it depends only on the grid spacing of each direction, meaning that in practice procedure 1 is performed only once before the main time stepping loop commences. Here the eigendecomposition is being considered for the correction pressure ϕ . Calculating the eigendecomposition of the discretization matrices of the second-order derivatives for the velocity fields $\mathbf{T}_x^{u_k}$ and $\mathbf{T}_y^{u_k}$, where $u_k = \{u, v, w\}$, requires updating the $\mathbf{\Lambda}_{xy}$ matrix each time step by the scalar $-\frac{1}{\Delta t}$, which is not covered below but can be consulted in the repository².

Obtaining the eigendecomposition of $\mathbf{T}$ is rather straightforward as it follows closely to the mathematical procedure described in chapter 4. Calculating this eigendecomposition numerically creates additional overhead that is otherwise not present in the DT-based operator, meaning that it is of interest in quantifying the wall-clock time spent performing the main operations of the numerical eigendecomposition. Fig. 5.2 shows the evolution of the wall-clock time spent on calculating the eigendecomposition compared to the size of the computational domain.

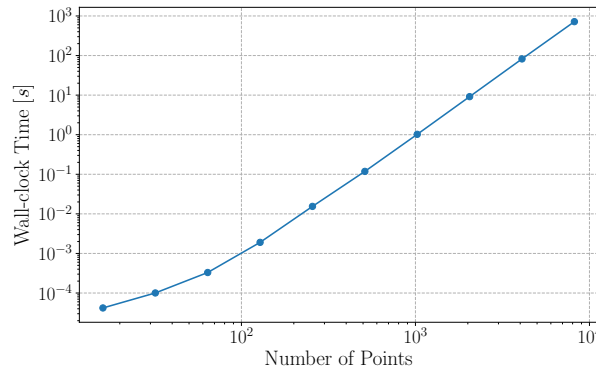


Figure 5.2: Timings for obtaining the eigendecomposition of $\mathbf{T}$ numerically with respect to the number of grid cells.

Algorithm 1 describes the LAPACK sub-routines used for calculating the numerical eigendecomposition of $\mathbf{T}_x$ and $\mathbf{T}_y$ depending on the direction's boundary conditions. Demmel et al in [104] describes how, for a symmetric tridiagonal matrix, the divide-and-conquer algorithm [105], implemented by the `stedc` sub-routine in the LAPACK

²<https://github.com/CaNS-World/CaNS>.

Algorithm 1 Obtaining the eigendecomposition for directions x and y , along with $\mathbf{T}_z$. `stedc` – Numerical eigendecomposition of tridiagonal matrix. `syevd` – Numerical eigendecomposition of symmetric matrix.

```

if is_poisson_fft(1) = T then
   $\Lambda_x \leftarrow$  DT-based Transform
else
  Calculate  $\mathbf{T}_x$  ▷ Here  $\mathbf{T}_x$  is always tridiagonal, regardless of BCs.
   $\mathbf{T}_x^* \leftarrow$  Similarity Transformation( $\mathbf{T}_x$ )
  if BCs(1)  $\neq$  P-P then
     $\Lambda_x^*, \mathbf{Q}_x^* \leftarrow \text{stedc}(\mathbf{T}_x^*)$ 
     $\Lambda_x \leftarrow \Lambda_x^*$ 
  else
     $\mathbf{T}_x^* \leftarrow$  Add non-tridiagonal coefficients( $\mathbf{T}_x^*$ )
     $\Lambda_x, \mathbf{Q}_x^* \leftarrow \text{syevd}(\mathbf{T}_x^*)$ 
  end if
   $\mathbf{Q}_x^{-1} \leftarrow \mathbf{Q}_x^{*T} \mathbf{D}_x^{-1}$ 
   $\mathbf{Q}_x \leftarrow \mathbf{D}_x \mathbf{Q}_x^*$ 
end if
if is_poisson_fft(2) = T then
   $\Lambda_y \leftarrow$  DT-based Transform
else
  Calculate  $\mathbf{T}_y$  ▷ Here  $\mathbf{T}_y$  is always tridiagonal, regardless of BCs.
   $\mathbf{T}_y^* \leftarrow$  Similarity Transformation( $\mathbf{T}_y$ )
  if BCs(2)  $\neq$  P-P then
     $\Lambda_y^*, \mathbf{Q}_y^* \leftarrow \text{stedc}(\mathbf{T}_y^*)$ 
     $\Lambda_y \leftarrow \Lambda_y^*$ 
  else
     $\mathbf{T}_y^* \leftarrow$  Add non-tridiagonal coefficients( $\mathbf{T}_y^*$ )
     $\Lambda_y, \mathbf{Q}_y^* \leftarrow \text{syevd}(\mathbf{T}_y^*)$ 
  end if
   $\mathbf{Q}_y^{-1} \leftarrow \mathbf{Q}_y^{*T} \mathbf{D}_y^{-1}$ 
   $\mathbf{Q}_y \leftarrow \mathbf{D}_y \mathbf{Q}_y^*$ 
end if
 $\Lambda_{xy} = \Lambda_x + \Lambda_y$ 
  Calculate  $\mathbf{T}_z$  ▷ Here  $\mathbf{T}_z$  is always tridiagonal, regardless of BCs.

```

library, is both fast and accurate compared to similar algorithms for calculating the matrices' eigendecomposition. Although no comparative reference was found regarding the algorithms available in LAPACK for calculating the eigendecomposition of a symmetric matrix, [104] was an incentive for choosing a similar divide-and-conquer approach for these matrices, implemented by the `syevd` sub-routine in the LAPACK library.

Although the time spent in this operation is deemed negligible compared to the wall-clock time dedicated to solving the Poisson equation through all time steps (compare figs. 5.2 and figs. 6.1), the former operation can add up in time were variable-coefficient discretization matrices considered, which will require calculating the aforementioned eigendecomposition every time step. Fortunately, that is not the case for the numerical method here implemented.

With the numerical eigendecomposition obtained, one may proceed to solving the discretized Poisson equation with the matrix-based operator. The following section describes the inner workings of the algorithm responsible for performing this task.

5.2. Computational algorithm

In practice, the discretized Poisson equation is solved in parallel across a number of processes. As described earlier, the application of the operators requires communication of the results of each of the pencils to different alignments, ensuring that the processes have the information they need to perform the operations in local memory. Algorithm 2 describes these communication steps as transposes of different variables that each indicate the direction in which the pencils are aligned. In this sense, `fx` represents the right-hand side of the discretized Poisson equation with pencils aligned with the x direction, meaning that `fx` has dimensions $\text{ng}(1) \times \text{n}(2)\text{n}(3)$, where $\text{ng}(k)$ represents $N_{x_k} \wedge k \in \{1, 2, 3\}$; and $\text{n}(k)$ represents the pencil size along the $k \in \{1, 2, 3\}$ direction. As an example, `py` represents the correction pressure decomposed with pencils aligned along the y direction, with dimensions $\text{ng}(2) \times \text{n}(1)\text{n}(3)$.

Algorithm 2 describes the multiple steps necessary to solving the discretized Poisson equation, with computational operations `fft` and `gemm` describing the sub-routines used for the CPU-based implementation. Although the sub-routines themselves are different, the GPU-based implementation follows the same procedure at solving the equation as the CPU-based implementation. As a note, the matrix-based operator here is shown to require a local transpose and de-transpose in order to ensure data contiguity when performing the matrix-matrix multiplication. Data contiguity is also required in the DT-based operator but the transposes necessary to ensure this are wrapped under the calls from the library responsible for applying the forward and backward FFT transforms (*CaNS* uses the FFTW3 library [106] for such transforms).

With the algorithm for solving the discretized Poisson equation settled, the next sections describe the CPU and GPU-based implementations of the algorithm, along with the libraries used for each respective version.

5.2.1. CPU-based implementation

As mentioned, the matrix-matrix multiplication necessary in the matrix-based operator is performed by the `gemm` sub-routine from the LAPACK library [107]. From here on out, the GEMM-based operator represents the CPU-based implementation of the matrix-based operator. In a similar fashion, the discrete transforms of the DT-based operators leverage FFT algorithms for computing these transforms with lower operation counts and memory footprint. Thus, the FFT-based operator represents the CPU-based implementation of the DT-based operator. The versions used for LAPACK, BLAS and OpenBLAS³ libraries are available under the Reproducibility Appendix.

In order to ensure the capability of solving the discretized Poisson equation in `single` and `double` precision, an interface was created in a separate submodule which points to the right sub-routine from LAPACK based on the precision used. In this case, by default the equation is solved using `double` precision, meaning that the `gemm` call is pointing to a `dgemm` call. Similarly, when solving with `single` precision, the `gemm` call is pointing to the `sgemm` sub-routine. Similar interfaces were created for the `stedc` and `syevd` sub-routines when calculating the eigendecomposition of $\mathbf{T}$ in algorithm 1.

As previously mentioned, the matrix-based algorithm has a larger memory footprint compared to the DT-based operator, and this translated to the GEMM-based and FFT-based operators respectively. This represented a challenge when managing available memory for large arrays as the eigenvector matrices $\mathbf{Q}$, and ended up being a

³OpenBLAS encapsulates the LAPACK and BLAS libraries into a single module in most HPC environments.

Algorithm 2 Solver of the discretized Poisson equation, assuming `ipencil_axis = 1`. `gemm` – GEneral Matrix-matrix Multiply sub-routine from the LAPACK library. `fwd/bwd_fft` – Forward and backward DT-based transforms from the FFTW3 library.

```

fx ← f
if is_poisson_fft(1) = T then
   $\mathcal{F}^x(\text{fx}) \leftarrow \text{fwd\_fft}(\text{fx})$ 
else
   $\mathcal{F}^x(\text{fx}) \leftarrow \text{gemm}(\mathbf{Q}_x^{-1}, \text{fx})$ 
end if
 $\mathcal{F}^x(\text{fy}) \leftarrow \text{transpose\_x\_to\_y}(\mathcal{F}^x(\text{fx}))$ 
if is_poisson_fft(2) = T then
   $\mathcal{F}^y(\mathcal{F}^x(\text{fy})) \leftarrow \text{fwd\_fft}(\mathcal{F}^x(\text{fy}))$ 
else
  Local transpose of  $\mathcal{F}^x(\text{fy})$ 
   $\mathcal{F}^y(\mathcal{F}^x(\text{fy})) \leftarrow \text{gemm}(\mathbf{Q}_y^{-1}, \mathcal{F}^x(\text{fy}))$ 
  Local de-transpose of  $\mathcal{F}^y(\mathcal{F}^x(\text{fy}))$ 
end if
 $\mathcal{F}^y(\mathcal{F}^x(\text{fz})) \leftarrow \text{transpose\_y\_to\_z}(\mathcal{F}^y(\mathcal{F}^x(\text{fy})))$ 
 $\mathcal{F}^y(\mathcal{F}^x(\text{pz})) \leftarrow \text{Gauss Elimination}(\mathbf{\Lambda}_{xy} + (\mathbf{I}_x \otimes \mathbf{I}_y \otimes \mathbf{T}_z), \mathcal{F}^y(\mathcal{F}^x(\text{fz})))$ 
 $\mathcal{F}^y(\mathcal{F}^x(\text{py})) \leftarrow \text{transpose\_z\_to\_y}(\mathcal{F}^y(\mathcal{F}^x(\text{pz})))$ 
if is_poisson_fft(2) = T then
   $\mathcal{F}^x(\text{py}) \leftarrow \text{bwd\_fft}(\mathcal{F}^y(\mathcal{F}^x(\text{py})))$ 
else
  Local transpose of  $\mathcal{F}^y(\mathcal{F}^x(\text{py}))$ 
   $\mathcal{F}^x(\text{py}) \leftarrow \text{gemm}(\mathbf{Q}_y, \mathcal{F}^y(\mathcal{F}^x(\text{py})))$ 
  Local de-transpose of  $\mathcal{F}^x(\text{py})$ 
end if
 $\mathcal{F}^x(\text{px}) \leftarrow \text{transpose\_y\_to\_x}(\mathcal{F}^x(\text{py}))$ 
if is_poisson_fft(1) = T then
   $\text{px} \leftarrow \text{bwd\_fft}(\mathcal{F}^x(\text{px}))$ 
else
   $\text{px} \leftarrow \text{gemm}(\mathbf{Q}_x, \mathcal{F}^x(\text{px}))$ 
end if

```

bottleneck on the computational size of the domains tested, partially explaining the upper limit of np tested in the weak scaling tests of figs. 6.3 and 6.8. A future implementation of the matrix-based operator could utilize the distribution of the larger arrays across various memory pools available in the system by using the SCALAPACK library [108]. In this case, one may have larger arrays divided into smaller chunks of memory that are stored across the local memory pools of the various processes, instead of having copies of the same large array in each local memory pool (in most cases, this means having copies across each of the nodes allocated), which are then communicated to the respective processes when performing the respective operations. As expected, this would yield in a higher communication bottleneck in the solver, but may allow for solving larger computational domains which have yet to be tackled in the literature.

Although all tests in the present work were performed solely on the CPU-based implementation due to time constraints, the GPU-based implementation was adapted from the procedure described in algorithm 2 by Dr. Pedro Costa. The next section explores the main differences between the CPU-based implementation here described and the GPU-based implementation.

5.2.2. GPU-based implementation

The GPU-based implementation differentiates itself from the CPU-based implementation mainly in terms of the explicit data movement necessary between the host and device, and in terms of the libraries and sub-routines needed. The present implementation utilizes OpenACC's directive-based approach to handling the memory spaces of the host and the device, automating much of the generation of the GPU-based code and emphasizing portability across systems.

The matrix-based operator was implemented using the cuBLAS library [109], a linear algebra library for NVIDIA GPUs. As in the CPU-based implementation, the `gemm` call is pointed to the respective call based on the precision of the solver, `cublasDgemm_v2` for double precision and `cublasSgemm_v2` for single precision. Unlike the CPU-based implementation, the GPU-based implementation does not require the local transposes for the y direction which ensure data contiguity, as the matrix-matrix multiplication is decomposed into multiple sub-matrix multiplication through tiled matrix multiplications, meaning that these tiles are simultaneously in shared-memory, negating the need for data contiguity in a column-major format [110]. It also means that very large matrix-matrix multiplications do not require for both matrices to reside in device memory in their entirety throughout the entire multiplication process, indicating that matrix-matrix multiplication of matrices larger than on-chip memory are possible [110]. Note that some algorithms for GEMM in GPUs might not fully utilize a tiling in a third dimension, meaning that a future implementation of an operator in the z direction might bring the need for explicitly transposing the large arrays in the host before communicating them to the device [110].

With both implementations completed, one needs to verify that these reproduce results in the literature, as means of establishing a reference point for future applications for which results are not yet known. The following section explores the verification and validation steps in the development of the matrix-based operator.

5.3. Verification & Validation

The verification of the solution to the Poisson equation in the present case is rather simple. As is the goal of the correction pressure to project the velocity field into a divergence-free space, one must make sure that the divergence obtained in the velocity field after the projection step is close to machine precision since the rest of the Navier-Stokes has been previously validated in [1, 97]. For both implementations this was observed.

The validation of the solution to the Poisson equation and of the Navier-Stokes solver is done by comparing the results obtained with the application of the matrix-based operator with established results in the literature with regards to various canonical flows. In order to compare the results of the present work with the work of Costa [1], a lid-driven cavity and a turbulent square duct were chosen as the physical test cases. Special interest is given to the turbulent square duct as it is a wall-bounded turbulent flow in two directions, meaning that it benefits from having non-uniform meshes along those two directions to fully discretize the turbulent boundary layer while keeping the inner core of the flow with a lower resolution. A visualization of the turbulent square duct done with the validation data obtained in section 5.3.2 is seen in fig. 5.3.

All input files necessary to recreate the setups below are given in the Reproducibility Appendix. The system used to simulate both validation cases is as described in Appendix D.

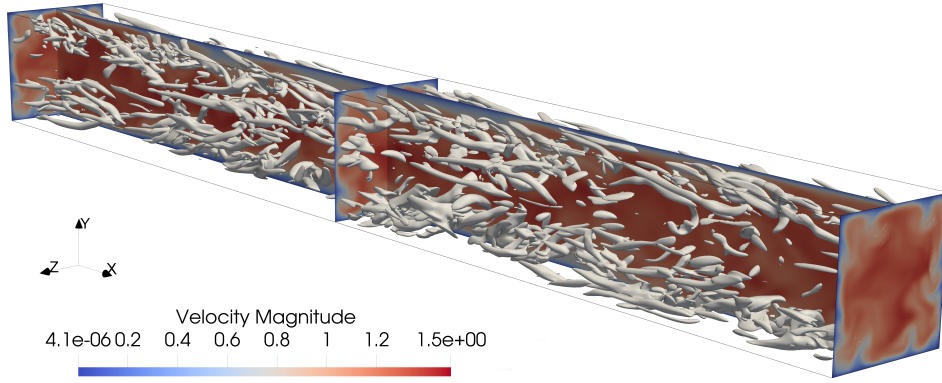


Figure 5.3: Visualization of the simulation of the turbulent square duct, with the contours refer to the velocity magnitude in the planes. Iso-contours of the Q -criterion shown with $Q = 5$.

5.3.1. Lid-driven cavity

A lid-driven cavity flow is simulated in a cubic domain with dimensions $[-\frac{h}{2}, \frac{h}{2}]^3 \wedge h = 1$, boasting homogeneous Dirichlet velocity boundary conditions along all walls and velocity components, except at the upper wall perpendicular to the z direction, which moves with velocity $\mathbf{u} = (0, 0, U_L) \wedge U_L = 1$. As per [1], the Reynolds number is prescribed as $Re = \frac{U_L h}{\nu} = 1000$. The flow is initialized with a zero-velocity field, meaning that in the first instances of the flow there is significant demand on the projection of the correction pressure onto the velocity field to guarantee mass conservation near the moving wall and its surroundings. Fig. 5.4 shows the velocity profiles taken from the centrelines $u(0, 0, z)$ and $w(x, 0, 0)$ of the lid-driven cavity flow.

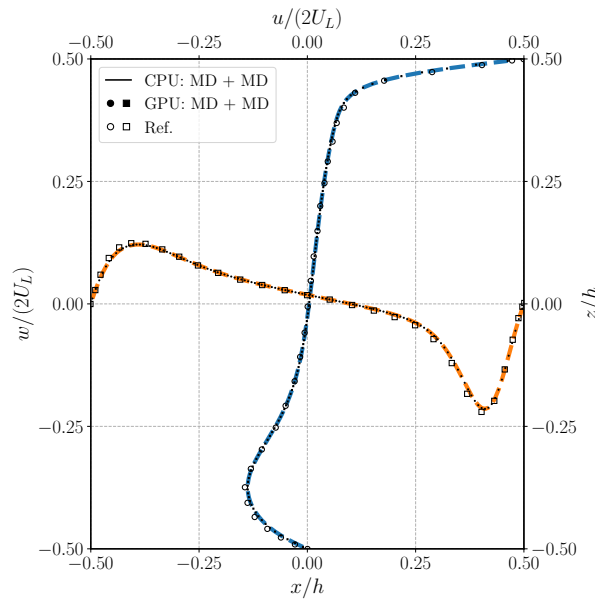


Figure 5.4: Distributions of the velocity components along the cavity centrelines for the lid-driven case at $Re = 1000$. Here MD + MD means both the x and y directions make use of the matrix-based operators for the forward and backward transformations of the right hand side of the discretized Poisson equation and the correction pressure, respectively. Reference data from [111].

It is clear that both the CPU-based and the GPU-based implementation agree with the reference data of Ku et al. [111] to a good degree. In this case, the three directions of the flow were discretized using uniform grid cell spacings as obtaining the solution to the lid-driven cavity flow does not require a high degree of discretization near the walls to ensure the flow is fully resolved (as empirically proven here). However, that is not the case of the turbulent square duct that is analysed in detail in the next section.

5.3.2. Turbulent square duct

A turbulent square duct of dimensions $10 \times 1 \times 1$ was simulated, with periodic boundary conditions in the x direction and homogeneous Dirichlet velocity boundary conditions for all other walls parallel to the x direction, in all velocity components. A computational domain size of $512 \times 128 \times 128$ grid points was considered, with a non-uniform discretization in the y and z directions. A grid stretching parameter of $gr=1$ was used for both directions with a mapping function that clusters points near both ends of the directions. This corresponds to a maximum to minimum grid cell spacing of $R = 1.27$.⁴ The following mapping function is used for the present validation case [112]:

$$z = \frac{1}{2} \cdot \frac{1 + \tanh\left(z_0 - \frac{1}{2}\right) gr}{\tanh\left(\frac{gr}{2}\right)}, \quad (5.1)$$

where gr is the grid stretching parameter ($gr=0$ yields a uniform grid). It provides a mapping from the uniform computational domain $z_0 \in [0, 1]$ to the non-uniform computational domain $z \in [0, 1]$.

As the x direction was discretized with a uniform grid spacing and the y direction was discretized with a non-uniform grid spacing, the operator chosen for the x direction was the DT-based operator, and for the y direction was the matrix-based operator (`is_poisson_fft(:) = T, F`). For this validation test, the GPU-based implementation was used, as the lid-driven cavity flow simulation showed no discernible differences between implementations.

The flow is initiated with a plane Poiseuille flow in the x direction with a mean velocity $u_{REF} = 1$. To drive the flow, a volume force is added as a pressure gradient that maintains a bulk streamwise velocity of $U_b = 1$. The stabilization of this force over time will determine when the flow reaches a statistically steady state, and thus, time averaging can be performed to obtain the flow statistics. The time evolution of this pressure gradient (represented as the Reynolds friction number) is available in fig. 5.5.

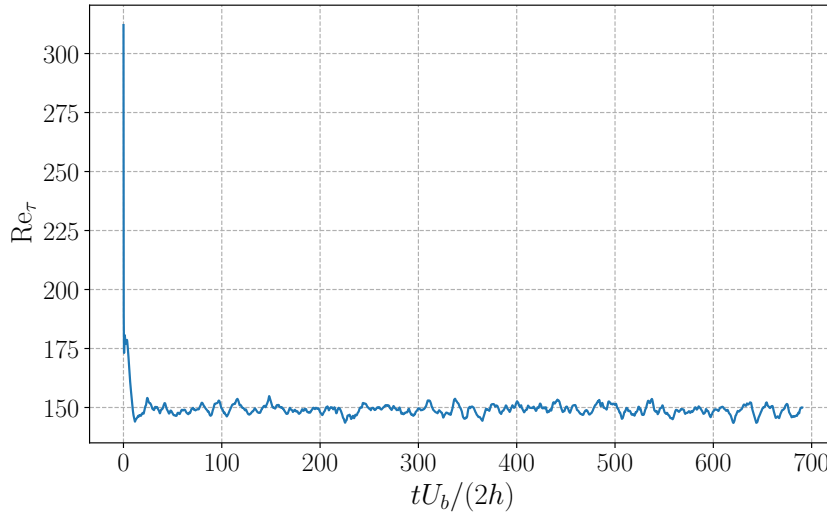


Figure 5.5: Time evolution of the Reynolds friction number $Re_\tau = \frac{u_\tau h}{\nu}$, where $u_\tau = \sqrt{-(dp/dx)h/2}$. A steady state value of $Re_\tau = 149.1$ is observed, near the $Re_\tau \approx 150$ reported by Gavrilakis [2].

The bulk Reynolds number is chosen as $Re = 4410$ in accordance with the works of Gavrilakis [2] and Costa [1]. Likewise, the steady state friction Reynolds number obtained is of $Re_\tau = 149.1$, in direct accordance with the simulations performed exclusively using the DT-based operator in [1] and close to the $Re_\tau \approx 150$ stated by Gavrilakis [2]. 200000 time steps were simulated over the total simulation duration. Velocity field data was extracted every 1000 time steps, but the ensemble averaging was only done from time step 25000 to guarantee the flow is statistically steady state. This means only 175 samples were used to obtain the mean flow necessary for the contour of the mean flow of fig. 5.6a and the mean streamwise velocity profile along the duct diagonal of fig. 5.6b. Due to the low sampling, an average of the four quadrants of the cross-section of the duct had to be done in order for the results to closely match those of Gavrilakis [2], in a "folding" and "unfolding" along the vertical and horizontal symmetry axis of the flow. The results agree to a good degree with the works of Gavrilakis [2].

⁴A detailed description of the non-uniform mesh can be found in Appendix C.

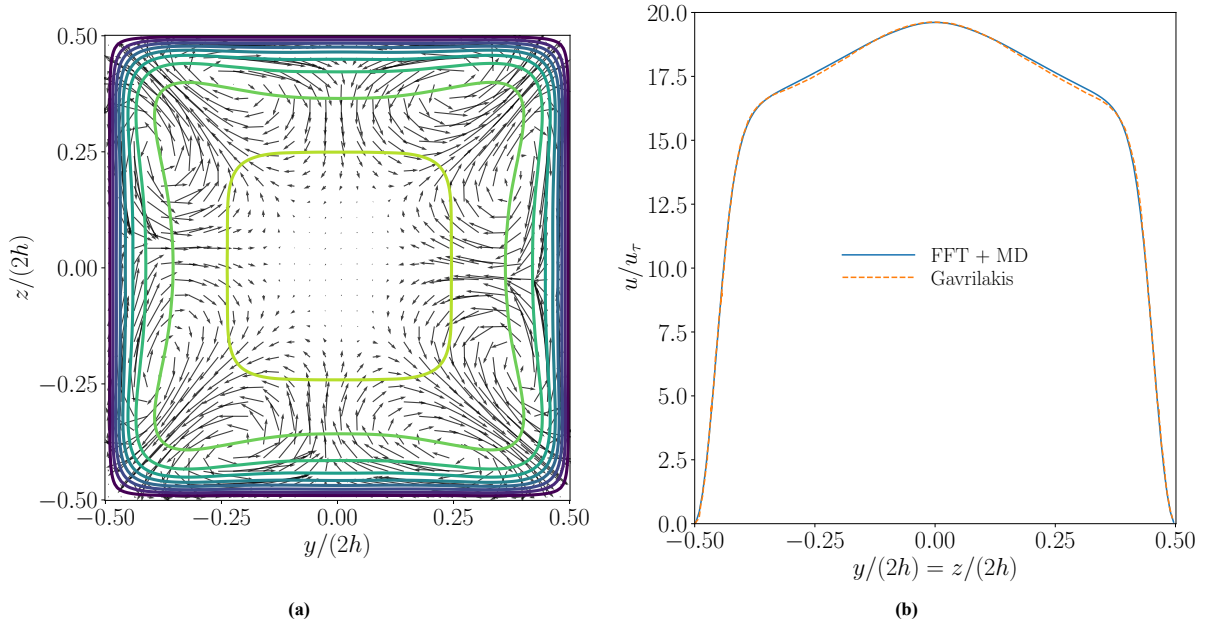


Figure 5.6: (a): Turbulent square duct mean flow represented by streamwise-velocity contours and in-plane velocity vectors. The contour levels start at $2u_\tau$ and increase with constant spacing. (b): Mean streamwise velocity profile sampled along the duct diagonal. Reference data from Gavrilakis [2].

The validation work presented shows that the implemented GEMM-based operator, along with its similarity transformation, corroborates with data from established works such as Gavrilakis [2], indicating the solution to the discretized pressure Poisson equation is being obtained in a reliable and correct manner. The next chapter thus explores the computational performance of the CPU-based implementation through various scaling tests, comparing the results with multigrid solvers and pre-conditioners. On top of that, it investigates the environmental and capital costs of implementing this operator on a large-scale environment.

6

Results

In the preceding chapters, the need for an algorithm that solves the Poisson equation in a fast, massively parallel manner has been described. This need has been partially met by the current work, with its mathematical description and computational implementation being previously dissected. In this chapter, the qualitative and quantitative performance of the current implementation of the developed algorithm is analysed, with emphasis on the measurement of the wall-clock time progression over strong and weak scaling setups. Moreover, its results are compared against the DNS solver *SNaC* [3] that leverages multigrid methods to solve the discretized Poisson equation, establishing a comparison between iterative and direct Poisson solvers. Finally, the environmental and capital costs of deploying this algorithm in a large-scale are examined.

6.1. CPU-based Computational Performance

As the aforementioned implementation is intended for deployment in a cluster-like environment, it is crucial to understand how the wall-clock time of the Poisson solver varies with increasing computational resources. Strong and weak scaling tests describe this evolution.

These tests were performed on the Dutch National supercomputer *Snellius*, in the rome partition. A job script is available in the Reproducibility Appendix, which includes the necessary libraries and their respective versions. All solutions were obtained with double precision for the respective Poisson equations.

6.1.1. Scaling Tests

The strong scaling test measures how wall-clock time changes with increased computational resources for a given computational domain size [113]. By selecting a constant computational setup and progressively increasing computational power, wall-clock time is expected to decrease, as each processor experiences a reduced workload. Ideally, this wall-clock time is inversely proportional to the number of processes, meaning that for a perfectly strong scaling algorithm, its wall-clock time reduces N -fold as the number of processes increases N -fold [113]. As $N \rightarrow \infty$, this proportionality will not hold, as the communication time between processes is expected to dominate the compute time.

In the present work, this test was performed to emulate the computational setup of the strong-scaling test of [1] and serve as a benchmark for the expected order of magnitude of the wall-clock time. The input files and parameters are available in the Reproducibility Appendix. In summary, a computational domain of size 1024^3 was considered for a physical domain of $l_k = 1 \wedge k \in \{1, 2, 3\}$, with all three directions being discretized using a uniform mesh. A zero initial velocity field is considered for the domain, with no slip and no penetration velocity boundary conditions in the walls perpendicular to the x and y directions. Accordingly, homogeneous Neumann-Neumann boundary conditions are set for the pressure in the x and y directions. The z direction is defined with periodic boundary conditions for both the velocity and pressure fields. A pressure gradient that balances the total wall shear forces the flow in the z direction, reaching a target bulk velocity of $U_b = 1$. A total of 50 time steps are simulated, ensuring timing results are independent from startup overhead.

The node was intentionally saturated by the computational setup chosen with an even decomposition, in the sense

that this was the setup with the largest amount of grid cells possibly accommodated by a single node's memory, and that the total number of grid cells is divisible by the number of cores in the node, thus its best scaling behaviour for lower core counts. The domain is decomposed in a *pencil*-like manner, where the variable `dims` represents the processor grid. The number of processes is factored into the two largest powers of 2 possible and the largest power is attributed to the second decomposed direction, meaning that either $2 \cdot \text{dims}(1) = \text{dims}(2)$ if $\sqrt{np} \bmod 1 \neq 0$ or $\text{dims}(1) = \text{dims}(2)$ if $\sqrt{np} \bmod 1 = 0$ such that $\text{prod}(\text{dims}(:)) = np$, where `np` is the number of processes allocated.

Fig. 6.1 describes the evolution of the wall-clock time to solve a single Poisson equation (also named the Poisson solve wall-clock time) with an increasing number of processes.

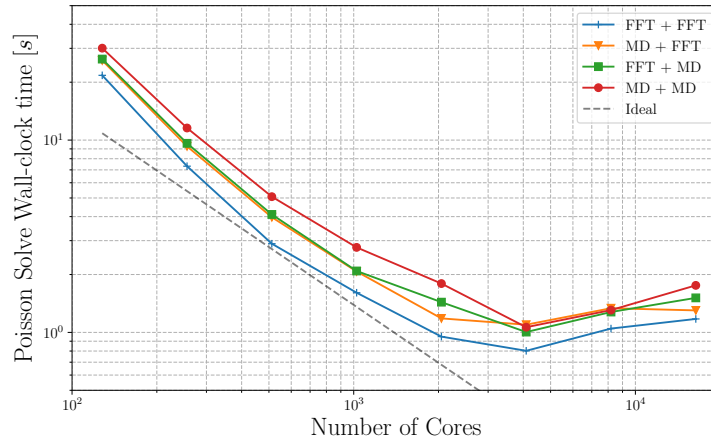


Figure 6.1: Strong scaling test of *CaNS* for all algorithmic configurations up to 16384 cores in a domain with 1024^3 grid cells in a uniform mesh. Data averaged from all (150) calls of the solver sub-routine. Performed in the *Snellius* supercomputer, in its *rome* partition.

Firstly and most evidently, utilizing FA in both decomposition directions (`is_poisson_fft = TT`) grants the lowest wall-clock time when compared to the GEMM-based algorithm (`is_poisson_fft = FF`). This goes in accordance with the operation count of both algorithms described in Chapter 4.

Secondly, all operator configurations show linear to supra-linear scaling performance up to 4096 cores, where performance degrades substantially thereafter. The aforementioned supra-linear behaviour can be explained by a potential cache bound or by the low communication overhead in the lower core counts. In the higher core counts, the communication bottleneck becomes significant to the point where the reduction in the local problem with increasing core count overwhelms the compute overhead of the algorithm(s).

To fully grasp the communication bottleneck in the CPU implementation, one may dissect the inner timings of the solver sub-routine for the respective FFT-based and GEMM-based algorithms:

Note that in the strong scaling test, the z direction is solving its tridiagonal system using a Gauss elimination kernel. This direction is periodic in its boundary conditions, meaning that the wall-clock time of the Gauss elimination component above reflects two passes of the respective operator.

It is now clear how communication becomes the greatest bottleneck in the Poisson solver, regardless of algorithm, with global transposes (the communication between processes) representing $\approx 99\%$ of the total wall-clock time for the FFT-based Poisson solver and $\approx 95\%$ of the total wall-clock time for the GEMM-based Poisson solver. This means that both solvers are heavily communication bottlenecked for higher core counts and compute bottlenecked for lower core counts.

The communication bottleneck observed more clearly from `np = 8192` can be hypothesized to be an artifact of the domain size chosen. Were the computational domain larger, then the total wall-clock time spent on computation would increase, regardless of the number of processes. This would mean that the current implementation would be compute bottlenecked until a higher core count, potentially not even showcasing the communication bottleneck observed within the range of number of processes tested.

Interestingly, one would expect the communication workload for each number of processes to be the same regardless of algorithm since the array being communicated in both cases has the same dimensions and is of the same

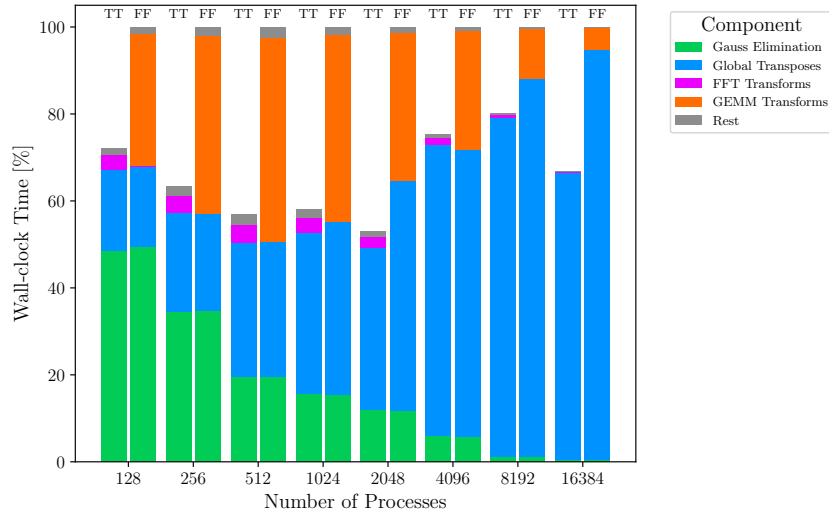


Figure 6.2: Timings of the solver sub-routine of *CaNS* for the cases of FFT + FFT and MD + MD. Data from the strong scaling tests of fig. 6.1.

type. It so happens that from $np = 1024$, great disparities in the wall-clock time spent on communication are observed between the FFT-based algorithm and the GEMM-based algorithm. Two hypotheses can explain this phenomenon.

Firstly, the tests for the FFT-based and GEMM-based algorithms, despite being run on the Snellius supercomputer, were not run on the exact same nodes. If one assumes that communication (latency) between nodes is not uniform, then the FFT-based tests may be performed on nodes with systematically lower communication overhead than the GEMM-based tests. This explains the difference in communication wall-clock time between $np = 1024$ and might even explain why, at $np = 4096$, a higher communication wall-clock time was observed for the FFT-based algorithm compared with the GEMM-based algorithm.

Secondly, external memory fragmentation could be occurring in differing degrees between the two algorithms. Since the algorithms have different memory footprints (see Chapter 4), the available memory chunks at the time of communication are different in number and size, potentially resulting in an allocation of memory which is less contiguous in the GEMM-based algorithm. Moreover, the variables being communicated (px , py and pz) might not fill entirely their last chunk of memory regardless of the number of processes being used, meaning that every instance of these variables creates internal fragmentation in the memory space. As one increases the number of processes used so does the local size of these variables reduce, potentially progressively creating a memory space with a higher degree of internal fragmentation. In turn, this could cause performance degradation in the communication workload as the number of processes increases [114], correlating to what is being observed in fig. 6.2.

A common critique of this test is that it can be seen as redundant [115]. One could argue that if a given setup is solvable with limited computational resources, then dedicating more resources to the problem is unnecessary. It is tempting to think that by increasing the available computational resources, one can simulate increasingly larger problems. Although this is true to a certain degree, the Poisson solve wall-clock time will only exclusively reduce in the parts of the solver where it is parallelizable when given more processes [115]. This process can be quantified in a weak scaling test, which sets the total number of grid cells in the domain as a function of the number of processes, thereby maintaining a constant compute workload per process. It is described in Fig. 6.3 for *CaNS* and its four algorithmic configurations.

For this scaling test, the physical domain is the same as the strong scaling test, being $l_k = 1 \wedge k \in \{1, 2, 3\}$. The computational domain depends on the growth direction of the domain, with the other two directions having 1024 cells. The direction of growth has $2 * np$ grid cells. The boundary conditions and initial conditions of the weak scaling test are the same as the strong scaling test. The same flow forcing is applied to the z direction, driving the flow. The same number of time steps (50) are simulated for the weak scaling test in comparison to the strong scaling test. Finally, the domain decomposition is set in the same fashion as described for the strong

scaling test. The Reproducibility Appendix describes the input and job script files required to recreate these tests on the *Snellius* supercomputer.

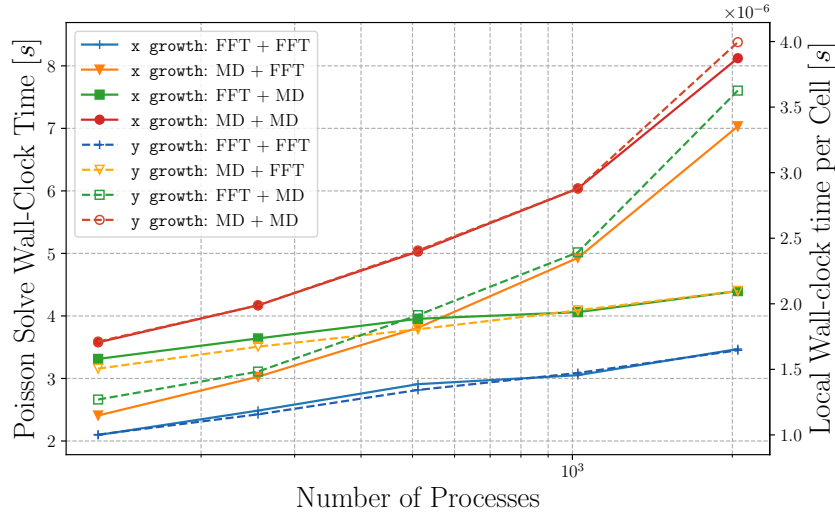


Figure 6.3: Weak scaling test of *CaNS* for all algorithmic configurations with domain growth in x and y and up to 2048 cores in a domain with $2 \cdot np \cdot 1024^2$ grid cells in a uniform mesh. Note how the FFT + MD and MD + FFT curves alternate positions according to the direction of domain growth. Data averaged from 150 calls of the solver sub-routine. Performed in the *Snellius* supercomputer, in its *rome* partition.

Ideally, since the compute workload is constant, one could expect to have a constant wall-clock time with an increasing number of processes. Despite this, broadcasting the solution of each sub-domain becomes progressively more expensive, resulting in an increase in wall-clock time.¹ In fact, as discussed in Chapter 4, the algorithms have differing computational complexities, meaning that the wall-clock time of the FFT-based operator scales by $\mathcal{O}(N \log N)$ and the GEMM-based operator by $\mathcal{O}(N^2)$. In the weak scaling test as the number of processes increases so does the problem size, meaning that one could also expect the wall-clock time of the solver to increase by $\mathcal{O}(np \log np)$ and $\mathcal{O}(np^2)$, respectively. It so happens that this is not the case for the current weak-scaling test.² One may estimate for this to be the case due to the fact that the Poisson solve wall-clock time encapsulates all operations made inside the solver sub-routine, whose components are described fig. 6.2. As such, the wall-clock time of the Gauss elimination kernel scales with $\mathcal{O}(N)$, and the scaling behaviour of the communication between processes with the total number of grid cells in the domain is unknown.

However, if the FFT and GEMM kernels are separated from the Poisson solver, a different picture emerges. Fig. 6.4 represents a fitting of the theoretical operation count of each computational operation (FFT and GEMM) to the respective timings obtained in the weak scaling test of fig. 6.3. Clearly, the timings obtained for the FFT-based operator relate closely to its theoretical operation count while the timings obtained for the GEMM-based operator do not relate to its operation count, as in the present weak scaling test the size of computational domain is proportional to the number of processes allocated. This could be due to the fact that the tests are being made in a domain which is not large enough for asymptotical operation count scaling to be observed. One could also explain these observations with the findings of [102]. Here, the GEMM kernel is observed to have a higher arithmetic intensity³ compared to the FFT kernel, noting that the GEMM kernel is compute bound in comparison to the memory bound of the FFT kernel. This means that increasing the number of processes will not grant improvements in memory traffic for the FFT kernel but will grant those to the GEMM kernel. As such, since the FLOPs⁴ are constant throughout the range of processes tested, the wall-clock time of the GEMM kernel could scale in a different manner compared to the scaling of its operation count.

As described in Section 5.2.1, the local transpose of the *py* variable between *i* and *j* ensures contiguity of the data needed for the GEMM-based operator when applied in the *y* direction, minimizing the potential communication

¹ Assuming homogeneous latency in the cluster, a finer domain decomposition results in higher communication workloads.

² Multiple attempts were made at fitting said curves to the scaling tests of FFT + FFT and MD + MD but none were successful.

³ Arithmetic intensity I - Ratio of work W to memory traffic Q . Denotes the ratio of floating point operations to total data movement (FLOPs/byte).

⁴ FLOPs - Floating Point Operations per Second.

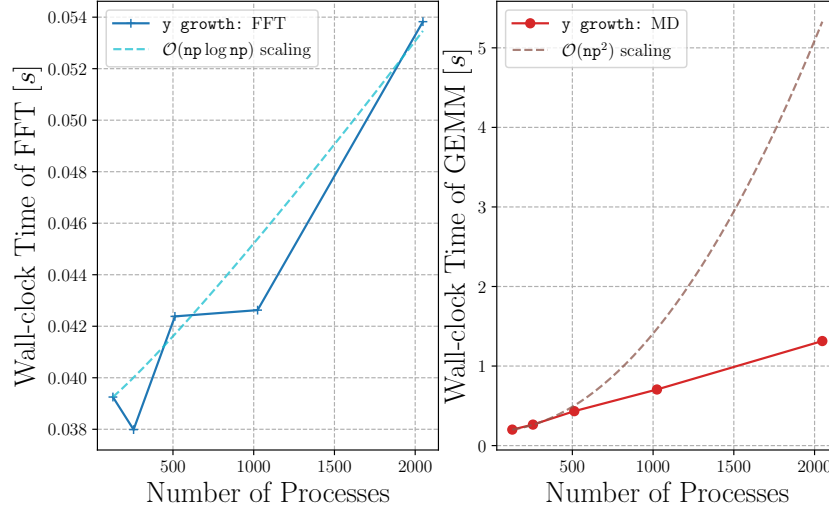


Figure 6.4: Fitting of theoretical operation count of each operator to timings obtained in the weak scaling test of fig. 6.3. *Left:* Fit of $\mathcal{O}(np \log np)$ to the timings of the FFT-based operator applied to the y direction. *Right:* Fit of $\mathcal{O}(np^2)$ to the timings of the GEMM-based operator applied to the y direction.

bottleneck that could be then created. As a corollary, the GEMM-based operator was reported to have lower wall-clock time when applied to the x direction instead of the y direction. When deploying the GEMM-based algorithm, if possible, an effort should be made to ensure that this operator is first applied in the x direction, $ng(1) < ng(2)$ and then in the y direction, since the wall-clock time of the operator scales quadratically with the number of grid cells in the direction it is operating in.

Despite the overall poorer performance of the GEMM-based algorithm, its need comes from the fact that the FFT-based operator requires a uniform mesh in the direction it is operating in, contrary to the GEMM-based operator. Nonetheless, there are situations where one may replace a non-uniform mesh with a uniform one simply for the performance gains (in wall-bounded flows, for example). Of course, this will require a higher number of grid cells to maintain the same degree of resolution in the areas of interest, potentially offsetting the performance gains acquired from applying the FFT-based operator. Determining the Pareto front of this trade-off is possible.

For it, consider a set of data points from fig. 6.1 of the FFT+FFT and MD+MD series. The ratio of the wall-clock time of these points gives the speedup between using the FFT-based approach in comparison to the GEMM-based approach, $s = \frac{t_w^{GEMM}(N_1)}{t_w^{FFT}(N_1)}$. Note how, in this case, $N_1 = 1024^3$. Defining N_2 as the computational domain size such that $t_w^{FFT}(N_2) = t_w^{GEMM}(N_1)$, one expects $N_2 > N_1$. Therefore, $t_w^{FFT}(N_2) = s t_w^{FFT}(N_1)$. In order to obtain N_2 , the relationship $t_w^{FFT} = t_w^{FFT}(N)$ for a given number of processes np must be known. Unfortunately, with the set of tests performed in the present work, this relationship is not explicitly known from the timing data. As such, the relationship is considered to be linear, such that $t_w^{FFT}(N) = \alpha N$, where $\alpha = \frac{t_w^{FFT}(N_1)}{N_1}$. N_2 is then found by $N_2 = \frac{N_1 t_w^{FFT}(N_2)}{t_w^{FFT}(N_1)} = s N_1$.

If this procedure is applied to the results obtained for $np=512$, one finds that a computational domain of size 513^3 has to be considered for the GEMM-based approach to yield the same wall-clock time as the FFT-based approach. If one considers $np=4096$, this computational domain size increases to 772^3 . $R = \frac{\Delta z_M}{\Delta z_m}$ can be chosen such that these reductions in resolution are usually valid for non-uniform grids [116].

Finally, special consideration should be given to the weak-scaling test above when evaluating GPU-based algorithm performance. It is intended for the GPU to be fully saturated when computing solutions (have 100% utilization) for two reasons: 1) this ensures that kernel launching/scheduling wall-clock time is kept to a minimum compared to the compute wall-clock time, 2) a GPU at high utilization gives the best performance per Watt of energy, minimising any potential idling resources. This makes the weak scaling test ideal for assessing performance for GPU-based implementations, and the aforementioned CPU-based weak scaling test the best approximation of the computational performance of the GPU-based implementation at the moment.

These tests reveal a harsh but reassuring truth: the GEMM-based operator scales and performs worse than the

FFT-based operator computationally. Despite this, the increased versatility of the GEMM-based operator grants it use in cases where two or more directions of the flow require a non-uniform mesh. Such cases include canonical flows such as the turbulent square duct analysed earlier in Section 5.3, but also enable the study of previously complex physical cases where proper conditions for the contemporary direct methods were not met. Regardless, some of these cases are already being simulated with high-performing state-of-the-art iterative solvers, such as those that utilize multigrid preconditioning. The next section aims at comparing the GEMM-based and FFT-based operators with the already established MG methods.

6.2. Alternative Methods - Multigrid

In the present work, the aforementioned CPU-based implementation of the FFT and GEMM-based operators have been compared against a high-performance implementation of iterative solvers featuring geometric multigrid methods *SNaC* [3]. This implementation utilizes the library *hypre* [42] for solving the Poisson equation which originates much in the same way as described in Chapter 4. Apart from the fact that it uses an iterative solver for the Poisson equation, *SNaC* is a multi-block code created to simulate otherwise impractical geometries such as a T-junction, which in *CaNS* have to be simulated using Immersed Boundary methods [117, 118]. Due to the versatility of the *hypre* library, *SNaC* is capable of solving the Poisson equation with various degrees of (relative) tolerance and with various solvers/pre-conditioners, being these controlled through the variables `hypre_tol` and `hypre_solver_i` in the `hypre.in` file.

Unfortunately, *SNaC* does not support GPU computing due to the inherent manner in which calculations are performed under geometric MG methods. Each time the fine grid is coarsened, the number of unknowns decreases roughly by $8\times$ (considering isotropic coarsening). GPUs excel under massive throughputs of data, meaning that while the finest level may run efficiently, the coarser levels result in an underutilized GPU [119]. Moreover, each coarsening and smoothing step represents a separate kernel to be launched and synchronized in the GPU, with such overhead potentially dominating computation in coarser grids. These factors contribute for the restricted use of geometric MG methods in modern GPU architectures.

The results that follow are given in terms of the ratio of the maximum to minimum cell spacing in the direction of consideration. A formula for this ratio is provided in the Appendix. The condition number is not given for the various test cases because for non-uniform meshes, denser areas of the mesh are considered higher frequency modes and lower frequency elsewhere. The assumption that coarse grids represent the low-frequency error of the fine grid breaks down for non-uniform meshes under geometric MG methods, making the condition number of the matrix an inadequate measure for the convergence rate in non-uniform meshes. On uniform grids, high and low frequency modes are cleanly separated in the fine and coarse grids. In non-uniform grids, the assumption that coarse grids represent low-frequency errors and fine grids represent high frequency errors is no longer valid as the locality of the non-uniform grid creates heterogeneity in the effectiveness of correcting errors through the smoothing/coarsening kernels.

6.2.1. Precision and tolerance effects

As aforementioned, *SNaC* allows for the Poisson equation to be solved with a user-chosen relative tolerance by controlling the `hypre_tol` variable. As expected, varying this tolerance projects the velocity onto a divergence-free space with different efficacies, a development which can be analysed in fig. 6.5. Remember that all solutions for the direct methods of *CaNS* were obtained with `double` precision, resulting in a divergence of the velocity field of *at least* $\mathcal{O}(10^{-9})$. Were the solutions obtained by solving the Poisson equation with `single` precision, a divergence of the velocity field of *at least* $\mathcal{O}(10^{-5})$.

A relative tolerance of $\mathcal{O}(10^{-6})$ for *SNaC* will then correspond to solving the Poisson equation with a direct method in `double` precision, and a relative tolerance of $\mathcal{O}(10^{-2})$ will correspond to solving the Poisson equation with a direct method in `single` precision. Fig. 6.5 also reveals the immense impact of grid stretching on the wall-clock time and iterations necessary for the MG solver to obtain the desired maximum divergence. As mentioned earlier, the coarser grid used to solve low-frequency errors in the fine grid might still have lower frequency errors that are not fully solved due to the very low density zones of the grid. In turn, this will demand a higher number of iterations for information to travel through the domain and thus a higher Poisson solve wall-clock time.

Now that a baseline for comparing results between the direct methods implemented and the alternative iterative methods with regard to their precision has been established, a reference comparison in terms of their computational performance can also be determined in the form of their single-core performance.

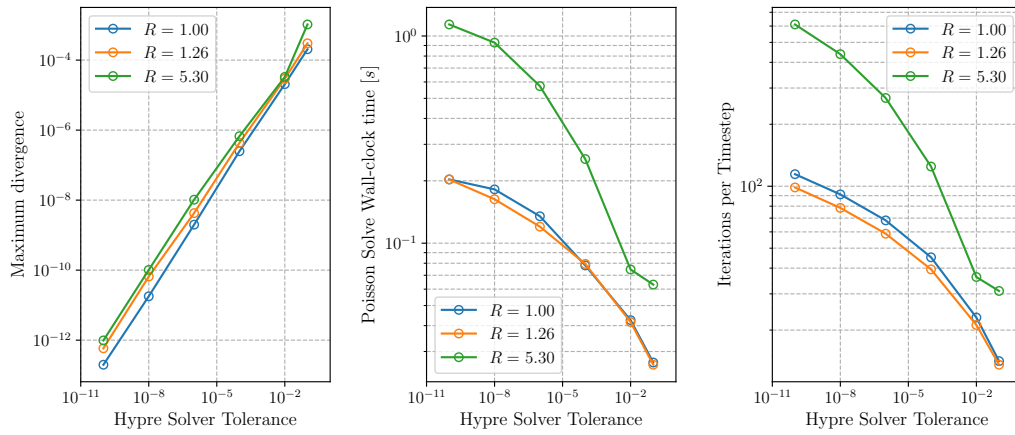


Figure 6.5: Precision analysis of *SNaC* from trial simulations of a lid-driven cavity of domain size 64^3 solved using PFMG with $np=8$. Simulations with $R = 2.32$ did not finish due to divergence of the solver. Values averaged over all (150) time steps. *Left:* Average of domain-maximum divergence per time step. *Centre:* Average Poisson solve wall-clock per time step. *Right:* Average iterations per time step.

6.2.2. Single-core performance

Although essentially no CFD simulations are currently performed in a sequential manner, evaluating the single-core performance of the algorithms allows separation between their inherent parallelizability and their purely serial convergence. If an algorithm has poor asymptotic convergence but a high degree of parallelizability, the performance obtained in scaling tests will not fully reflect all potential disadvantages of using this algorithm. Thus table 6.1 demonstrates this comparison.

Table 6.1: Single-core comparison between direct (FFT and MD) and iterative (MG) methods for a lid-driven cavity with $Re = 1000$. t_w represents the Poisson solve wall-clock time in seconds. s is the speedup, the ratio of the t_w of the method in question to the t_w of fastest method tested (FFT + FFT) in their respective conditions. DNF - Did not finish.

Domain size	R		Method							
			FFT	FFT	MD	MD	MG	MG	MG	MG
			+	+	+	+	SMG	PFMG	GMRES	BiCGSTAB
		FFT	MD	FFT	MD					
64^3	1	t_w	0.023	0.060	0.034	0.062	0.688	0.173	0.327	0.230
		s	1	2.61	1.47	2.69	29.95	7.52	14.22	10.03
	1.26	t_w	0.023	0.060	0.034	0.062	1.358	0.151	DNF	DNF
		s	1	2.61	1.47	2.69	59.14	6.59	DNF	DNF
	2.32	t_w	0.023	0.060	0.034	0.062	DNF	DNF	DNF	DNF
		s	1	2.61	1.47	2.69	DNF	DNF	DNF	DNF
128^3	1	t_w	0.228	0.546	0.401	0.744	7.397	1.739	3.096	2.140
		s	1	2.39	1.76	3.25	32.38	7.61	13.55	9.37
	1.27	t_w	0.228	0.546	0.401	0.744	DNF	1.578	3.314	2.167
		s	1	2.39	1.76	3.25	DNF	6.91	14.51	9.49
	2.35	t_w	0.228	0.546	0.401	0.744	DNF	1.697	DNF	DNF
		s	1	2.39	1.76	3.25	DNF	7.43	DNF	DNF

This reference benchmark enlightens the position of MG methods in the scope of the direct methods tested, on top of clarifying how the GEMM-based operator compares to the FFT-based operator on a small scale level.

Most obviously, the GEMM-operator degrades its performance to a worse degree than that observed in the large-scale scaling tests before. An explanation for this is the fact that the GEMM kernel is compute bound [102]. This means that it will take more advantage of increasing computing power (i.e. increasing parallelism) in comparison to the FFT kernel, which is memory bound [102]. Therefore, the speedup is roughly expected to decrease with increasing core count.

It is here clear how the worse asymptotical convergence of the GEMM-based operator results in wall-clock times that are over $1.5\times$ those of the FFT-based operator given the worsening of the performance when the number of grid cells in the domain is increased.

Regardless, the MG methods do not perform adequately in comparison to the direct methods tested, especially if one is not to use the most performant method of *SNaC*, PFMG. The minimum speedup one would obtain when computing the lid-driven cavity in the tested domain sizes would be of $s = 6.59$ when using the FFT-based operators and $s = 2.13$ when using the GEMM-based operators. Striking were the issues related to the inability of the simulations to finish, even for these single-core runs, a problem that direct methods do not suffer from. The reasons as to why this happened could be related to stability issues of the MG methods, but most likely are directly caused by the implementation here considered, which utilizes the *hypre* library. The inherent asymptotical operation count of the MG methods of $\mathcal{O}(N)$ still puts these methods as genuinely attractive, as the argument that the domains here tested (64^3 and 128^3) are insufficiently small for the difference between its asymptotical operation count and the operation count of the FFT-based operator $\mathcal{O}(N \log N)$ to become relevant. It should be reiterated that MG methods are capable of solving complex geometries where most direct methods fail, thereby earning their place among the plethora of available solvers.

Although from a mathematical perspective, these results are surprising, they demonstrate the practicality of the current direct methods and the limitations of relying solely on asymptotic operation counts. Table 6.1 illustrates how ignoring constant prefactors and implementation overhead in an algorithm's operation count can produce misjudgements, as asymptotical scaling often fails to predict behaviour at the scale of HPC environments. Realistic performance comparisons should use the total operation count (including constant prefactors and practical overheads) rather than asymptotical complexity alone.

As so, the following is an attempt at dissecting how both the FFT and GEMM-based operator, and the aforementioned MG methods have their computational performance evolve over the course of two different scales: 1) a small scale multi-core scenario, one of interest to enthusiasts who run CFD software locally; 2) a large scale massively-parallel scenario similar to most HPC deployments, for simulating setups which require funded computational time, one where minimising wall-clock time is a priority.

6.2.3. Multi-core performance

Frequently, full access to HPC environments is unnecessary to obtain useful solutions. Figure 6.6 thus compares the established and implemented direct and iterative methods in terms of the resulting Poisson solve wall-clock time in relation to the number of processes used. These simulations were performed locally on an AMD Ryzen AI 5 340 with 64 GB of memory.

Unsurprisingly, parallelism benefits (almost) all the methods tested, regardless of whether they are direct or iterative in their convergence. An exception is made for the MG test with a maximum to minimum grid cell spacing ratio of $R = 13.3$. Since the implementation of MG in *SNaC* is library-based, it is difficult to understand in-depth why this phenomenon is occurring.

A hypothesis comes from the fact that, for high mesh stretching, MG methods are not as efficient in solving the discretized Poisson equation: highly dense grid areas, when coarsened, result in cell spacing that might resemble the grid spacing of the fine grid in low density areas. To combat this, the selective coarsening of highly dense areas of the grid could be implemented in the form of agglomerated multigrid, but it is not the present case. An example of this in a turbulent square duct would be by coarsening the grid in the same way that it was clustered, meaning that areas only close to the walls are coarsened first. Subsequent coarsening steps could ideally target obtaining a uniform grid along all directions and then, if necessary, perform an isotropic coarsening cycle.

Another hypothesis stems from the manner in which parallelization could be implemented in *hypre*. It may happen that some MPI ranks are solving highly dense areas of the mesh while others are kept in an idle state waiting, as the solutions of the less dense areas of the mesh have been achieved. A possible remedy to this is the attribution of MPI ranks based on the density of the mesh, and not only exclusively based on the number of cells that the mesh has. Intuitively, if the mesh is denser near walls and these regions require more computational resources, the distribution of MPI ranks should be adjusted accordingly. Note how the present work, the computational domain considered is extremely small due to time constraints. This means that the gain expected from parallelizing the solver is not expected to be significant, which is roughly what is observed.

Regardless, the FFT-based operator and even the GEMM-based operator outperform the MG implementation in

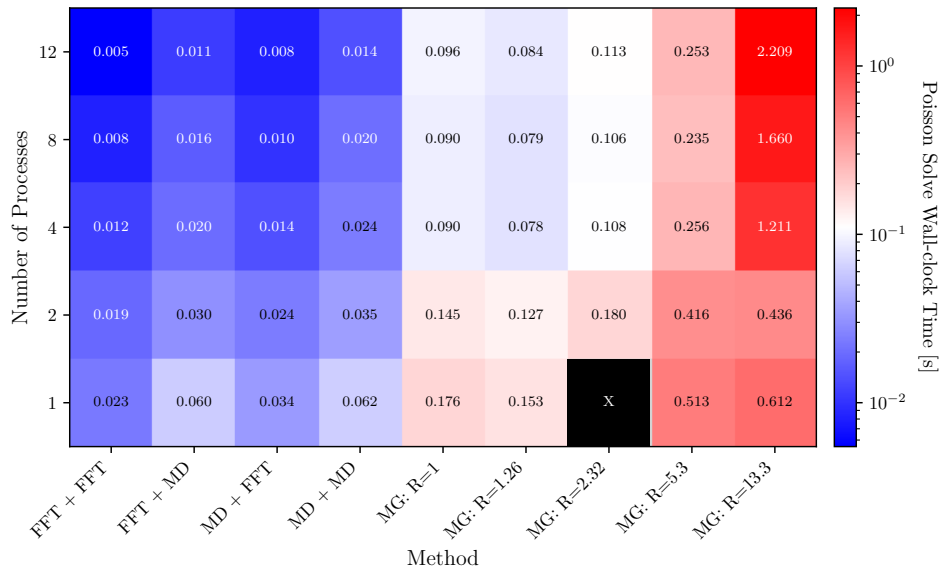


Figure 6.6: Multi-core comparison for a lid-driven cavity with domain size of 64^3 , $Re = 1000$, $hypre_tol = 1.e-4$ and $hypre_solver_i = \text{PFMG}$. For the direct methods, $X + Y$ means operator X was applied to direction x and operator Y was applied to direction y . R is the ratio of maximum to minimum grid cell spacing in the domain. The same ratio is applied to all three directions of the flow. Values averaged over all (150) calls of the solver sub-routine.

every aspect of this test, with the slowest test of a direct method being faster than the fastest test of an iterative method, irrespective of the number of processes or grid stretching (or lack thereof). For the current implementation of an iterative method, some solutions diverged or did not finish for various reasons, as observed here for $R = 2.32$ and $np=1$, which are not relevant in the context of these direct methods.

Once more, it could be argued that the domain size tested here is insufficient for the MG setup/smoothing processes to be close to negligible relative to the time spent computing the solution. While this argument is valid, the following strong and weak scaling tests of figures 6.7 and 6.8 performed on the *Snellius* supercomputer shed light on the scaling behaviour of the MG implementation at hand on a much larger domain.

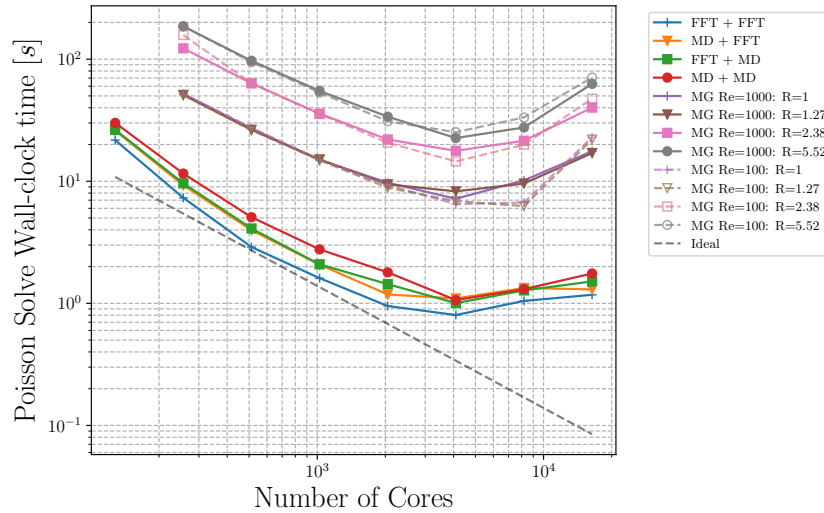


Figure 6.7: Strong scaling test of direct and iterative methods up to 16384 cores in a domain with 1024^3 grid cells. $hypre_solver_i = \text{PFMG}$ and $hypre_tol = 1.e-4$ constitute the iterative solver settings. For the direct methods, $X + Y$ means operator X was applied to direction x and operator Y was applied to direction y . R is the ratio of maximum to minimum grid cell spacing in the domain. The same ratio is applied to all three directions of the flow. Data averaged from all (150) calls of the solver sub-routine. Performed in the *Snellius* supercomputer, in its *rome* partition.

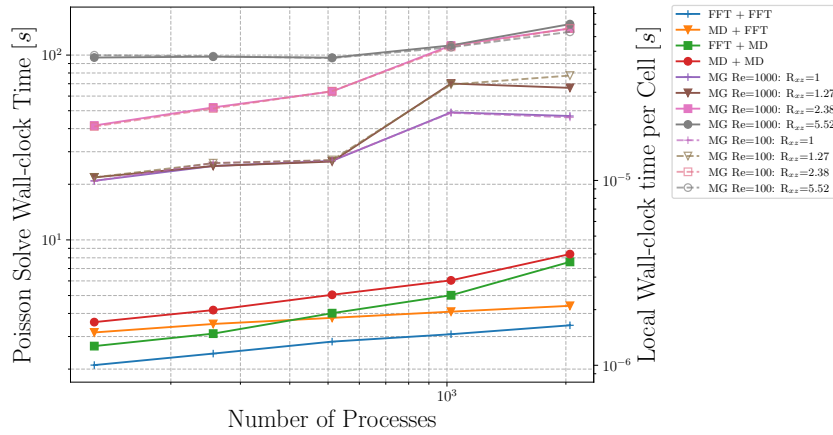


Figure 6.8: Weak scaling test of direct and iterative methods up to 2048 cores in a domain with $2 \cdot np \cdot 1024^2$ grid cells. `hybre_solver_i=PFMG` and `hybre_tol=1.e-4` constitute the iterative solver settings. For the direct methods, $X + Y$ means operator X was applied to direction x and operator Y was applied to direction y . R_{xz} is the ratio of maximum to minimum grid cell spacing in the x and z directions. In the y direction, R depends on the number of cores np used. Consult Appendix C. Data averaged from all (150) calls of the solver sub-routine. Performed in the *Snellius* supercomputer, in its *rome* partition.

It is undeniable that a point of diminishing returns was observed for all methods, regardless of asymptotic computational complexity or Poisson solve wall-clock time. At roughly $np=4096$, all methods experience non-negligible degradation in their performance despite having more computational resources available. In these tests, the MG method tested can be faster than a direct method when it comes to their raw Poisson solve wall-clock time, mentioning how for low grid stretching ratios the $np=4096$ and $np=8192$ tests were faster than the $np=128$ tests of direct methods. In spite of this, the differences in wall-clock time are minute, and the direct methods consume considerably less computational time and energy for the same solution. Even so, a domain with 1024^3 appears not to be sufficiently large to demonstrate the superior asymptotic convergence rate of the MG implementation, possibly justified by the potential orders of magnitude in the methods' prefactors. Regrettably, this domain size is approaching the limits of some clusters' ability to simulate in reasonable wall-clock time. Until severe hardware overhauls allow for more efficient communication between ranks, these pre-factors will dominate the asymptotic convergence rate.

Lastly, these tests were also an opportunity to evaluate the influence of the Reynolds number on the Poisson solve wall-clock time of the MG methods. It would seem that the values of Re tested did not allow for a significant difference to be observed between the wall-clock times of both values. This was to be expected, as the grid was not adjusted to compensate for the necessary resolution of both cases in terms of capturing the inertial and viscous length scales.

Compared with the multigrid implementation of *SNaC*, the direct methods developed here deliver superior computational performance by exploiting the highly consistent structure of the discretized Poisson equation on a Cartesian grid. Although (geometric) multigrid is versatile with favourable asymptotic convergence, it is not competitive with direct methods in this real-world case.

The last section of this chapter assesses the environmental and capital costs of using the developed methods, showing that deploying these algorithms on state-of-the-art HPC systems will have real consequences on carbon emissions and electricity consumption.

6.3. Environmental and capital cost of simulations

Having established that the implemented method is suited for massively-parallel environments, it is worth assessing the environmental and financial impact that deploying this algorithm could have. For this, a retrospective is made of the costs of testing the present implementation, in comparison to the established FFT-based operator and the iterative MG method. Moreover, a mock deployment of a problem is recreated as a way of establishing a reference budget for what a full-scale project could require. All costs/budgets will be given in kWh, € and SBU⁵,

⁵SBU – standard billing units, synonym for core hours for the *Snellius* supercomputer.

with thorough calculations being shown in the Appendix.

With this said, table 6.2 presents the costs of testing the methods with regard to their validity and computational performance. The results shown demonstrate how minimizing the wall-clock time of solving the Poisson equation can lower the environmental impact, on the funding of the project, and on the duration it may take in the future.

Table 6.2: Environmental and financial cost analysis of the tests performed for the present work. Calculation details available in the Appendix. (X*) means that test was repeated X times to assess different configurations or parameters in the computational setup. NA represents that the data is Not Available.

Program	Test	SBU	kWh	€	CO ₂ emitted (kg)	Elapsed (h)
CaNS	Strong Scaling	8162	45.7	6.03	14.5	4.06
	Weak Scaling (3*)	34807	197.2	26.03	62.7	10.10
	Turbulent square duct	NA	2.2	0.29	0.7	7.15
	Lid-driven cavity (4*)	NA	1.5	0.20	0.5	6.63
SNaC	Strong Scaling (2*)	122447	686.3	90.59	218.2	34.67
	Weak Scaling (2*)	208735	1193.7	157.57	379.6	61.79
	Rest	151550	888.0	117.21	282.4	37.00
	Total	525700	3014.6	397.93	958.6	161.39

Although interesting, these results lack relevance to most users. A reference large-scale simulation on an HPC environment is more fruitful to the average researcher when committing to a value of computational hours requested when funding computational projects.

For this purpose, a large Reynolds number turbulent square duct is used as a mock problem. In order to compare to established literature on the matter (see [120], [121]), a simulation at a friction Reynolds number $Re_\tau = 2000$ is chosen. Following the insight of Gavrilakis [2], the computational domain is set as $1(:) = 10., 1., 1..$ In order to achieve statistical convergence of the turbulent variables, 100000 time steps are simulated, in accordance to the procedure taken in section 5.3. After satisfying the necessary viscous and inertial length scales, one may arrive at the fact that $ng(:) = 4096, 1024, 1024$ sufficiently discretizes the flow, with x being the flow's direction with periodic boundary conditions.

Since one may have a non-uniform grid only along y and z , the x direction can have a uniform grid. This means that, for solving the pressure Poisson equation, one may solve the x direction with the FFT-based operator, the y direction with the GEMM-based operator, and the remaining tri-diagonal system of z with Gauss Elimination.

The described setup is exactly similar to the one used for the weak scaling test of *CaNS* (fig. 6.3) for $np=2048$, where a Poisson solve wall-clock time of $t_w = 4.4$ s was reported. If one decides to use more cores and assumes that performance degrades after $np=4096$ cores, the data from the strong scaling test of fig. 6.1 suggests a reduction in wall-clock time of around 30%, thus extrapolating to a Poisson solve wall-clock time in this scenario of $t_w = 3.08$ s. For the aforementioned 100000 time steps, the energy and capital costs of this mock simulation are as in table 6.3.

Table 6.3: Estimated environmental and financial cost of performing the simulation of turbulent square duct of domain size $ng(:) = 4096, 1024, 1024$. Note that the estimated wall-clock time does not take output routines into account. Parameters used for calculations available in the Appendix.

np	SBU	€	kWh	CO ₂ emitted [kg]	Estimated wall-clock time [h]
2048	250311	187	1416.6	450.3	122.2
4096	350435	262	1983.2	630.7	85.6

Conclusion

The present thesis aimed at developing a massively-parallel eigendecomposition-based framework for performing DNS of turbulent flows in non-uniform grids. So far, most DNS codes that solve the discretized pressure Poisson equation directly focused exclusively on utilizing FFT-based algorithms to efficiently obtain the eigendecomposition of the 1D second-order spatial derivative of the correction pressure [1, 16], represented as the matrix $\mathbf{T}$. Unfortunately, this approach requires for the direction in which this calculation is performed to be discretized using a uniform mesh. Often, this creates over-resolved regions in the flow, such as in the core region of wall-bounded flows. In these particular situations, iterative solvers that utilize multigrid (MG) methods are capable of solving the discretized pressure Poisson equation in non-uniform meshes, but these have been repeatedly been shown to boast higher wall-clock times compared to their direct counterparts [20]. To the best of our knowledge, there was no eigendecomposition-based implementation of a direct Poisson solver in a Navier-Stokes solver in both a many-CPU and many-GPU framework in the literature, where computational performance, scalability and cost was centrally evaluated.

Here, a generalization of the FFT-based approach of direct solvers of the discretized pressure Poisson equation was considered by calculating the eigendecomposition of $\mathbf{T}$ numerically with the aid of the LAPACK library [107], with consideration for the boundary conditions of the flow. This approach boasts a higher asymptotical operation count of $\mathcal{O}(N^2)$ for a 1D problem, compared to the asymptotical operation count of $\mathcal{O}(N \log N)$ for the FFT-based approach, and $\mathcal{O}(N)$ for a MG-based approach, where N is the number of grid cells.

The generalization of the FFT-based approach was implemented into the established DNS code *CaNS* [1], whose computational details can be seen in Chapter 5. In agreement, this generalization was implemented in a CPU-based and GPU-based framework. Both implementations have been verified to correctly solve the discretized Poisson equation by verifying that the velocity field is projected into a divergence-free space. These implementations were validated against established results in the literature by comparing velocity profiles of a lid-driven cavity and of a turbulent square duct. Both validations were successful, with special attention being given to the simulation of the turbulent square duct, which benefits from having non-uniform discretization in its cross-section.

For the CPU-based implementation, strong scaling tests of the eigendecomposition-based approach show good scaling up to $n_p=4096$ cores, with serious degradation from there on after. Fig. 6.2 demonstrates a communication bottleneck for higher core counts, where compute time represents only 5% of the total wall-clock time of the Poisson solver for the eigendecomposition-based approach and 1% of the total wall-clock time of the Poisson solver for the FFT-based approach. This has been hypothesized to be a result of the small domain tested (1024^3 grid cells) and possible memory fragmentation.

Although scaling tests were not performed for the GPU-based implementation, the weak scaling tests of the CPU-based implementation show that the eigendecomposition-based approach scales poorly in comparison to the FFT-based approach, indicating that such a result would also be observed for the GPU-based scaling tests.

In addition, these results were compared against an implementation that solves the discretized Poisson equation using iterative solvers that leverage MG methods. Regardless of the number of processes tested, the direct methods implemented in *CaNS* show lower wall-clock times compared to the solutions obtained with the iterative

solver. The scales tested ranged from a single core use case, to a multi-core use case, to testing in a HPC environment where these implementations have been designed to perform their best. The only criteria that allowed MG methods to be competitive in this specific scenario was by changing the relative tolerance to which they solve the discretized Poisson equation. Unfortunately, no tests were made comparing the performance of the direct solver obtaining solutions in single precision to the performance of the MG implementation when its relative tolerance is reduced to comparable levels.

Finally, the environmental and capital costs of deploying the eigendecomposition-based approach have been assessed. In the present work, 525000 core hours have been used in order to implement, debug and evaluate the performance of the solver, constituting ≈ 3000 kWh of energy, close to 1 ton of CO₂ being emitted, ≈ 400 € spent on electricity and over 160 hours of elapsed compute time. Deploying this method in large-scale turbulent square duct simulation would require between 300000 to 400000 core hours to obtain results over a period of time that allows for statistical convergence.

All in all, the eigendecomposition-based approach from the present work is deemed suitable to perform large-scale simulations on a many-CPU framework, with considerable speed-ups achieved when compared to iterative methods. Unfortunately, the FFT-based approach boasts still a lower wall-clock time in every scenario tested, meaning that it should be preferred to the eigendecomposition-based approach when uniform meshes are suitable. Although it was the goal of this work to be as complete as possible, many are the aspects in which it can be improved. Chapter 8 describes the future evolution of the project given more time and computational resources.

8

Recommendations

As it stands, the present implementation of *CaNS* is a powerful tool to perform DNS in uniform and non-uniform grids. The present work could act as a basis for future work related to the Poisson solver and its applications. Below are listed some short-term and long-term goals that could be achieved in the future.

Short-term goals:

1. *GPU-based performance*: In chapter 6, the computational and environmental performance of the CPU-based implementation was assessed through testing of the scaling behaviour at multiple scales. However, chapter 5 describes that a GPU-based implementation was also incorporated by Dr. Pedro Costa. It is still unknown how the GPU-based implementation performs under a strong and weak scaling test. This information could prove to be essential for future researchers who intend on using *CaNS*, as more often than not CFD codes are being deployed in GPU-based nodes [23].
2. *Precision-based performance*: All tests on *CaNS* were performed using double precision for the calculation of the correction pressure from the Poisson equation. As *CaNS* supports calculations using single precision [1], it would be of great interest to understand how the performance differs of the matrix-based operator changes with solver precision.
3. *Simulating complex non-uniform domains*: In chapter 3, it was the intention of the present work to develop an eigendecomposition-based Poisson solver such that DNS could be performed in non-uniform grids. Now that this has been accomplished, certain setups that were impracticable to simulate previously are now possible with the present work. Any physical setup that might require a non-uniform grid in more than one direction is suitable for this. An example of this is the turbulent square duct. Another example of this is the simulation of jets and plumes, that benefit from grid refinement in both the radial and downstream directions.

Long-term goals:

1. *Simulating larger domains*: In chapter 4, it is mentioned that the matrix-based operator has a larger memory footprint compared to the DT-based operator. In turn, this resulted in the simulation of smaller domains than those allowed by the DT-based operator due to out of memory (OOM) errors encountered. Reducing this memory footprint is essential in enabling the simulation of larger computational domains with the matrix-based operator. At the moment, the eigendecomposition of the discretized 1D second-order spatial derivative of the correction pressure is calculated simultaneously in each process, resulting in np copies of the eigendecomposition across the entire system. This could be mitigated by storing a portion of the eigendecomposition in the local memory pool of each process, such that the matrix-matrix multiplication is done in a distributed manner. The library ScaLAPACK [108] has this feature built-in in the form of SUMMA [122], algorithm that performs the GEMM call in a scalable manner.
2. *Simulating different geometries*: *CaNS* only simulates box-type geometries discretized with Cartesian grids. This is a severe limitation of the tool as most engineering applications require complex geometries for its physics to be reflected. *SNaC*, the multi-block solver that utilizes multigrid methods to solve the discretized

Poisson equation, circumvents this issue by enabling geometries divided into blocks, such as T-junctions [3]. It would be of great interest to implement the matrix-based operator as a substitute for the multigrid methods used here.

3. *Implementing variable-coefficients:* Many are the situations in which the coefficients of the pressure Poisson equation vary in time. These include the simulation of multiphase flows or stable stratified flows beyond the Oberbeck-Boussinesq assumptions without a pressure-splitting algorithm [123]. Chapter 4 describes how the present implementation takes advantage of the eigendecomposition calculated at the beginning of simulation and reuses it each time the matrix-based operator is applied (as the eigendecomposition of the matrix of the discretized pressure Poisson equation currently depends only on the geometry of the problem). While this would incur significant overhead in the Poisson solver (see fig. 5.2 for explicit timings of obtaining an eigendecomposition), it would allow for simulating computational setups in reduced wall-clock times that have yet not been solved in the literature.

Reproducibility

In the present work, the DNS tools *CaNS* and *SNaC* were used extensively for both implementing the matrix-based operator as well as benchmarking said implementation. The input files vary between the two tools, and thus their input parameters have different meanings. Regardless of the DNS code, the file `INFO_INPUT.md` of each repository dissects the physical and computational meaning of these input parameters, present in the following input files.

CaNS can be consulted at <https://github.com/CaNS-World/CaNS> and *SNaC* can be consulted at <https://github.com/p-costa/SNaC>. The CPU and GPU-based implementations were built on top of *CaNS* version 3.1.0. The new matrix-based approach will be made available in a separate release of *CaNS*.

Some alterations were made to both tools' code in order to time the many operations and sub-routines. These were timed using the sub-routines of the `timer.f90` file, which originate from the `first-timer` repository. To make use of them, the `TIMER` pre-processing macro has to be defined in the `build.conf` file (seen below), along with defining it in the `configs/flags.mk` file.

Verification & Validation

As mentioned, both the verification and the validation of the CPU and GPU-based implementations of the matrix-based operator was made in a workstation. The libraries installed in this workstation, at the time of benchmarking, are as follows:

- GCC version 13.3.0
- Open MPI version 4.1.6
- FFTW3 version 3.3.10
- OpenBLAS (BLAS/LAPACK) version 0.3.26
- CUDA/CUDA Compiler/cuFFT/cuBLAS Version 12.9.41
- NVHPC version 25.5.0
- CMake version 3.28.3
- NCCL version 2.26.05

Its system details are as follows:

- CPU: Intel Xeon W-2223
- GPU: NVIDIA RTX A4000
- Memory: 16 GB

Lid-driven cavity

The version of *CaNS* used for simulating the lid-driven cavity from the CPU-based implementation can be seen below. The input file describes all input parameters necessary to define the computational system of the lid-driven cavity. The lid-driven cavity was also simulated using the GPU-based implementation, and the respective `build.conf` file can be seen in the turbulent square duct section of this appendix.

Listing 8.1: `build.conf` file used for performing all CPU-based runs regardless of system.

```
1 #
2 # compiler and compiling profile
3 #
4 FCOMP=GNU           # options: GNU, NVIDIA, INTEL
5 FFLAGS_OPT=1        # for production runs
6 FFLAGS_OPT_MAX=0    # for production runs (more aggressive optimization)
7 FFLAGS_DEBUG=0      # for debugging
```

```

8 FFLAGS_DEBUG_MAX=0 # for thorough debugging
9 #
10 # defines
11 #
12 SINGLE_PRECISION=0 # perform the whole calculation in single precision
13 GPU=0               # GPU build
14 TIMER=1

```

Listing 8.2: input.nml file used for the validation of the CPU and GPU-based implementation of the matrix-based operator with the lid-driven cavity.

```

1 &dns
2 ng(1:3) = 128, 128, 128
3 l(1:3) = 1., 1., 1.
4 gtype(1:3) = 1, 1, 1, gr(1:3) = 0., 0., 0.
5 cfl = 0.95, dtmax = 1.e5, dt_f = -1.
6 visci = 1000.
7 inivel = 'zer'
8 is_wallturb = F
9 nstep = 20000, time_max = 100., tw_max = 0.1
10 stop_type(1:3) = T, F, F
11 restart = F, is_overwrite_save = T, nsaves_max = 0
12 icheck = 10, iout0d = 10, iout1d = 100, iout2d = 500, iout3d = 1000, isave = 2000
13 cbcvel(0:1,1:3,1) = 'D','D', 'D','D', 'D','D'
14 cbcvel(0:1,1:3,2) = 'D','D', 'D','D', 'D','D'
15 cbcvel(0:1,1:3,3) = 'D','D', 'D','D', 'D','D'
16 bcpre(0:1,1:3) = 'N','N', 'N','N', 'N','N'
17 bcvel(0:1,1:3,1) = 0.,0., 0.,0., 0.,1.
18 bcvel(0:1,1:3,2) = 0.,0., 0.,0., 0.,0.
19 bcvel(0:1,1:3,3) = 0.,0., 0.,0., 0.,0.
20 bcpre(0:1,1:3) = 0.,0., 0.,0., 0.,0.
21 bforce(1:3) = 0., 0., 0.
22 is_forced(1:3) = F, F, F
23 velf(1:3) = 0., 0., 0.
24 dims(1:2) = 2, 4, ipencil_axis = 1
25 /
26
27 &cudecomp
28 cudecomp_t_comm_backend = 0, cudecomp_is_t_enable_nccl = T, cudecomp_is_t_enable_nvshmem = T
29 cudecomp_h_comm_backend = 0, cudecomp_is_h_enable_nccl = T, cudecomp_is_h_enable_nvshmem = T
30 /
31
32 &numerics
33 is_impdiff = F, is_impdiff_1d = F
34 is_poisson_pcr_tdma = F, is_poisson_fft = F, F
35 /
36
37 &other_options
38 is_debug = T, is_timing = F
39 /

```

Turbulent square duct -- GPU-based implementation

The version of *CaNS* used for simulating the turbulent square duct from the GPU-based implementation can be seen below. The input file describes all input parameters necessary to define the computational system of the turbulent square duct. Finally, a bash script is shown that links the necessary libraries to *CaNS*. This is done by running said script in the root of the repository.

Listing 8.3: build.conf file used for performing GPU-based runs regardless of system.

```

1 #
2 # compiler and compiling profile
3 #
4 FCOMP=NVIDIA      # options: GNU, NVIDIA, INTEL
5 FFLAGS_OPT=1      # for production runs
6 FFLAGS_OPT_MAX=0  # for production runs (more aggressive optimization)
7 FFLAGS_DEBUG=0    # for debugging
8 FFLAGS_DEBUG_MAX=0 # for thorough debugging
9 #
10 # defines

```



```

11 #
12 SINGLE_PRECISION=0 # perform the whole calculation in single precision
13 GPU=1              # GPU build
14 TIMER=1
15 USE_NVTX=1

```

Listing 8.4: Bash script responsible for pointing *CaNS* to the installations of NVHPC along with dynamically linking the cuDecomp and other libraries.

```

1 #!/bin/bash
2 YEAR=2025
3 NVHPC_INSTALL_DIR=$HOME/software/nvidia/hpc_sdk
4 NVARCH=`uname -s`_`uname -m`; export NVARCH
5 NVCOMPILERS=$NVHPC_INSTALL_DIR; export NVCOMPILERS
6 MANPATH=$MANPATH:$NVCOMPILERS/$NVARCH/$YEAR/compiler/man; export MANPATH
7 PATH=$NVCOMPILERS/$NVARCH/$YEAR/compiler/bin:$PATH; export PATH
8
9 export PATH=$NVCOMPILERS/$NVARCH/$YEAR/comm_libs/mpi/bin:$PATH
10 export MANPATH=$MANPATH:$NVCOMPILERS/$NVARCH/$YEAR/mpi:$MANPATH
11 export PATH=$NVCOMPILERS/$NVARCH/$YEAR/profilers/Nsight_Systems/bin:$PATH
12 #
13 # communication libraries
14 #
15 export LD_LIBRARY_PATH=$NVHPC_INSTALL_DIR/$NVARCH/$YEAR/compiler/lib:$LD_LIBRARY_PATH
16 export LD_LIBRARY_PATH=$NVHPC_INSTALL_DIR/$NVARCH/$YEAR/comm_libs/mpi/lib:$LD_LIBRARY_PATH
17 export LD_LIBRARY_PATH=$NVHPC_INSTALL_DIR/$NVARCH/$YEAR/math_libs/lib64:$LD_LIBRARY_PATH
18 export LD_LIBRARY_PATH=$NVHPC_INSTALL_DIR/$NVARCH/$YEAR/comm_libs/nccl/lib:$LD_LIBRARY_PATH
19 export LD_LIBRARY_PATH=$NVHPC_INSTALL_DIR/$NVARCH/$YEAR/comm_libs/nvshmem/lib:
20   $LD_LIBRARY_PATH
21 export LD_LIBRARY_PATH=$PWD/dependencies/cuDecomp/build/lib:$LD_LIBRARY_PATH

```

Listing 8.5: input.nml file used for validating the GPU-based implementation of the matrix-based operator with a turbulent square duct.

```

1 &dns
2 ng(1:3) = 512, 128, 128
3 l(1:3) = 10., 1., 1.
4 gtype(1:3) = 1, 1, 1, gr(1:3) = 0., 1., 1.
5 cfl = 0.95, dtmax = 1.e5, dt_f = -1.
6 visci = 4410.
7 inivel = 'poi'
8 is_wallturb = T
9 nstep = 200000, time_max = 100., tw_max = 0.1
10 stop_type(1:3) = T, F, F
11 restart = F, is_overwrite_save = T, nsaves_max = 0
12 icheck = 200, iout0d = 0, iout1d = 0, iout2d = 50, iout3d = 1000, isave = 0
13 cbcvel(0:1,1:3,1) = 'P','P', 'D','D', 'D','D'
14 cbcvel(0:1,1:3,2) = 'P','P', 'D','D', 'D','D'
15 cbcvel(0:1,1:3,3) = 'P','P', 'D','D', 'D','D'
16 bcpre(0:1,1:3) = 'P','P', 'N','N', 'N','N'
17 bcvel(0:1,1:3,1) = 0.,0., 0.,0., 0.,0.
18 bcvel(0:1,1:3,2) = 0.,0., 0.,0., 0.,0.
19 bcvel(0:1,1:3,3) = 0.,0., 0.,0., 0.,0.
20 bcpre(0:1,1:3) = 0.,0., 0.,0., 0.,0.
21 bforce(1:3) = 0., 0., 0.
22 is_forced(1:3) = T, F, F
23 velf(1:3) = 1., 0., 0.
24 dims(1:2) = 0, 0, ipencil_axis = 1
25 /
26
27 &cudecomp
28 cudecomp_t_comm_backend = 0, cudecomp_is_t_enable_nccl = T, cudecomp_is_t_enable_nvshmem = T
29 cudecomp_h_comm_backend = 0, cudecomp_is_h_enable_nccl = T, cudecomp_is_h_enable_nvshmem = T
30 /
31
32 &numerics
33 is_impdiff = F, is_impdiff_1d = F
34 is_poisson_pcr_tdma = F, is_poisson_fft = T, F
35 /
36
37 &other_options

```

```

38 is_debug = T, is_timing = F
39 /

```

Results

All figures in chapter 6 were made using solutions calculated in the Dutch national supercomputer *Snellius*. The job scripts below indicate the versions considered for the libraries necessary to carry out the scaling tests. The system details of the rome partition can be seen in [124].

CaNS

As mentioned previously, the `build.conf` file considered for running the CPU-based implementation is the same as the one used in the workstation. Here example `input.nml` files can be seen for the strong and weak scaling tests of *CaNS*. One may obtain all results by changing the `is_poisson_fft` variable, the `ng` variable and the `dims` variable. For the weak scaling tests, depending on the direction of growth of the computational domain, `ng=2*np` for the direction of growth, and `ng=1024` for the other two directions. The domain decomposition, for both the strong and weak scaling test, are as explained in chapter 6.

Listing 8.6: Example `input.nml` file for the strong and weak scaling test of *CaNS* for `np=512` and applying the GEMM-based operator in the x and y directions.

```

1 &dns
2 ng(1:3) = 1024, 1024, 1024
3 l(1:3) = 1., 1., 1.
4 gtype(1:3) = 1, 1, 1, gr(1:3) = 0., 0., 0.
5 cfl = 0.95, dtmax = 1.e5, dt_f = -1.
6 visci = 1000.
7 inivel = 'zer'
8 is_wallturb = T
9 nstep = 50, time_max = 100., tw_max = 0.1
10 stop_type(1:3) = T, F, F
11 restart = F, is_overwrite_save = T, nsaves_max = 0
12 icheck = 10, iout0d = 50, iout1d = 0, iout2d = 0, iout3d = 0, isave = 0
13 cbcvel(0:1,1:3,1) = 'D','D', 'D','D', 'P','P'
14 cbcvel(0:1,1:3,2) = 'D','D', 'D','D', 'P','P'
15 cbcvel(0:1,1:3,3) = 'D','D', 'D','D', 'P','P'
16 bcpre(0:1,1:3) = 'N','N', 'N','N', 'P','P'
17 bcvel(0:1,1:3,1) = 0.,0., 0.,0., 0.,0.
18 bcvel(0:1,1:3,2) = 0.,0., 0.,0., 0.,0.
19 bcvel(0:1,1:3,3) = 0.,0., 0.,0., 0.,0.
20 bcpre(0:1,1:3) = 0.,0., 0.,0., 0.,0.
21 bforce(1:3) = 0., 0., 0.
22 is_forced(1:3) = F, F, T
23 velf(1:3) = 0., 0., 1.
24 dims(1:2) = 16, 32, ipencil_axis = 1
25 /
26
27 &cudecomp
28 cudecomp_t_comm_backend = 0, cudecomp_is_t_enable_nccl = T, cudecomp_is_t_enable_nvshmem = T
29 cudecomp_h_comm_backend = 0, cudecomp_is_h_enable_nccl = T, cudecomp_is_h_enable_nvshmem = T
30 /
31
32 &numerics
33 is_impdiff = F, is_impdiff_1d = F
34 is_poisson_pcr_tdma = F, is_poisson_fft = F, F
35 /
36
37 &other_options
38 is_debug = T, is_timing = T
39 /

```

Listing 8.7: Example job script for the strong scaling test of *CaNS* applying the GEMM-based operator in the x and y directions on the *Snellius* supercomputer.

```

1 #!/bin/bash
2 #
3 #SBATCH --job-name="FFT-FF-STRONG-SCALING-NP-512"
4 #SBATCH --partition=rome

```

```

5 #SBATCH --time=01:00:00
6 #SBATCH --ntasks=512
7 #SBATCH --exclusive
8 #SBATCH --mem=0
9
10 module purge
11 module load 2025
12 module load GCC/14.2.0
13 module load OpenMPI/5.0.7-GCC-14.2.0
14 module load FFTW/3.3.10-GCC-14.2.0
15 module load OpenBLAS/0.3.29-GCC-14.2.0
16
17 srun ./cans >> $SLURM_JOB_NAME.out

```

SNaC

For the present scaling tests, *SNaC* version 0.2.0 is considered. *SNaC* expects a version of hypre [42] installed, with its path indicated in the `src/Makefile`. For the present test, hypre version 3.0.0 was considered.

A `git diff` describes some minor alterations made to the `initflow.f90` and `param.f90` files in order to guarantee that the solution of the Poisson equation is non-trivial, regardless of boundary conditions. The `dns.in` and `block.001` files define the global and block-specific input parameters, respectively. Finally, an example job script can be seen for running *SNaC* on *Snellius*, which includes the modules loaded for said runs.

```

1 diff --git a/src/initflow.f90 b/src/initflow.f90
2 index a9f3167..a17443e 100644
3 --- a/src/initflow.f90
4 +++ b/src/initflow.f90
5 @@ -129,6 +129,7 @@ module mod_initflow
6     end if
7     if(is_wallturb) is_pair = .true.
8     if(is_pair) then
9 +   if(.false.) then
10        !
11        ! initialize a streamwise vortex pair for a fast transition
12        ! to turbulence in a pressure-driven channel:
13 @@ -154,26 +155,28 @@ module mod_initflow
14         end do
15     end do
16 end do
17 + else
18     !
19     ! alternatively, using a Taylor-Green vortex
20     ! for the cross-stream velocity components
21 -   ! (commented below)
22 +   ! (uncommented below)
23     !
24 -   !do k=lo(3),hi(3)
25 -   !   zcl = zc(k)/l(3)*2._rp*pi
26 -   !   zfl = zf(k)/l(3)*2._rp*pi
27 -   !   do j=lo(2),hi(2)
28 -   !       ycl = yc(j)/l(2)*2._rp*pi
29 -   !       yfl = yf(j)/l(2)*2._rp*pi
30 -   !       do i=lo(1),hi(1)
31 -   !           xcl = xc(i)/l(1)*2._rp*pi
32 -   !           xfl = xf(i)/l(1)*2._rp*pi
33 -   !           v(i,j,k) = sin(xcl)*cos(yfl)*cos(zcl)
34 -   !           w(i,j,k) = -cos(xcl)*sin(ycl)*cos(zfl)
35 -   !           p(i,j,k) = 0._rp!(cos(2._rp*xcl)+cos(2._rp*ycl))*(cos(2._rp*zcl)+2._rp)/16._rp
36 -   !       end do
37 -   !   end do
38 -   !end do
39 +   do k=lo(3),hi(3)
40 +   !   zcl = zc(k)/l(3)*2._rp*pi
41 +   !   zfl = zf(k)/l(3)*2._rp*pi
42 +   !   do j=lo(2),hi(2)
43 +   !       ycl = yc(j)/l(2)*2._rp*pi
44 +   !       yfl = yf(j)/l(2)*2._rp*pi

```

```

45 +      do i=lo(1),hi(1)
46 +          xc1 = xc(i)/l(1)*2._rp*pi
47 +          xf1 = xf(i)/l(1)*2._rp*pi
48 +          v(i,j,k) = sin(xc1)*cos(yf1)*cos(zc1)
49 +          w(i,j,k) = -cos(xc1)*sin(yf1)*cos(zf1)
50 +          p(i,j,k) = 0._rp!(cos(2._rp*xc1)+cos(2._rp*yf1))*(cos(2._rp*zc1)+2._rp)/16._rp
51 +      end do
52 +  end do
53 +  end do
54 +  end if
55 +  end if
56      deallocate(uid)
57  end subroutine initflow
58 diff --git a/src/param.f90 b/src/param.f90
59 index a07a77f..eae7aa7 100644
60 --- a/src/param.f90
61 +++ b/src/param.f90
62 @@ -44,6 +44,7 @@ character(len=1 ), dimension(0:1, 3) :: cbcpre
63 real(rp), dimension(0:1, 3) :: bcpre
64 integer, dimension(0:1, 3) :: inflow_type
65 character(len=100) :: inivel
66 +logical, , protected :: is_wallturb
67 !
68 real(rp) :: vol_all
69 integer :: my_block,id_first,nblocks,nrank
70 @@ -136,6 +137,7 @@ contains
71      read(iunit,*) bcpre(0,1 ), bcpre(1,1 ), bcpre(0,2 ), bcpre(1,2 ), bcpre(0,3 ), bcpre
72      (1,3 )
73      read(iunit,*) inflow_type(0,1),inflow_type(1,1),inflow_type(0,2),inflow_type(1,2),
74      inflow_type(0,3),inflow_type(1,3)
75      read(iunit,*) inivel
76 +      read(iunit,*) is_wallturb
77      my_block = iblock
78      id_first = sum(nranks(1:iblock-1))
79      do q=1,3

```

Listing 8.8: Example dns.in file for the strong and weak scaling tests of *SNaC* for np=512. dns.in define the shared/global parameters of the simulation.

```

1 .95 1.0e5 ! cfl, dtmin
2 1. 1. 1000. ! uref, lref, rey
3 50 100. 0.1 ! nstep,time_max,tw_max
4 T F F ! stop_type(1:3)
5 F F 0 ! restart,is_overwrite_save,nsaves_max
6 10 10 2000 5000000 100 200 ! icheck, iout0d, iout1d, iout2d, iout3d, isave
7 0. 0. 1. ! bforce(1:3)

```

Listing 8.9: Example block.001 file for the strong and weak scaling test of *SNaC* for np=512. block.XXX files define the local parameters of block XXX.

```

1 8 8 8 ! dims(1:3)
2 1024 1024 1024 ! ng(1:3)
3 0. 0. 0. ! lmin(1:3)
4 1. 1. 1. ! lmax(1:3)
5 0 0 0 ! gt(1:3)
6 0. 0. 0. ! gr(1:3)
7 F F D D D D ! cbcvel(0:1,1:3,1) [u BC type]
8 F F D D D D ! cbcvel(0:1,1:3,2) [v BC type]
9 F F D D D D ! cbcvel(0:1,1:3,3) [w BC type]
10 F F N N N N ! cbcpre(0:1,1:3 ) [p BC type]
11 1. 1. 0. 0. 0. 0. ! bcvel(0:1,1:3,1) [u BC value]
12 1. 1. 0. 0. 0. 0. ! bcvel(0:1,1:3,2) [v BC value]
13 1. 1. 0. 0. 0. 0. ! bcvel(0:1,1:3,3) [w BC value]
14 1. 1. 0. 0. 0. 0. ! bcpre(0:1,1:3 ) [p BC value]
15 0 0 0 0 0 0 ! inflow_type(0:1,1:3)
16 zer ! inivel
17 T ! is_wallturb
18 1 ! id

```

Listing 8.10: Example job script for the strong scaling test of *SNaC* for np=512 and gr(:) = 0. 0. 0..

```
1 #!/bin/bash
2 #SBATCH --job-name=SNAC-STRONG-SCALING-WALLTURB-T-NP-512-GR000
3 #SBATCH --partition=rome
4 #SBATCH --time=06:00:00
5 #SBATCH --ntasks=512
6 #SBATCH --cpus-per-task=1
7 #SBATCH --exclusive
8 #SBATCH --mem=0
9
10 module purge
11 module load 2025
12 module load GCC/14.2.0
13 module load OpenMPI/5.0.7-GCC-14.2.0
14 module load FFTW/3.3.10-GCC-14.2.0
15 module load OpenBLAS/0.3.29-GCC-14.2.0
16
17 export OMP_NUM_THREADS=1
18 srun --mpi=pmix -n $SLURM_NTASKS stdbuf -oL ./snac |& tee $SLURM_JOB_NAME.out
```


AI disclaimer

Throughout the development of the present work, some contributions were made using artificial intelligence. The model preferred was ChatGPT-5 Thinking mini from OpenAI. The following prompt was given to write the present disclaimer: *Summarize into bullet points the main contributions made by artificial intelligence in the thesis thus far. Take into consideration the many previous conversations about the subject.*

The response is as follows:

- Assisted in the conceptual design of extending a GPU-accelerated DNS solver (CaNS) to support non-uniform meshes by replacing FFT-based solvers with matrix-based formulations.
- Helped explore and compare numerical strategies for solving the Poisson and Helmholtz equations on non-uniform grids, including hybrid direct-iterative and domain-decomposed approaches.
- Supported the formulation of multi-block and interface-coupling ideas for structured grids with different spacings, including discussion of interpolation operators and interface continuity conditions.
- Provided guidance on performance-aware algorithm design for GPUs, including memory layout considerations and solver structure compatible with accelerator architectures.
- Assisted in identifying and refining physically relevant test cases (e.g., turbulent square duct flow) suitable for validating the modified numerical framework.¹
- Helped locate and contextualize relevant literature on iterative solvers, mixed discretizations, and numerical methods for non-uniform grids in CFD.
- Contributed to the clarification and professional phrasing of technical arguments, motivations, and limitations for inclusion in the thesis text.
- Supported the structuring and summarization of scientific material to improve clarity and coherence.

End of response.

Despite the many contributions of artificial intelligence, at any point was the present report written through it (with the exception of the previous points). All chapters in the present work were done exclusively by the author and revised by the supervisor. Beyond the aforementioned contributions, the following were also identified as relevant, which were not considered by the GPT-5 Thinking mini model:

- Clarified and explained the DT-based operator from relevant literature, including mathematical derivations on the eigendecomposition of the 1D second-order spatial derivative of the correction pressure.
- Clarified the inner workings of modern cyclic reduction algorithms, by dissecting the procedure, asymptotic operation counts and memory footprint.
- Assisted in prototyping a proof-of-concept Python script that demonstrated in an inefficient manner that the matrix-based operator solved the discretized Poisson equation.
- Developed many parsers for outputs from *CaNS* and *SNAC*, which allowed for the figures in chapter 6. On top of that, GPT-5 Thinking mini contributed to these figures by tackling unusual plots, such as fig. 6.2.

¹Note how the model, despite its best efforts, could not generate new physical test cases beyond those suggested by the author. However, it still claims credit for outputting its input.

References

- [1] P. Costa. “A FFT-based finite-difference solver for massively-parallel direct numerical simulations of turbulent flows”. In: *Computers & Mathematics with Applications* 76.8 (2018), pp. 1853–1862. ISSN: 0898-1221. DOI: 10.1016/j.camwa.2018.07.034. URL: <https://www.sciencedirect.com/science/article/pii/S089812211830405X>.
- [2] S. Gavrilakis. “Numerical simulation of low-Reynolds-number turbulent flow through a straight square duct”. In: *Journal of Fluid Mechanics* 244 (1992), pp. 101–129. DOI: 10.1017/S0022112092002982.
- [3] P. Costa. “A FFT-accelerated multi-block finite-difference solver for massively parallel simulations of incompressible flows”. In: *Computer Physics Communications* 271 (2022), p. 108194. ISSN: 0010-4655. DOI: 10.1016/j.cpc.2021.108194. URL: <https://www.sciencedirect.com/science/article/pii/S0010465521003064>.
- [4] D. Deming. “The aqueducts and water supply of Ancient Rome”. In: *Ground water* 58.1 (2019), p. 152.
- [5] A. N. Angelakis et al. “Water Dams: From Ancient to Present Times and into the Future”. In: *Water* 16.13 (2024). ISSN: 2073-4441. DOI: 10.3390/w16131889. URL: <https://www.mdpi.com/2073-4441/16/13/1889>.
- [6] A. Rizzi and J. M. Luckring. “Historical development and use of CFD for separated flow simulations relevant to military aircraft”. In: *Aerospace Science and Technology* 117 (2021), p. 106940. ISSN: 1270-9638. DOI: 10.1016/j.ast.2021.106940. URL: <https://www.sciencedirect.com/science/article/pii/S1270963821004508>.
- [7] R. Boukhanouf and A. Haddad. “A CFD analysis of an electronics cooling enclosure for application in telecommunication systems”. In: *Applied Thermal Engineering* 30.16 (2010). Selected Papers from the 12th Conference on Process Integration, Modelling and Optimisation for Energy Saving and Pollution Reduction, pp. 2426–2434. ISSN: 1359-4311. DOI: 10.1016/j.applthermaleng.2010.06.012. URL: <https://www.sciencedirect.com/science/article/pii/S1359431110002590>.
- [8] D. M. Sforza, C. M. Putman, and J. R. Cebal. “Computational fluid dynamics in brain aneurysms”. In: *International Journal for Numerical Methods in Biomedical Engineering* 28.6-7 (2012), pp. 801–808. DOI: 10.1002/cnm.1481. eprint: <https://onlinelibrary.wiley.com/doi/pdf/10.1002/cnm.1481>. URL: <https://onlinelibrary.wiley.com/doi/abs/10.1002/cnm.1481>.
- [9] P. G. Drazin and N. Riley. *The Navier-Stokes equations: a classification of flows and exact solutions*. 334. Cambridge University Press, 2006.
- [10] T. Kajishima and K. Taira. *Computational fluid dynamics: Incompressible turbulent flows*. Springer, 2016.
- [11] M. Honkanen. *Direct optical measurement of fluid dynamics and dispersed phase morphology in multi-phase flows*. Tampere University of Technology, 2006.
- [12] D. S. N. Abg Shamsuddin et al. “Computational fluid dynamics modelling approaches of gas explosion in the chemical process industry: A review”. In: *Process Safety and Environmental Protection* 170 (2023), pp. 112–138. ISSN: 0957-5820. DOI: 10.1016/j.psep.2022.11.090. URL: <https://www.sciencedirect.com/science/article/pii/S0957582022010643>.
- [13] A. Katal, M. Mortezaadeh, and L. (Wang. “Modeling building resilience against extreme weather by integrated CityFFD and CityBEM simulations”. In: *Applied Energy* 250 (2019), pp. 1402–1417. ISSN: 0306-2619. DOI: 10.1016/j.apenergy.2019.04.192. URL: <https://www.sciencedirect.com/science/article/pii/S0306261919308499>.
- [14] F. T. M. Nieuwstadt, J. Westerweel, and B. J. Boersma. *Turbulence: introduction to theory and applications of turbulent flows*. Springer, 2016.

- [15] J. Kim and P. Moin. “Application of a fractional-step method to incompressible Navier-Stokes equations”. In: *Journal of Computational Physics* 59.2 (1985), pp. 308–323. ISSN: 0021-9991. DOI: 10.1016/0021-9991(85)90148-2. URL: <https://www.sciencedirect.com/science/article/pii/0021999185901482>.
- [16] X. Zhu et al. “AFiD-GPU: A versatile Navier–Stokes solver for wall-bounded turbulent flows on GPU clusters”. In: *Computer Physics Communications* 229 (2018), pp. 199–210. ISSN: 0010-4655. DOI: 10.1016/j.cpc.2018.03.026. URL: <https://www.sciencedirect.com/science/article/pii/S0010465518300985>.
- [17] A. McAdams, E. Sifakis, and J. Teran. “A parallel multigrid Poisson solver for fluids simulation on large grids”. In: *Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. SCA ’10. Madrid, Spain: Eurographics Association, 2010, pp. 65–74.
- [18] C. Dick, M. Rogowsky, and R. Westermann. “Solving the Fluid Pressure Poisson Equation Using Multigrid—Evaluation and Improvements”. In: *IEEE Transactions on Visualization and Computer Graphics* 22.11 (2016), pp. 2480–2492. DOI: 10.1109/TVCG.2015.2511734.
- [19] C. Buneman. “A compact non-iterative Poisson solver”. In: *Stanford Univ. Inst. Plasma Res. Rep.* 294 (1969). URL: <https://cir.nii.ac.jp/crid/1571135649204959360>.
- [20] R. W. Hockney. “A Fast Direct Solution of Poisson’s Equation Using Fourier Analysis”. In: *Journal of the ACM* 12.1 (Jan. 1965), pp. 95–113. ISSN: 0004-5411. DOI: 10.1145/321250.321259. URL: <https://doi.org/10.1145/321250.321259>.
- [21] S. Ha, J. Park, and D. You. “A GPU-accelerated semi-implicit fractional-step method for numerical solutions of incompressible Navier–Stokes equations”. In: *Journal of Computational Physics* 352 (2018), pp. 246–264. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2017.09.055. URL: <https://www.sciencedirect.com/science/article/pii/S0021999117307283>.
- [22] B. L. Buzbee. “A Fast Poisson Solver Amenable to Parallel Computation”. In: *IEEE Transactions on Computers* C-22.8 (1973), pp. 793–796. DOI: 10.1109/TC.1973.5009160.
- [23] K. E. Niemeyer and C.-J. Sung. “Recent progress and challenges in exploiting graphics processors in computational fluid dynamics”. In: *The Journal of Supercomputing* 67.2 (Feb. 2014), pp. 528–564. ISSN: 1573-0484. DOI: 10.1007/s11227-013-1015-7. URL: <https://doi.org/10.1007/s11227-013-1015-7>.
- [24] S. Mukherjee et al. “Turbulent Flows Are Not Uniformly Multifractal”. In: *Physical Review Letters* 132 (18 May 2024), p. 184002. DOI: 10.1103/PhysRevLett.132.184002. URL: <https://link.aps.org/doi/10.1103/PhysRevLett.132.184002>.
- [25] C. M. Rhie and W.-L. Chow. “Numerical study of the turbulent flow past an airfoil with trailing edge separation”. In: *AIAA Journal* 21.11 (1983), pp. 1525–1532.
- [26] P. Wesseling. *Principles of computational fluid dynamics*. Vol. 29. Springer Science & Business Media, 2009.
- [27] J. C. Chien. “A general finite-difference formulation with application to Navier-Stokes equations”. In: *Computers & Fluids* 5.1 (1977), pp. 15–31. ISSN: 0045-7930. DOI: 10.1016/0045-7930(77)90003-2. URL: <https://www.sciencedirect.com/science/article/pii/0045793077900032>.
- [28] C. Brehm et al. “A comparison of higher-order finite-difference shock capturing schemes”. In: *Computers & Fluids* 122 (2015), pp. 184–208. ISSN: 0045-7930. DOI: 10.1016/j.compfluid.2015.08.023. URL: <https://www.sciencedirect.com/science/article/pii/S0045793015002996>.
- [29] R. Verstappen and A. Veldman. “Symmetry-preserving discretization of turbulent flow”. In: *Journal of Computational Physics* 187.1 (2003), pp. 343–368. ISSN: 0021-9991. DOI: 10.1016/S0021-9991(03)00126-8. URL: <https://www.sciencedirect.com/science/article/pii/S0021999103001268>.
- [30] S. K. Pandit, J. C. Kalita, and D. Dalal. “A fourth-order accurate compact scheme for the solution of steady Navier–Stokes equations on non-uniform grids”. In: *Computers & Fluids* 37.2 (2008), pp. 121–134. ISSN: 0045-7930. DOI: 10.1016/j.compfluid.2007.04.002. URL: <https://www.sciencedirect.com/science/article/pii/S0045793007000916>.

- [31] P. Moin and R. Verzicco. “On the suitability of second-order accurate discretizations for turbulent flow simulations”. In: *European Journal of Mechanics - B/Fluids* 55 (2016). Vortical Structures and Wall Turbulence, pp. 242–245. ISSN: 0997-7546. DOI: 10.1016/j.euromechflu.2015.10.006. URL: <https://www.sciencedirect.com/science/article/pii/S0997754615302582>.
- [32] A. J. Chorin. “Numerical solution of the Navier-Stokes equations”. In: *Mathematics of Computation* 22.104 (1968), pp. 745–762.
- [33] H. Qian, S. S. Sapatnekar, and E. Kursun. “Fast poisson solvers for thermal analysis”. In: *ACM Transactions on Design Automation of Electronic Systems* 17.3 (July 2012). ISSN: 1084-4309. DOI: 10.1145/2209291.2209305. URL: <https://doi.org/10.1145/2209291.2209305>.
- [34] M. A. Zaman. “Numerical Solution of the Poisson Equation Using Finite Difference Matrix Operators”. In: *Electronics* 11.15 (2022). ISSN: 2079-9292. DOI: 10.3390/electronics11152365. URL: <https://www.mdpi.com/2079-9292/11/15/2365>.
- [35] B. L. Buzbee, G. H. Golub, and C. W. Nielson. “On Direct Methods for Solving Poisson’s Equations”. In: *SIAM Journal on Numerical Analysis* 7.4 (1970), pp. 627–656. ISSN: 00361429. URL: <http://www.jstor.org/stable/2949380> (visited on 06/05/2025).
- [36] S. Yoon and D. Kwak. “Three-dimensional incompressible Navier-Stokes solver using lower-upper symmetric-Gauss-Seidel algorithm”. In: *AIJA Journal* 29.6 (1991), pp. 874–875. DOI: 10.2514/3.10671. eprint: <https://doi.org/10.2514/3.10671>. URL: <https://doi.org/10.2514/3.10671>.
- [37] N. Chentanez and M. Müller-Fischer. “A multigrid fluid pressure solver handling separating solid boundary conditions”. en. In: *IEEE Trans. Vis. Comput. Graph.* 18.8 (Aug. 2012), pp. 1191–1201.
- [38] R. Borrell et al. “Parallel direct Poisson solver for discretisations with one Fourier diagonalisable direction”. In: *Journal of Computational Physics* 230.12 (2011), pp. 4723–4741. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2011.02.042. URL: <https://www.sciencedirect.com/science/article/pii/S002199911100132X>.
- [39] R. Fedkiw, J. Stam, and H. W. Jensen. “Visual simulation of smoke”. In: *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH ’01. New York, NY, USA: Association for Computing Machinery, 2001, pp. 15–22. ISBN: 158113374X. DOI: 10.1145/383259.383260. URL: <https://doi.org/10.1145/383259.383260>.
- [40] N. Foster and R. Fedkiw. “Practical animation of liquids”. In: *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH ’01. New York, NY, USA: Association for Computing Machinery, 2001, pp. 23–30. ISBN: 158113374X. DOI: 10.1145/383259.383261. URL: <https://doi.org/10.1145/383259.383261>.
- [41] D. Mavriplis. “Multigrid Strategies for Viscous Flow Solvers on Anisotropic Unstructured Meshes”. In: *Journal of Computational Physics* 145.1 (1998), pp. 141–165. ISSN: 0021-9991. DOI: 10.1006/jcph.1998.6036. URL: <https://www.sciencedirect.com/science/article/pii/S0021999198960363>.
- [42] R. D. Falgout and U. M. Yang. “hype: A Library of High Performance Preconditioners”. In: *Computational Science — ICCS 2002*. Ed. by P. M. A. Sloot et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 632–641. ISBN: 978-3-540-47789-1.
- [43] A. H. Baker et al. “Scaling Hypre’s Multigrid Solvers to 100,000 Cores”. In: *High-Performance Scientific Computing: Algorithms and Applications*. Ed. by M. W. Berry et al. London: Springer London, 2012, pp. 261–279. ISBN: 978-1-4471-2437-5. DOI: 10.1007/978-1-4471-2437-5_13. URL: https://doi.org/10.1007/978-1-4471-2437-5_13.
- [44] P. Muldowney et al. “Preparing an incompressible-flow fluid dynamics code for exascale-class wind energy simulations”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC ’21. St. Louis, Missouri: Association for Computing Machinery, 2021. ISBN: 9781450384421. DOI: 10.1145/3458817.3476185. URL: <https://doi.org/10.1145/3458817.3476185>.
- [45] N. Margenberg et al. “A neural network multigrid solver for the Navier-Stokes equations”. In: *Journal of Computational Physics* 460 (2022), p. 110983. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2022.110983. URL: <https://www.sciencedirect.com/science/article/pii/S0021999122000456>.

- [46] U. Schumann and R. A. Sweet. "A direct method for the solution of poisson's equation with neumann boundary conditions on a staggered grid of arbitrary size". In: *Journal of Computational Physics* 20.2 (1976), pp. 171–182. ISSN: 0021-9991. DOI: 10.1016/0021-9991(76)90062-0. URL: <https://www.sciencedirect.com/science/article/pii/0021999176900620>.
- [47] F. W. Dorr. "The Direct Solution of the Discrete Poisson Equation on a Rectangle". In: *SIAM Review* 12.2 (1970), pp. 248–263. DOI: 10.1137/1012045. eprint: <https://doi.org/10.1137/1012045>. URL: <https://doi.org/10.1137/1012045>.
- [48] B. Buzbee, G. H. Golub, and C. Nielson. *The method of off/even reduction and factorization with application to Poisson's equation*. Tech. rep. Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 1969.
- [49] P. N. Swarztrauber. "The Methods of Cyclic Reduction, Fourier Analysis and the FACR Algorithm for the Discrete Solution of Poisson's Equation on a Rectangle". In: *SIAM Review* 19.3 (1977), pp. 490–501. ISSN: 00361445. URL: <http://www.jstor.org/stable/2029616> (visited on 06/05/2025).
- [50] R. B. Wilhelmson and J. H. Ericksen. "Direct solutions for Poisson's equation in three dimensions". In: *Journal of Computational Physics* 25.4 (1977), pp. 319–331. ISSN: 0021-9991. DOI: 10.1016/0021-9991(77)90001-8. URL: <https://www.sciencedirect.com/science/article/pii/0021999177900018>.
- [51] T. Hawkins. "Cauchy and the spectral theory of matrices". In: *Historia Mathematica* 2.1 (1975), pp. 1–29. ISSN: 0315-0860. DOI: 10.1016/0315-0860(75)90032-4. URL: <https://www.sciencedirect.com/science/article/pii/0315086075900324>.
- [52] B. Bialecki, G. Fairweather, and A. Karageorghis. "Matrix decomposition algorithms for elliptic boundary value problems: a survey". In: *Numerical Algorithms* 56.2 (Feb. 2011), pp. 253–295. ISSN: 1572-9265. DOI: 10.1007/s11075-010-9384-y. URL: <https://doi.org/10.1007/s11075-010-9384-y>.
- [53] R. E. Lynch, J. R. Rice, and D. H. Thomas. "Tensor product analysis of partial difference equations". en. In: *Bulletin of the American Mathematical Society* 70.3 (May 1964), pp. 378–385.
- [54] M. A. Hyman. "Non-iterative numerical solution of boundary-value problems". In: *Applied Scientific Research, Section B* 2.1 (Dec. 1952), pp. 325–351. ISSN: 0365-7140. DOI: 10.1007/BF02919780. URL: <https://doi.org/10.1007/BF02919780>.
- [55] W. G. Bickley, J. McNamee, and G. F. J. Temple. "Matrix and other direct methods for the solution of systems of linear difference equations". In: *Philosophical Transactions of the Royal Society of London. Series A, Mathematical and Physical Sciences* 252.1005 (1960), pp. 69–131. DOI: 10.1098/rsta.1960.0001. eprint: <https://royalsocietypublishing.org/doi/pdf/10.1098/rsta.1960.0001>. URL: <https://royalsocietypublishing.org/doi/abs/10.1098/rsta.1960.0001>.
- [56] M. R. Osborne. "Direct methods for the solution of finite-difference approximations to separable partial differential equations". In: *The Computer Journal* 8.2 (Jan. 1965), pp. 150–156. ISSN: 0010-4620. DOI: 10.1093/comjnl/8.2.150. eprint: <https://academic.oup.com/comjnl/article-pdf/8/2/150/1102074/080150.pdf>. URL: <https://doi.org/10.1093/comjnl/8.2.150>.
- [57] R. E. Lynch, J. R. Rice, and D. H. Thomas. "Direct solution of partial difference equations by tensor product methods". In: *Numerische Mathematik* 6.1 (Dec. 1964), pp. 185–199. ISSN: 0945-3245. DOI: 10.1007/BF01386067. URL: <https://doi.org/10.1007/BF01386067>.
- [58] E. Egerváry. "Über eine Methode zur numerischen Lösung der Poissonschen Differenzengleichung für beliebige Gebiete". In: *Acta Mathematica Academiae Scientiarum Hungarica* 11.3 (Sept. 1960), pp. 341–361. ISSN: 1588-2632. DOI: 10.1007/BF02020949. URL: <https://doi.org/10.1007/BF02020949>.
- [59] G. H. Golub et al. "A Fast Solver for Incompressible Navier–Stokes Equations with Finite Difference Methods". In: *SIAM J. Sci. Stat. Comput.*(submitted) (1994).
- [60] Y. Zhuang and X.-H. Sun. "A High-Order Fast Direct Solver for Singular Poisson Equations". In: *Journal of Computational Physics* 171.1 (2001), pp. 79–94. ISSN: 0021-9991. DOI: 10.1006/jcph.2001.6771. URL: <https://www.sciencedirect.com/science/article/pii/S0021999101967713>.

- [61] C. Temperton. "Direct methods for the solution of the discrete Poisson equation: Some comparisons". In: *Journal of Computational Physics* 31.1 (1979), pp. 1–20. ISSN: 0021-9991. DOI: 10.1016/0021-9991(79)90059-7. URL: <https://www.sciencedirect.com/science/article/pii/0021999179900597>.
- [62] M. K. Kadalbajoo and K. Bharadwaj. "Fast elliptic solvers—an overview". In: *Applied Mathematics and Computation* 14.4 (1984), pp. 331–355. ISSN: 0096-3003. DOI: 10.1016/0096-3003(84)90007-9. URL: <https://www.sciencedirect.com/science/article/pii/0096300384900079>.
- [63] M. Vajtersic. *Algorithms for elliptic problems*. en. Mathematics and its Applications. Dordrecht, Netherlands: Springer, Dec. 2010.
- [64] Z. Bai et al. "The Spectral Decomposition of Nonsymmetric Matrices on Distributed Memory Parallel Computers". In: *SIAM Journal on Scientific Computing* 18.5 (1997), pp. 1446–1461. DOI: 10.1137/S1064827595281368. eprint: <https://doi.org/10.1137/S1064827595281368>. URL: <https://doi.org/10.1137/S1064827595281368>.
- [65] J. A. George. *The use of direct methods for the solution of the discrete Poisson equation on non-rectangular regions*. Tech. rep. Stanford, CA, USA: Stanford University, 1970.
- [66] B. L. Buzbee et al. "The Direct Solution of the Discrete Poisson Equation on Irregular Regions". In: *SIAM Journal on Numerical Analysis* 8.4 (1971), pp. 722–736. ISSN: 00361429. URL: <http://www.jstor.org/stable/2949602> (visited on 06/06/2025).
- [67] L. Farnell. "Solution of poisson equations on a nonuniform grid". In: *Journal of Computational Physics* 35.3 (1980), pp. 408–425. ISSN: 0021-9991. DOI: 10.1016/0021-9991(80)90078-9. URL: <https://www.sciencedirect.com/science/article/pii/0021999180900789>.
- [68] V. Babu and S. A. Korpela. "On the Direct Solution of Poisson's Equation on a Non-uniform Grid". In: *Journal of Computational Physics* 104.1 (1993), pp. 93–98. ISSN: 0021-9991. DOI: 10.1006/jcph.1993.1011. URL: <https://www.sciencedirect.com/science/article/pii/S0021999183710119>.
- [69] X. Wang et al. "A high-order compact difference scheme for 2D Laplace and Poisson equations in non-uniform grid systems". In: *Communications in Nonlinear Science and Numerical Simulation* 14.2 (2009), pp. 379–398. ISSN: 1007-5704. DOI: 10.1016/j.cnsns.2007.07.017. URL: <https://www.sciencedirect.com/science/article/pii/S1007570407002493>.
- [70] J. Castagna and Y. Yao. "Development of a multi-block high-order DNS code for turbulent flow simulations". In: *IOP Conference Series: Materials Science and Engineering* 10 (June 2010), p. 012027.
- [71] M. Saczek et al. "Hybrid MPI/Open-MP acceleration approach for high-order schemes for CFD". In: *Journal of Physics: Conference Series* 1101 (Oct. 2018), p. 012031.
- [72] N. Li and S. Laizet. "2decomp & fft—a highly scalable 2d decomposition library and fft interface". In: *Cray user group 2010 conference*. 2010, pp. 1–13.
- [73] A. Mayo and A. Greenbaum. "Fast Parallel Iterative Solution of Poisson's and the Biharmonic Equations on Irregular Regions". In: *SIAM Journal on Scientific and Statistical Computing* 13.1 (1992), pp. 101–118. DOI: 10.1137/0913006. eprint: <https://doi.org/10.1137/0913006>. URL: <https://doi.org/10.1137/0913006>.
- [74] S. F. Ashby and R. D. Falgout. "A Parallel Multigrid Preconditioned Conjugate Gradient Algorithm for Groundwater Flow Simulations". In: *Nuclear Science and Engineering* 124.1 (1996), pp. 145–159. DOI: 10.13182/NSE96-A24230. eprint: <https://doi.org/10.13182/NSE96-A24230>. URL: <https://doi.org/10.13182/NSE96-A24230>.
- [75] A. McAdams, E. Sifakis, and J. Teran. "A Parallel Multigrid Poisson Solver for Fluids Simulation on Large Grids." In: *Symposium on Computer Animation*. Vol. 65. 2010, p. 74.
- [76] M. Ament et al. "A Parallel Preconditioned Conjugate Gradient Solver for the Poisson Problem on a Multi-GPU Platform". In: *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*. 2010, pp. 583–592. DOI: 10.1109/PDP.2010.51.
- [77] J. Xu. "Fast Poisson-based Solvers for Linear and Nonlinear PDEs". In: *Proceedings of the International Congress of Mathematicians 2010 (ICM 2010)*. 2010, pp. 2886–2912. DOI: 10.1142/9789814324359_0171. eprint: https://www.worldscientific.com/doi/pdf/10.1142/9789814324359_0171. URL: https://www.worldscientific.com/doi/abs/10.1142/9789814324359_0171.

- [78] Y. Notay and A. Napov. “A massively parallel solver for discrete Poisson-like problems”. In: *Journal of Computational Physics* 281 (2015), pp. 237–250. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2014.10.043. URL: <https://www.sciencedirect.com/science/article/pii/S0021999114007256>.
- [79] H. Zolfaghari and D. Obrist. “A high-throughput hybrid task and data parallel Poisson solver for large-scale simulations of incompressible turbulent flows on distributed GPUs”. In: *Journal of Computational Physics* 437 (2021), p. 110329. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2021.110329. URL: <https://www.sciencedirect.com/science/article/pii/S0021999121002242>.
- [80] R. W. Hockney. “Characterizing Computers and Optimizing the FACR(I) Poisson-Solver on Parallel Uni-computers”. In: *IEEE Transactions on Computers* C-32.10 (1983), pp. 933–941. DOI: 10.1109/TC.1983.1676137.
- [81] H. S. Stone. “Parallel Tridiagonal Equation Solvers”. In: *ACM Transactions on Mathematical Software* 1.4 (Dec. 1975), pp. 289–307. ISSN: 0098-3500. DOI: 10.1145/355656.355657. URL: <https://doi.org/10.1145/355656.355657>.
- [82] T. Rossi and J. Toivanen. “A Parallel Fast Direct Solver for Block Tridiagonal Systems with Separable Matrices of Arbitrary Dimension”. In: *SIAM Journal on Scientific Computing* 20.5 (1999), pp. 1778–1793. DOI: 10.1137/S1064827597317016. eprint: <https://doi.org/10.1137/S1064827597317016>. URL: <https://doi.org/10.1137/S1064827597317016>.
- [83] Y. Zhang, J. Cohen, and J. D. Owens. “Fast tridiagonal solvers on the GPU”. In: *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPOPP ’10. Bangalore, India: Association for Computing Machinery, 2010, pp. 127–136. ISBN: 9781605588773. DOI: 10.1145/1693453.1693472. URL: <https://doi.org/10.1145/1693453.1693472>.
- [84] P. Valero-Lara, A. Pinelli, and M. Prieto-Matias. “Fast finite difference Poisson solvers on heterogeneous architectures”. In: *Computer Physics Communications* 185.4 (2014), pp. 1265–1272. ISSN: 0010-4655. DOI: 10.1016/j.cpc.2013.12.026. URL: <https://www.sciencedirect.com/science/article/pii/S0010465513004384>.
- [85] T. F. Chan and D. C. Resasco. “A Domain-Decomposed Fast Poisson Solver on a Rectangle”. In: *SIAM Journal on Scientific and Statistical Computing* 8.1 (1987), s14–s26. DOI: 10.1137/0908010. eprint: <https://doi.org/10.1137/0908010>. URL: <https://doi.org/10.1137/0908010>.
- [86] M. V. Pham, F. Plourde, and S. D. Kim. “Strip decomposition parallelization of fast direct Poisson solver on a 3D Cartesian staggered grid”. In: *International Journal of Computer Science and Engineering* 1.1 (2007), pp. 31–40.
- [87] E. P. van der Poel et al. “A pencil distributed finite difference code for strongly turbulent wall-bounded flows”. In: *Computers & Fluids* 116 (2015), pp. 10–16. ISSN: 0045-7930. DOI: 10.1016/j.compfluid.2015.04.007. URL: <https://www.sciencedirect.com/science/article/pii/S004579301501164>.
- [88] M. S. Dodd and A. Ferrante. “A fast pressure-correction method for incompressible two-fluid flows”. In: *Journal of Computational Physics* 273 (2014), pp. 416–434. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2014.05.024. URL: <https://www.sciencedirect.com/science/article/pii/S0021999114003702>.
- [89] R. D. Sanhueza, J. Peeters, and P. Costa. *A pencil-distributed finite-difference solver for extreme-scale calculations of turbulent wall flows at high Reynolds number*. 2025. arXiv: 2502.06296 [physics.flu-dyn]. URL: <https://arxiv.org/abs/2502.06296>.
- [90] V. Fuka. “PoisFFT – A free parallel fast Poisson solver”. In: *Applied Mathematics and Computation* 267 (2015). The Fourth European Seminar on Computing (ESCO 2014), pp. 356–364. ISSN: 0096-3003. DOI: 10.1016/j.amc.2015.03.011. URL: <https://www.sciencedirect.com/science/article/pii/S0096300315003057>.
- [91] J. J. Hasbestan, C.-N. Xiao, and I. Senocak. “PittPack: An open-source Poisson’s equation solver for extreme-scale computing with accelerators”. In: *Computer Physics Communications* 254 (2020), p. 107272. ISSN: 0010-4655. DOI: 10.1016/j.cpc.2020.107272. URL: <https://www.sciencedirect.com/science/article/pii/S0010465520300898>.

- [92] R. D. Budiardja and C. Y. Cardall. “Parallel FFT-based Poisson solver for isolated three-dimensional systems”. In: *Computer Physics Communications* 182.10 (2011), pp. 2265–2275. ISSN: 0010-4655. DOI: 10.1016/j.cpc.2011.05.014. URL: <https://www.sciencedirect.com/science/article/pii/S0010465511001809>.
- [93] P. N. Swarztrauber and R. A. Sweet. “Vector and parallel methods for the direct solution of Poisson’s equation”. In: *Journal of Computational and Applied Mathematics* 27.1 (1989). Special Issue on Parallel Algorithms for Numerical Linear Algebra, pp. 241–263. ISSN: 0377-0427. DOI: 10.1016/0377-0427(89)90369-5. URL: <https://www.sciencedirect.com/science/article/pii/0377042789903695>.
- [94] N. Brown. *A comparison of techniques for solving the Poisson equation in CFD*. 2020. arXiv: 2010.14132 [cs.DC]. URL: <https://arxiv.org/abs/2010.14132>.
- [95] G. Alfonsi et al. “GPGPU implementation of mixed spectral-finite difference computational code for the numerical integration of the three-dimensional time-dependent incompressible Navier–Stokes equations”. In: *Computers & Fluids* 102 (2014), pp. 237–249. ISSN: 0045-7930. DOI: 10.1016/j.compfluid.2014.07.005. URL: <https://www.sciencedirect.com/science/article/pii/S0045793014002849>.
- [96] M. A. U. Zapata and F. J. Hernández-López. “A GPU parallel finite volume method for a 3D Poisson equation on arbitrary geometries”. In: *International Journal of Combinatorial Optimization Problems and Informatics* 9.1 (2018), p. 3.
- [97] P. Costa et al. “GPU acceleration of CaNS for massively-parallel direct numerical simulations of canonical fluid flows”. In: *Computers & Mathematics with Applications* 81 (2021). Development and Application of Open-source Software for Problems with Numerical PDEs, pp. 502–511. ISSN: 0898-1221. DOI: 10.1016/j.camwa.2020.01.002. URL: <https://www.sciencedirect.com/science/article/pii/S0898122120300092>.
- [98] G. Lupo, P. Wellens, and P. Costa. *CaNS-Fizzy: A GPU-accelerated finite difference solver for turbulent two-phase flows*. 2025. arXiv: 2502.04189 [physics.flu-dyn]. URL: <https://arxiv.org/abs/2502.04189>.
- [99] M. Kreer. “Analytic birth—death processes: A Hilbert-space approach”. In: *Stochastic Processes and their Applications* 49.1 (1994), pp. 65–74. ISSN: 0304-4149. DOI: 10.1016/0304-4149(94)90112-0. URL: <https://www.sciencedirect.com/science/article/pii/0304414994901120>.
- [100] S. Liu et al. “Professor Heinz Neudecker and matrix differential calculus”. In: *Statistical Papers* 65.4 (June 2024), pp. 2605–2639. ISSN: 1613-9798. DOI: 10.1007/s00362-023-01499-w. URL: <https://doi.org/10.1007/s00362-023-01499-w>.
- [101] U. Schumann and R. A. Sweet. “Fast Fourier transforms for direct solution of Poisson’s equation with staggered boundary conditions”. In: *Journal of Computational Physics* 75.1 (1988), pp. 123–137. ISSN: 0021-9991. DOI: 10.1016/0021-9991(88)90102-7. URL: <https://www.sciencedirect.com/science/article/pii/0021999188901027>.
- [102] G. Ofenbeck et al. “Applying the roofline model”. In: *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2014, pp. 76–85. DOI: 10.1109/ISPASS.2014.6844463.
- [103] R. Farber. *Parallel programming with OpenACC*. Newnes, 2016.
- [104] J. W. Demmel et al. “Performance and Accuracy of LAPACK’s Symmetric Tridiagonal Eigensolvers”. In: *SIAM Journal on Scientific Computing* 30.3 (2008), pp. 1508–1526. DOI: 10.1137/070688778. eprint: <https://doi.org/10.1137/070688778>. URL: <https://doi.org/10.1137/070688778>.
- [105] J. J. Cuppen. “A divide and conquer method for the symmetric tridiagonal eigenproblem”. In: *Numerische Mathematik* 36.2 (1980), pp. 177–195.
- [106] M. Frigo and S. G. Johnson. “The design and implementation of FFTW3”. In: *Proceedings of the IEEE* 93.2 (2005), pp. 216–231.
- [107] E. Anderson et al. “LAPACK: A portable linear algebra library for high-performance computers”. In: *Proceedings of the 1990 ACM/IEEE conference on Supercomputing*. IEEE Computer Society Press. 1990, pp. 2–11.

- [108] J. Choi et al. “ScaLAPACK: a scalable linear algebra library for distributed memory concurrent computers”. In: *The Fourth Symposium on the Frontiers of Massively Parallel Computation*. Los Alamitos, CA, USA: IEEE Computer Society, Oct. 1992, pp. 120, 121, 122, 123, 124, 125, 126, 127. DOI: 10.1109/FMPC.1992.234898. URL: <https://doi.ieeeecomputersociety.org/10.1109/FMPC.1992.234898>.
- [109] NVIDIA. *cuBLAS*. NVIDIA Developer. URL: <https://developer.nvidia.com/cublas>.
- [110] T. Faingnaert, T. Besard, and B. De Sutter. “Flexible Performant GEMM Kernels on GPUs”. In: *IEEE Transactions on Parallel and Distributed Systems* 33.9 (2022), pp. 2230–2248. DOI: 10.1109/TPDS.2021.3136457.
- [111] H. C. Ku, R. S. Hirsh, and T. D. Taylor. “A pseudospectral method for solution of the three-dimensional incompressible Navier-Stokes equations”. In: *Tenth International Conference on Numerical Methods in Fluid Dynamics*. Ed. by F. G. Zhuang and Y. L. Zhu. Berlin, Heidelberg: Springer Berlin Heidelberg, 1986, pp. 391–397. ISBN: 978-3-540-47233-9.
- [112] P. Orlandi. *Fluid flow phenomena: a numerical toolkit*. Vol. 55. Springer Science & Business Media, 2012.
- [113] G. Hager and G. Wellein. *Introduction to High Performance Computing for Scientists and Engineers*. CRC Press, July 2010. ISBN: 9781439811931. DOI: 10.1201/ebk1439811924. URL: <http://dx.doi.org/10.1201/EBK1439811924>.
- [114] A. Mamirov. *Reducing Fragmentation and Starvation in GPU Clusters through Dynamic Multi-Objective Scheduling*. 2025. arXiv: 2512.10980 [cs.DC]. URL: <https://arxiv.org/abs/2512.10980>.
- [115] J. L. Gustafson. “Reevaluating Amdahl’s law”. In: *Communications of the ACM* 31.5 (May 1988), pp. 532–533. ISSN: 0001-0782. DOI: 10.1145/42411.42415. URL: <https://doi.org/10.1145/42411.42415>.
- [116] S. Pirozzoli and P. Orlandi. “Natural grid stretching for DNS of wall-bounded flows”. In: *Journal of Computational Physics* 439 (2021), p. 110408.
- [117] M. Uhlmann. “An immersed boundary method with direct forcing for the simulation of particulate flows”. In: *Journal of Computational Physics* 209.2 (2005), pp. 448–476. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2005.03.017. URL: <https://www.sciencedirect.com/science/article/pii/S0021999105001385>.
- [118] W.-P. Breugem. “A second-order accurate immersed boundary method for fully resolved simulations of particle-laden flows”. In: *Journal of Computational Physics* 231.13 (2012), pp. 4469–4498. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2012.02.026. URL: <https://www.sciencedirect.com/science/article/pii/S0021999112001374>.
- [119] J. Sebastian, N. Sivadasan, and R. Banerjee. “GPU accelerated three dimensional unstructured geometric multigrid solver”. In: *2014 International Conference on High Performance Computing & Simulation (HPCS)*. 2014, pp. 9–16. DOI: 10.1109/HPCSim.2014.6903663.
- [120] H. Zhang et al. “Direct numerical simulation of a fully developed turbulent square duct flow up to $Re_\tau=1200$ ”. In: *International Journal of Heat and Fluid Flow* 54 (2015), pp. 258–267. ISSN: 0142-727X. DOI: 10.1016/j.ijheatfluidflow.2015.06.003. URL: <https://www.sciencedirect.com/science/article/pii/S0142727X15000685>.
- [121] D. Modesti and S. Pirozzoli. “Direct numerical simulation of forced thermal convection in square ducts”. In: *12th International Symposium on Turbulence and Shear Flow Phenomena, TSFP 2022*. 2022.
- [122] R. A. Van De Geijn and J. Watts. “SUMMA: scalable universal matrix multiplication algorithm”. In: *Concurrency: Practice and Experience* 9.4 (1997), pp. 255–274. DOI: 10.1002/(SICI)1096-9128(199704)9:4<255::AID-CPE250>3.0.CO;2-2. eprint: <https://onlinelibrary.wiley.com/doi/pdf/10.1002/%28SICI%291096-9128%28199704%299%3A4%3C255%3A%3AAID-CPE250%3E3.0.CO%3B2-2>. URL: <https://onlinelibrary.wiley.com/doi/abs/10.1002/%28SICI%291096-9128%28199704%299%3A4%3C255%3A%3AAID-CPE250%3E3.0.CO%3B2-2>.
- [123] S. Kotturshettar. “Turbulent channel flow with stable stratification beyond the Oberbeck-Boussinesq assumptions: a DNS study”. MA thesis. Technical University of Delft, 2023. URL: <https://resolver.tudelft.nl/uuid:5cfe9305-304f-4b9e-8bf0-2ba7cad59e86>.

- [124] SURF. *Snellius hardware*. Snellius hardware - SURF User Knowledge Base. URL: <https://servicedesk.surf.nl/wiki/spaces/WIKI/pages/30660208/Snellius+hardware>.
- [125] P. Orlandi. *Fluid Flow Phenomena*. Springer Netherlands, 2000. ISBN: 9789401142816. DOI: 10.1007/978-94-011-4281-6. URL: <http://dx.doi.org/10.1007/978-94-011-4281-6>.
- [126] AMD. *Server Processor Specifications*. AMD Server Processor Specifications. URL: <https://www.amd.com/en/products/specifications/server-processor.html>.
- [127] Crucial. *How much power does memory use?* Crucial - How much power does memory use? URL: <https://www.crucial.com/support/articles-faq-memory/how-much-power-does-memory-use>.
- [128] Intel. *Intel® Xeon® W-2223 processor (8.25m cache, 3.60 GHz) - product specifications*. Xeon® W-2223 processor (8.25m cache, 3.60 GHz) - product specifications. URL: <https://www.intel.com/content/www/us/en/products/sku/198018/intel-xeon-w2223-processor-8-25m-cache-3-60-ghz/specifications.html>.
- [129] NVIDIA. *Nvidia RTX A4000 graphics card*. NVIDIA RTX A4000 graphics card. URL: <https://www.nvidia.com/en-us/products/workstations/rtx-a4000/>.
- [130] T. Gerhardsen. *Electricity spot prices in Netherlands today, hour by hour*. Energy prices.eu. URL: <https://www.energyprices.eu/electricity/netherlands>.
- [131] ENTSO-E. *Current emissions in Netherlands*. Nowtricity. URL: <https://www.nowtricity.com/country/netherlands/>.

A

Similarity transformation of $\mathbf{T}_{x_j}$

The present appendix demonstrates how the similarity transformation of $\mathbf{T}_{x_j}$ is obtained. Kreer first described this procedure in [99]. The goal of the similarity transformation of $\mathbf{T}_{x_j}$, where $j = \{1, 2, \dots, n\}$ is to make a symmetric matrix from it:

$$\mathbf{T}_{x_j} = \begin{bmatrix} b_1 & c_1 & 0 & 0 & \cdots & 0 \\ a_2 & b_2 & c_2 & 0 & \cdots & 0 \\ 0 & a_3 & \ddots & \ddots & \cdots & 0 \\ 0 & 0 & \ddots & \ddots & c_{N_{x_j}-2} & 0 \\ \vdots & \vdots & \vdots & a_{N_{x_j}-1} & b_{N_{x_j}-1} & c_{N_{x_j}-1} \\ 0 & 0 & 0 & 0 & a_{N_{x_j}} & b_{N_{x_j}} \end{bmatrix}. \quad (\text{A.1})$$

The similarity transformation is as follows:

$$\mathbf{T}_{x_j}^* = \mathbf{D}_{x_j}^{-1} \mathbf{T}_{x_j} \mathbf{D}_{x_j}, \quad (\text{A.2})$$

where

$$\mathbf{D}_{x_j} = \text{diag}(\delta_1, \delta_2, \dots, \delta_{N_{x_j}}). \quad (\text{A.3})$$

Calculating the product of matrices of eq. A.2 gives then:

$$\mathbf{T}_{x_j}^* = \begin{bmatrix} b_1 & c_1 \frac{\delta_2}{\delta_1} & 0 & 0 & \cdots & 0 \\ a_2 \frac{\delta_1}{\delta_2} & b_2 & c_2 \frac{\delta_3}{\delta_2} & 0 & \cdots & 0 \\ 0 & a_3 \frac{\delta_2}{\delta_3} & \ddots & \ddots & \cdots & 0 \\ 0 & 0 & \ddots & \ddots & c_{N_{x_j}-2} \frac{\delta_{N_{x_j}-1}}{\delta_{N_{x_j}-2}} & 0 \\ \vdots & \vdots & \vdots & a_{N_{x_j}-1} \frac{\delta_{N_{x_j}-2}}{\delta_{N_{x_j}-1}} & b_{N_{x_j}-1} & c_{N_{x_j}-1} \frac{\delta_{N_{x_j}}}{\delta_{N_{x_j}-1}} \\ 0 & 0 & 0 & 0 & a_{N_{x_j}} \frac{\delta_{N_{x_j}-1}}{\delta_{N_{x_j}}} & b_{N_{x_j}} \end{bmatrix}. \quad (\text{A.4})$$

For $\mathbf{T}_{x_j}^*$ to be symmetric:

$$\left\{ \begin{array}{l} a_2 \frac{\delta_1}{\delta_2} = c_1 \frac{\delta_2}{\delta_1} \\ a_3 \frac{\delta_2}{\delta_3} = c_2 \frac{\delta_3}{\delta_2} \\ \vdots \\ a_{N_{x_j}-1} \frac{\delta_{N_{x_j}-2}}{\delta_{N_{x_j}-1}} = c_{N_{x_j}-2} \frac{\delta_{N_{x_j}-1}}{\delta_{N_{x_j}-2}} \\ a_{N_{x_j}} \frac{\delta_{N_{x_j}-1}}{\delta_{N_{x_j}}} = c_{N_{x_j}-1} \frac{\delta_{N_{x_j}}}{\delta_{N_{x_j}-1}} \end{array} \right. \quad (\text{A.5})$$

This produces $N_{x_j} - 1$ equations and N_{x_j} unknowns. For simplicity, $\delta_1 = 1$. Recursively:

$$\frac{\delta_{i+1}}{\delta_i} = \sqrt{\frac{a_{i+1}}{c_i}} \quad \text{where } i \in [1, N_{x_j}] \wedge i \in \mathbb{N}. \quad (\text{A.6})$$

This could be rewritten as:

$$\delta_i = \sqrt{\prod_{k=1}^{i-1} \frac{a_{k+1}}{c_k}}, \quad (\text{A.7})$$

since $\delta_1 = 1$.

Obtaining the eigendecomposition of $\mathbf{T}_{x_j}$ is a matter of solving for this matrix in equation A.2, thus resulting in $\mathbf{T}_{x_j} = \mathbf{D}_{x_j} \mathbf{Q}_{x_j}^* \mathbf{\Lambda}_{x_j}^* \mathbf{Q}_{x_j}^{*\top} \mathbf{D}_{x_j}^{-1}$.

B

Eigendecomposition proofs

B.1. Eigendecomposition of a sum of tensor products

The relationship between the eigendecomposition of the matrices which constitute the tensor product and the eigendecomposition of the tensor product itself is known. If two square matrices $\mathbf{B}$ and $\mathbf{C}$ of dimensions $b \times b$ and $c \times c$, with eigenvalues λ and μ and eigenvectors $\mathbf{v}_i$ and $\mathbf{w}_j$, respectively, then the eigendecomposition of their eigendecomposition is as follows:

$$\begin{cases} \sigma(\mathbf{B} \otimes \mathbf{C}) = \lambda \otimes \mu \\ \text{eigenvectors}(\mathbf{B} \otimes \mathbf{C}) = \mathbf{v}_i \otimes \mathbf{w}_j \quad \text{where } i = \{1, 2, \dots, b\} \text{ and } j = \{1, 2, \dots, c\} \end{cases}$$

Recall that the discretization matrix of the Poisson equation can be written as:

$$\mathbf{A} = \sum_{i=1}^n \left(\bigotimes_{k=1}^n \mathbf{M}_{x_k} \right)_i \quad \text{where} \quad \mathbf{M}_{x_k} = \begin{cases} \mathbf{T}_{x_k} & \text{if } i = k \\ \mathbf{I}_{x_k} & \text{if } i \neq k \end{cases}$$

The spectral theorem [51] for a real, symmetric matrix states that an orthonormal eigendecomposition of $\mathbf{T}_{x_j}^*$ exists, where $\mathbf{T}_{x_j}^*$ is the matrix resulting from the similarity transformation of $\mathbf{T}_{x_j}$. Also noteworthy, albeit rather trivial, is that:

$$\mathbf{I}_{x_i} = \mathbf{Q}_{x_i} \mathbf{I}_{x_i} \mathbf{Q}_{x_i}^T. \quad (\text{B.1})$$

Noting:

$$\mathbf{A} = \sum_{k=1}^n \mathbf{A}_k, \quad (\text{B.2})$$

Each term $\mathbf{A}_k$ is written explicitly as:

$$\mathbf{A}_k = \mathbf{I}_{x_1} \otimes \dots \otimes \mathbf{T}_{x_k} \otimes \dots \otimes \mathbf{I}_{x_n}. \quad (\text{B.3})$$

Substituting the respective eigendecompositions:

$$\mathbf{A}_k = \mathbf{Q}_{x_1} \mathbf{I}_{x_1} \mathbf{Q}_{x_1}^T \otimes \dots \otimes \mathbf{Q}_{x_k} \mathbf{A}_{x_k} \mathbf{Q}_{x_k}^T \otimes \dots \otimes \mathbf{Q}_{x_n} \mathbf{I}_{x_n} \mathbf{Q}_{x_n}^T, \quad (\text{B.4})$$

With the mixed-product property of the tensor product [100], one is able to write:

$$\mathbf{A}_k = \left(\bigotimes_{i=1}^n \mathbf{Q}_{x_i} \right) \left(\bigotimes_{i=1}^n \mathbf{M}_{x_i} \right) \left(\bigotimes_{i=1}^n \mathbf{Q}_{x_i} \right)^T \quad \text{where} \quad \mathbf{M}_{x_i} = \begin{cases} \mathbf{A}_{x_i} & \text{if } i = k \\ \mathbf{I}_{x_i} & \text{if } i \neq k \end{cases} \quad (\text{B.5})$$

This means that every partial sum of the full matrix $\mathbf{A}$ is simultaneously diagonalisable, an important result when defining the sum of spectrums of matrices. This also means that the eigendecomposition of $\mathbf{A}_k$ is written as:

$$\begin{cases} \sigma(\mathbf{A}_k) = \bigotimes_{i=1}^n \sigma(\mathbf{M}_{x_i}) & \text{where } \mathbf{M}_{x_i} = \begin{cases} \mathbf{A}_{x_i} & \text{if } i = k \\ \mathbf{I}_{x_i} & \text{if } i \neq k \end{cases} \\ \text{eigenvectors}(\mathbf{A}_k) = \bigotimes_{i=1}^n \mathbf{Q}_{x_i} \end{cases} \quad (\text{B.6})$$

Since every term of the sum $\mathbf{A}_k$ is simultaneously diagonalisable, one may write directly:

$$\mathbf{A} = \sum_{x_i} \left[\left(\bigotimes_{i=1}^n \mathbf{Q}_{x_i} \right) \left(\bigotimes_{i=1}^n \mathbf{M}_{x_i} \right) \left(\bigotimes_{i=1}^n \mathbf{Q}_{x_i} \right)^T \right] = \left(\bigotimes_{i=1}^n \mathbf{Q}_{x_i} \right) \left[\sum_{k=1}^n \left(\bigotimes_{i=1}^n \mathbf{M}_{x_i} \right) \right] \left(\bigotimes_{i=1}^n \mathbf{Q}_{x_i} \right)^T. \quad (\text{B.7})$$

This proves the eigendecomposition of $\mathbf{A}$:

$$\begin{cases} \mathbf{A} = \sigma(\mathbf{A}) = \sum_{k=1}^n \bigotimes_{i=1}^n \sigma(\mathbf{M}_{x_i}) & \text{where } \mathbf{M}_{x_i} = \begin{cases} \mathbf{A}_{x_i} & \text{if } i = k \\ \mathbf{I}_{x_i} & \text{if } i \neq k \end{cases} \\ \mathbf{Q} = \text{eigenvectors}(\mathbf{A}) = \bigotimes_{i=1}^n \mathbf{Q}_{x_i} \end{cases},$$

where

$$\mathbf{A} = \mathbf{Q} \mathbf{\Lambda} \mathbf{Q}^{-1}. \quad (\text{B.8})$$

B.2. Linearity of eigendecomposition

Suppose the $m \times m$ matrix $\mathbf{M}$ has eigenvectors $\mathbf{v}_i$ with corresponding eigenvalues λ_i where $i = \{1, 2, \dots, m\}$. For any constants $\alpha, \beta \in \mathbb{R}$, the matrix $\alpha\mathbf{M} + \beta\mathbf{I}$ has eigenvalues $\alpha\lambda_i + \beta$ with corresponding eigenvectors $\mathbf{v}_i$.

The proof is simple:

$$\begin{aligned} (\alpha\mathbf{M} + \beta\mathbf{I})\mathbf{v}_i &= \alpha\mathbf{M}\mathbf{v}_i + \beta\mathbf{I}\mathbf{v}_i, \\ &= \alpha(\lambda_i\mathbf{v}_i) + \beta\mathbf{v}_i. \end{aligned}$$

Therefore:

$$(\alpha\mathbf{M} + \beta\mathbf{I})\mathbf{v}_i = (\alpha\lambda_i + \beta)\mathbf{v}_i. \quad (\text{B.9})$$

For the case of calculating the eigendecomposition of $(-\frac{1}{\Delta t}\mathbf{I} + \mathbf{A}_{x_i})$, $\alpha = 1$ and $\beta = -\frac{1}{\Delta t}$. Therefore:

$$\begin{cases} \sigma(-\frac{1}{\Delta t}\mathbf{I} + \mathbf{A}_{x_i}) = -\frac{1}{\Delta t} + \sigma(\mathbf{A}_{x_i}) \\ \text{eigenvectors}(-\frac{1}{\Delta t}\mathbf{I} + \mathbf{A}_{x_i}) = \text{eigenvectors}(\mathbf{A}_{x_i}) \end{cases}$$

C

Describing non-uniform grids

For the present work, non-uniform grids were always prescribed using a mapping function that transforms a uniform grid into a grid which is clustered on both ends of the direction being applied to, originally from [125].

The mapping function used is as follows [112]:

$$z = \frac{1}{2} \cdot \frac{1 + \tanh\left(z_0 - \frac{1}{2}\right) \text{gr}}{\tanh\left(\frac{\text{gr}}{2}\right)},$$

where gr is the grid stretching parameter ($\text{gr}=0$ yields a uniform grid). It provides a mapping from the uniform computational domain $z_0 \in [0, 1]$ to the non-uniform computational domain $z \in [0, 1]$.

The number of grid cells of said direction is given by ng , meaning that $z_0 = \left\{\frac{1}{\text{ng}}, \frac{2}{\text{ng}}, \dots, \frac{\text{ng}-1}{\text{ng}}, 1\right\}$. Since this mesh clusters both ends of the direction, the maximum cell spacing will be found at the mid point of the direction and the minimum cell spacing will be found near the wall (at the start or the end of the direction).

Thus:

$$\frac{\Delta z_M}{\Delta z_m} = \frac{z(\frac{1}{2}) - z(\frac{1}{2} + \frac{1}{\text{ng}})}{z(\frac{1}{\text{ng}}) - z(0)} = \frac{\tanh\left(\frac{\text{gr}}{\text{ng}}\right)}{\tanh\left[\left(\frac{1}{\text{ng}} - \frac{1}{2}\right) \text{gr}\right] + \tanh\left(\frac{\text{gr}}{2}\right)}. \quad (\text{C.1})$$

Throughout this text, this ratio is also given as R , meaning that this value of $\frac{\Delta z_M}{\Delta z_m}$ is applied to the three directions of the flow. When this is not the case, the direction in which the ratio is being applied is given as an index to the ratio, such as R_{xz} signifies the ratio that is being applied to both the x and z directions.

Below are some example ratios for differing values of ng and gr :

Table C.1: Example ratios of maximum to minimum cell spacing for differing values of ng and gr . Left column represents varying ng and top row represents varying gr . Note the dependency of the ratio with gr compared to the influence of ng .

ng	0	1	2	3	4
64	1	1.26	2.32	5.3	13.3
128	1	1.27	2.35	5.42	13.73
1024	1	1.27	2.38	5.52	14.1

For the weak scaling test of *SNaC*, the ratio of maximum to minimum grid cell spacing in the y direction R_y depends on the number of processes np of the test. Table C.2 summarizes the small but relevant variations that this ratio has throughout the data points of the weak scaling test of fig. 6.8.

Table C.2: Ratio of maximum to minimum grid cell spacing in the y direction or R_y for the weak scaling test of *SNaC* with respect to the ratio of maximum to minimum grid cell spacing of directions x and z or R_{xz} and the number of processes np allocated in that test.

gr	R_{xz}	Number of Processes np				
		128	256	512	1024	2048
1	1.27	1.27	1.27	1.27	1.27	1.27
2	2.38	2.37	2.37	2.38	2.38	2.38
3	5.52	5.48	5.5	5.52	5.53	5.53

D

Calculating environmental and financial costs

Three different systems were used in the present work. Large scale scaling tests were performed in the Dutch national supercomputer *Snellius* in their *rome* partition. The validation of the algorithm was performed on a workstation with a Intel Xeon W-2223 CPU, a NVIDIA RTX A4000 GPU and 16 GB of memory.

The same formula can be used to compute the power consumption for both systems:

$$P_{avg} = N (P_{CPU}\%CPU + P_{MEM}\%MEM + P_{GPU}\%GPU + P_{REST}), \quad (D.1)$$

where P_{avg} is the average power consumption in the simulation, P_{CPU} , P_{GPU} and P_{MEM} are the power consumptions of the CPU, GPU and memory respectively; P_{REST} is the power consumption of the rest of the components at idle; $\%CPU$, $\%GPU$ and $\%MEM$ are the utilization of the CPU, GPU and memory respectively; and N is the number of *nodes* being used. For the workstation, $N = 1$.

Table D.1 summarizes the power consumption of components of each system, along with the utilization of the components:

Table D.1: Parameters used for calculating the average power consumption of the *Snellius* supercomputer and the workstation. Input means that these values are given as single *inputs* from each scaling test performed. References for *Snellius rome*: P_{CPU} : See [124] and [126]; P_{MEM} : Estimate based on [124] and [127]; P_{REST} : Empirical estimate. References for workstation: P_{CPU} : See [128]; $\%CPU$: If the run is CPU-based, $\%CPU = 100$. Else, $\%CPU = 50$; P_{GPU} : See [129]; P_{MEM} : Estimate based on [127]; $\%MEM$: Based on total occupancy of the domain in the available memory; P_{REST} : Empirical estimate.

Parameter	<i>Snellius</i> <i>rome</i>	Workstation
P_{CPU}	2*280	120
$\%CPU$	100	100 or 50
P_{GPU}	0	140
$\%GPU$	0	100
P_{MEM}	5*16	5
$\%MEM$	Input	50
P_{REST}	150	100
N	Input	1

From each simulation, one extracts the wall-clock time of the entire program (including output routines), the number of cores/nodes used and percentage of memory used. With this, it is possible to assess for each simulation the average power consumption P_{avg} .

Calculating the energy spent is a matter of multiplying the average power consumed by the wall-clock time of the simulation $E(J) = P\Delta t_w$. With this, calculating the energy in kWh comes from $E(kWh) = E(J) \cdot 3.6 \times 10^{-6}$.

Then, the cost of the electricity is obtained by $\text{Cost}(\text{€}) = E(\text{kWh}) \cdot \text{Cost}(\text{€/kWh})$. $\text{Cost}(\text{€/kWh})$ is obtained from [130] as the monthly average value from November 2025. Determining the cost in SBU (Standard Billing Units) is a matter of multiplying the number of allocated CPU cores (in the case of the `rome` partition) by the elapsed time of each simulation in hours. Finally, the CO_2 emitted is calculated by $\text{CO}_2(\text{kg}) = E(\text{kWh}) \cdot \text{CO}_2(\text{g/kWh})/10^3$. $\text{CO}_2(\text{g/kWh})$ is obtained from [131] as the monthly average value of January 2026.

D.1. Estimating the cost of a mock simulation

For defining the grid spacing along the cross-section of the duct, the major challenge comes from providing enough resolution near the walls of the duct while keeping the core region of as low density in the grid as possible to still capture the Kolmogorov scale.

To satisfy the necessary resolution near the wall, the cross-sectional grid can be set by defining the first grid cell spacing of at a minimum of $y_m^+ < 1$, where y^+ represents the dimensionless wall unit. Thus:

$$y^+ = \frac{yu_\tau}{\nu} \Rightarrow \Delta y_m = \frac{y_m^+ \nu}{u_\tau}, \quad (\text{D.2})$$

where u_τ is the friction velocity, Δy_m the grid spacing of the cell near the wall of the duct and ν the kinematic viscosity. For the current setup, the characteristic length is taken as $h = \frac{1}{2}$. Since $Re_\tau = \frac{hu_\tau}{\nu} = 2000$, it may be written:

$$u_\tau = 2Re_\tau \nu \Rightarrow \Delta y_m = \frac{1}{2 \cdot 2000} = 0.00025. \quad (\text{D.3})$$

Assuming the mapping function used in Appendix C, the maximum cell spacing and minimum cell spacing are respectively given by:

$$\Delta z_M = \frac{1}{2} \left(\frac{\tanh\left(\frac{\text{gr}}{\text{ng}}\right)}{\tanh\left(\frac{\text{gr}}{2}\right)} \right), \quad (\text{D.4})$$

$$\Delta z_m = \frac{1}{2} \left(\frac{\tanh\left[\left(\frac{1}{\text{ng}} - \frac{1}{2}\right)\text{gr}\right] + \tanh\left(\frac{\text{gr}}{2}\right)}{\tanh\left(\frac{\text{gr}}{2}\right)} \right). \quad (\text{D.5})$$

Modesti and Pirozzoli in [121] keep the ratio R of the maximum to minimum grid cell spacing to ≈ 9 , thus a maximum value of $R < 15$ was deemed empirically appropriate for the purpose of this thought experiment. With the aforementioned grid spacings and this maximum ratio, by trial and error the values of $\text{ng}(2,3)=1024$, 1024 and $R = 14.1$ (which corresponds to a grid stretching ratio of $\text{gr}=4$) were chosen.¹

For the streamwise direction, the recommendation of Modesti and Pirozzoli in [121] was followed in regards to the number of grid cells, resulting thus in a chosen $\text{ng}(1)=4096$. Gavrilakis in [2] had a ratio of $\frac{\text{ng}(1)}{\text{ng}(2)} = \frac{1000}{127}$ which might suggest this value might not be enough for the high friction Reynolds number here chosen. Nonetheless, the computational time requested should ensure sufficient leverage to the researcher to perform small scale experiments, such as estimating the necessary streamwise resolution.

¹Note how no multigrid method from the *SNaC* implementation was capable of simulating such skewed grids.