POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Topological optimisation for power electronics for Advanced Air Mobility

Course:
High Performance Scientific Computing in Aerospace

Group ID: 1

Group Members:
Ettore Cirillo
Mattia Gotti
Giulio Martella
Michele Milani
Stefano Pedretti
Daniele Piano
Federico Pinto

Advisors:
Prof. Franco Auteri

Academic year:
2025-2026

Contents

Introduction

The design of lightweight and efficient cooling solutions is a key challenge in power-electronics systems for **Advanced Air Mobility**. Although heat exchangers represent a well-covered topic in the literature, results may lack generality and often rely on empiricism or strong turbulence modeling assumptions. In this context, **topology optimization** based on a high-fidelity solver provides a systematic way to identify optimal material distributions.

The present design task focuses on the development of a high-fidelity Full Order Model (FOM), while the definition of an optimization strategy is only briefly discussed. The latter may rely on different approaches, including gradient-based topology optimization and data-driven approaches based on neural networks. Significant computational savings can be achieved by integrating Model Order Reduction techniques within shape optimization frameworks; however, such approaches are currently well established mainly for low–Reynolds-number flows [1], while their extension to high–Reynolds-number turbulent regimes remains an open research challenge despite recent advances in reduced-order modeling for turbulent Navier–Stokes equations [2].

While several commercial codes address Conjugate Heat Transfer (CHT), most rely on the Finite Volume Method (FVM), complicating fluid–solid interface handling. In addition, turbulence modeling is commonly employed, although it is known to be unreliable in flows characterized by separation and recirculation, which directly influence the objective function.

To overcome the computational burden of body-fitted meshing and the inaccuracies of Reynolds-Averaged Navier-Stokes (RANS) modeling, our FOM accounts for topology changes by adopting a fixed-grid strategy inspired by the Immersed Boundary Method (IBM; see [3] for an overview and [4] for a more recent implementation). The Navier-Stokes-Brinkman framework is considered, where internal boundary conditions are imposed via a Darcy term penalty as in [5]. In this setup, a porosity field is defined over the entire domain: its scalar value is set to near-zero in solid regions and to a very high value in fluid zones.

Aim

The primary objective of this work is to design and implement a core **Stokes-Brinkman** solver to serve as the computational kernel for a topology optimization framework. While the convective term is neglected in this implementation for simplicity, its future inclusion is straightforward and limited to the formulation of the differential operator.

The success of this implementation is quantified by three specific performance metrics. First, **accuracy** is validated by verifying **second-order convergence** rates for both velocity and pressure fields. Second, **computational efficiency** is measured against a strict performance target, aiming for an execution time of less than 10^{-6} **seconds per time step per cell**. Finally, **parallel performance** is evaluated by demonstrating **near-linear scalability** as the number of processing units increases.

Overview

To meet the stringent requirements of accuracy, efficiency and scalability outlined above, the development focuses on a highly optimized C++ **finite-difference** solver architecture. The core algorithm exploits the **direction splitting** technique developed by Guermond and Minev [6, 7]. This strategy serves a dual purpose: it effectively decouples the pressure and viscous steps through a second-order **Fractional Step Method** (FSM), and it dramatically reduces computational complexity by splitting 3D operators into a sequence of three successive 1D problems.

From a numerical perspective, spatial discretization is performed using a second-order asymmetric **Marker-And-Cell** (MAC) stencil, while temporal integration relies on the unconditionally stable Crank-Nicolson scheme; further details on discretization accuracy are provided in [8].

To ensure scalability across HPC environments, the solver is parallelized using **MPI**. This implementation utilizes a domain decomposition approach based on the **Schur complement method** to efficiently manage linear systems across multiple processes, adopting the block decomposition techniques defined in [9, 10].

This report is organized as follows; in the first part, we describe the governing equations and the direction splitting algorithm, with details on spatial discretization, boundary condition enforcement, and temporal integration. The second part discusses the code implementation and data structures. The third part presents results on convergence, stability, and parallel consistency and scalability.

1. Governing equations and numerical methods

The purpose of this section is to formulate the numerical problem setup and describe the implemented algorithm.

1.1. The continuous problem

We consider the time-dependent dimensional **incompressible Navier-Stokes-Brinkman** equations in the box $\Omega = [0, L_x] \times [0, L_y] \times [0, L_z] \subset \mathbb{R}^3$:

$$\begin{cases} \partial_t \mathbf{u} + (\mathbf{u} \cdot \nabla)\mathbf{u} + \nabla p - \nu \Delta \mathbf{u} + \dfrac{\nu}{\kappa}\mathbf{u} = \boldsymbol{f} & \text{in } \Omega \times (0, T], \\ \nabla \cdot \mathbf{u} = 0 & \text{in } \Omega \times (0, T], \\ \mathbf{u}\big|_{\partial\Omega} = \mathbf{a} & \text{on } \partial\Omega \times (0, T], \\ \mathbf{u}\big|_{t=0} = \mathbf{u}_0 & \text{in } \Omega \times \{0\}. \end{cases} \tag{1}$$

Here, $\boldsymbol{f}$ is the source term, $\kappa = \kappa(\mathbf{x})$ is the porosity field, $\mathbf{a}$ is the boundary data, and $\mathbf{u}_0$ is the initial condition.

We employ an incremental pressure-correction scheme. Starting from a linearized version of Eq. 1, we consider a singular perturbation of the Poisson equation for the pressure:

$$\begin{cases} \partial_t \mathbf{u}_\epsilon + (\mathbf{u}_\epsilon \cdot \nabla)\mathbf{u}_\epsilon + \nabla p_\epsilon - \nu \Delta \mathbf{u}_\epsilon + \dfrac{\nu}{\kappa}\mathbf{u}_\epsilon = \boldsymbol{f} & \text{in } \Omega \times (0, T], \quad \mathbf{u}_\epsilon|_{\partial\Omega} = \mathbf{a}, \quad \mathbf{u}_\epsilon|_{t=0} = \mathbf{u}_0 \\ -\Delta t \Delta \phi_\epsilon + \nabla \cdot \mathbf{u}_\epsilon = 0 & \text{in } \Omega \times (0, T], \quad \partial_n \phi_\epsilon|_{\partial\Omega} = 0 \\ \Delta t \partial_t p_\epsilon = \phi_\epsilon - \chi \nu \nabla \cdot \mathbf{u}_\epsilon & \qquad\qquad\quad p_\epsilon|_{t=0} = p_0 \end{cases} \tag{2}$$

Here, Δt is the perturbation parameter (i.e., $\epsilon := \Delta t$) and $\chi \in [0, 1]$ is an adjustable parameter.

Provided that the limit solution is smooth enough, an alternative singular perturbation is:

$$\begin{cases} \partial_t \mathbf{u}_\epsilon + (\mathbf{u}_\epsilon \cdot \nabla)\mathbf{u}_\epsilon + \nabla p_\epsilon - \nu \Delta \mathbf{u}_\epsilon + \dfrac{\nu}{\kappa}\mathbf{u}_\epsilon = \boldsymbol{f} & \text{in } \Omega \times (0, T], \quad \mathbf{u}_\epsilon|_{\partial\Omega} = \mathbf{a}, \quad \mathbf{u}_\epsilon|_{t=0} = \mathbf{u}_0 \\ -\Delta t A \phi_\epsilon + \nabla \cdot \mathbf{u}_\epsilon = 0 & \text{in } \Omega \times (0, T], \quad \phi_\epsilon \in D(A), \\ \Delta t \partial_t p_\epsilon = \phi_\epsilon - \chi \nu \nabla \cdot \mathbf{u}_\epsilon & \qquad\qquad\quad p_\epsilon|_{t=0} = p_0 \end{cases} \tag{3}$$

Here, the Laplacian has been replaced by a more general operator A. It is only required that the bilinear form $a(p, q) := \int_\Omega qAp \, d\mathbf{x}$ is symmetric and satisfies the coercivity condition:

$$\|q\|^2_{L^2(\Omega)} \leq a(q, q), \qquad \forall q \in D(A). \tag{4}$$

Here A is taken as $A := (1 - \partial_{xx})(1 - \partial_{yy})(1 - \partial_{zz})$.

1.2. Time integration scheme

Let us generalize the momentum equation as the evolution of a generic dynamical system:

$$\frac{\partial \mathbf{u}}{\partial t} = \mathcal{N}(\mathbf{u}, t) \tag{5}$$

where $\mathcal{N}$ is the Navier-Stokes-Brinkman operator. Time discretization is performed using the Crank–Nicolson scheme, which is second-order accurate in time and A-stable. As a consequence, the method ensures the boundedness of numerical perturbations.

$$(\mathbf{u}^{n+1} - \mathbf{u}^n)\frac{2}{\Delta t} = \mathcal{N}^{n+1} + \mathcal{N}^n \tag{6}$$

This leads to:

$$\left(1 + \frac{\Delta t\, \nu}{2\kappa} - \frac{\Delta t\, \nu}{2}\Delta\right)(\mathbf{u}^{n+1} - \mathbf{u}^n) = \Delta t\, \mathbf{g}^{n+\frac{1}{2}} \tag{7}$$

$$\mathbf{g}^{n+\frac{1}{2}} = \boldsymbol{f}^{n+\frac{1}{2}} - \nabla p_*^{n+\frac{1}{2}} - \left(\frac{\nu}{\kappa} - \nu\Delta\right)\mathbf{u}^n - \mathsf{NL}\left(\mathbf{u}^{n+1}, \mathbf{u}^n\right) \tag{8}$$

Here, $\mathsf{NL}(\mathbf{u}^*, \mathbf{u}^n)$ denotes the non-linear convective operator. It is less stiff than the Stokes operator and is therefore often integrated explicitly. From this point forward, it will be neglected, as our focus is exclusively on the Stokes–Brinkman component.
The expression is manipulated up to:

$$(1 - \gamma\Delta)\,(\mathbf{u}^{n+1} - \mathbf{u}^n) = \frac{\Delta t}{\beta}\,\mathbf{g}^{n+\frac{1}{2}}$$

Where we have introduced the coefficients: $\gamma = \frac{\Delta t\,\nu}{2\beta}$, $\beta = 1 + \frac{\Delta t\,\nu}{2\kappa}$. This equation is solved by means of the directional splitting of Douglas [11]:

$$(1 - \gamma\Delta) \approx (1 - \gamma\partial_{xx})(1 - \gamma\partial_{yy})(1 - \gamma\partial_{zz})$$

1.3. The direction-splitting algorithm

We now describe the employed algorithm.

Pressure predictor The pressure predictor, denoted by p_*, is computed from the old pressure and pressure increment. The algorithm is initialized by setting $p^{-\frac{1}{2}} = p_0$ and $\phi^{-\frac{1}{2}} = 0$.

$$p_*^{n+\frac{1}{2}} = p^{n-\frac{1}{2}} + \phi^{n-\frac{1}{2}} \tag{9}$$

Velocity update The velocity update is computed by solving a series of one-dimensional problems. The algorithm is initialized setting $\boldsymbol{\xi}^0 = \boldsymbol{\eta}^0 = \boldsymbol{\zeta}^0 = \mathbf{u}^0$.

$$\mathbf{g}^{n+\frac{1}{2}} = \mathbf{f}^{n+\frac{1}{2}} - \nabla p_\star^{n+\frac{1}{2}} - \frac{\nu}{\kappa}\mathbf{u}^n + \nu(\partial_{xx}\boldsymbol{\eta}^n + \partial_{yy}\boldsymbol{\zeta}^n + \partial_{zz}\mathbf{u}^n). \tag{10}$$

$$\begin{aligned}
&\boldsymbol{\xi}^{n+1} = \mathbf{u}^n + \frac{\Delta t}{\beta}\mathbf{g}^{n+1/2}, && \boldsymbol{\xi}^{n+1}|_{\partial\Omega} = \mathbf{a},\\
&(I - \gamma\partial_{xx})\boldsymbol{\eta}^{n+1} = \boldsymbol{\xi}^{n+1} - \gamma\partial_{xx}\boldsymbol{\eta}^n, && \boldsymbol{\eta}^{n+1}|_{x=0,L_x} = \mathbf{a},\\
&(I - \gamma\partial_{yy})\boldsymbol{\zeta}^{n+1} = \boldsymbol{\eta}^{n+1} - \gamma\partial_{yy}\boldsymbol{\zeta}^n, && \boldsymbol{\zeta}^{n+1}|_{y=0,L_y} = \mathbf{a},\\
&(I - \gamma\partial_{zz})\mathbf{u}^{n+1} = \boldsymbol{\zeta}^{n+1} - \gamma\partial_{zz}\mathbf{u}^n && \boldsymbol{u}^{n+1}|_{z=0,L_z} = \mathbf{a}.
\end{aligned} \tag{11}$$

In Eq. 10, the Laplacian has been replaced by the operator $(\partial_{xx}\boldsymbol{\eta}^n + \partial_{yy}\boldsymbol{\zeta}^n + \partial_{zz}\mathbf{u}^n)$, as suggested by Angot et al. [12]. This is because the Douglas direction splitting guarantees smoothness of $\mathbf{u}$ only along the z-direction; with this approach, each derivative can instead be computed explicitly using the most recent intermediate variable which is guaranteed to be smooth.
The equations for $\mathbf{g}$ and $\boldsymbol{\xi}$ are explicit and they can be computed simultaneously.

Penalty step The pressure increment $\phi^{n+\frac{1}{2}}$ is computed via the following sequence of one-dimensional problems:

$$\begin{aligned}
&\psi - \partial_{xx}\psi = -\frac{1}{\Delta t}\nabla\cdot\boldsymbol{u}^{n+1}, && \partial_x\psi|_{x=0,L_x} = 0;\\
&\varphi - \partial_{yy}\varphi = \psi, && \partial_y\varphi|_{y=0,L_y} = 0.\\
&\phi^{n+\frac{1}{2}} - \partial_{zz}\phi^{n+\frac{1}{2}} = \varphi, && \partial_z\phi^{n+\frac{1}{2}}|_{z=0,L_z} = 0.
\end{aligned} \tag{12}$$

Pressure update We finish the timestep by updating the pressure as follows:

$$p^{n+\frac{1}{2}} = p^{n-\frac{1}{2}} + \phi^{n+\frac{1}{2}} - \chi\nu\nabla\cdot\left(\frac{1}{2}(\boldsymbol{u}^{n+1} + \boldsymbol{u}^n)\right), \tag{13}$$

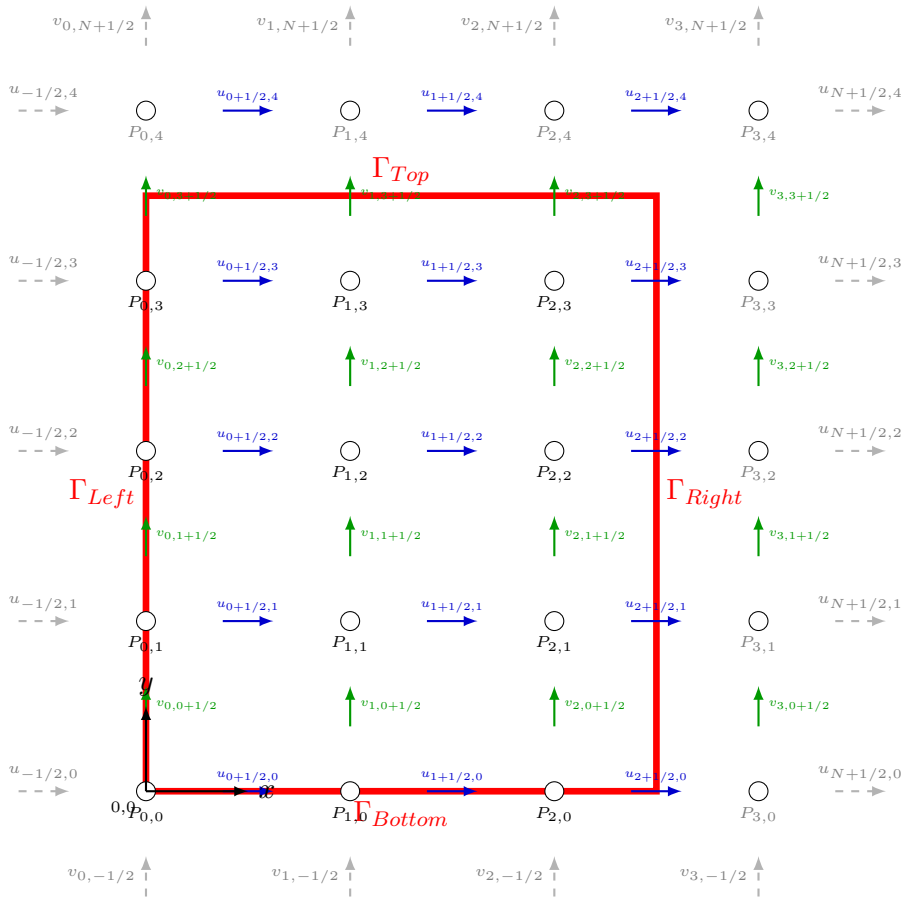
where the parameter χ is set to 0 in all the simulations reported hereafter.

1.4. Spatial discretization

1.4.1 Staggered grid

In order to satisfy the inf-sup condition (LBB condition, see [13–15]), for the Stokes operator, without any particular filtering or stabilization technique (see [16, 17]), we use a **Marker-And-Cell** (MAC) stencil. The implemented asymmetric staggered grid is represented in Figure 1.

Figure 1: Two dimensional asymmetric staggered grid scheme.



1.4.2 Finite differences

To ensure **second-order spatial accuracy**, **central finite differences** are employed for all explicit derivative computations.

The general central difference approximation for a **first derivative** at a generic node i is defined as:

$$\left.\frac{\partial f}{\partial x}\right|_i \approx \frac{f_{i+1} - f_{i-1}}{2\Delta x}. \tag{14}$$

On a staggered grid, this stencil is adapted to exploit the interleaved variable layout.

In the momentum equation, the **pressure gradient** drives the velocity. It is therefore evaluated exactly at the velocity nodes $(i + \frac{1}{2})$, midway between two pressure nodes:

$$\left.\frac{\partial p}{\partial x}\right|_{i+\frac{1}{2}} \approx \frac{p_{i+1} - p_i}{\Delta x}. \tag{15}$$

Conversely, the **velocity divergence** is evaluated at the pressure cell center (i) to compute the net flux for mass conservation, using velocity components on the faces:

$$\left.\frac{\partial u}{\partial x}\right|_i \approx \frac{u_{i+\frac{1}{2}} - u_{i-\frac{1}{2}}}{\Delta x}. \tag{16}$$

It is important to note that despite resembling first-order stencils, the equations 15 and 16 achieve second-order accuracy because they are evaluated at the geometric center of the sampling nodes.

For the **second derivative** terms, standard second-order central difference is used:

$$\left.\frac{\partial^2 f}{\partial x^2}\right|_i \approx \frac{f_{i+1} - 2f_i + f_{i-1}}{\Delta x^2}. \tag{17}$$

In our algorithm, this is applied to the staggered velocity components to compute the **viscous diffusion** contribution:

$$\left.\frac{\partial^2 u}{\partial x^2}\right|_{i+\frac{1}{2}} \approx \frac{u_{i+\frac{3}{2}} - 2u_{i+\frac{1}{2}} + u_{i-\frac{1}{2}}}{\Delta x^2}. \tag{18}$$

1.4.3 Tridiagonal matrices

The spatial discretization of the operator $(I-\gamma\partial_{xx})$ using second-order central differences relies on a **3-point stencil**. This scheme couples every grid node i strictly to its immediate neighbors $i-1$ and $i+1$. Consequently, when applied to a grid line of size N (in whatever direction, at each fractional step), the problem reduces to a sparse, **tridiagonal linear system $\mathbf{Ax}=\mathbf{d}$**. The system can be expressed in scalar form as:

$$a_i x_{i-1} + b_i x_i + c_i x_{i+1} = d_i, \quad \text{for } i = 1, \dots, N \tag{19}$$

where $\mathbf{x}$ is the vector of unknowns, and a_i, b_i and c_i represent the sub-diagonal, main diagonal, and super-diagonal elements respectively.
Due to the physical boundaries of the domain, the off-diagonal terms a_1 (left boundary) and c_N (right boundary) are zero, giving the coefficient matrix $\mathbf{A}$ the following band diagonal form:

$$\mathbf{A} = \begin{bmatrix} b_1 & c_1 & 0 & \cdots & 0 \\ a_2 & b_2 & c_2 & \ddots & \vdots \\ 0 & a_3 & b_3 & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & c_{N-1} \\ 0 & \cdots & 0 & a_N & b_N \end{bmatrix} \tag{20}$$

Direction-splitting mapping This formulation allows for a direct translation of the direction-splitting sweep stencil into matrix coefficients. For instance, the discretized equation for an x-direction sweep of the generic operator $(I-\gamma\partial_{xx})$ becomes:

$$\underbrace{-\frac{\gamma_{i-1}}{\Delta x^2}}_{a_i} X_{i-1} + \underbrace{\left(1+\frac{2\gamma_i}{\Delta x^2}\right)}_{b_i} X_i + \underbrace{-\frac{\gamma_{i+1}}{\Delta x^2}}_{c_i} X_{i+1} = d_i \tag{21}$$

1.5. Boundary Conditions

Pressure Homogeneous Neumann boundary conditions are imposed on the pressure field, namely

$$\left.\frac{\partial p}{\partial n}\right|_{\partial\Omega} = 0. \tag{22}$$

A schematic representation of the corresponding stencil is shown in Figure 3. On the left boundary, the zero normal derivative implies $P_{-1} = P_1$, while on the right boundary it yields $P_{N-1} = P_N$.
As a consequence, when solving a one–dimensional problem of the form

$$(1-\partial_{xx})\psi = rhs, \qquad \partial_x\psi|_{x=0,L_x} = 0, \tag{23}$$

and defining the coefficient $\theta = 1/\Delta x_i^2$, the resulting linear system is characterized by the matrix

$$\mathbf{A} = \begin{bmatrix} 1+2\theta & -2\theta & & & \\ -\theta & 1+2\theta & -\theta & & \\ & \ddots & \ddots & \ddots & \\ & & -\theta & 1+2\theta & -\theta \\ & & & -\theta & 1+\theta \end{bmatrix}. \tag{24}$$

Velocity Dirichlet boundary conditions are imposed on the velocity field. Due to the asymmetry of the grid, two configurations are possible at both the left and right boundaries. The velocity unknown may be located exactly on the boundary line Γ, which trivially yields $u_0 = u_{ex}$ or $u_{N-1} = u_{ex}$. Alternatively, the velocity may be located inside the domain, at an offset $\delta = \Delta x/2$ from the boundary.
A schematic representation of the left boundary configuration is shown in Figure 2; the same arguments apply to the right boundary.
Our objective is to derive an expression for the second derivative u_0''. This quantity cannot be evaluated directly, as it involves the ghost value u_{-1}, which lies outside the physical domain. To overcome this difficulty, a second–order accurate Taylor expansion is employed for the terms appearing in the discrete stencil.

$$u_{-1} = u_{ex} - u'_{ex}\,\delta + \frac{1}{2}u''_{ex}\,\delta^2 + \mathcal{O}(\delta^3), \tag{25}$$

$$u_0 = u_{ex} + u'_{ex}\,\delta + \frac{1}{2}u''_{ex}\,\delta^2 + \mathcal{O}(\delta^3), \tag{26}$$

$$u_1 = u_{ex} + 3u'_{ex}\,\delta + \frac{9}{2}u''_{ex}\,\delta^2 + \mathcal{O}(\delta^3). \tag{27}$$

The relations above lead to a second–order accurate quadratic extrapolation for the ghost value,

$$u_{-1} = \frac{8}{3}u_{ex} - 2u_0 + \frac{1}{3}u_1, \tag{28}$$

which in turn yields the discrete second derivative at the first interior point,

$$u''_0 = \frac{4}{3\Delta x^2}u_1 - \frac{4}{\Delta x^2}u_0 + \frac{8}{3\Delta x^2}u_{ex}. \tag{29}$$

These expressions are employed in the evaluation of the explicit viscous contribution in $\mathbf{g}$, in the computation of the velocity divergence at the boundaries, and in the assembly of the tridiagonal linear systems arising from the momentum equation.
As an example, consider the tridiagonal system

$$(I - \gamma\partial_{xx})\eta^{n+1} = \xi^{n+1} - \gamma\partial_{xx}\eta^n, \tag{30}$$

supplemented with the boundary condition

$$\eta^n_\Gamma = u^n_{ex}. \tag{31}$$

The first row of the resulting linear system reads

$$\left(1 + \frac{4\gamma}{\Delta x^2}\right)\eta^{n+1}_0 - \frac{4\gamma}{3\Delta x^2}\eta^{n+1}_1 = \frac{8\gamma}{3\Delta x^2}\left(u^{n+1}_{ex} - u^n_{ex}\right) + \xi^{n+1}_0 + \frac{4\gamma}{\Delta x^2}\eta^n_0 - \frac{4\gamma}{3\Delta x^2}\eta^n_1. \tag{32}$$

In the case where the velocity is located exactly on the boundary, the same row reduces to

$$\eta^n_0 = u^n_{ex}. \tag{33}$$

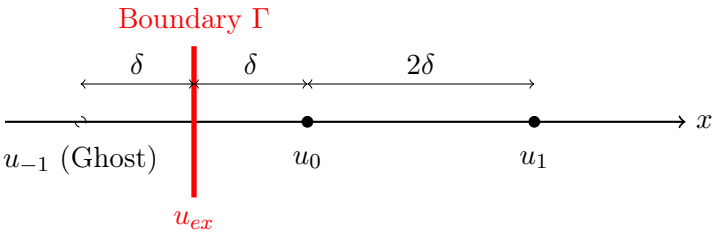


Figure 2: Stencil for left-side normal Dirichlet boundary condition

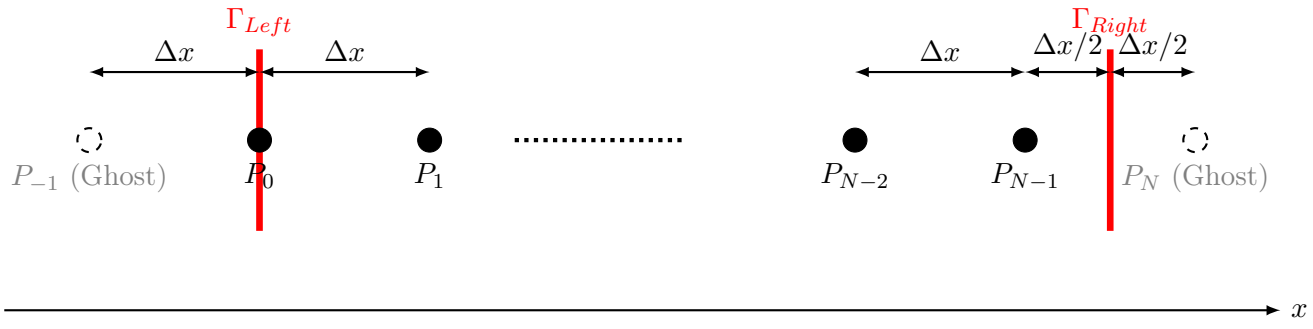


Figure 3: Stencil for pressure Neumann conditions

1.6. Thomas algorithm

The **Thomas algorithm** (TDMA) is a specialized direct method for solving systems of linear algebraic equations where the coefficient matrix is **tridiagonal**. For a system of size N, defined by $\mathbf{Ax} = \mathbf{d}$, the equations take the scalar form:

$$a_i x_{i-1} + b_i x_i + c_i x_{i+1} = d_i, \quad \text{for } i = 1, \ldots, N \tag{34}$$

where x is the unknown vector, and a_i, b_i, c_i represent the sub-diagonal, main diagonal, and super-diagonal coefficients, respectively (with $a_1 = 0$ and $c_N = 0$).

To solve this system efficiently, the algorithm exploits the **matrix sparsity to reduce the computational complexity** from $\mathcal{O}(N^3)$ to $\mathcal{O}(N)$, making it feasible to solve thousands of independent systems per time step - although it requires several divisions, which are generally more expensive than multiplications in terms of floating-point operations. The procedure consists of two strictly sequential phases.

1. Forward Elimination The goal of this phase is to eliminate the lower diagonal term a_i, effectively transforming the matrix into an upper bidiagonal form. This is achieved by computing modified coefficients c'_i and a modified source vector d'_i recursively:

$$c'_i = \begin{cases} \frac{c_1}{b_1} & i = 1 \\ \frac{c_i}{b_i - a_i c'_{i-1}} & i > 1 \end{cases}, \qquad d'_i = \begin{cases} \frac{d_1}{b_1} & i = 1 \\ \frac{d_i - a_i d'_{i-1}}{b_i - a_i c'_{i-1}} & i > 1 \end{cases} \tag{35}$$

The denominator $(b_i - a_i c'_{i-1})$ acts as the pivot. For the algorithm to be numerically stable, the system must generally satisfy the condition of diagonal dominance ($|b_i| \geq |a_i| + |c_i|$).

2. Backward Substitution Once the forward sweep reaches the boundary $i = N$, the system is solved in reverse order to obtain the final solution $\mathbf{x}$:

$$x_N = d'_N, \qquad x_i = d'_i - c'_i x_{i+1} \quad \text{for } i = N-1, \ldots, 1 \tag{36}$$

This formulation provides the exact solution (within machine precision) in a deterministic number of operations, avoiding the convergence issues associated with iterative solvers for 1D problems.

Scalability limits While mathematically optimal, the TDMA is inherently **sequential**. The forward elimination and backward substitution phases are strictly recursive: the calculation at grid point i depends immediately on the result at $i-1$ (or $i+1$). On a single processor, this is not a limitation.

To handle high-resolution spatial grids efficiently, a **domain decomposition** strategy would allow the global physical domain to be sliced into multiple sub-domains - hence distributing the computational load. This would lead coefficients from the original global matrices to be scattered across disjoint processors. However, matrices would still be **globally coupled**, and the TDMA application would still rely on a strictly sequential dependency chain: this effectively **negates** the benefits of **parallelism and** compromises **scalability**.

1.7. Schur complement method

To reconcile the globally coupled system with the distributed storage stemming from domain decomposition, we employ the **Schur complement method**. This technique mathematically reorders the system to decouple *internal* nodes of each sub-domain from *shared-interface* nodes that connect them, enabling high parallel efficiency.

Mathematical formulation Consider the global system distributed across P processors. We classify the unknown vector $\mathbf{x}$ into two distinct sets:

- **internal nodes** $\mathbf{x}_i$, residing entirely within a sub-domain and do not interact directly with neighboring processors;
- $P+1$ **shared interface nodes** $\mathbf{x}_s$, lying at partition edges and coupling the sub-domains.

By permuting the rows and columns of the matrix $\mathbf{A}$ to group these sets together, the linear system takes a block-structured form:

$$\begin{bmatrix} \mathbf{A}_{ii} & \mathbf{A}_{is} \\ \mathbf{A}_{si} & \mathbf{A}_{ss} \end{bmatrix} \begin{bmatrix} \mathbf{x}_i \\ \mathbf{x}_s \end{bmatrix} = \begin{bmatrix} \mathbf{d}_i \\ \mathbf{d}_s \end{bmatrix} \tag{37}$$

Crucially, $\mathbf{A}_{ii}$ is a block-tridiagonal matrix, in which each block corresponds to the internal nodes of a single processor. Since these blocks are mathematically uncoupled, **operations involving $\mathbf{A}_{ii}^{-1}$ can be performed perfectly in parallel**.

The reduced system To solve this system, we first eliminate the internal variables $\mathbf{x}_i$ by expressing them as a function of the interface variables $\mathbf{x}_s$ - using the first row of the block system:

$$\mathbf{x}_i = \mathbf{A}_{ii}^{-1}(\mathbf{d}_i - \mathbf{A}_{is}\mathbf{x}_s) \tag{38}$$

Substituting this expression into the second row yields a **reduced system** that governs only the interface nodes:

$$\underbrace{(\mathbf{A}_{ss} - \mathbf{A}_{si}\mathbf{A}_{ii}^{-1}\mathbf{A}_{is})}_{\mathbf{S}}\mathbf{x}_s = \underbrace{\mathbf{d}_s - \mathbf{A}_{si}\mathbf{A}_{ii}^{-1}\mathbf{d}_i}_{\tilde{\mathbf{d}}_s} \tag{39}$$

where $\mathbf{S}$ is known as the **Schur matrix**. This system allows to solve interface solutions alone, and is significantly smaller than the original (size $P+1$ vs size N), making it fast and less memory-demanding.

Algorithmic phases The solution is obtained through three strictly defined phases:

1. **Local condensation (parallel)** - Each processor effectively performs a partial Gaussian elimination on its local internal nodes ($\mathbf{A}_{ii}^{-1}$). This results in the concurrent calculation of local 2×2 **Schur complement** matrices - which condense local physics into coupling values for left and right boundaries - and **local RHS contributions** to the reduced RHS $\tilde{\mathbf{d}}_s$.
2. **Interface solution (global communication)** - The small reduced system $\mathbf{S}\mathbf{x}_s = \tilde{\mathbf{d}}_s$ is solved. This step requires communication (i.e., gathering Schur complements into a global **Schur matrix**, and gathering local RHS contributions) to determine the correct values at the partition boundaries.
3. **Local expansion (parallel)** - With the interface values $\mathbf{x}_s$ now known, each processor independently performs a backward substitution to resolve its internal nodes $\mathbf{x}_i$.

2. Code implementation

2.1. Solver overview

The codebase is architected following strict software engineering principles, prioritizing **separation of concerns** and **data encapsulation**.

The system is organized into hierarchical layers, where each layer abstracts some complexity from the layer above. This modular design ensures that the high-level physics orchestration is decoupled from low-level memory management, parallel cooperation and coordination, and linear algebra routines.

- `core` **module** - This module establishes the foundational infrastructure for parallel execution and spatial discretization, alongside robust data containers and helpers to manage physical quantities across the distributed domain.
- `numerics` **module** - The numerical engine encapsulates the core mathematical machinery required by the solver, including finite difference operators, custom linear algebra structures, and tridiagonal system solvers.
- `simulation` **module** - This module segregates the physical data into a centralized structure, and implements the driver and the fractional steps required for a Navier-Stokes-Brinkman simulation.
- `io` **module** - The module handles the interaction between the solver and the external environment, managing configuration loading, simulation feedback (logging), and result visualization.

2.2. Domain decomposition: topology and grid management

This module establishes the foundational infrastructure for parallel execution and spatial discretization. Crucially, in alignment with the aforementioned architectural principles, the following abstractions allow the high-level numerical solver to remain agnostic to the underlying parallelization strategy, effectively decoupling the mathematical formulation from the complexities of the distributed memory environment.

2.2.1 MPI environment wrapper: `core/MpiEnv` class

The `MpiEnv` class serves as a robust **Resource Acquisition Is Initialization** (RAII) wrapper around the MPI library. This design pattern ensures that the MPI environment is safely initialized upon object construction and strictly finalized upon destruction, preventing resource leaks or undefined states.

Beyond basic lifecycle management, the class is responsible for abstracting the hardware topology into a logical structure suitable for the solver.

Cartesian topology The class employs `MPI_Cart_create` to map physical processes onto a logical **3D Cartesian grid**. It automatically determines optimal dimensions if none are provided and handles rank reordering to enhance memory locality by aligning logical neighbors with physical hardware interconnects.

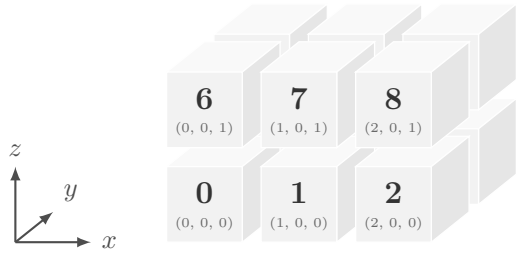


Figure 4: **3D Cartesian topology** for a decomposition of dimensions $N_x = 3, N_y = 2, N_z = 2$.

Directional sub-communicators To facilitate direction splitting algorithms, the class generates specialized **sub-communicators for each coordinate axis** using `MPI_Cart_sub`. This capability allows the solver to execute collective operations - such as parallel tridiagonal solves - along isolated process rows or columns, thereby significantly reducing communication overhead by avoiding global synchronization.

Figure 5: **Directional sub-communicators for the reference rank** $(2, 0, 0)$**.** Left: the X-communicator aggregates processes varying only in x. Center: the Y-communicator aggregates processes varying only in y. Right: the Z-communicator aggregates processes varying only in z.

2.2.2 Parallel staggered grids: `core/Grid` class

The `Grid` class acts as the process-specific immutable, geometrical descriptor of the computational domain.

Geometric abstraction The class encapsulates:

- **global parameters** - such as total physical lengths (L_x, L_y, L_z), global point counts (N_x, N_y, N_z), spatial discretization steps (Δx, Δy, Δz);
- **local parameters** derived from the domain decomposition - such as process-specific point count (N_x^{loc}, N_y^{loc}, N_z^{loc}) and the index offsets with respect to global discretization (i_{start}, j_{start}, k_{start}).

Overlapping domain decomposition Upon initialization with the `MpiEnv`, the grid automatically calculates its local geometry using an **overlapping decomposition** strategy. Rather than partitioning the nodes directly, the logic distributes the global spatial *intervals* among the available processes. This ensures adjacent sub-domains share a common interface boundary: the last grid node of rank P is physically coincident with the first grid node of rank $P + 1$. This overlap is a structural prerequisite for the **Schur complement method**, and is handled transparently alongside the allocation of **halo layers** for stencil operations at the interfaces.

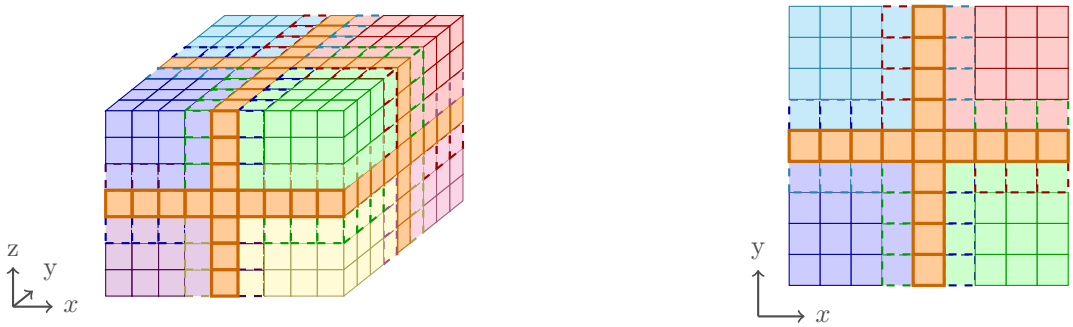


Figure 6: **3D** (left) **and 2D** (right) **overlapping domain decomposition.** Ranks are separated by **shared interfaces** (orange cells). Internal nodes immediately adjacent to an interface (dashed borders matching the neighbor's color) serve as **halo** sources.

Staggered grid support The class supports MAC staggering. Coordinate retrieval methods (i.e., mapping from `ijk`- linear indices to `XYZ`- global coordinates, useful for boundary conditions evaluation) accept a `GridStaggering` enum, allowing precise differentiation between face-centered and cell-centered nodes.

2.3. Distributed data structures

With the topology and grid geometry established, the solver requires robust data containers to manage physical quantities across the distributed domain. This functionality is encapsulated in the `Field` and `VectorField` classes, while inter-process halo synchronization is delegated to the `HaloHandler`.

2.3.1 Scalar and vector field abstractions: `core/Field` and `core/VectorField` classes

The `Field` class serves as the primary container for scalar quantities. It is designed to handle the specific requirements of staggered finite-difference schemes on a distributed memory architecture.

Complementing this, the `VectorField` class encapsulates vector quantities by aggregating distinct `Field` instances for each spatial component. This design effectively implements a **Structure of Arrays** (SoA) memory layout. Compared to the Array of Structures (AoS), this layout significantly improves memory access patterns and enhances cache locality during component-wise operations (e.g., $\partial u/\partial x$), thereby facilitating better compiler auto-vectorization.

Memory layout and indexing To maximize cache locality, the class stores data in a flattened, one-dimensional `std::vector<double>` using a **row-major** indexing scheme. The class provides two distinct access modalities to balance ease of use with computational efficiency.

- **Triple linear indexing** (i, j, k), for readability and rapid prototyping of complex 3D stencils. The mapping to the linear address is handled internally via precomputed strides:

$$\text{idx}(i, j, k) = (k + n_{halo}) \cdot S_z + (j + n_{halo}) \cdot S_y + (i + n_{halo}) \tag{40}$$

  where S_y and S_z are the strides along the Y and Z axes, and n_{halo} represents the ghost layer padding.
- **Raw linear indexing**, for performance-critical loops. This allows optimized stencils to bypass the integer arithmetic of coordinate mapping, facilitating vectorization and reducing overhead during dense iterative operations.

Staggered grid support Reflecting the MAC grid strategy, each `Field` and `VectorField` component stores metadata regarding its physical centering (`GridStaggering`). This allows the class to transparently handle coordinate retrieval, hiding the staggering logic.

Functional initialization To facilitate the setup of **initial conditions** or **time-dependent constant** values, the classes employ a high-level **populate** mechanism. The classes store a user-defined analytical function $f(x, y, z, t)$ (encapsulated as a `std::function`); when the `populate(time)` method is invoked, the solver iterates over the local grid, automatically converting logical indices to physical coordinates based on the field's specific staggering. This design effectively decouples the physical definition of the problem from the underlying discrete memory representation, allowing users to apply complex initial states without manually managing loop bounds or ghost layer offsets.

2.3.2 Halo exchange mechanism: `HaloHandler` class

In a domain decomposition approach, the consistency of the global solution relies on the frequent synchronization of the *halo* cells - the layers of memory padding that replicate data owned by neighboring processes. The `HaloHandler` implements a non-blocking communication strategy.

Buffering and packing strategy To handle strided data belonging to Y and Z faces, the handler employs a naive **packing/unpacking** strategy, where data is explicitly copied from field arrays to contiguous communication buffers and vice versa prior to transmission.
While this strategy significantly simplifies the MPI calls by allowing them to operate on contiguous blocks of `MPI_DOUBLE`, the approach incurs memory and CPU overhead due to the requirements for duplicate buffer storage and the cost of explicit memory copy operations.
A promising avenue for future optimization involves the implementation of **MPI Derived Datatypes**, such as `MPI_Type_vector` or `MPI_Type_create_subarray`. Adopting these would allow the MPI implementation to read strided data directly from `Field` memory in a "zero-copy" manner.

Overlapping synchronization logic Crucially, the `HaloHandler` respects the overlapping domain decomposition strategy. Unlike standard ghost cell exchanges that send boundary nodes, this handler transmits *internal* nodes. Specifically, to fill the left halo $u[-1]$ of a process, the handler fetches $u[N-2]$ from the left neighbor (rather than $u[N-1]$). This ensures that the interface points are treated as internal points by the receiving process, maintaining the strict continuity required for the interface linear system.

2.4. The numerical engine

The numerical engine encapsulates the core mathematical machinery required by the solver, including finite difference operators, custom linear algebra structures, and tridiagonal system solvers. Designed for **high cohesion** and **loose coupling**, this module strictly isolates numerical operations from the solver logic, physics implementation, allowing them to be validated, optimized, or replaced independently of the governing equations.

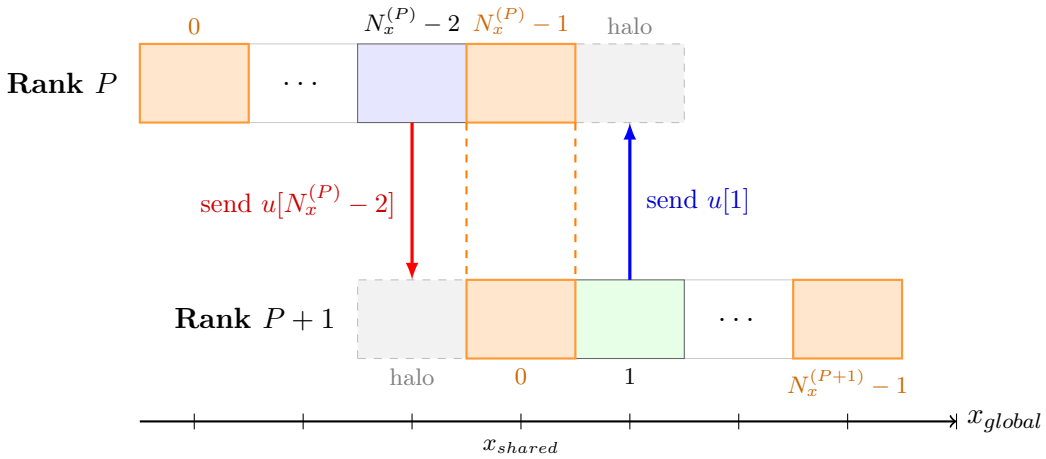


Figure 7: **Halo exchange in an overlapping decomposition setting.** The overlapping interface is defined at node $N_x^{(P)} - 1$ of rank P and node 0 of rank $P+1$. The halo exchange utilizes the internal nodes ($N_x^{(P)} - 2$ and 1) to populate the respective ghost cells.

2.4.1 Finite differences: `numerics/Derivatives`

The `Derivatives` class provides a stateless interface for computing gradients and divergences, designed to operate directly on `Field` and `VectorField` data containers.

Operational strategy A core design decision in this module is the use of **explicit full-field storage**: the methods evaluate the differential operator over the entire domain and populate complete output buffers (e.g., a `VectorField &gradP` for the pressure gradient).

This approach offers distinct trade-offs. On the one hand - as just mentioned - it maximizes **modularity** and **testability**. On the other hand, it imposes a substantial memory footprint and bandwidth cost, as every intermediate derivative requires the **allocation** and **global write-back** of a grid-sized array.

Stencils The module translates the mathematical discretizations defined in Section 1.4.2 into specific algorithmic kernels. While the underlying operations achieve **second-order accuracy** on the staggered grid, the implementation distinguishes them based on their indexing patterns:

- **1st-order Forward Difference** - Implements the stencil $f_{i+1} - f_i$. As established in Eq. 15, this operation is used to compute the **gradient** of pressure-like fields.
- **1st-order Backward Difference** - Implements the stencil $f_i - f_{i-1}$. Corresponding to Eq. 16, this is applied to the **divergence** of the velocity vector field.
- **2nd-order Central Difference** - Implements the standard 3-point stencil $f_{i+1} - 2f_i + f_{i-1}$. This corresponds to Eq. 18 and is used to compute the **diffusive terms**.

Boundary and interface enforcement Beyond internal discretization, the module handles two distinct types of boundaries. **Physical boundaries** are consistently enforced employing **Taylor series expansions** with **ghost nodes** contributions. Conversely, for internal **shared interfaces** arising from domain decomposition, the derivative stencils impose a strict precondition: a **halo exchange** with neighboring sub-domains must be successfully completed prior to execution, ensuring ghost layers contain valid, up-to-date field values.

Naive implementation A baseline implementation prioritizes code **readability** and safety, by employing the high-level `Field::operator()` accessor and three nested loops over (i, j, k) to cover the whole domain. However, this approach incurs overhead from **repeated 3D-to-1D index calculations** and **non-contiguous memory access**, which limits compiler optimization.

Optimized implementation To mitigate performance penalties associated with high-level abstractions, the optimized implementation transitions from object-oriented field access to a low-level, data-oriented design.

- **Pointer arithmetic and striding** - Operates on **raw pointers** using pre-calculated **strides**, effectively bypassing expensive 3D coordinate recalculations.
- **Linearized iteration** - Always reduces nested loops structure (specific to the derivative direction) to strided yet sequential access of contiguous memory (row-major, X-lines). This maximizes **cache locality** and enables **SIMD** auto-vectorization.
- **Loop peeling for boundaries** - Decouples boundary logic from the main loop, keeping the internal domain processing streamlined and **branch-free**.

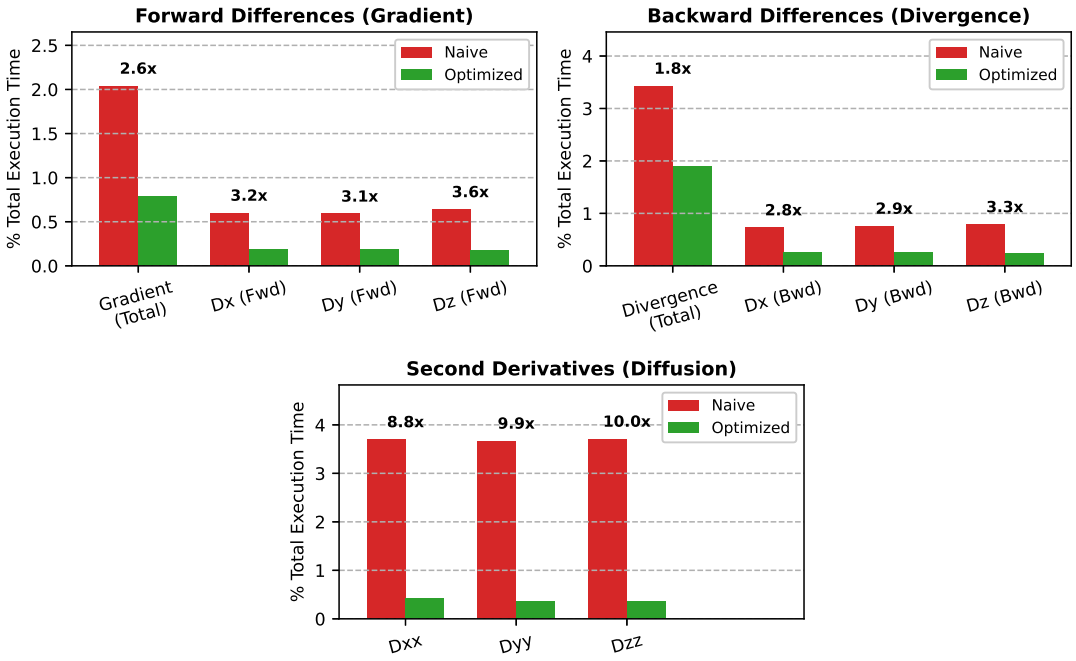


Figure 8: **Performance comparison between the naive and optimized implementations of the finite difference stencils.**

Experimental alternative: on-the-fly computation An alternative strategy to be explored is computing derivatives **point-wise on-the-fly** within the solver kernels (e.g., inside the momentum equation loop). This experiment would test the trade-off between increased arithmetic operations (recalculating derivatives) and reduced memory traffic (avoiding global write-backs) and pressure, which is often beneficial on bandwidth-bound modern architectures - though it may complicate the code structure.

2.4.2 Optimized matrix storage: `numerics/TridiagMat`

The `numerics/TridiagMat` class serves as the fundamental data structure for the solver's implicit numerics, underpinning both the viscous and pressure steps differential equations.

Compressed storage The class **minimizes memory footprint** by avoiding dense matrix storage, instead maintaining three distinct `std::vector<double>` to represent non-zero diagonals (`diag`, `subdiag`, `supdiag`).

This storage pattern reduces memory complexity to $\mathcal{O}(N_d^{(loc)})$, making it very cheap to store matrix coefficients even for high-resolution domains.

Reference-based access The class exposes its internal storage through a strictly typed, reference-based interface. This design pattern enables the physics solvers to **populate matrix coefficients in-place** during the line-sweeping process, effectively eliminating data copying overhead.

2.4.3 Tridiagonal solver: `numerics/ThomasSolver`

The `ThomasSolver` class provides the implementation of the TDMA. With the theoretical derivation established in Section 1.6, the design focuses purely on the architectural optimizations required to efficiently solve thousands of independent linear systems per simulation step - prioritizing **memory reuse** and **instruction-level efficiency**.

Persistent scratch memory The standard TDMA requires a temporary array to store modified coefficients (c') during the forward elimination phase. To avoid the overhead of dynamic memory allocation in every time step (which would occur thousands of times), the class maintains a **persistent scratch buffer** `c_prime_` as a member variable. This buffer is reused across calls and only resized if the domain resolution increases, effectively eliminating heap fragmentation and allocation costs during the time loop.

Low-level optimizations The `solve()` method bypasses high-level container abstractions in favor of raw performance:

- **Pointer aliasing hints** - The kernel operates on **raw pointers** decorated with the `__restrict__` keyword. This informs the compiler that the coefficient arrays do not overlap, enabling aggressive optimization of load/store sequences and eliminating `std::vector` bounds-checking overhead.
- **Division minimization** - Division operations are computationally expensive. The solver minimizes them by computing the **inverse of the pivot** once per row and replacing subsequent divisions with faster multiplication instructions.
- **In-place solution** - The solver writes the **result directly back into the RHS** vector `d`. This reduces memory traffic by eliminating the need for a separate output buffer.

2.4.4 Distributed solver: `numerics/SchurSolver` class

The `SchurSolver` class encapsulates the complexity of the parallel tridiagonal solution, providing a high-level interface `solve()` that hides the underlying domain decomposition and inherently-required communication from the physics modules.

Embarrassingly parallel workload: memory management The recurrent, per-process independent, abstract operation $\mathbf{y} = \mathbf{A}_{ii}^{-1}\hat{\mathbf{f}}$ is never performed by explicit matrix inversion, which would be computationally expensive and destroy sparsity. Instead, it is reduced to solving the equivalent linear system $\mathbf{A}_{ii}\mathbf{y} = \hat{\mathbf{f}}$ using the Thomas algorithm.
To support this efficiently, the class maintains a **persistent solver instance** `thomas_in_` tailored to the inner system size ($N_d^{(loc)} - 2$). Furthermore, to eliminate allocation overhead per time step and across time steps, the class pre-allocates member variables to serve as **persistent scratchpads** for all intermediate data structures:

- `a_in_, b_in_, c_in_`: for the inner matrix coefficients, avoiding repeated extractions from the full matrix;
- `f_in_`: a buffer for an inner RHS vector;
- `y_`: a buffer for the inner "solution" vector.

Phase 1: Local matrix condensation (or *Preprocessing*) A `preprocess()` method constructs the **Schur matrix S**. This phase is computationally intensive but fully parallel. First, each process computes the response of its own internal system to unit perturbations at the boundaries (the vectors E_{left} and E_{right}) by executing `thomas_in_` twice. Then, the 2×2 Schur complement matrices are derived. Finally, local matrices are gathered to the line-master process (rank 0 in the line sub-communicator) using `MPI_Gather` to be assembled into the global block-tridiagonal system S (stored in persistent `Sa_glob_`, `Sb_glob_`, `Sc_glob_` buffers).

This design provides a critical optimization path: by isolating the matrix factorization, the expensive global assembly can be performed just once for problems with stationary matrix coefficients. Conversely, if the physics dictates time-varying or spatially evolving coefficients, `preprocess()` must be re-invoked at every time step and every spatial line to update the global couplings.

Phase 2: Interface solution (global coupling) The `solveInterface()` method handles the global synchronization:

1. **Local RHS condensation** - The method `condenseRHS()` reduces the local force vector $\mathbf{f}$ to two values representing the accumulated flux at the sub-domain boundaries.
2. **Gather** - These condensed values are sent to the line-master process: the communication volume is minimal (2 values per process), ensuring low latency.
3. **Reduced solve** - The line-master solves the global system $\mathbf{S}\mathbf{x}_s = \tilde{\mathbf{f}}_s$ using a dedicated, persisted Thomas solver instance `thomas_s_glob_`. Since the system size is proportional to the processor count ($P + 1$), this step is negligible in cost compared to the recurrent local workload.

4. **Scatter** - The computed boundary values are distributed back to their respective processors.

Phase 3: Local expansion Once the interface values x_0 and x_{N-1} are known for each sub-domain, the `solveInterior()` method computes the final values for the internal nodes. It corrects the local RHS by subtracting the boundary contributions (e.g., $f_{in}[0] \leftarrow f_{in}[0] - a_1 x_0$) and performs one final back-substitution using `thomas_in_`. This step is performed in parallel by all processors.

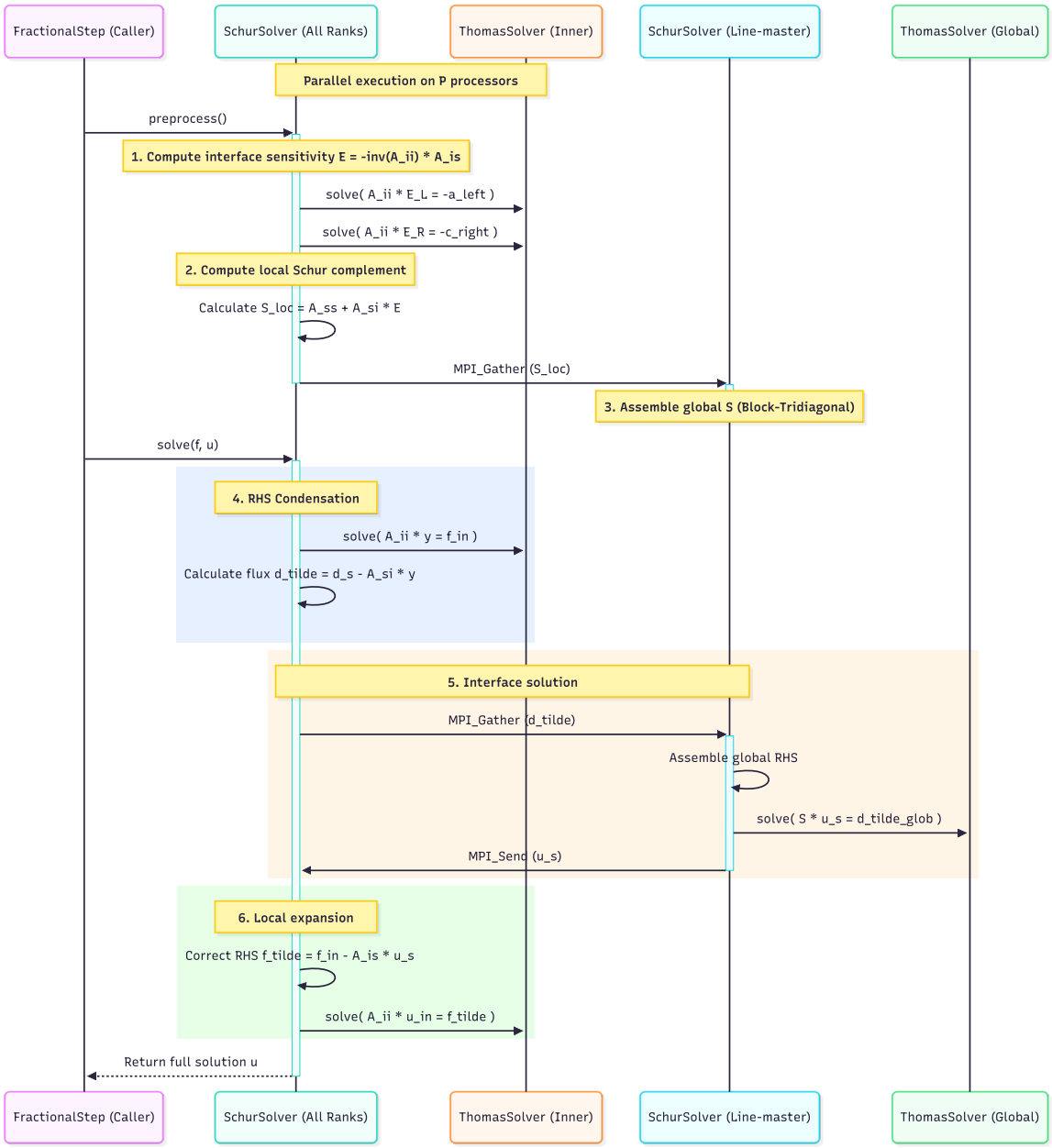


Figure 9: **Sequence diagram of the `SchurSolver` operations.**

2.5. Simulation orchestration

To maintain a modular architecture, the code strictly separates the simulation state from the control logic. This design segregates the transient physical data into a centralized structure, while the `NSBSolver` class acts as the driver responsible for time integration and I/O.

2.5.1 Centralized state: `simulation/SimulationData`

The `SimulationData` structure serves as the central data hub for the entire application, holding all physical fields and parameters required by the fractional steps. By passing this single context object to the physics modules, the code avoids cumbersome argument lists. The structure encapsulates:

- **Grid and Time** - The global mesh topology and the time-stepping state (`currTime`, `dt`, `currStep`).
- **Core fields** - The velocity vector field **u**, the intermediate fractional step fields $\vec{\eta}$ and $\vec{\zeta}$, the pressure scalar field p, and the pressure predictor *predictor*.
- **Boundary conditions** - The functional definitions for the velocity boundary conditions (`bcu, bcv, bcw`), allowing the solver to evaluate Dirichlet constraints at any spatial coordinate and time.
- **Physical properties** - The spatially varying permeability field `inv_k` (representing the porous medium) and the body force field **f**. The porosity is stored as its **inverse** to avoid otherwise frequent, computationally expensive division operations inside the time-stepping loop - allowing the Brinkman penalization term to be evaluated via **efficient multiplication**.

Data model analysis: modularity vs. efficiency trade-off The current design of `SimulationData` prioritizes modularity and code readability over raw hardware efficiency - yet achieving promising performance, as later discussed in Section 3. However, because each `Field` manages its own memory allocation internally, the result is a **sparse allocation** pattern, with simulation **data fragmented across the heap**. This memory layout increases the risk of **Translation Lookaside Buffer** (TLB) **misses**, as accessing correlated variables (e.g., u, v, w during a divergence calculation) requires the CPU to juggle multiple, potentially disjoint memory pages. Furthermore, the **lack of locality** between interacting fields (e.g., `p` and `u`) hinders the efficacy of the hardware prefetcher, which thrives on predictable, linear access patterns within contiguous memory blocks.

Optimized future alternatives To mitigate these bottlenecks, a superior strategy involves allocating a single, contiguous memory block (e.g., a linearized `std::vector<double>`) to hold all simulation fields. By mapping fields to specific offsets within this "super-array", the solver ensures that all data resides in the same virtual memory region, drastically reducing TLB thrashing and allocation overhead. Regarding layout, three distinct strategies offer specific advantages:

- **Structure of Arrays (SoA)** - Storing components contiguously (e.g., all u then all v) - which closely mirrors the current logic - is generally preferred for SIMD vectorization, as it allows loading packed vectors without complex shuffling.
- **Array of Structures (AoS)** - Interleaving data (e.g., u_i, v_i, w_i, p_i per cell) optimizes cache locality for operations that require all variables simultaneously, reducing cache line evictions during multi-variable physics updates.
- **Directional grouping: hybrid layout** - Since the FSM decouples the solving directions, a hybrid approach could group fields by their **sweep dependency**. For instance, storing all X-related fields ($\vec{\eta}, \psi$, etc.) in one contiguous block, followed by Y-related fields ($\vec{\zeta}, \phi$, etc.), etc. This strategy combines the benefits of both worlds: it preserves the **contiguous memory** streams required for vectorization (SoA-like) while ensuring that **only the relevant data** for the current sweep is loaded **into the cache** (AoS-like locality), minimizing cache pollution from variables not currently in use.

2.5.2 Data factory: `simulation/Initializer`

The `Initializer` namespace is responsible for the systematic population of the simulation data. It acts as a factory that reads the configuration parameters and functional initial conditions, and constructs the `SimulationData` instance.

A critical task performed here is the initialization of the permeability field. Since the Brinkman penalization term involves division by permeability k, numerical stability requires operating on the inverse k^{-1}: the initializer computes this field, clamping the values to a maximum penalization threshold (10^{10}) in solid regions where $k \to 0$, thus preventing division-by-zero errors during the solve.

2.5.3 Workflow orchestrator: `simulation/NSBSolver`

The `NSBSolver` class acts as the main orchestrator of the simulation. It owns the `SimulationData`, the physics steps (`ViscousStep`, `PressureStep`), and the I/O handlers. Its lifecycle is controlled by two primary methods.

- **Setup** - The `setup()` method establishes the MPI topology, initializes the simulation data via the `Initializer`, and constructs the physics step objects.
- **Execution** - The `solve()` method executes the main time-stepping loop. As illustrated in Figure 10, each iteration advances the simulation time, updates boundary conditions, and sequentially invokes the fractional steps.

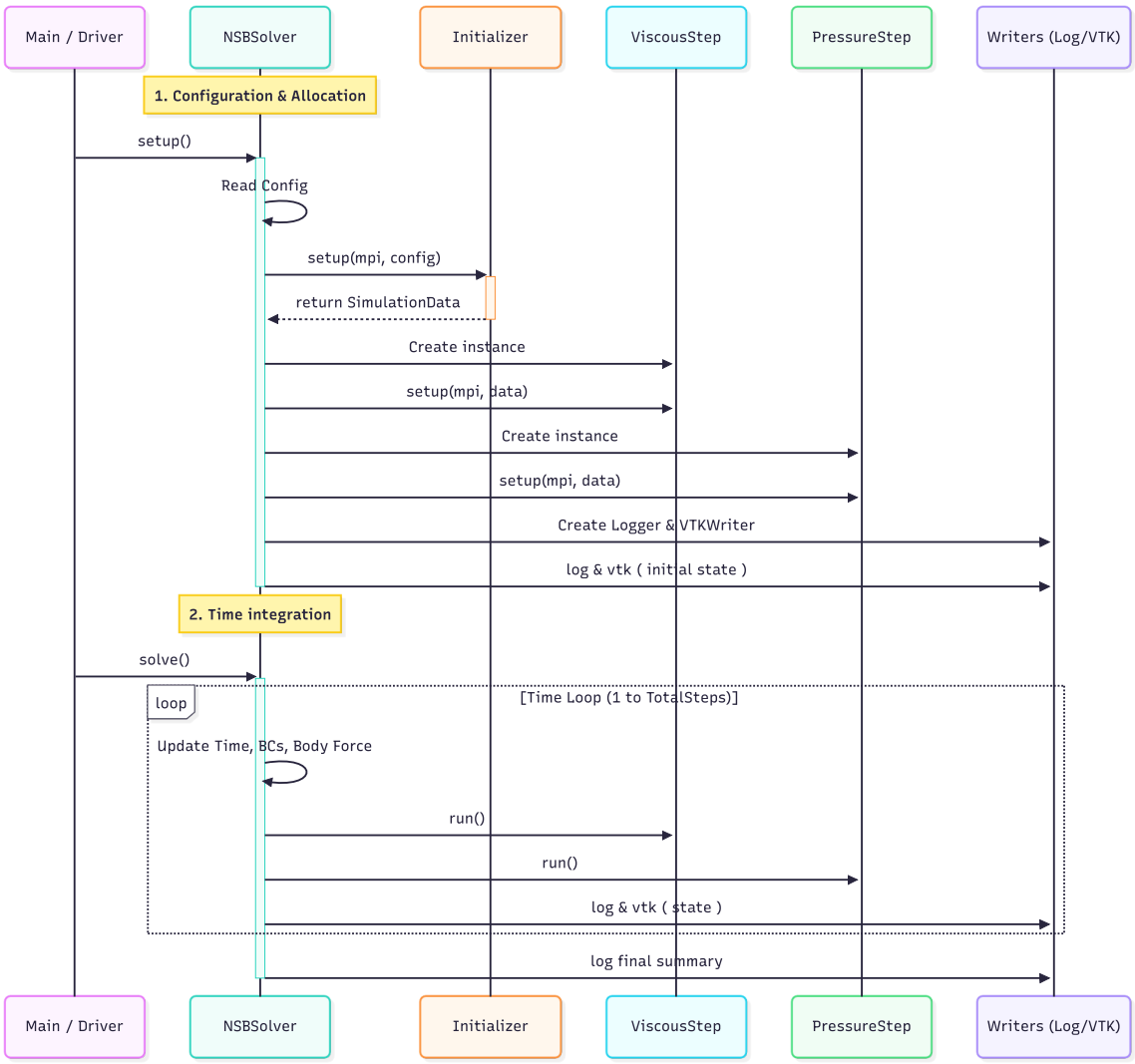


Figure 10: **Control flow of the `NSBSolver` lifecycle.**

From a software engineering perspective, this design provides a high degree of abstraction: it enables the user to employ the solver through a minimal, declarative interface, hiding the underlying complexity of the time loop and memory management.

```
MpiEnv mpi(argc, argv);
NSBSolver problem(configFilename, mpi);
problem.setup();
problem.solve();
```

Listing 1: **High-level usage of the `NSBSolver` API.**

2.5.4 Momentum equation: `simulation/ViscousStep`

The `ViscousStep` class implements the time advancement of the momentum equation, employing an ADI scheme to decouple the three spatial dimensions. The implementation is designed to minimize memory footprint while maximizing parallel consistency.

Memory management and resource reusage To maintain a lean memory profile, the solver differentiates between persistent state and scratchpad memory. While the heavy 3D fields (u, p, η, ζ) are stored in a `SimulationData` instance, the `ViscousStep` class manages its own resources for linear algebra operations.

- **Solvers** - One `SchurSolver` instance per direction (`solver_x`, `solver_y`, `solver_z`) is allocated, to enable reuse of internal buffers without reallocation overhead across time steps.
- **Scratchpads** - Auxiliary matrices and vectors for the linear systems (`matrix`, `rhs`, `unknown_u`, `unknown_v`, `unknown_w`) are pre-allocated during the `setup()` phase to the maximum required dimension.
- **Derivative and explicit terms** - Full 3D fields for derivatives and explicit terms (`xi`, `dxxEta`, `dxxZeta`, `dzzU`, `gradP`) are maintained as member variables.

Explicit predictor ξ The method `computeXi()` constructs the field ξ (as defined in Equation 11) which acts as the source term for the first implicit sub-problem. This phase is optimized for both correctness and memory efficiency.

- **Halo exchange** - Before computing derivatives, the method invokes `HaloHandler::exchange()` on the velocity components and pressure predictor. This synchronization is strictly necessary to compute accurate gradients and second derivatives at the partition boundaries.
- **Inline g computation** - The explicit term g (as defined in Equation 10) is computed inline within the ξ calculation loop. By avoiding the storage of g as a separate 3D field, the solver saves $O(3 \cdot N_x^{(loc)} N_y^{(loc)} N_z^{(loc)})$ memory.
- **Boundary conditions** - Dirichlet boundary conditions are finally enforced to the ξ boundaries too.

Implicit ADI sweeps The implicit solution phase is orchestrated through three distinct directional sweeps, executed sequentially to resolve the coupled velocity components. For each direction (X, Y, Z), the algorithm iterates over the perpendicular plane (e.g., for the X-sweep, looping over all j, k indices) to isolate a single 1D line of cells. Inside this innermost loop, the solver performs a rigorous sequence of operations:

1. **Component-specific, line-specific system assembly** - The `assembleLocalSystem()` method is called for each iterated line, once for each field component.
2. **Schur preprocessing** - Since the matrix coefficients change in space, time and with the boundary type, the `SchurSolver::preprocess()` method must be invoked immediately after each assembly to update the global interface blocks.
3. **Solution** - Solutions for the three components are sequentially computed using the proper solver instance.

Assembly and boundary conditions The method `assembleLocalSystem()` constructs the matrix coefficients and the RHS for a specific line in space and time, enforcing boundary conditions discussed in Section 1.5.

Furthermore, for internal shared interfaces, the method handles the related coefficients in the main diagonal and the RHS by scaling them by 0.5: this ensures "half contributions" that sum to unity when the Schur system is later solved, recovering the correct central stencil.

Assembly analysis A notable performance trade-off in the current `assembleLocalSystem` implementation is the memory access pattern during the Y and Z sweeps.

- **Bottleneck** - The simulation data is stored in a standard row-major, X-fastest, format. While assembling X-lines accesses memory contiguously, assembling Y or Z lines requires accessing data with large strides. This **non-contiguous access pattern** leads to frequent cache misses, as fetching a single double for the line requires loading an entire cache line, most of which is unused for that specific iteration.
- **Optimizations** - A common optimization in HPC to mitigate this is **global transposition**. By reordering the data such that the solve direction is always the fastest-varying index in memory (e.g., transposing $(x, y, z) \to (y, x, z)$ before the Y-sweep), the CPU can utilize vectorization and hardware prefetching more effectively. However, this introduces the overhead of the transposition operation itself, representing a classic compute-vs-memory trade-off.

2.5.5 Decoupled pressure update: `simulation/PressureStep`

The `PressureStep` enforces the divergence-free constraint - again, employing an ADI scheme to decouple the three spatial dimensions required by the Laplacian operator.

Memory management and resource reusage Similar to the viscous step, the pressure solver manages its resources to minimize allocation overhead during the time loop.

- **Solvers** - Three `SchurSolver` instances (`solver_x`, `solver_y`, `solver_z`) are allocated. Unlike the viscous solvers, these are initialized and preprocessed **once** during the step's `setup()` phase, as the pressure **matrix coefficients are constant in time and space**.
- **Scratchpads** - Shared vectors (`rhs`, `solution`) are reserved to the maximum dimension needed and then used for all linear system solves.
- **Intermediate fields** - Full 3D scalar fields are maintained as member variables for intermediate potentials (`psi`, `phi`), the final pressure correction (`pcr`) and the velocity divergence `divU`.

Explicit source The divergence of the predicted velocity field $\mathbf{u}$ is fully computed and scaled by $-1/\Delta t$ to form the RHS of the first sub-problem (X-sweep).

- **Halo exchange** - The `HaloHandler` updates the ghost layers of $\mathbf{u}$ to ensure accurate differentiation at partition boundaries.

Implicit ADI sweeps As in the viscous step, three directional sweeps are executed sequentially - as described in Equation 12.

Assembly and boundary conditions Unlike the viscous step, specifically to the derivative direction, the method constructs a constant matrix that remains valid for the entire simulation.

Neumann boundary conditions and half contributions are enforced respectively at physical boundaries and partition interfaces.

Critical analysis A key performance distinction between the two physics steps lies in the matrix stability:

- **Static matrices** - Since the matrix coefficients depend only on the grid geometry, the Schur complement matrices are factorized and assembled only **once** in `setup()`. Inside the time loop, the solver performs only the relatively cheap forward/backward substitutions. This makes the pressure step significantly faster per iteration than the viscous step, which requires constant re-factorization.
- **Access bottleneck** - Despite the algorithmic efficiency, the Y and Z sweeps suffer from the same **strided memory access** issues identified in the viscous step.

2.6. Input and output management

The I/O module handles the interaction between the solver and the external environment, managing configuration loading, simulation feedback (logging), and result visualization. The design isolates these responsibilities into three distinct classes - `InputReader`, `LogWriter`, and `VTKWriter` - ensuring that the core physics kernels remain agnostic to file formats.

2.6.1 Input configuration and analytical definitions: `io/InputReader`

The solver utilizes a hybrid configuration strategy that combines runtime parameters with compile-time analytical functions.

JSON parameter parsing The `InputReader` class uses the `nlohmann/json` library to parse the `config.json` file. This file defines the numerical environment, including grid resolution, time-stepping parameters and physical constants.

Analytical field definitions Spatially- and time-varying fields (such as boundary conditions and initial states) are defined in `configFunctions.hpp` as C++ functions. These functions are bound to the `SimulationData` functors during initialization, providing a high-performance mechanism to evaluate boundary conditions without the overhead of parsing mathematical expressions at runtime.

2.6.2 Visualization output: `io/VTKWriter`

The `VTKWriter` class manages the export of simulation fields to the legacy VTK format (`STRUCTURED_POINTS`) at a configured `output_frequency`, allowing visualization in tools like *ParaView*.

2.6.3 Logging and performance metrics: `io/LogWriter`

The `LogWriter` provides runtime feedback to the console and log files. To avoid clutter in a parallel environment, output is strictly guarded so that only the **master rank (rank 0)** writes to the stream.

Performance tracking Beyond simple status updates, the logger acts as a profiler, tracking the execution time of each time step and computes a standardized performance metric at the end of the simulation:

$$\text{Metric} = \frac{\text{Total CPU Time}}{\text{Total Steps} \times \text{Total Cells}} \tag{41}$$

This metric enables objective performance comparisons across different mesh sizes and processor counts.

2.7. Full code documentation

To ensure maximum **transparency**, **maintainability**, and **comprehensibility** of the source code, the entire solver structure has been rigorously commented. Subsequently, a formal documentation suite was generated using *Doxygen* (adhering to the standards introduced in the course modules).

Implementation details The documentation, derived directly from the source code annotations, provides a comprehensive and formal overview of the implementation, specifically detailing:

- **architectural hierarchy**: a structural breakdown of class inheritance and relationships;
- **function specifications**: detailed descriptions of usage, parameters, and return values for each method;
- **data abstractions**: in-depth explanations of core data structures and types.

Dependencies and control flow To facilitate the understanding of complex interactions, the documentation includes **call graphs** and **caller graphs** for every function. Furthermore, collaborative classes feature **collaboration diagrams** to illustrate coupling between components (e.g., the `NSBSolver` class diagram). Additionally, the `Files` section maps the project structure, allowing users to inspect the **dependency graph** of any specific file to trace direct and indirect inclusions.

The complete documentation is hosted on GitHub Pages for immediate accessibility and can be reached at the following address:

access the complete Doxygen documentation

Strategic value While extensive, this level of documentation is critical. It serves not merely as a **learning aid**, but as a fundamental tool for **organizing development**, **explaining architectural decisions**, and efficiently **identifying bugs or performance bottlenecks** during the optimization phase.

3. Results

In this section, we assess the numerical and performance validation of the Stokes-Brinkman solver. First, the spatial and temporal convergence of the scheme was assessed through a convergence study using the Method of Manufactured Solutions (MMS), i.e., a solution obtained by introducing a suitable body force in the right-hand side of the equations. Then, we tested our code on two laminar test cases: the channel flow and the pipe flow. Lastly, the results concerning the scalability of the parallel implementation are discussed.

3.1. Convergence test

To verify that the implementation achieves the theoretical second-order accuracy in both space ($\mathcal{O}(h^2)$) and time ($\mathcal{O}(\Delta t^2)$), the algorithm has been tested numerically on a manufactured solution.

3.1.1 Manufactured Solutions

The following analytical velocity and pressure fields were defined:

$$u = \sin(x)\cos(t+y)\sin(z) \tag{42}$$
$$v = \cos(x)\sin(t+y)\sin(z) \tag{43}$$
$$w = 2\cos(x)\cos(t+y)\cos(z) \tag{44}$$

$$p = \frac{3}{Re}\cos(x)\cos(t+y)\cos(z) \tag{45}$$

$$\kappa = 10\,(2+\cos(x)\cos(y)\cos(z)) \tag{46}$$

where the permeability field $\kappa(\boldsymbol{x})$ is assumed to be strictly positive ($\kappa > 0$) throughout the domain to ensure the Darcy term is well-defined. By substituting these fields into the Navier-Stokes-Brinkman equations, the forcing term $\mathbf{f}$ was derived. The forcing term is constructed by summing the individual contributions without grouping coefficients, to maintain clarity of the physical terms:

$$\boldsymbol{f} = \underbrace{\frac{\partial \boldsymbol{u}}{\partial t}}_{\text{Transient}} + \underbrace{3\nu\boldsymbol{u}}_{\text{Viscous}} + \underbrace{\frac{\nu}{\kappa}\boldsymbol{u}}_{\text{Darcy}} + \underbrace{\nabla p}_{\text{Pressure}} \tag{47}$$

This term includes the transient, viscous, Darcy, and pressure gradient contributions, evaluated at time $t^{n+1/2}$ to preserve the temporal accuracy of the Crank-Nicolson scheme.

3.1.2 Convergence Study

The numerical accuracy of the solver is evaluated in the cubic domain $\Omega = [0, 6]^3$ up to a final physical time $T_{end} = 0.5s$, with the Reynolds number fixed at $Re = 1$.
To assess the performance of the algorithm, a coupled space-time refinement study was carried out using eleven uniform grids of increasing resolution: 20^3, 30^3, 40^3, 50^3, 60^3, 70^3, 80^3, 90^3, and 100^3 points. In accordance with the coupled nature of the test, the time step was refined proportionally to the grid size: the coarsest grid ($n_x^0 = 20$) used $\Delta t^0 = 0.01s$, and subsequent time steps were scaled linearly.Results are presented in Tables 1, 2, and Figure 11.
The spatial convergence has been tested with different grid spacing, but keeping fixed $\Delta t = 0.001s$ and $T_{end} = 0.05s$. Results are presented in Tables 3, 4, and Figure 12.
The temporal convergence has been tested with constant grid spacing ($n_x = 100$). The physical time is set to $T_{end} = 1.0s$. Time step scales from $\Delta t = 0.2s$ up to $\Delta t = 0.02s$. Results are presented in Tables 5, 6 and Figure 13.
The results show that both approximations, for velocity and pressure, are second-order accurate. It is worth noting that the temporal convergence results are less robust than the spatial and coupled space–time ones. This behavior is likely not intrinsic to the time integration scheme, but rather a consequence of the relatively coarse spatial resolution adopted in the temporal refinement study.

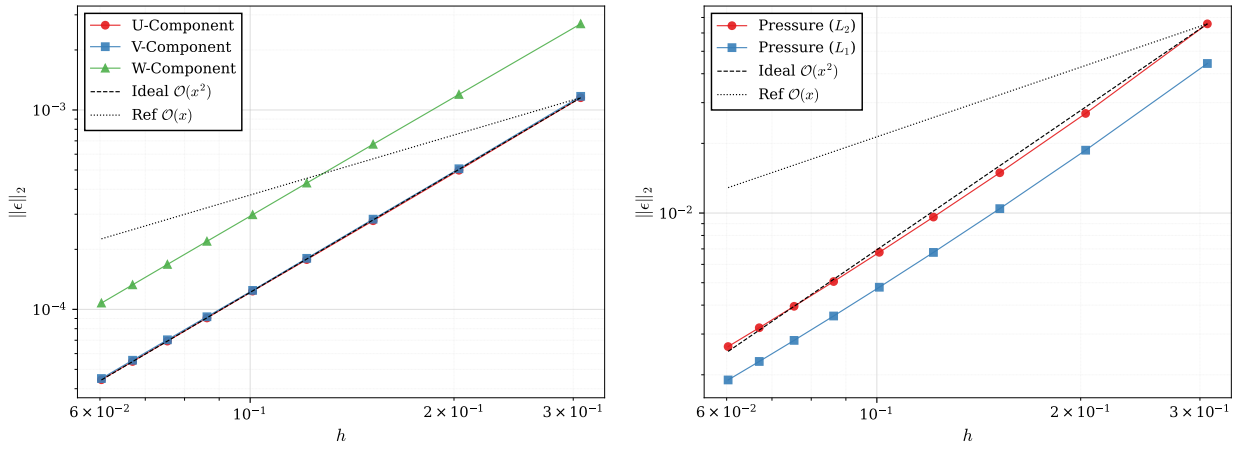


Figure 11: Convergence analysis for the 3D Manufactured Solution. L_2 error norms for velocity components (left) and pressure field (right) as a function of the grid spacing h. $\Delta t \sim \Delta x$

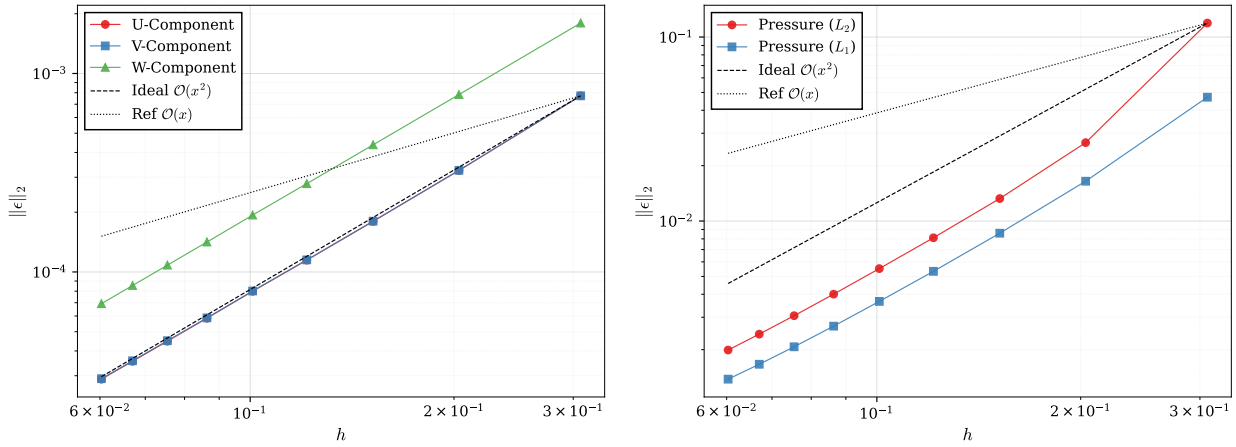


Figure 12: Spatial convergence analysis for the 3D Manufactured Solution. L_2 error norms for velocity components (left) and pressure field (right) as a function of the grid spacing h. Time step is fixed, only Δx varies.

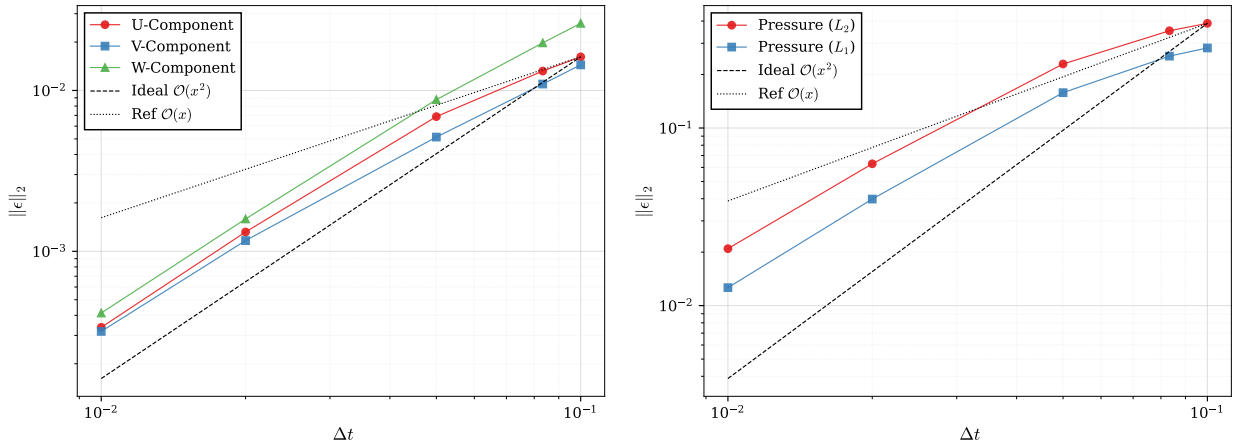


Figure 13: Temporal convergence analysis for the 3D Manufactured Solution. L_2 error norms for velocity components (left) and pressure field (right) as a function of the grid spacing h. Grid spacing is fixed; only Δt varies.

3.2. Comparison with Laminar solutions

Building on the satisfactory convergence results obtained using the manufactured solution, additional tests were carried out. Given the absence of the nonlinear convective term, only a limited set of test cases could be considered. We therefore validated the code against analytical solutions of the Navier–Stokes equations for laminar flows (see Chapter 9 in [18]).
The focus of these tests was on the imposition of a pressure gradient through a prescribed forcing term, the enforcement of no-slip and no-penetration boundary conditions, and the use of a permeability field to accurately represent solid boundaries.

3.2.1 Plane Couette-Poiseuille Flow

The first test case consists of a fluid confined between two infinite parallel plates. We prescribe a time-dependent driving force and boundary conditions to match the analytical solution for a combined Couette-Poiseuille profile. The analytical streamwise velocity $u(y,t)$ is defined as the superposition of two components:

1. **Couette component:** Driven by the motion of the upper wall with velocity U_{max}.
2. **Poiseuille component:** Driven by a prescribed pressure gradient G.

Both components are modulated by a harmonic function $\sin(t)$ to test the solver's transient response:

$$u_{analytical}(y,t) = \sin(t)\left(\frac{yU_{max}}{L_x} - \frac{Gy}{2\nu}(L_x - y)\right) \tag{48}$$

where L_x is the channel height, ν is the kinematic viscosity, and G is the pressure gradient amplitude. In our setup, we set $U_{max} = 0.5$, $G = 10.0$, and $\nu = 1.0$. A scheme is presented in Figure 14.

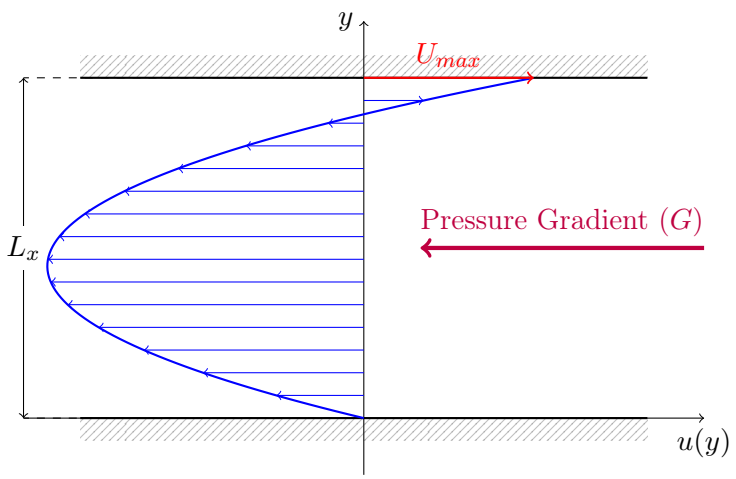


Figure 14: Analytical solution for Plane Couette-Poiseuille Flow. The profile results from the competition between the upper wall velocity U_{max} and a strong, opposing pressure gradient G, causing significant backflow.

Simulation Setup The domain is a channel where the fluid region is characterized by high permeability ($\kappa = 10^{15}$) to recover the standard Stokes equations. The driving force f_x is applied as a source term $G\sin(t)$. No-slip boundary conditions are imposed at the walls by directly setting the analytical velocity at $y = 0$ and $y = L_x$. Final physical time has been set to $T_{end} = 1s$.
In both the streamwise and spanwise directions, the physically consistent choice would be to enforce periodic boundary conditions. However, since periodicity implementation has not been finalized in the present solver, the analytical solution is prescribed at the corresponding lateral boundaries. The implementation of periodic boundary conditions is straightforward and does not affect the scope of the present analysis.

Results As shown in Table 7, the streamwise velocity component exhibits second-order convergence. The analytical values of the other two velocity components are identically zero; however, their L_2 norms fluctuate around 10^{-3}. This effect is expected to vanish once periodic boundary conditions are implemented. The analytical pressure field was not considered, but the observed values indicate that it remains bounded.

3.2.2 Circular Pipe Flow (Hagen-Poiseuille)

The second test case focuses on the flow through a circular pipe, providing a critical assessment of the Brinkman penalization method's ability to enforce complex internal boundaries on a fixed Cartesian mesh.

Geometry and Penalization The computational domain is a cube $\Omega = [0,1]^3$ as shown in Figure 15. A straight cylindrical vessel of radius $R = 0.25$ is centered at $(y_c, z_c) = (0.5, 0.5)$. The geometry is implicitly defined by the permeability field κ, which is set to a high value ($k_{\text{fluid}} = 10^{20}$) inside the cylinder to represent the fluid and to a near-zero value ($k_{\text{solid}} = 10^{-20}$) outside to represent the solid walls. To avoid numerical oscillations at the fluid-solid interface, a sigmoid-based smoothing is applied:

$$I(\mathbf{x}) = \frac{1}{2}\left[1 - \tanh\left(\frac{r - R}{\epsilon}\right)\right] \tag{49}$$

where r is the radial distance from the cylinder axis and $\epsilon_{\text{interface}}$ is the interface width. Adjusting the porosity values and the interface width did not yield any significant results.

Analytical Setup The flow is driven by a time-varying body force $f_x = G\cos(t)$. The analytical solution for the streamwise velocity in a circular pipe is given by:

$$u_{analytical}(r,t) = \cos(t)\frac{G}{4\nu}(R^2 - r^2) \tag{50}$$

where $G = 10$ is the pressure gradient amplitude and $\nu = 1$ is the kinematic viscosity. This profile is used to prescribe the inlet boundary conditions and the initial state of the system.

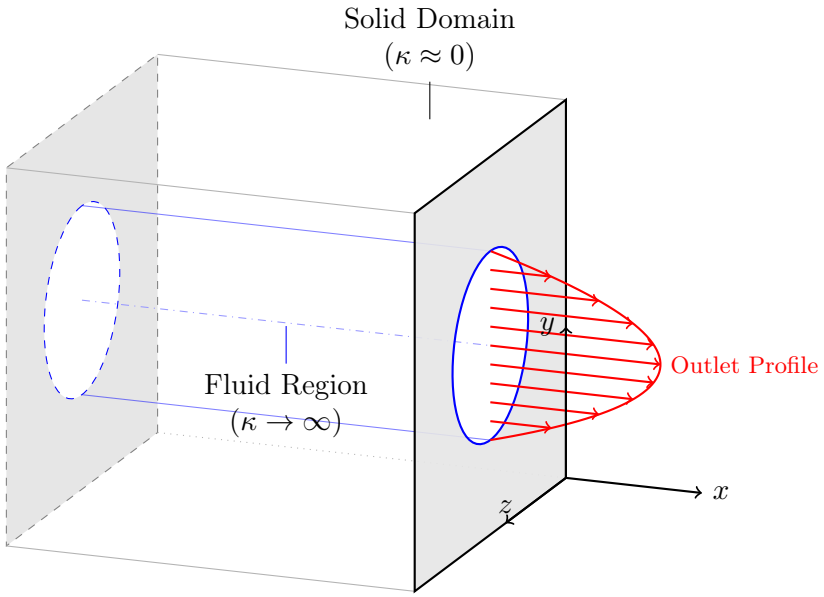


Figure 15: Computational setup for the Hagen-Poiseuille flow. The domain is a cube $[0,1]^3$ where the solid walls are modeled via permeability penalization (gray region). The fluid flows inside a cylindrical pipe of radius R along the x-axis, developing the analytical parabolic velocity profile shown in red.

Results The final physical time was set to $T_{\text{end}} = 0.5\,\text{s}$. As reported in Table 8, the streamwise velocity component shows no convergence in L_2 norm, maintaining a roughly constant error of about 3×10^{-2}. The analytical values of the other two velocity components are identically zero; nevertheless, their L_2 norms fluctuate around 10^{-4}. The analytical pressure field was not specified, but the computed values remain bounded throughout the simulation.

Three hypotheses have been proposed to explain the lack of observed convergence. The first relates to topology-induced aliasing, caused by the inability of a Cartesian grid to accurately represent circular geometries, leading to a staircase approximation. Although this error is expected to scale at first order, it may still significantly affect the observed convergence behavior. The second pertains to the interface width, which is theoretically expected to scale with the mesh size h. This scaling, however, has been neglected because the porosity function is compiled prior to the code receiving the grid information. The third concerns the error evaluation itself, which

may be dominated by contributions from the interface region, thereby obscuring the asymptotic convergence behavior of the bulk solution.

3.3. Performance and parallelism

To rigorously assess the high-performance capabilities of the solver, a comprehensive scalability analysis was conducted targeting both strong and weak scaling regimes. Dedicated Python automation scripts were developed to validate parallel solutions correctness and to orchestrate these benchmarks, managing the submission of MPI jobs and deriving the following key performance indicators:

- **Execution time (T_p)**, the wall-clock time required to complete a fixed number of time steps;
- **Time per cell-step (τ_p)**, the execution time normalized per time step per cell, defined as $\tau_{cell} = T_p/(N_{steps} \cdot N_{cells})$, enabling objective performance comparisons across disparate grid resolutions;
- **Speedup (S_p)**, defined as $S_p = T_1/T_p$, representing the relative performance gain over the serial implementation;
- **Efficiency (E_p)**, defined as $E_p = S_p/p$, measuring the effective utilization of allocated resources.

3.3.1 Parallel consistency

To ensure the correctness of the distributed implementation, a dedicated verification framework was developed using a Python script. This tool guarantees domain decomposition introduces no numerical artifacts by comparing the distributed VTK outputs of a parallel execution against a strictly sequential reference.

The parallel solver has been proven to strictly inherit the convergence order and stability properties of the underlying sequential algorithm.

3.3.2 Strong scalability

This analysis evaluates the solver's ability to reduce time-to-solution for a fixed total workload. The global domain size was held constant while the number of MPI processes was progressively increased.
In this regime, the **ideal outcome is linear speedup** ($S_p \approx p$), where the execution time decreases in direct proportion to the number of processors added. Deviations from this ideal are typically caused by parallel overheads, such as communication latency and non-parallelizable I/O operations, which become more dominant as the local workload per core decreases.

Performance was evaluated on a $64 \times 64 \times 64$ ($N_{cell} \approx 2.6 \times 10^5$) MMS test case. The quantitative results are detailed in Table 9, while Figure 16 visually contrasts the observed performance against the ideal linear trends.

Results The solver maintains a trajectory close to the ideal linear line for low core counts, achieving $S_2 = 1.92$ and $S_4 = 3.30$. This efficient scaling demonstrates the communication overhead from halo exchanges and Schur complement solves remains sub-dominant compared to the computational workload. Consequently, parallel efficiency remains high, starting at 95.8% for 2 cores and staying above 82% at 4 cores.

However, a significant deviation occurs at $N_p = 8$, where the speedup plateaus at $S_8 = 3.13$, causing the efficiency to drop drastically to 39.1%. This behavior is characteristic of parallel resource saturation on the test machine rather than a flaw in the parallel algorithm itself.

3.3.3 Weak scalability

This analysis addresses the requirement of leveraging larger computational resources to achieve higher spatial resolution, alongside more efficient resource utilization as the number of processors scales up.
In this regime, the **ideal outcome is a constant execution time** ($T_p \approx T_1$), which demonstrates the solver can maintain a fixed time-to-solution even as the problem complexity scales up with the available hardware.

Tests were performed by maintaining a constant workload per processor of approximately 2.6×10^5 cells (starting from a $64 \times 64 \times 64$ baseline); as the number of cores increased, the global domain size grew proportionally. Results are summarized in Table 10 and visualized in Figure 17.

Results For low core counts, the solver demonstrates robust weak scaling characteristics. The execution time increases only moderately, resulting in a parallel efficiency above 81%. This relative flatness in execution time confirms that the algorithmic overheads do not grow prohibitively as the problem size expands.

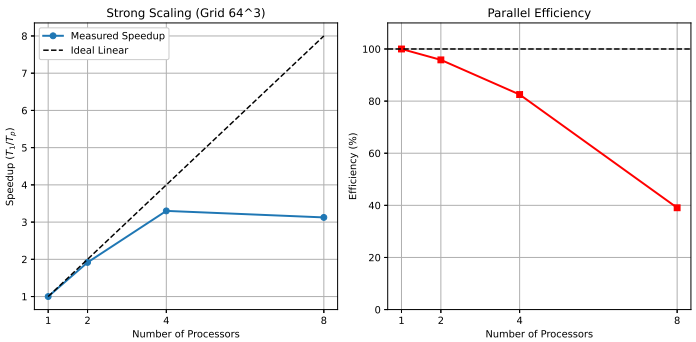


Figure 16: **Strong scalability analysis on a 64^3 grid.** Left: observed speedup against ideal linear scaling. Right: degradation of parallel efficiency.

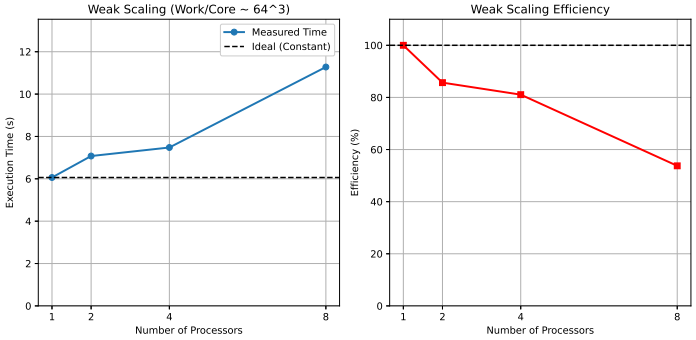


Figure 17: **Weak scalability analysis on a $\approx 2.6 \times 10^5$ cells-per-process workload.** Left: execution time against ideal constant (sequential baseline). Right: degradation of parallel efficiency.

3.3.4 Target metrics analysis

The **time per cell-step** metric serves as a normalized indicator of the solver's computational efficiency, independent of the total problem size. It effectively quantifies the cost of updating a single finite volume cell by one time interval, incorporating both the algebraic operations and the parallel communication overhead. Maintaining τ_{cell} close to the serial baseline is the primary objective for high-performance execution.

Baseline efficiency In the serial configuration ($N_p = 1$), the solver achieves a τ_{cell} of approximately $\mathbf{0.46 - 0.48\,\mu s}$. This value represents the intrinsic computational **cost of** the algorithm's **numerics** - specifically, the construction of the tridiagonal systems and the explicit derivative computations - on the test hardware.

Impact of parallelism As the processor count increases, τ_{cell} provides insight into the **parallel overhead**.

- **Low-count stability** - In the efficient scaling regime ($N_p \leq 4$), τ_{cell} undergoes only a marginal increase, rising to $\approx 0.54\,\mu s$ in both strong and weak settings. This slight uplift (+12%) reflects the unavoidable cost of halo exchanges and the Schur complement interface solve. The stability of this metric confirms that the communication latency is well-amortized by the local computation.
- **Hardware saturation** - At $N_p = 8$, the metric reveals the precise nature of the bottleneck. In the strong scaling case, τ_{cell} jumps to $\mathbf{1.12\,\mu s}$ (a 2.3× increase). Since the total number of floating-point operations remains constant, this drastic increase indicates that either/both the communication overhead is increasing or/and the CPU is stalling (symptom of memory-boundness).

4. Conclusions

The primary objective of this project was to design, implement, and validate a high-performance parallel solver for the Navier-Stokes-Brinkman equations. The development focused on balancing mathematical rigor with software engineering best practices, resulting in a solver that is both numerically robust and computationally efficient.

The numerical fidelity of the implementation was rigorously validated using the MMS, demonstrating the expected **second-order spatial convergence** for both velocity and pressure fields. This result confirms that the operator splitting and finite difference discretizations were correctly implemented and are capable of resolving flow features with high precision without numerical artifacts.

In terms of computational performance, the solver successfully exceeded its strict design targets. A primary goal was to achieve an **execution time lower than 10^{-6} seconds per cell-step**. The implementation significantly outperformed this benchmark, achieving a baseline metric of approximately $\mathbf{4.6 \times 10^{-7}}$ **s/step/cell** in serial execution. Even under the added overhead of parallel communication, the metric remained well below the target threshold, validating the efficacy of the optimized memory management and pre-computed linear algebra structures.

Finally, the distributed memory implementation, orchestrated via MPI, proved to be both numerically consistent and scalable. The solver exhibited **near-linear strong scalability** and **robust weak scaling** on the test hardware. While symptoms of hardware saturation were observed at higher core counts, the algorithmic design demonstrated the capacity to scale effectively given sufficient hardware resources.

Ultimately, while the `NSBSolver` successfully delivers a promising and verified tool, it is essential to acknowledge the inherent trade-off between software abstraction and raw performance. The modular design - while crucial for maintainability and ease of use - inevitably places a ceiling on the achievable throughput compared to a bare-metal implementation. Consequently, despite the efficiency demonstrated in this study, the solver's performance could be significantly higher. Unlocking this theoretical maximum would require sacrificing some of the current architectural elegance in favor of more aggressive optimizations and flattened memory structures.

Contributions

The development of the solver was a collaborative effort. A more specific division of labor, broken down by component and task, is detailed below.

Core architecture & Parallelism

- **Distributed data structures** (`core/Field`, `core/VectorField`)
  → Daniele Piano, Federico Pinto, Stefano Pedretti
- **MPI environment wrapper & Halo exchange** (`core/MpiEnv`, `core/HaloHandler`)
  → Daniele Piano, Ettore Cirillo

Numerical engine

- **Finite differences** (`numerics/Derivatives`)
  → Mattia Gotti, Stefano Pedretti, Giulio Martella
- **Matrix storage** (`numerics/TridiagMat`)
  → Mattia Gotti, Federico Pinto, Stefano Pedretti
- **Thomas solver** (`numerics/ThomasSolver`)
  → Mattia Gotti, Federico Pinto
- **Sequential Schur solver**
  → Michele Milani
- **Distributed Schur solver** (`numerics/SchurSolver`)
  → Daniele Piano, Ettore Cirillo

Physics & Orchestration

- **Simulation driver** (`simulation/NSBSolver`)
  → Federico Pinto, Giulio Martella
- **Fractional steps & Boundary conditions handling** (`simulation/ViscousStep`, `simulation/PressureStep`)
  → Giulio Martella

Input/Output & Initialization

- **Data factory** (`simulation/Initializer`)
  → Ettore Cirillo, Michele Milani
- **I/O management** (Configuration, `io/LogWriter`, `io/VTKWriter`)
  → Ettore Cirillo, Michele Milani

Profiling & Performance optimization

- **Code profiling**
  → Michele Milani
- **Code optimization**
  → Daniele Piano

Validation, Results & Documentation

- **Testing**
  → Mattia Gotti, Daniele Piano, Stefano Pedretti
- **Convergence study**
  → Michele Milani
- **Consistency & Scalability study**
  → Daniele Piano
- **Manufactured Solutions & Laminar study cases**
  → Giulio Martella
- **Code documentation, GitHub workflow**
  → Mattia Gotti

Appendix

Convergence results

Table 1: Numerical errors for velocity and pressure with space–time coupled refinement. Linear scaling $\Delta t \sim \Delta x$.

N_x	h	Δt	L_2 Err_u	L_2 Err_v	L_2 Err_w	L_2 Err_p	L_1 Err_p
20	3.0769e-01	1.0000e-02	1.1522e-03	1.1704e-03	2.7032e-03	6.5699e-02	4.4287e-02
30	2.0339e-01	6.6667e-03	4.9855e-04	5.0709e-04	1.1963e-03	2.6942e-02	1.8689e-02
40	1.5190e-01	5.0000e-03	2.7834e-04	2.8333e-04	6.7257e-04	1.4934e-02	1.0447e-02
50	1.2121e-01	4.0000e-03	1.7768e-04	1.8025e-04	4.2986e-04	9.6085e-03	6.7581e-03
60	1.0084e-01	3.3333e-03	1.2343e-04	1.2449e-04	2.9818e-04	6.7645e-03	4.7795e-03
70	8.6331e-02	2.8571e-03	9.0535e-05	9.1802e-05	2.1935e-04	5.0587e-03	3.5899e-03
80	7.5472e-02	2.5000e-03	6.9314e-05	7.0358e-05	1.6803e-04	3.9533e-03	2.8165e-03
90	6.7039e-02	2.2222e-03	5.4772e-05	5.5555e-05	1.3275e-04	3.1941e-03	2.2838e-03
100	6.0302e-02	2.0000e-03	4.4372e-05	4.4985e-05	1.0752e-04	2.6485e-03	1.8998e-03

Table 2: Convergence order analysis for velocity and pressure with space–time coupled refinement. Linear scaling $\Delta t \sim \Delta x$.

Variable	Global Order	Local Orders
Velocity u	2.00	2.02, 2.00, 1.99, 1.98, 2.00, 1.99, 1.99, 1.99
Velocity v	2.00	2.02, 1.99, 2.00, 2.01, 1.96, 1.98, 1.99, 1.99
Velocity w	1.98	1.97, 1.97, 1.98, 1.99, 1.98, 1.98, 1.99, 1.99
Pressure (L_2)	1.96	2.15, 2.02, 1.95, 1.91, 1.87, 1.83, 1.80, 1.77
Pressure (L_1)	1.93	2.08, 1.99, 1.93, 1.88, 1.84, 1.80, 1.77, 1.74

Table 3: Numerical errors for velocity and pressure at different mesh resolutions with fixed time step ($\Delta t = 1.0 \times 10^{-3}$). Spatial analysis.

N_x	h	Δt	L_2 Err_u	L_2 Err_v	L_2 Err_w	L_2 Err_p	L_1 Err_p
20	3.0769e-01	1.0000e-03	7.7215e-04	7.7185e-04	1.7952e-03	1.1912e-01	4.7105e-02
30	2.0339e-01	1.0000e-03	3.2458e-04	3.2504e-04	7.8298e-04	2.6675e-02	1.6461e-02
40	1.5190e-01	1.0000e-03	1.7977e-04	1.8026e-04	4.3698e-04	1.3253e-02	8.5890e-03
50	1.2121e-01	1.0000e-03	1.1465e-04	1.1501e-04	2.7852e-04	8.1115e-03	5.3291e-03
60	1.0084e-01	1.0000e-03	7.9920e-05	8.0167e-05	1.9341e-04	5.5143e-03	3.6587e-03
70	8.6331e-02	1.0000e-03	5.8439e-05	5.8698e-05	1.4148e-04	4.0052e-03	2.6843e-03
80	7.5472e-02	1.0000e-03	4.4796e-05	4.5025e-05	1.0821e-04	3.0587e-03	2.0731e-03
90	6.7039e-02	1.0000e-03	3.5478e-05	3.5670e-05	8.5443e-05	2.4285e-03	1.6660e-03
100	6.0302e-02	1.0000e-03	2.8826e-05	2.8990e-05	6.9194e-05	1.9905e-03	1.3825e-03

Table 4: Convergence order analysis for velocity and pressure at fixed time step with varying mesh resolution. Spatial analysis.

Variable	Global Order	Local Orders
Velocity u	2.01	2.09, 2.02, 1.99, 1.96, 2.02, 1.98, 1.97, 1.96
Velocity v	2.01	2.09, 2.02, 1.99, 1.96, 2.01, 1.97, 1.97, 1.96
Velocity w	2.00	2.00, 2.00, 2.00, 1.98, 2.01, 1.99, 1.99, 1.99
Pressure (L_2)	2.41	3.61, 2.40, 2.18, 2.10, 2.06, 2.01, 1.95, 1.88
Pressure (L_1)	2.14	2.54, 2.23, 2.11, 2.04, 1.99, 1.92, 1.85, 1.76

Table 5: Numerical errors for velocity and pressure at fixed mesh resolution ($N_x = 100$) for different time steps. Temporal analysis.

N_x	h	Δt	L_2 Err$_u$	L_2 Err$_v$	L_2 Err$_w$	L_2 Err$_p$	L_1 Err$_p$
100	6.0302e-02	1.0000e-02	3.3679e-04	3.1775e-04	4.1348e-04	2.0926e-02	1.2624e-02
100	6.0302e-02	2.0000e-02	1.3188e-03	1.1663e-03	1.5888e-03	6.2890e-02	3.9809e-02
100	6.0302e-02	5.0000e-02	6.8783e-03	5.1321e-03	8.7672e-03	2.2936e-01	1.5815e-01
100	6.0302e-02	8.3333e-02	1.3201e-02	1.0964e-02	1.9760e-02	3.5278e-01	2.5392e-01
100	6.0302e-02	1.0000e-01	1.6179e-02	1.4425e-02	2.6138e-02	3.8845e-01	2.8235e-01

Table 6: Convergence order analysis for velocity and pressure with varying time step at fixed mesh resolution. Temporal analysis.

Variable	Global Order	Local Orders
Velocity u	1.69	1.97, 1.80, 1.28, 1.12
Velocity v	1.64	1.88, 1.62, 1.49, 1.50
Velocity w	1.80	1.94, 1.86, 1.59, 1.53
Pressure (L_2)	1.28	1.59, 1.41, 0.84, 0.53
Pressure (L_1)	1.37	1.66, 1.51, 0.93, 0.58

Table 7: Convergence results for the streamwise velocity u in the Couette-Poiseuille (channel) flow.

N_x	20	40	60	80	100
h	5.1282e-2	2.5316e-2	1.6807e-2	1.2579e-2	1.0050e-2
dt	1.0000e-1	5.0000e-2	3.3333e-2	2.5000e-2	2.0000e-2
L_2 Err$_u$	1.5752e-1	4.0040e-2	1.6668e-2	9.1202e-3	5.8031e-3
Local order	-	1.94	2.01	2.08	2.14
Global order			2.03		

Table 8: Convergence results for the streamwise velocity u in Hagen-Poiseuille (pipe) flow.

Nx	20	40	60	80	100
h	5.1282e-2	2.5316e-2	1.6807e-2	1.2579e-2	1.0050e-2
dt	1.0000e-2	5.0000e-3	3.3333e-3	2.5000e-3	2.0000e-3
L_2 Err$_u$	3.2184e-2	3.3617e-2	3.4041e-2	3.4316e-2	3.4452e-2
Local order	-	-0.06	-0.03	-0.03	-0.02
Global order			-0.04		

Table 9: Strong scalability results on a fixed 64^3 grid. Efficiency is relative to the single-core baseline.

p	T_p $[s]$	Speedup S_p	Ideal S_p	E_p	τ_p $[\mu s/(\text{cells} \cdot \text{steps})]$
1	6.27	1.00	1.00	100.0%	0.47846
2	3.27	1.92	2.00	95.8%	0.48419
4	1.90	3.30	4.00	82.5%	0.54519
8	2.01	3.13	8.00	39.1%	1.1166

Table 10: Weak scalability results on a fixed $\approx 2.6 \times 10^5$ cells-per process workload. Efficiency is relative to the single-core baseline.

p	N	N_{cell}	T_p $[s]$	E_p	τ_{cell} $[\mu s/(\text{cells} \cdot \text{steps})]$
1	64	262,144	6.06	100.0%	0.462
2	82	551,368	7.08	85.7%	0.501
4	102	1,061,208	7.48	81.1%	0.542
8	128	2,097,152	11.28	53.7%	0.821

References

[1] Andrea Manzoni. *Reduced Models for Optimal Control, Shape Optimization and Inverse Problems in Haemodynamics*. PhD thesis, École Polytechnique Fédérale de Lausanne, 2012. Thèse No. 5402.

[2] Maciej J. Balajewicz, Earl H. Dowell, and Bernd R. Noack. Low-dimensional modelling of high-Reynolds-number shear flows incorporating constraints from the Navier–Stokes equations. *Journal of Fluid Mechanics*, 729:285–308, 2013.

[3] Charles S. Peskin. The immersed boundary method. *Acta Numerica*, 11:479–517, 2002.

[4] Paolo Luchini, Davide Gatti, Alessandro Chiarini, Federica Gattere, Marco Atzori, and Maurizio Quadrio. A simple and efficient second-order immersed-boundary method for the incompressible Navier–Stokes equations. *Journal of Computational Physics*, 539:114245, 2025.

[5] T. Gornak, Jean-Luc Guermond, O. Iliev, and Peter D. Minev. A direction splitting approach for incompressible Brinkman flow. *Computers & Mathematics with Applications*, 74(1):1–13, 2017.

[6] Jean-Luc Guermond, Peter D. Minev, and Jie Shen. An overview of projection methods for incompressible flows. *Computer Methods in Applied Mechanics and Engineering*, 195(44-47):6011–6045, 2006.

[7] Jean-Luc Guermond and Peter D. Minev. A new class of fractional step techniques for the incompressible Navier–Stokes equations using direction splitting. *Comptes Rendus Mathematique*, 348(9-10):581–585, 2010.

[8] Randall J. LeVeque. *Finite difference methods for ordinary and partial differential equations: steady-state and time-dependent problems*. SIAM, 2007.

[9] Alessandro Chiarini, Maurizio Quadrio, and Franco Auteri. A direction-splitting Navier–Stokes solver on co-located grids. *Journal of Computational Physics*, 429:109996, 2021.

[10] Jean-Luc Guermond and Peter D. Minev. Start-up flow in a three-dimensional lid-driven cavity by means of a massively parallel direction splitting algorithm. *International Journal for Numerical Methods in Fluids*, 68(7):856–871, 2012.

[11] Jim Douglas, Jr. On the numerical integration of $u_{xx} + u_{yy} = u_t$ by implicit methods. *Journal of the Society for Industrial and Applied Mathematics*, 3(1):42–65, 1955.

[12] Philippe Angot, Johnwill Keating, and Peter D. Minev. A direction splitting algorithm for incompressible flow in complex geometries. *Computer Methods in Applied Mechanics and Engineering*, 217:111–120, 2012.

[13] Olga A. Ladyzhenskaya. *The mathematical theory of viscous incompressible flow*. Gordon and Breach, New York, second english edition edition, 1969.

[14] Ivo Babuška. The finite element method with Lagrangian multipliers. *Numerische Mathematik*, 20:179–192, 1973.

[15] Franco Brezzi. On the existence, uniqueness and approximation of saddle-point problems arising from Lagrangian multipliers. *RAIRO Analyse numérique*, 8(R2):129–151, 1974.

[16] S. Shashank, J. Larsson, and G. Iaccarino. A co-located incompressible Navier–Stokes solver with exact mass, momentum and kinetic energy conservation in the inviscid limit. *Journal of Computational Physics*, 229(12):4425–4430, 2010.

[17] C. M. Rhie and W. L. Chow. Numerical study of the turbulent flow past an airfoil with trailing edge separation. *AIAA Journal*, 21(11):1525–1532, 1983.

[18] Pijush K. Kundu and Ira M. Cohen. *Fluid Mechanics*. Academic Press, San Diego, 4th edition, 2008.