LIÈGE université Library

https://lib.uliege.be

https://matheo.uliege.be

Comparison of linear solvers for the incompressible Navier-Stokes equations in PFEM simulations

Auteur : Ghalem, Rayane
Promoteur(s) : Boman, Romain
Faculté : Faculté des Sciences appliquées
Diplôme : Master en ingénieur civil en aérospatiale, à finalité spécialisée en "aerospace engineering"
Année académique : 2025-2026
URI/URL : http://hdl.handle.net/2268.2/26083

Avertissement à l'attention des usagers :

Tous les documents placés en accès ouvert sur le site le site MatheO sont protégés par le droit d'auteur. Conformément aux principes énoncés par la "Budapest Open Access Initiative"(BOAI, 2002), l'utilisateur du site peut lire, télécharger, copier, transmettre, imprimer, chercher ou faire un lien vers le texte intégral de ces documents, les disséquer pour les indexer, s'en servir de données pour un logiciel, ou s'en servir à toute autre fin légale (ou prévue par la réglementation relative au droit d'auteur). Toute utilisation du document à des fins commerciales est strictement interdite.

Par ailleurs, l'utilisateur s'engage à respecter les droits moraux de l'auteur, principalement le droit à l'intégrité de l'oeuvre et le droit de paternité et ce dans toute utilisation que l'utilisateur entreprend. Ainsi, à titre d'exemple, lorsqu'il reproduira un document par extrait ou dans son intégralité, l'utilisateur citera de manière complète les sources telles que mentionnées ci-dessus. Toute utilisation non explicitement autorisée ci-avant (telle que par exemple, la modification du document ou son résumé) nécessite l'autorisation préalable et expresse des auteurs ou de leurs ayants droit.

UNIVERSITÉ DE LIÈGE - FACULTÉ DES SCIENCES APPLIQUÉES



Comparison of linear solvers for the incompressible Navier-Stokes equations in PFEM simulations

Master's thesis presented by Rayane Ghalem in order to obtain the degree of
Master of Science in Aerospace Engineering

Thesis supervisor:
BOMAN Romain

Academic year 2025 - 2026

Acknowledgements

First and foremost, I would like to thank my supervisor, Professor Romain Boman, for his guidance, availability, and continued support throughout this master thesis. His expertise has been essential in shaping this work.

The PFEM team is full of hardworking and passionate people, and I am deeply grateful to Dorian Bogucki and Martin Lacroix for their support, their invaluable feedback, and the stimulating discussions that have greatly contributed to this work, and nourished further my passion for numerical physical modeling.

I am profoundly grateful to my mother, my brother and my sister for their unconditional love. Knowing they were always there for me gave me the strength to keep moving forward even in my darkest moments.

Finally, this last word is for my father. You made all of this possible by giving me the opportunity to pursue my studies in Belgium. Seeing you at the hospital was one of the hardest moments of my life, but it also showed me how strong you are. I want you to know that from now on, everything will be okay. It is my turn to take care of us, and I promise I will work hard every day to make you proud and protect you as you have always protected me. I love you.

Contents

Introduction **13**

1 Fluid Mechanics **15**
1.1 Incompressible Navier-Stokes equations . . . 15
1.2 Weak formulation . . . 16

2 The Particle Finite Element Method **17**
2.1 Space discretization . . . 17
2.2 Time integration . . . 18
2.3 Remeshing . . . 20
2.4 Implementation . . . 21
2.5 Benchmark . . . 23

3 Presentation of Sparse Linear Solvers **27**
3.1 Overview . . . 27
3.1.1 Condition number . . . 28
3.1.2 Matrix storage schemes . . . 28
3.1.3 BLAS . . . 30
3.2 Direct sparse solvers . . . 30
3.2.1 Triangular systems . . . 30
3.2.2 Gaussian elimination . . . 31
3.2.3 LU factorization . . . 32
3.2.4 QR factorization . . . 32
3.2.5 Ordering . . . 33
3.2.6 Elimination trees . . . 34
3.2.7 Supernodal methods . . . 35
3.2.8 Resolution steps in a sparse direct solver . . . 36
3.3 Iterative sparse solvers . . . 36
3.3.1 Krylov-subspace methods . . . 36
3.3.2 GMRES . . . 38
3.3.3 BiConjugate Gradient . . . 38
3.3.4 BiConjugate Gradient Stabilized . . . 39
3.3.5 Iterative refinement . . . 40
3.3.6 Preconditioning . . . 40
3.3.7 Jacobi Preconditioner . . . 41

3.3.8 Flexible GMRES . . . 41
3.3.9 ILU Preconditioners . . . 42
3.4 Presentation of Sparse Linear Solver Libraries . . . 42
3.4.1 Ordering libraries . . . 42
3.4.2 Eigen . . . 43
3.4.3 MKL . . . 43
3.4.4 PaStiX . . . 45
3.4.5 AMGCL . . . 45
3.4.6 SuperLU . . . 46
3.4.7 Panua PARDISO . . . 46

4 Comparison of Linear Solvers **47**
4.1 Methodology . . . 47
4.1.1 Computer Specifications . . . 48
4.1.2 Benchmark Linear Systems . . . 48
4.1.3 Speedup, Parallel Efficiency and Performance Increase Factor . . . 50
4.1.4 Linear Solvers Investigated . . . 50
4.1.5 Eigen Solvers as Performance Baseline . . . 51
4.2 Direct Solvers . . . 52
4.2.1 SuperLU based solvers . . . 52
4.2.2 PARDISO-based Solvers . . . 54
4.2.3 Time Complexities . . . 57
4.2.4 Parallel efficiency . . . 61
4.2.5 Memory Requirements . . . 62
4.3 Iterative Solvers . . . 64
4.3.1 Loose Tolerance with Native Preconditioners . . . 64
4.3.2 Tolerance Setting . . . 67
4.3.3 ILUT Preconditioners . . . 75
4.3.4 Restarted GMRES . . . 78
4.3.5 Memory Requirements . . . 81
4.4 Conclusions . . . 84

5 Laser track benchmark **87**

6 Conclusion **91**

List of Figures

2.1 PFEM remeshing procedure adapted from Cerquaglia [1]. 20
2.2 Flowchart of the PFEM algorithm. 22
2.3 Geometry of the 3D dam break. 23
2.4 Evolution of the fluid domain over time from $t = 0$ s to $t = 0.3$ s. 24
2.5 Sparsity representation of the linear system matrix at the first iteration of the nonlinear algorithm at $t = 0$. It is divided in 16 blocks which couple the different types of degrees of freedom. Blue dots represent non-zeros entries. 25

3.1 Sparsity pattern of matrix $\mathbf{L}+\mathbf{U}$ after factorization of $\mathbf{A}$ into $\mathbf{LU}$ without reordering. $\mathbf{L}$ and $\mathbf{U}$ are summed to represent visually the increase of nonzeros. 33
3.2 Sparsity pattern of matrix $\mathbf{L}+\mathbf{U}$ after factorization of $\mathbf{A}$ into $\mathbf{LU}$ using COLAMD algorithm for reordering. $\mathbf{L}$ and $\mathbf{U}$ are summed to represent visually the increase of nonzeros. 34
3.3 Matrix with fill-in $\mathbf{F} = \mathbf{L}+\mathbf{U}$ with the elimination tree $\mathcal{T}(\mathbf{A})$ resulting from the factorization of a matrix $\mathbf{A}$. Arrows represent data dependencies between the different elimination steps required to factorize $\mathbf{A}$ for both the matrix with fill-in $\mathbf{F}$ and the elimination tree: from the elimination of children of node j, elimination step at column j can be performed. 35
3.4 Resolution of a simple linear system of solution $\tilde{\mathbf{x}}$ using the Generalized Minimal Residual method (GMRES), which is a Krylov-subspace method. Vectors $\mathbf{k}_1 = \mathbf{r}_0$ and $\mathbf{k}_2 = \mathbf{r}_0 + \mathbf{A}\mathbf{r}_0$ generate the search space on which the approximation of $\mathbf{x}$ is obtained. The approximation corresponds to the projection of $\mathbf{x}$ onto $\mathbf{x}_0 + \mathbf{K}$. 37

4.1 Selection of time instants for linear solver comparison. (a) Evolution of the linear solver computation time throughout the simulation for a mesh with $n = 10{,}000$ particles. (b)–(f) Snapshots for a mesh with $n = 50{,}000$ particles at the five selected time instants. (g) Pressure colorbar, where the color is proportional to pressure intensity (high pressure in red, low in blue). 49
4.2 Speedup S as a function of the number of threads j ($n = 50{,}000$ particles, 5 time instants). 51
4.3 Computational cost of PFEM3D using the Eigen linear solvers on a dam break benchmark with 50 000 particles over 10 time steps. *Left*: estimated total computation time, obtained by scaling the 10-step duration by a factor of 120. *Right*: breakdown of computation time for Eigen BiCGStab, with computation time decomposed into LU factorization and solve phases. 52

4.4 Performance Increase Factor of SuperLU Dist relative to Eigen LU as a function of the number of threads (left) and peak memory requirement (right), for n = 50,000 particles. . . . 53
4.5 Total computation time of the DSS solver as a function of the number of threads for MKL in non-optimized and optimized configurations. Each subfigure corresponds to a different mesh size. . . . 55
4.6 Computation time of MKL DSS and Panua PARDISO as a function of the number of threads for linear systems arising from the dam break test case in PFEM3D. . . . 56
4.7 Comparison of MKL DSS (2019 optimized) and SuperLU Dist against Eigen LU for n = 50,000 particles, as a function of the number of threads j. . . . 57
4.8 Total computation time of all direct solvers as a function of the number of threads. Subfigure (a) includes Eigen LU and SuperLU Dist, which are only evaluated at n = 50,000; subfigures (b)–(e) show PaStiX LU, MKL DSS, and Panua Pardiso across all mesh sizes. . . . 58
4.9 Total computation time of MKL DSS, PaStiX LU, and Panua Pardiso as a function of the number of non-zeros (NNZ), plotted on a log-log scale for each thread count j. Dashed lines indicate the fitted power-law model. . . . 60
4.10 Parallel efficiency as a function of the number of threads for different linear solvers. Each subfigure corresponds to a different mesh size. . . . 61
4.11 Memory consumption of the direct solvers as a function of the mesh size. . . . 62
4.12 Log-log plot of peak memory consumption as a function of NNZ for the three direct solvers. Solid lines show measured data (j = 1), shaded bands span the min/max across all thread counts, dashed lines extrapolate the fitted power law, and the orange tick marks the estimated NNZ at which memory reaches 256 GB. . . . 63
4.13 Total computation time of the three iterative solvers as a function of the number of threads. Hatched bars indicate that at least one time step encountered a non-convergence. Each subfigure corresponds to a different mesh size. . . . 65
4.14 Performance Increase Factor relative to Eigen BiCGStab (j = 1) as a function of the number of threads (left) and peak memory requirement (right), for n = 300,000 particles. . . . 66
4.15 Total computation time of all direct and iterative solvers as a function of the number of threads for each mesh size. Each subfigure corresponds to a different mesh size. 67
4.16 Total CPU time (left column) and number of nonlinear iterations (right column, with CPU time shown in grey on the secondary axis) as a function of the convergence tolerance for BiCGStab and GMRES with ILU(0) preconditioner (j = 1 thread). Rows correspond to mesh sizes of n = 50,000, 90,000, 150,000, and 300,000 particles. The shaded region indicates the identified optimal tolerance range $[10^{-6}, 10^{-1}]$. . . . 69
4.17 Schematic illustration of the effect of linear solver tolerance on Newton–Raphson convergence. Each row represents one nonlinear iteration; coloured squares denote linear iterations required to solve the corresponding linear system. A tight tolerance ($\varepsilon_{\text{lin}} = 10^{-14}$) yields few nonlinear iterations but many linear iterations per system, while a relaxed tolerance ($\varepsilon_{\text{lin}} = 0.1$) increases the nonlinear iteration count but reduces the number of linear iterations per system. . . . 70

4.18 Average number of linear iterations per system (left column) and time per linear iteration (right column) as a function of the convergence tolerance for BiCGStab and GMRES with ILU(0) preconditioner ($j = 1$ thread). Rows correspond to mesh sizes of n = 50,000, 90,000, 150,000, and 300,000 particles. The shaded region indicates the identified optimal tolerance range $[10^{-6}, 10^{-1}]$. 71
4.19 Simulation time of complete PFEM3D simulations for the selected tolerance values across all mesh sizes (n = 50,000, 90,000, 150,000, and 300,000 particles). The shaded region indicates the tolerance range $[10^{-6}, 10^{-1}]$. Darker bars are extrapolated from five specific time instants. 73
4.20 Performance Improvement Factor (PIF) relative to Eigen LU for all solver libraries at $j = 1$ thread (a) and $j = 32$ threads (b). Hatched bars denote iterative solvers. An inset provides a zoomed view of the direct solver range for readability. 74
4.21 Total computation time as a function of the ILUT fill factor ρ for four values of the drop tolerance τ (n = 50,000 particles, $j = 1$ thread). The ILU(0) result is shown as a reference on the left of each panel. 76
4.22 Eigenvalue distribution of the system matrix $\mathbf{A}$ and the preconditioned matrices $\mathbf{M}^{-1}\mathbf{A}$ for three preconditioners: Jacobi ($\rho = 0$), ILU(0), and ILUT with fill factor $\rho = 8$. All eigenvalues are pooled across multiple checkpoints (n = 50,000 particles, $j = 32$ threads). 77
4.23 The direct-iterative continuum: as the quality of the preconditioner improves, the number of iterations required to reach convergence decreases. The iteration counts are approximate values observed at n = 50,000 particles. 78
4.24 GMRES(k) with ILU(0) preconditioner, $j = 32$ threads. Left: cost per linear iteration and solver iteration count. Right: average CPU time per time step. Horizontal lines: BiCGStab reference ($\varepsilon = 10^{-4}$). Dotted line: boundary of *Unrestarted* regime. Rows: n = 50,000, 90,000, 150,000, 300,000 particles. 80
4.25 Maximum memory usage as a function of the restart parameter for the GMRES implementations of MKL and AMGCL. Each subfigure corresponds to a different mesh size. 81
4.26 Peak memory consumption as a function of mesh size for direct solvers, Eigen BiCGStab, and AMGCL iterative solvers with ILU(0) preconditioning. 82
4.27 Log-log plot of peak memory consumption as a function of NNZ for the two AMGCL iterative solvers. Solid lines show measured data ($j = 1$), shaded bands span the min/max across all thread counts, dashed lines extrapolate the fitted power law, and the red cross marks the estimated NNZ at which memory reaches 256 GB. 83
4.28 Decision tree for selecting a linear solver for incompressible fluid PFEM3D simulations, based on the empirical findings of this chapter. 85

5.1 Initial configuration and boundary conditions of the laser track test. Figure adapted from Bogucki *et al.* [75] 87
5.2 Temperature (top), Von Mises stress (left) and pressure (right) distributions at $t = 1$ ms of the laser track test. Figure adapted from Bogucki *et al.* [75]. 89
5.3 Cumulative CPU time as a function of the simulation time for five solver configurations for the laser track test. 90

List of Tables

2.1 Key simulation properties for the 3D dam break Koshizuka. 23

3.1 CSR representation of a sparse matrix with diagonal and off-diagonal non-zero elements. 29
3.2 CSC representation of the same sparse matrix, organized by columns instead of rows. 29

4.1 Hardware specifications of the computational platform. 48
4.2 Ratio of Pardiso PIF to MKL DSS PIF across mesh sizes and thread counts. Values above 1 indicate Panua PARDISO is faster; values below 1 indicate MKL DSS is faster. 56
4.3 Empirical time complexity exponent α such that $T(\text{nnz}) \propto \text{nnz}^{\alpha}$, estimated from the slope of a linear regression of $\log T$ vs $\log(\text{nnz})$ across all problem sizes (n = 50 000, 90 000, 150 000, 300 000). Columns indicate the number of threads; the last column is the average across all thread counts. Cells are colored from green (lowest exponent, best) to red (highest exponent, worst). 59
4.4 Ratio of PaStiX LU PIF to MKL DSS PIF across mesh sizes and thread counts. Green: PaStiX faster (ratio ≥ 1). Red: MKL DSS faster (ratio < 1). 61
4.5 Empirical memory complexity exponent α such that $M(\text{nnz}) \propto \text{nnz}^{\alpha}$, estimated from the slope of a linear regression of $\log M$ vs $\log(\text{nnz})$ across all problem sizes (n = 50 000, 90 000, 150 000, 300 000). Columns indicate the number of threads and the rightmost column is the average across all thread counts. 64
4.6 Empirical memory complexity exponent α such that $M(\text{nnz}) \propto \text{nnz}^{\alpha}$, estimated from the slope of a linear regression of $\log M$ vs $\log(\text{nnz})$ across all problem sizes (n = 50 000, 90 000, 150 000, 300 000). Columns indicate the number of threads and the rightmost column is the average across all thread counts. Cells are colored from green (closest to linear) to red (furthest from linear). 83

5.1 Key simulation properties for the laser track test case. 88

Introduction

The Navier–Stokes equations describe the motion of fluids and are at the core of many engineering and scientific problems. Their resolution is essential for modeling a wide range of phenomena such as wave impact, dam breaks, water sloshing and hydraulic infrastructure design.

To address these problems, the LTAS-MN2L laboratory of the University of Liège is developing PFEM3D, an in-house C++ software based on the Particle Finite Element Method (PFEM). In the PFEM framework, the equations are expressed in Lagrangian form, allowing the computational mesh to follow the fluid particles. This method has been designed to tackle problems involving free surfaces, thanks to a remeshing procedure, enabling a natural handling of moving boundaries. Although PFEM3D supports various types of simulations including fluid-solid interaction and phase change, this work focuses on the resolution of the incompressible Navier–Stokes equations. When using an implicit time integration scheme, each time step requires solving a nonlinear system of equations, which in turn involves the repeated resolution of large sparse linear systems. In practice, a large part of the computation time is spent in the resolution of these linear systems, which can be achieved using one of the many sparse linear solvers interfaced in PFEM3D.

Despite the availability of these solvers, no systematic comparison has been performed to evaluate their performance within the specific context of PFEM. Such a comparison is essential to identify the most appropriate solver and its parameters depending on the matrix structure or the simulation constraints. Choosing the right solver based on its computational behavior can significantly help reduce simulation times.

The objective of this work is therefore to analyze the performance and scalability of the sparse linear solvers interfaced in PFEM3D, as well as newly interfaced solvers, on a representative free surface flow problem, and to identify the most appropriate solver depending on the nature of the problem by extending the comparison to benchmark cases of increasing physical complexity. Chapter 1 introduces the governing fluid mechanics equations. Chapter 2 presents the Particle Finite Element Method and the benchmark problem. Chapter 3 describes the sparse linear solvers investigated. Chapter 4 presents the comparison of the solvers on the benchmark problem. Chapter 5 extends the comparison to a more complex laser track benchmark involving phase change and thermo-mechanical coupling. Finally, a conclusion summarizes the main findings and perspectives.

Chapter 1

Fluid Mechanics

This chapter presents the fluid mechanics equations that govern the problems addressed in this work. Section 1.1 introduces the incompressible Navier-Stokes equations in their strong form, and Section 1.2 derives their weak formulation.

1.1 Incompressible Navier-Stokes equations

In the strong form, the incompressible Navier–Stokes equations are composed of the continuity equation and the momentum conservation equation. These equations are respectively

$$\nabla \cdot \boldsymbol{v} = 0 \tag{1.1}$$

$$\rho \frac{D\boldsymbol{v}}{Dt} = \rho \boldsymbol{f} - \nabla p + \mu \nabla^2 \boldsymbol{v} \tag{1.2}$$

where $\boldsymbol{v}$ is the velocity vector field, ρ is the fluid density, p is the pressure, μ is the dynamic viscosity and $\boldsymbol{f}$ is the body force per unit mass. These equations are defined inside the finite space domain $\Omega = \Omega(t) \subset \mathbb{R}^3$ which is the set of all the mass points of the fluid.

The momentum conservation expresses the rate of change of momentum of a fluid particle, based on Newton's second law. The right-hand side consists of three terms: the pressure gradient force, the viscous diffusion term, and the body forces. The pressure gradient and viscous terms in the momentum equation represent internal contact forces within the continuous fluid. In the above systems, the fluid properties and the body forces are known while the velocity and the pressure are both unknowns.

To obtain a closed system of equations, appropriate boundary and initial conditions must be enforced. The boundary of the domain Ω is denoted $\partial\Omega = S_D \cup S_N$ where S_N and S_D are respectively the surfaces where Neumann boundary conditions and Dirichlet boundary conditions are applied. The following conditions are thus applied:

$$\boldsymbol{v}(\boldsymbol{X}, 0) = \boldsymbol{v}_0 \qquad \forall \boldsymbol{X} \in \Omega(0) \tag{1.3}$$

$$\boldsymbol{v}(\boldsymbol{x}, t) = \boldsymbol{v}_b \qquad \forall \boldsymbol{x} \in S_D \tag{1.4}$$

$$\boldsymbol{\sigma} \cdot \boldsymbol{n} = \boldsymbol{t}_b \qquad \forall \boldsymbol{x} \in S_N \tag{1.5}$$

where $\boldsymbol{v}_0$ is the initial velocity, $\boldsymbol{v}_b$ the velocity at the boundaries, $\boldsymbol{\sigma}$ the deviatoric stress tensor and $\boldsymbol{t}_b$ the prescribed traction at the boundaries.

The strong form of the Navier–Stokes equations, as presented in Equations (1.1) and (1.2), requires sufficiently smooth solutions. Indeed, the velocity field $\boldsymbol{v}$ must be at least once differentiable in time and twice differentiable in space, while the pressure field p must be at least once differentiable in space. Such requirements can be weakened through the use of a weak formulation.

1.2 Weak formulation

Due to the high requirements of the strong form on the solution differentiability, a weak form of the Navier–Stokes equations is employed. The following derivation is adapted from Cerquaglia [1] and John [2]. Let us consider the following function spaces of test functions:

$$\mathcal{V} = \left\{ \boldsymbol{w} \in H^1(\Omega) \mid \boldsymbol{w} = \boldsymbol{0} \text{ on } S_D \right\},$$
$$\mathcal{Q} = \left\{ q \in L^2(\Omega) \right\}$$

where $L^2(\Omega)$ is the space of square integrable functions over Ω, and $H^1(\Omega)$ the space of all functions in L^2 for which the first derivatives are also in L^2.

By multiplying Equations (1.1) and (1.2) by the test functions q and $\boldsymbol{v}$, respectively, and by integrating them on the volume Ω, one can obtain the weak form:

$$\int_\Omega q(\nabla \cdot \boldsymbol{v})\, d\Omega = 0 \tag{1.6}$$

$$\int_\Omega \boldsymbol{w} \cdot \rho \frac{D\boldsymbol{v}}{Dt}\, d\Omega = \int_\Omega \boldsymbol{w} \cdot \rho \boldsymbol{f}\, d\Omega + \int_\Omega p(\nabla \cdot \boldsymbol{w})\, d\Omega - \int_\Omega \mu \nabla \boldsymbol{v} : \nabla \boldsymbol{w}\, d\Omega \tag{1.7}$$

The weak formulation in Equations (1.6) and (1.7) relaxes the differentiability requirements of the Navier-Stokes equations such that $\boldsymbol{v}$ does not require to be twice differentiable in space, and p does not require to be differentiable in space. Indeed, this formulation not only holds true for solutions of the strong form called *generalized solutions*, but it also holds true for solutions $\boldsymbol{v} \in H^1$ and $p \in L^2$.

Chapter 2

The Particle Finite Element Method

Various approaches are used in computational fluid dynamics (CFD), which can broadly be categorized into two groups depending on the adopted kinematic description. *Eulerian methods* rely on a fixed computational mesh through which the fluid flows. In this framework, physical properties such as velocity and pressure are computed at stationary nodes, which can be thought of as fixed probes in space. *Lagrangian methods*, in contrast, attach the computational nodes to material particles, allowing the mesh to move and deform with the fluid. This naturally captures domain changes and is particularly advantageous for problems involving free surfaces.

This chapter presents one of these Lagrangian methods, which is the Particle Finite Element Method (PFEM). Section 2.1 introduces the space discretization, covering the Galerkin method and the PSPG stabilization. Section 2.2 describes the time integration scheme and the Newton-Raphson algorithm used to solve the resulting nonlinear system. Section 2.3 presents the remeshing procedure and explains how the PFEM extends the classical FEM. Section 2.4 presents the implementation in the PFEM3D software and Section 2.5 introduces the benchmark problem used throughout this work.

2.1 Space discretization

Classical methods in CFD rely primarily on Eulerian methods such as the Finite Element Method (FEM), the Spectral Difference Method (SDM) and the Discontinuous Galerkin Method (DGM) [3, 4]. Lagrangian methods have only started gaining attention about two decades ago [5].

In Lagrangian methods, elements can deform naturally as the nodes move with the fluid. Unfortunately, this also means that excessively distorted elements can arise, even in simple situations such as a vortex. Moreover, regions with a too high or too low particle density can result in a poor approximation of the solution. To address these issues, alternative formulations have been developed such as the Smoothed Particle Hydrodynamics (SPH) and the PFEM [1, 6].

This section describes how the Lagrangian FEM formulation is obtained through the Galerkin method and the PSPG stabilization.
To obtain the semi-discrete system of equations, the unknowns and their corresponding test functions

are spatially discretized. The velocity $\boldsymbol{v}$ and its test function $\boldsymbol{w}$ are approximated using the shape functions $\mathbf{N}_{\boldsymbol{v}}$ while the pressure p and its test function q are approximated using the shape functions $\mathbf{N}_p$ [7]:

$$\boldsymbol{v} = \mathbf{N}_{\boldsymbol{v}}^{\mathrm{T}}\mathbf{v}, \qquad \boldsymbol{w} = \mathbf{N}_{\boldsymbol{v}}^{\mathrm{T}}\mathbf{w}, \tag{2.1}$$

$$p = \mathbf{N}_p\mathbf{p}, \qquad q = \mathbf{N}_p^{\mathrm{T}}\mathbf{q}, \tag{2.2}$$

where $\mathbf{v}$ and $\mathbf{p}$ are the vectors of nodal parameters and $\mathbf{w}$ and $\mathbf{q}$ are the test functions.

Injecting these expressions into Equations (1.6) and (1.7) of the weak form leads to the following semi-discrete system:

$$\mathbf{M}\frac{D\mathbf{v}}{Dt} + \mathbf{K}\mathbf{v} - \mathbf{D}^{\mathrm{T}}\mathbf{p} = \mathbf{f} \tag{2.3}$$

$$\mathbf{D}\mathbf{v} = \mathbf{0} \tag{2.4}$$

where $\mathbf{M}$ is the mass matrix and $\mathbf{D}$ is the discrete divergence operator [1]. A common and computationally attractive choice of shape functions is the use of equal-order linear interpolation for both velocity and pressure. This means that both $\boldsymbol{v}$ and p are interpolated using piecewise linear shape functions defined on the same mesh.

However, using equal-order interpolation for the velocity and the pressure can lead to spurious solutions as it leads to a violation of the Ladyzhenskaya–Babuška–Brezzi (LBB) condition [8]. When not satisfied, the discrete system may admit oscillatory or unstable pressure fields [9]. Therefore, although equal-order interpolation is appealing due to its simplicity and ease of implementation, it requires additional stabilization techniques to recover numerical stability and ensure reliable results.

The Pressure-Stabilizing Petrov–Galerkin (PSPG) method is a technique used in order to circumvent the LBB condition by injecting in the continuity equation a stabilization term using the residual of the momentum equation [2, 8]. Following [7] yields this system:

$$\mathbf{M}\frac{D\mathbf{v}}{Dt} + \mathbf{K}\mathbf{v} - \mathbf{D}^{\mathrm{T}}\mathbf{p} = \mathbf{f} \tag{2.5}$$

$$\mathbf{C}\frac{D\mathbf{v}}{Dt} - \mathbf{D}\mathbf{v} + \mathbf{L}_\tau\mathbf{p} = \mathbf{h} \tag{2.6}$$

where the vector $\mathbf{h}$, as well as the matrices $\mathbf{C}$ and $\mathbf{L}_\tau$ are stabilization terms which come from the PSPG stabilization.

2.2 Time integration

Although there exist multiple time integration schemes, the backward Euler method is chosen in this study for its simplicity. The material derivative of the velocity is discretized using a first-order finite difference. Denoting by n the reference configuration and by $n+1$ the current configuration, the material derivate is discretized as

$$\frac{D\mathbf{v}}{Dt} = \frac{\mathbf{v}^{n+1} - \mathbf{v}^n}{\Delta t}, \tag{2.7}$$

where Δt is the time step, and $\mathbf{v}^n$ and $\mathbf{v}^{n+1}$ are the velocity vectors at time t^n and t^{n+1} respectively, with $t^{n+1} = t^n + \Delta t$. Injecting this time integration into Equations (2.5) and (2.6) and evaluating the matrices at time step $n+1$ yields the following implicit integration scheme

$$\begin{bmatrix} \frac{1}{\Delta t}\mathbf{M}^{n+1} + \mathbf{K}^{n+1} & -\mathbf{D}^{\mathrm{T},n+1} \\ \frac{1}{\Delta t}\mathbf{C}^{n+1} - \mathbf{D}^{n+1} & \mathbf{L}_\tau^{n+1} \end{bmatrix} \begin{bmatrix} \mathbf{v}^{n+1} \\ \mathbf{p}^{n+1} \end{bmatrix} = \begin{bmatrix} \mathbf{f}^{n+1} + \frac{1}{\Delta t}\mathbf{M}^{n+1}\mathbf{v}^n \\ \mathbf{h}^{n+1} + \frac{1}{\Delta t}\mathbf{C}^{n+1}\mathbf{v}^n \end{bmatrix}. \tag{2.8}$$

It is important to note from Equation 2.8 that the coefficient matrix is nonsymmetric. Nonetheless, this matrix is *structurally symmetric*. A square matrix $\mathbf{A}$ is said to be *structurally symmetric* if

$$a_{ij} \neq 0 \iff a_{ji} \neq 0, \quad \forall i, j, \tag{2.9}$$

i.e., the locations of the nonzero entries are symmetric with respect to the diagonal [10]. Moreover, one can see that the system is nonlinear, since this matrix depends on the velocities through the positions of the nodes. This is a consequence of a Lagrangian formalism.

This system will thus be solved using the Newton-Raphson algorithm, which solves nonlinear problems by iteratively solving a series of linearized problems. The iterations of this algorithm will be referred to as nonlinear iterations.
The Newton-Raphson method [11] is used to solve the nonlinear system of Equation (2.8). From the solution of the next time step $\mathbf{q}^{n+1} = [\mathbf{v}^{n+1} \quad \mathbf{p}^{n+1}]^{\mathrm{T}}$, the residual vector of Equation (2.8) is defined as

$$\mathbf{r}^{n+1} = \begin{bmatrix} \frac{1}{\Delta t}\mathbf{M}^{n+1} + \mathbf{K}^{n+1} & -\mathbf{D}^{\mathrm{T},n+1} \\ \frac{1}{\Delta t}\mathbf{C}^{n+1} - \mathbf{D}^{n+1} & \mathbf{L}_\tau^{n+1} \end{bmatrix} \mathbf{q}^{n+1} - \begin{bmatrix} \mathbf{f}^{n+1} + \frac{1}{\Delta t}\mathbf{M}^{n+1}\mathbf{v}^n \\ \mathbf{h}^{n+1} + \frac{1}{\Delta t}\mathbf{C}^{n+1}\mathbf{v}^n \end{bmatrix}. \tag{2.10}$$

The Newton-Raphson method consists in linearizing the residual vector around the current estimate of the solution and by setting it to zero:

$$\mathbf{r}^{k+1} \approx \mathbf{r}^k + \left(\frac{\partial \mathbf{r}}{\partial \mathbf{q}}\right)^{k+1} (\mathbf{q}^{k+1} - \mathbf{q}^k) = 0 \Leftrightarrow \mathbf{T}^{k+1}\Delta\mathbf{q}^{k+1} = -\mathbf{r}^k \tag{2.11}$$

where $\mathbf{T}^{k+1} = \left(\frac{\partial \mathbf{r}}{\partial \mathbf{q}}\right)^{k+1}$ is the tangent matrix and $\Delta\mathbf{q}^{k+1} = \mathbf{q}^{k+1} - \mathbf{q}^k$ is the increment of the solution.

The linear system of Equation (2.11) is solved for the increment $\Delta\mathbf{q}^{k+1}$, which is used to update the solution at each iteration:

$$\mathbf{q}^{k+1} = \mathbf{q}^k + \Delta\mathbf{q}^{k+1}. \tag{2.12}$$

Steps (2.11) and (2.12) are repeated until a prescribed convergence criterion is satisfied. Deriving Equation 2.8 with respect to $\mathbf{q}$ yields the tangent matrix, which is by construction nonsymmetric but structurally symmetric.
In this work, convergence is evaluated based on the residual vector $\mathbf{r}^k$ of Equation (2.11). The residual measures how far the left hand-side of an equation is from the right hand-side for a given approximate solution. The residual consists of two parts $\mathbf{r} = [\mathbf{r}_v \quad \mathbf{r}_p]$ to account separately for the

velocity and pressure components of the residual vector.

Moreover, the norms of each part are normalized with respect to the part of the residual that corresponds to the external loadings $\mathbf{f}_v$ or $\mathbf{f}_p$. The total normalized residual is then computed as

$$\varepsilon = \max\left(\frac{\|\mathbf{r}_v\|}{\|\mathbf{f}_v\|}, \frac{\|\mathbf{r}_p\|}{\|\mathbf{f}_p\|}\right)$$

and convergence is reached once

$$\varepsilon \leq \varepsilon_{\min} \tag{2.13}$$

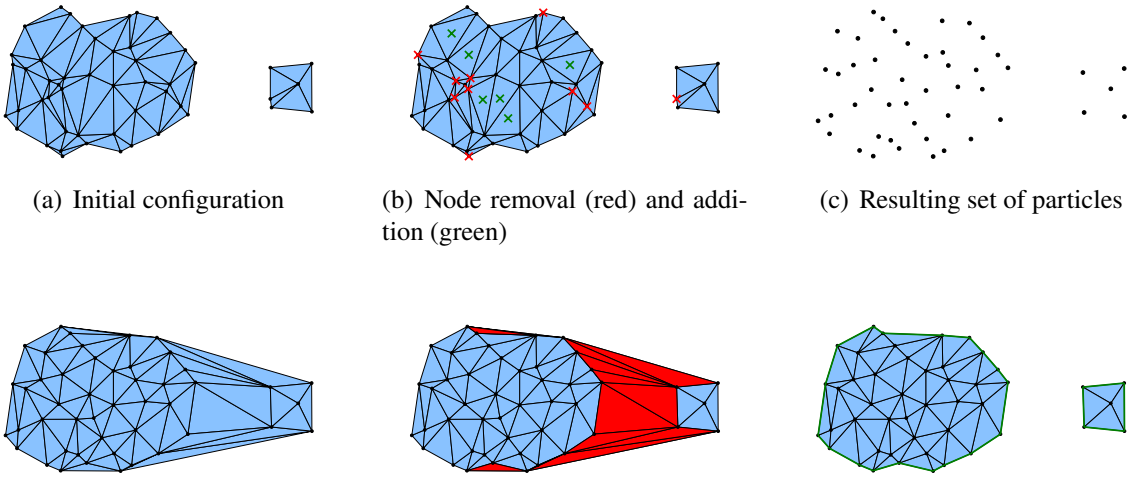
where $\varepsilon_{\min}$ is set by the user.

2.3 Remeshing

The PFEM distinguishes itself from classical FEM through a remeshing procedure applied at regular intervals, which relies primarily on three key algorithms: [1]

1. Particles are added or removed in order to maintain a uniform nodal density throughout the whole simulation.
2. Regular elements are generated from the current set of particles via a Delaunay tetrahedralization.
3. Excessively distorted tetrahedrons are suppressed from the mesh using the α-shape method.

The remeshing procedure in 2D is illustrated in 2.1 adapted from Cerquaglia [1]:



(a) Initial configuration

(b) Node removal (red) and addition (green)

(c) Resulting set of particles

(d) Delaunay tetrahedralization

(e) Detection of distorted triangles

(f) Final configuration with boundary identification

Figure 2.1: PFEM remeshing procedure adapted from Cerquaglia [1].

First, particles are added or removed to ensure a uniform nodal density. Then, the Delaunay triangulation is performed before detecting excessively distorted triangles and deleting them. This combination allows boundary detection and the modeling of free-surface flow. The final remeshed configuration is then obtained.

By considering each triangle e as being part of a circumcircle of radius r_e and of minimal side length h_e, one can define a measure of the aspect ratio of the triangle α_e by

$$\alpha_e = \frac{r_e}{h} \tag{2.14}$$

where h is the average of all h_e, with high values of α_e characterizing excessively distorted elements. The α-shape method consists therefore in eliminating all elements for which α_e is superior to a threshold α.

Concerning particle deletion, if two particles are too close to each other, one of them is removed if:

$$d < \gamma h, \tag{2.15}$$

where d is the distance between the two particles and γ is a user-defined parameter. Similarly, if an element has an area A_e that is too large, a new particle is added at its centroid if:

$$A_e > \omega h^2, \tag{2.16}$$

where ω is another user-defined parameter.

2.4 Implementation

The PFEM method has been implemented by the LTAS-MN2L lab of the University of Liège. The present master's thesis relies on the in-house C++ software for the subsequent analyses of the linear coefficient matrix. Figure 2.2 summarizes the resolution algorithm being used to solve one complete simulation. The PFEM3D software is complex and the flowchart represents only the features used in this work.

First, the physical and numerical parameters of the simulation are loaded, as well as the initial mesh which was generated using Gmsh [12].

Next, the time integration begins, where in each iteration the nonlinear and linear solvers are set up in order to solve the nonlinear system of Equations (2.5) and (2.6). Concerning the linear systems, they are solved using one of the many interfaced algorithms available, such as LU or GMRES. This master's thesis will focus on the study and comparison of the different linear solvers available.

Once the nonlinear system is solved, the entire process is repeated for the next time step until the solution has been fully obtained. Frequent remeshing of the problem is done throughout the simulation and the Delaunay tetrahedralization is performed.

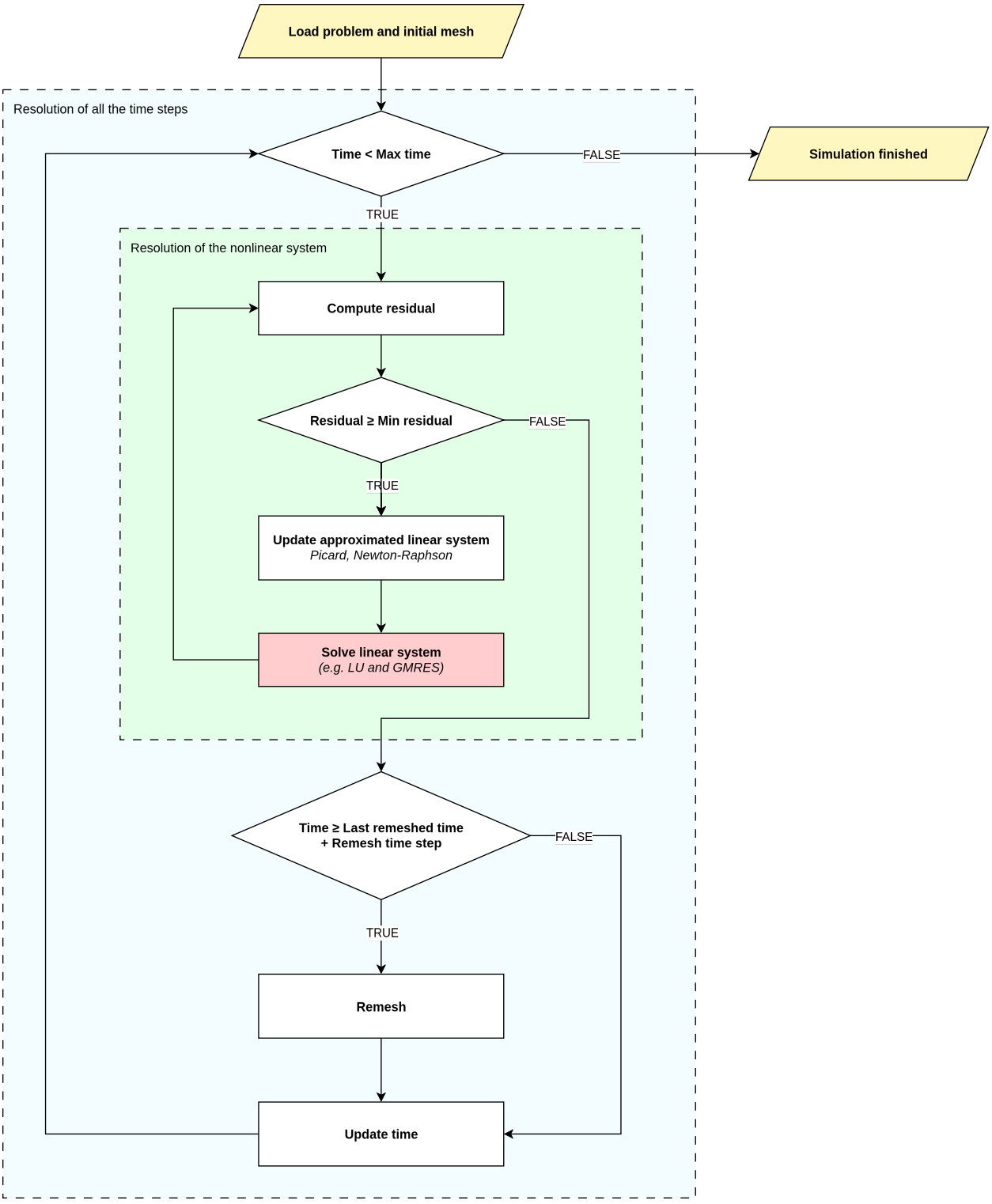


Figure 2.2: Flowchart of the PFEM algorithm.

2.5 Benchmark

The study and comparison of linear solvers will be carried out using the dam break example initially presented by Koshizuka [13] and generalized in 3D by Meduri *et al.* [14]. A water column is initially at rest as shown in Figure 2.3, and the algorithm described in Section 2.4 is applied with the numerical and physical properties listed in Table 2.1.

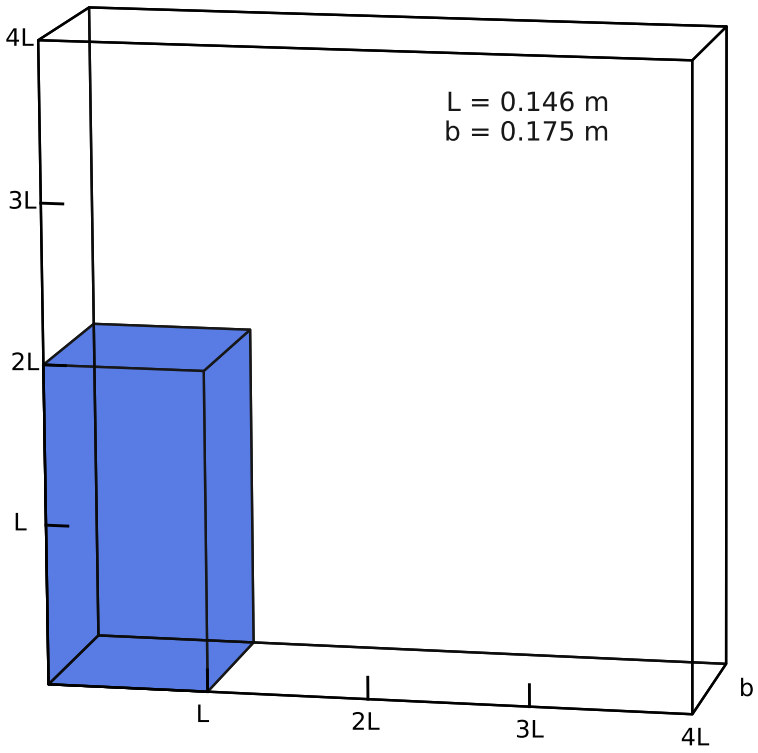


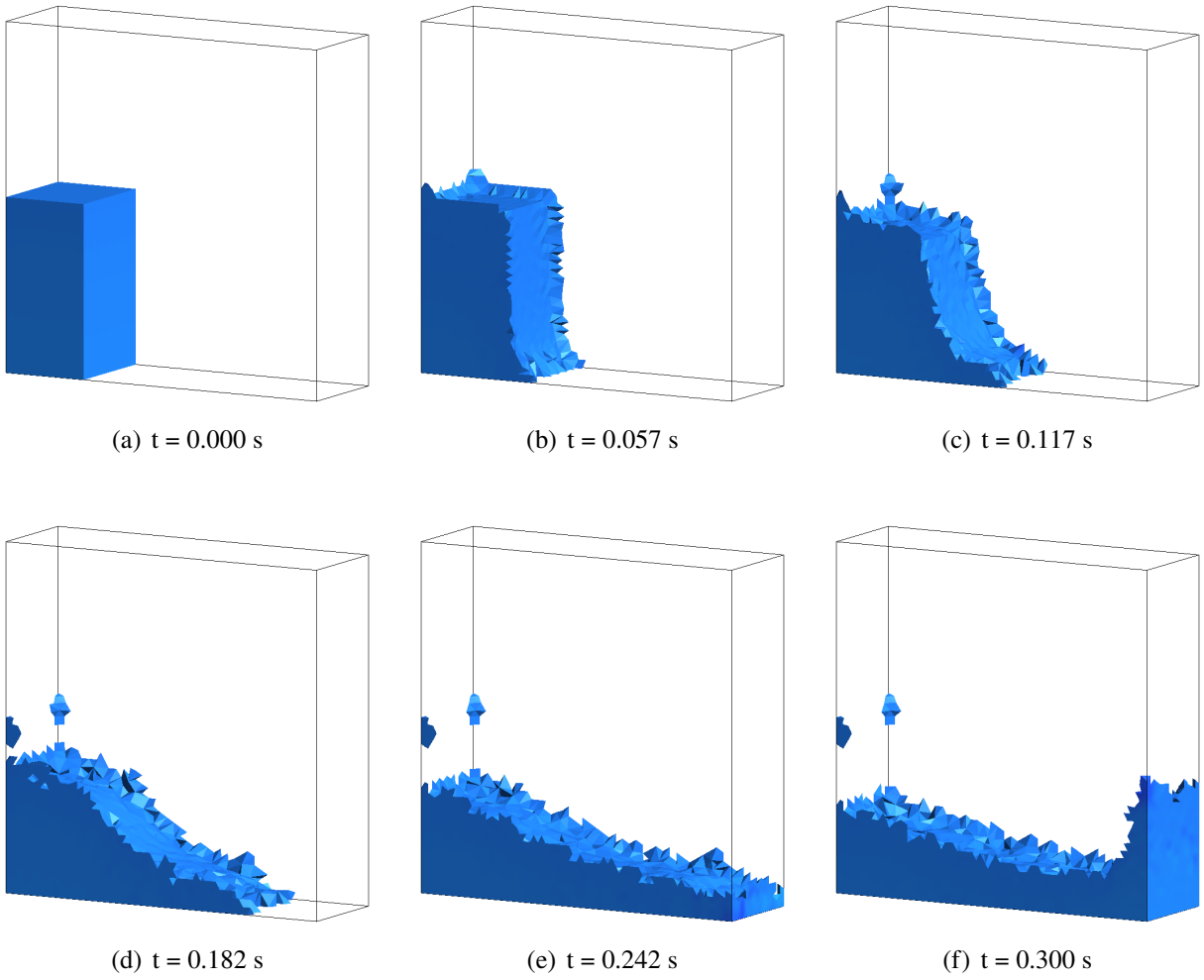
Figure 2.3: Geometry of the 3D dam break.

Property	**Value**
Element size (h_{char})	0.0146 m
Alpha (α)	1.2
Omega (ω)	0.7
Gamma (γ)	0.2
Time step Δt	$2.5 \cdot 10^{-4}$ s
Minimum residual ε_{min}	$1 \cdot 10^{-6}$

Table 2.1: Key simulation properties for the 3D dam break Koshizuka.

In the simulations carried out in this work, remeshing of the domain is performed at each time step. To illustrate this example in reasonable time, a very coarse mesh has been chosen by specifying a

large characteristic length. The motion of the fluid at different time steps is shown in Figure 2.4.



(a) t = 0.000 s (b) t = 0.057 s (c) t = 0.117 s

(d) t = 0.182 s (e) t = 0.242 s (f) t = 0.300 s

Figure 2.4: Evolution of the fluid domain over time from $t = 0$ s to $t = 0.3$ s.

Using Gmsh [12], the resulting mesh admits initially 7,138 nodes and 11,409 elements. Since there are 4 degrees of freedom per node (u, v, w and p), the linearized system of Equation (2.8) has $7,138 \times 4 = 28,552$ unknowns. Comparisons of the different sparse linear solvers will be performed with different mesh sizes.
To illustrate the properties of the system, the coefficient matrix computed at the first iteration of the first time step of the 3D dam break example has been extracted. This matrix has been reordered to group each velocity component and the pressures separately, i.e., the associated vector of nodal unknowns is $[u_1 \dots u_n v_1 \dots v_n w_1 \dots w_n p_1 \dots p_n]^T$.
The initial sparsity of the linear system matrix is therefore presented in Figure 2.5.

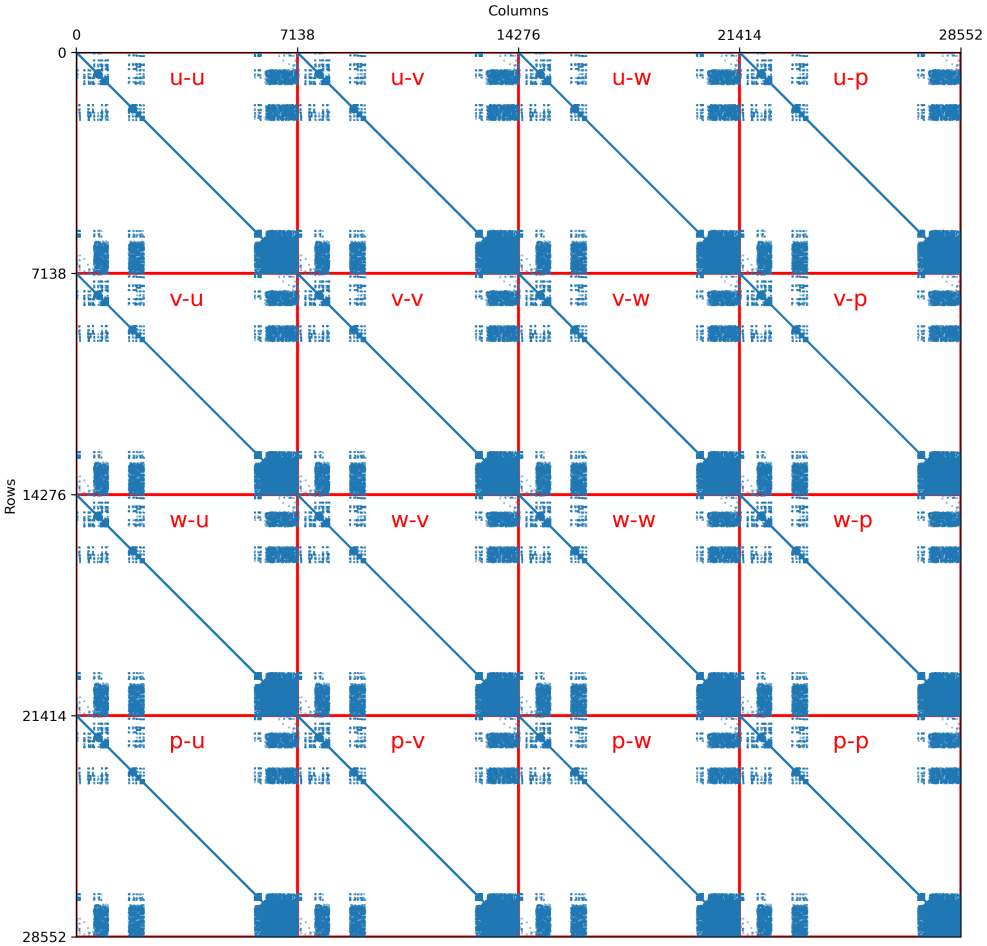


Figure 2.5: Sparsity representation of the linear system matrix at the first iteration of the nonlinear algorithm at $t = 0$. It is divided in 16 blocks which couple the different types of degrees of freedom. Blue dots represent non-zeros entries.

The coefficient matrix was obtained through assembling elementary matrices of each tetrahedron. In the above image, blue dots correspond to the location of a nonzero stored element in the sparse matrix. One can easily recognize from Equation (2.8) the coupling between the different velocities u, v and w through the matrices $\mathbf{M}$ and $\mathbf{K}$, while the nodal pressures p are coupled through $\mathbf{L}_\tau$. The velocities and pressures are coupled with $\mathbf{D}^{\mathrm{T}}$ in the top-right of the matrix and $\mathbf{D}$ and $\mathbf{C}$ in the bottom-left.

Regardless of the nonlinear algorithm used, the sparsity structure as well as the coupling between degrees of freedom are exactly the same for the linear system matrix. In addition, it is important to remark these following properties:

1. $\mathbf{A}$ is a structurally symmetric matrix.
2. All the sub-matrices $\mathbf{A}_{11}$ to $\mathbf{A}_{44}$ are structurally symmetric as well. Moreover, outside of the pressure-velocity coupling matrices, the sub-matrices are all symmetric in value.
3. The sparsity of all the sub-matrices is exactly the same.

According to the construction of Equations (2.8), these properties hold true regardless of the time step or the iteration count [7].

Chapter 3

Presentation of Sparse Linear Solvers

In this Chapter, different linear solvers are presented in order to solve the approximated $n \times n$ linear system of Equations (2.8), which will be written under the following form

$$\mathbf{A}\mathbf{x} = \mathbf{b}. \tag{3.1}$$

where $\mathbf{A}$ is the tangent matrix, being structurally symmetric, $\mathbf{x}$ is the increment of the solution, and $\mathbf{b}$ is the opposite of the residual vector.

This Section presents different linear solvers for solving the Newton-Raphson linear system without further consideration of the nonlinear context.

Additionally, this work will refer to *system properties* as properties concerning the coefficient matrix A. For example, a structurally symmetric linear system will refer to a linear system where A is a structurally symmetric matrix.

3.1 Overview

Linear solvers can be categorized into two classes of methods called direct methods and iterative methods [10, 15].

Direct methods transform the linear system into an equivalent one that is easier to solve. The exact solution is then directly obtained after a finite number of operations. Very large systems require large memory to store the transformed system [10, 15].

Iterative methods are methods in which an initial guess of the solution is refined until the desired accuracy is achieved. Contrary to direct methods, they require little memory and the number of steps is unknown, leading to performance that is highly dependent on the problem [10].

This section introduces three preliminary concepts common to both classes of solvers. Subsection 3.1.1 presents the condition number as a measure of the sensitivity of the linear system to perturbations. Subsection 3.1.2 then describes sparse matrix storage formats, covering the Compressed Sparse Row and Compressed Sparse Column representations. Finally, Subsection 3.1.3 introduces the Basic Linear Algebra Subprograms (BLAS) and their role in sparse linear algebra.

3.1.1 Condition number

This section presents the *condition number* and is adapted from the derivations of the book Matrix Computations [16].

When solving linear systems, it is of interest to know the sensitivity of the solution $\mathbf{x}$ with respect to $\mathbf{A}$ and $\mathbf{b}$. Indeed, round-off errors coming from machine limited precision can change the actual system one is trying to solve. Denoting by $\varepsilon\mathbf{F}$ and $\varepsilon\mathbf{f}$ small perturbations of $\mathbf{A}$ and $\mathbf{b}$, the linear system of Equation 3.1 is rewritten as [16]

$$(\mathbf{A} + \varepsilon\mathbf{F})\mathbf{x}(\varepsilon) = \mathbf{b} + \varepsilon\mathbf{f}, \quad \mathbf{x}(0) = \mathbf{x}, \tag{3.2}$$

where $\mathbf{F} \in \mathbb{R}^{n\times n}$ and $\mathbf{f} \in \mathbb{R}^n$. Assuming $\mathbf{A}$ is nonsingular, $\mathbf{x}(\varepsilon)$ is differentiable and $\dot{\mathbf{x}}$ can be obtained through differentiating Equation (3.2):

$$\dot{\mathbf{x}}(0) = \mathbf{A}^{-1}(\mathbf{f} - \mathbf{F}\mathbf{x}). \tag{3.3}$$

Inserting $\dot{\mathbf{x}}$ expression into the Taylor series expansion for $\mathbf{x}(\varepsilon)$ gives

$$\mathbf{x}(\varepsilon) = \mathbf{x} + \varepsilon\dot{\mathbf{x}}(0) + O(\varepsilon^2). \tag{3.4}$$

Using any vector norm and consistent matrix norm, the following condition is obtained:

$$\frac{\|\mathbf{x}(\varepsilon) - \mathbf{x}\|}{\|\mathbf{x}\|} \leq |\varepsilon|\,\|\mathbf{A}\|\|\mathbf{A}^{-1}\|\left(\frac{\|\mathbf{F}\|}{\|\mathbf{A}\|} + \frac{\|\mathbf{f}\|}{\|\mathbf{b}\|}\right) + O(\varepsilon^2). \tag{3.5}$$

Therefore, the relative error of $\mathbf{x}(\varepsilon)$ is bounded by the relative errors of $\delta\mathbf{A} = \mathbf{A} + \mathbf{F}$ and $\delta\mathbf{b} = \mathbf{b} + \mathbf{f}$, multiplied by

$$\kappa(\mathbf{A}) = \|\mathbf{A}\|\,\|\mathbf{A}^{-1}\| > 1 \quad \forall\mathbf{A} \in \mathbb{R}^{n\times n}. \tag{3.6}$$

This matrix property is called the *condition number*. It measures if small changes of $\mathbf{A}$ or $\mathbf{b}$ can lead to large changes in the solution $\mathbf{x}$. Systems with a low condition number have a solution which is not very sensitive to $\delta\mathbf{A}$ and $\delta\mathbf{b}$, those systems are said to be *well-conditioned*. Similarly, systems with a high condition number have a solution which is very sensitive and are said to be *ill-conditioned* [16].

3.1.2 Matrix storage schemes

Dense matrix representation consists in storing the values of all the elements of a matrix in memory. Although simple to use, such matrix storage scheme leads to a very high memory usage [17]. Consider the matrix in Figure 2.5, storing all the numerical values of this matrix leads to $28,552 \times 28,552 = 813,504,484$ values stored. With double-precision floating-point numbers, this corresponds to 813,504,484 × 8 bytes = 6.51 GB of RAM. This is indeed not practical for very large sparse matrices systems, especially since most values are zeros.

Therefore, a *sparse matrix* representation is used instead where all the zero values are not stored. A *sparse matrix* is a matrix in which most of the elements are null like the one shown in Figure

2.5. Exploiting this sparsity by not storing the zeros reduces considerably the memory usage while enabling more efficient algorithms for all related linear algebra operations [17].

There exist multiple sparse matrix representations for scientific computing software with probably the most common one being the Compressed Sparse Row format (CSR) [18]. This format uses three one-dimensional arrays, one to store the non-zero *values*, one to store the the different *column indices* which correspond to the values, and finally, one which stores the starting index in the values array of each row, called the *row pointers*. This format is illustrated with the following matrix associated with Table 3.1:

$$\mathbf{A} = \begin{bmatrix} 4 & & & & 9 \\ & 5 & & 2 & \\ & & 6 & & \\ 7 & & & 8 & \\ & 1 & & & 3 \end{bmatrix} \tag{3.7}$$

Array	**Values**
Values	[4, 9, 5, 2, 6, 7, 8, 1, 3]
Column indices	[0, 4, 1, 3, 2, 0, 3, 1, 4]
Row pointers	[0, 2, 4, 5, 7, 9]

Table 3.1: CSR representation of a sparse matrix with diagonal and off-diagonal non-zero elements.

Another common representation is the Compressed Sparse Column (CSC) format, which is very similar to the CSR format but organizes the matrix data by columns instead of rows.

The CSC format also uses three one-dimensional arrays:

- **values:** stores the non-zero values of the matrix, column by column.
- **row indices:** stores the row indices corresponding to each value.
- **column pointers:** stores the starting index in the values array of each column.

Using the same matrix as before, the CSC representation of this matrix is as follows:

Array	**Values**
`values`	[4, 7, 5, 1, 6, 2, 8, 9, 3]
`row_indices`	[0, 3, 1, 4, 2, 1, 3, 0, 4]
`column_pointer`	[0, 2, 4, 5, 7, 9]

Table 3.2: CSC representation of the same sparse matrix, organized by columns instead of rows.

3.1.3 BLAS

The Basic Linear Algebra Subprograms (BLAS) is a set of low-level routines that perform common linear algebra operations such as vector addition, dot products, and matrix multiplication [19]. Originally in the 1970s, BLAS has become a standard for building high-performance numerical software. Modern implementations of BLAS are highly optimized for specific hardware architectures and are available on most platforms [20].

BLAS routines are organized into three levels based on the type of operations they perform:

- **Level 1 BLAS:** These routines operate on vectors and perform operations such as scalar multiplication, vector addition, or dot products. They have a computational complexity of $O(n)$ and are memory-bound due to their low arithmetic intensity.
- **Level 2 BLAS:** These routines handle matrix-vector operations such as matrix-vector multiplication and triangular solves. Their complexity is $O(n^2)$, and while they offer better computational density than Level 1, they still suffer from bandwidth limitations on large systems.
- **Level 3 BLAS:** These are the most computationally intensive and performance-critical routines. They perform matrix-matrix operations, with a complexity of $O(n^3)$, and benefit greatly from optimized cache usage and parallel execution. Examples include dense matrix multiplication and LU or QR factorization kernels.

Level 3 routines in particular are highly optimized to take full advantage of CPU caches and vectorized operations. However, it is important to note that BLAS is inherently designed for *dense* linear algebra. Only limited sparse matrix operations are covered by BLAS. [19]

3.2 Direct sparse solvers

This section presents direct sparse solvers. Subsection 3.2.1 introduces the resolution of triangular systems, followed by the Gaussian elimination algorithm in Subsection 3.2.2. Then, the LU and QR factorizations are described in Subsections 3.2.3 and 3.2.4 as two common direct methods for solving sparse linear systems.

3.2.1 Triangular systems

A triangular system is a linear system for which $\mathbf{A}$ of Equation 3.1 is either an upper triangular matrix $\mathbf{U}$ or a lower triangular matrix $\mathbf{L}$, which are defined as

$$\mathbf{L} \in \mathbb{R}^{n \times n},\ l_{ij} = 0 \quad \forall i < j,$$

$$\mathbf{U} \in \mathbb{R}^{n \times n},\ u_{ij} = 0 \quad \forall i > j.$$

Solving such systems is a straightforward and efficient process [21]. Indeed, in a lower triangular system, component i of the solution vector $\mathbf{x}$ is obtained from the i-th equation, which depends only on the $k < i$ components of $\mathbf{x}$ which were already obtained. This process is known as "forward

substitution" since the already obtained components are substituted in the next equation. Similarly, in an upper triangular system, component i is obtained from the i-th equation, which depends only on the $m > i$ already obtained components. This process is called "backward substitution" as the system is solved by substituting the components in the previous equation.

It is worth noting that both forward and backward substitutions cannot be parallelized due to data dependency since each row in $\mathbf{L}$ ($\mathbf{U}$) depends on the upper (lower) solutions.

3.2.2 Gaussian elimination

The following presentation is adapted from the books Numerical Recipes [17] and Matrix Computations [16].

Gaussian elimination is an algorithm used to transform a linear system into an equivalent triangular system by applying linear combinations of the rows. For each column j of $\mathbf{A}$, linear combinations are applied such that all the elements below the diagonal a_{kj} (with $k \geq j+1$) become null, this is called an elimination step. The elimination step at column k requires dividing all the rows $k \geq j+1$ by the element a_{jj}, which is called the pivot. This is illustrated in the following example:

$$\begin{bmatrix} 2 & 1 & -1 \\ -3 & -1 & 2 \\ -2 & 1 & 2 \end{bmatrix} \xrightarrow[\substack{\mathbf{R}_3 \to \mathbf{R}_3 + \mathbf{R}_1 \\ \text{Elimination step 1}}]{\mathbf{R}_2 \to \mathbf{R}_2 + \frac{3}{2}\mathbf{R}_1} \begin{bmatrix} 2 & 1 & -1 \\ 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & 2 & 1 \end{bmatrix} \xrightarrow[\text{Elimination step 2}]{\mathbf{R}_3 \to \mathbf{R}_3 - 4\mathbf{R}_2} \begin{bmatrix} 2 & 1 & -1 \\ 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & -1 \end{bmatrix}$$

Following this process, the solution can then be easily obtained through backward or forward substitution. Evidently, the linear operations performed on the matrix must also be applied on the right-hand side $\mathbf{b}$ to obtain an equivalent system.

While simple conceptually, this method may fail or become numerically unstable if a pivot is too close to zero. Indeed, dividing the row k by a near zero value will lead to round-off errors since in floating-point arithmetic, every number is stored with a finite precision. These errors can get amplified while eliminating the succeeding rows. This issue is called round-off amplification.

To overcome this, the pivot can be chosen to be different than simply a_{jj} using pivoting strategies, which consist in swapping rows and/or columns to select pivots with better numerical properties. Different pivoting strategies are introduced. *No pivoting*, the pivot is taken as a_{jj} without modification. *Partial pivoting*, rows are interchanged so that the largest element in absolute value in column i is used as the pivot. *Complete pivoting*, both rows and columns can be interchanged to select the largest available element in the matrix.

Additionally, a pivoting strategy can be said to be *dynamic* or *static*. In *dynamic pivoting*, each pivot is chosen adaptively during the elimination depending on numerical values, allowing numerical stability. In *static pivoting*, all the pivots are chosen prior to the triangulation of A, allowing the rows and/or columns to swap to be predicted without looking at numerical values.

3.2.3 LU factorization

The LU factorization is a direct method used to solve nonsymmetric linear systems by factorizing **A** into **L** and **U** where **L** is a lower triangular matrix and **U** an upper triangular matrix. It relies on the Gaussian elimination in order to obtain both **L** and **U**.

The matrix **L** is usually considered to have an identity diagonal, there exists therefore only one LU factorization respecting this constraint [17]. The system of equations to solve become:

$$\mathbf{Ly} = \mathbf{b} \tag{3.8}$$

$$\mathbf{Ux} = \mathbf{y} \tag{3.9}$$

These equations are solved sequentially with forward and then backward substitution.

Since this factorization relies on the Gaussian elimination, this method can be unstable for ill-conditioned systems if the pivots are too close to zero, requiring some pivoting. The often used method for this purpose is partial pivoting, which is more efficient than complete pivoting since it does not require the examination of the complete matrix in order to select a pivot [17].

Concerning its sparse version, the elimination steps of the LU factorization can be very efficiently parallelized [15, 22].

3.2.4 QR factorization

This method factorizes the matrix **A** into $\mathbf{A} = \mathbf{QR}$ where **Q** is orthogonal (i.e., $\mathbf{Q}^T\mathbf{Q} = \mathbf{0}$) and **R** is an upper triangular matrix. It belongs to the subclass of methods based on orthogonal factorizations. Once **Q** and **R** have been determined, the equation to solve is simply

$$\mathbf{Rx} = \mathbf{Q}^T\mathbf{b} \tag{3.10}$$

where the solution is obtained similarly as with the LU method using backward substitution. The interest of **Q** being orthogonal allows to easily find its inverse to obtain the triangular system.
There exist multiple methods to find a suitable orthogonal matrix such as the *Givens rotation method*, the *Gram-Schmidt procedure* and the *Householder reflections* [16].

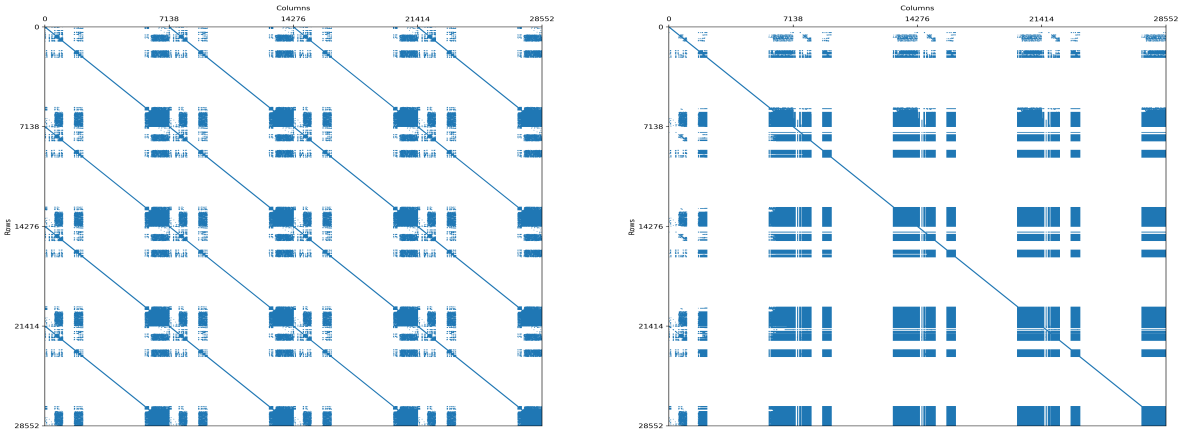
Compared to the LU decomposition, the QR method presents two advantages. Firstly, it can be used to solve problems with more equations than unknowns and is therefore used for least-squares problems. Secondly, it is significantly more stable than LU for ill-conditioned matrices, and it therefore does not require pivoting for stability. [21, 22]

Nevertheless, the QR decomposition generally requires about twice as many operations as LU decomposition and cannot be efficiently parallelized in its sparse version [22]. It is therefore rarely used in practice when LU is available [17]. Hence, this method should underperform LU in computation time for the square and structurally symmetric system of Equation (2.8).

3.2.5 Ordering

The factorization of the tangent matrix can lead to factors that are much less sparse than the original matrix, leading to the storage of many more elements. Consequently, more operations are also required in order to solve the system. This phenomenon is called "fill-in" and it is of interest to minimize it for the sake of performance [10, 15, 21].

To illustrate the fill-in problem, let's consider the LU decomposition of the matrix $\mathbf{A}$ shown in Figure 2.5 using `splu`[1] function defined in `scipy.sparse.linalg` in Python. As illustrated in Figure 3.1, the factorization leads to a much higher memory usage, with 56,431,124 non-zeros stored in $\mathbf{L}+\mathbf{U}$ when only 596,320 were required in the original matrix $\mathbf{A}$.



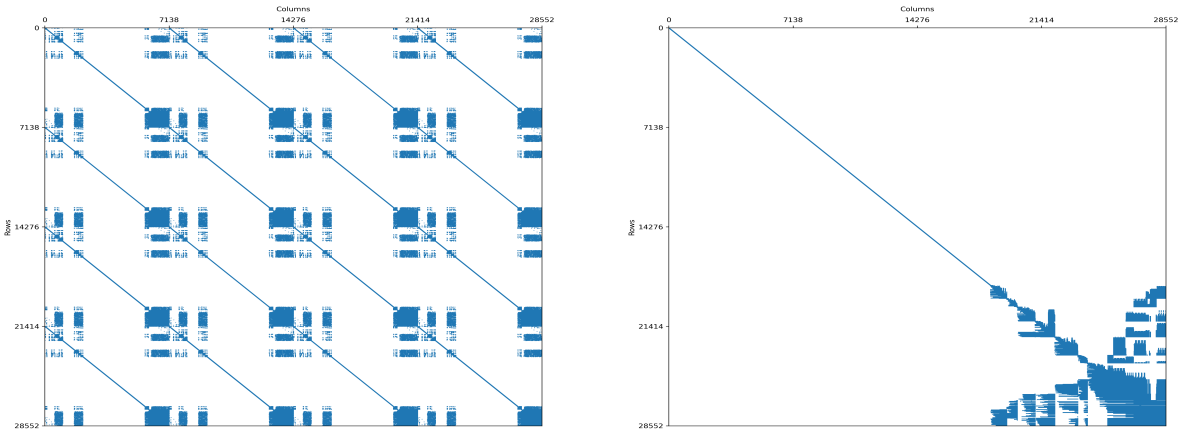
(a) $\mathbf{A}$. (b) $\mathbf{L}+\mathbf{U}$.

Figure 3.1: Sparsity pattern of matrix $\mathbf{L}+\mathbf{U}$ after factorization of $\mathbf{A}$ into $\mathbf{LU}$ without reordering. $\mathbf{L}$ and $\mathbf{U}$ are summed to represent visually the increase of nonzeros.

Avoiding fill-in and unstable substitution can be achieved by pivoting the rows and the columns of the linear system. Denoting the new system matrix by $\mathbf{PAP}^{\mathrm{T}} = \mathbf{LU}$ where $\mathbf{P}$ is the permutation matrix, the first phase of a direct sparse solver is the *ordering* in which an appropriate permutation matrix is found [15].

However, finding $\mathbf{P}$ such that the factorization induces minimum fill-in is computationally very expensive and therefore the ordering algorithms are *heuristics* [15]. A *heuristic* is a computationally efficient algorithm used to find approximate solutions rather than the actual optimum in an optimization problem [24].

[1] The `splu` function is an interface to the LU algorithm developed in the library SuperLU [23].



(a) **A**. (b) **L** + **U**.

Figure 3.2: Sparsity pattern of matrix **L** + **U** after factorization of **A** into **LU** using COLAMD algorithm for reordering. **L** and **U** are summed to represent visually the increase of nonzeros.

As shown in Figure 3.2, the choice of pivoting can lead to a drastically different non-zero pattern. Indeed, the ordering algorithm used led to a matrix **L** + **U** with 9,203,469 non-zeros, which is 6 times less than the matrix in Figure 3.1.

The ordering using COLAMD algorithm induces less fill-in and is therefore preferred for a *sequential* linear resolution. For efficient *parallel* resolutions, **P** is determined in order to avoid instability, avoid fill-in, and allow parallel numerical factorization thanks to a balanced elimination tree [10, 15, 21].

3.2.6 Elimination trees

Contrary to dense direct solvers, sparse direct solvers factorize the matrix **A** into other sparse matrices for which the memory must be allocated before proceeding to the factorization. This memory allocation step is known as the *symbolic factorization* step in a sparse direct solver. Symbolic factorization not only allocates the structure to store the factors, but also defines the dependencies between the elimination steps by building an *elimination tree* [10, 15, 21].

An elimination tree is a directed rooted tree which defines the order of the elimination steps to be performed and where each node corresponds to one column of the matrix with fill-in. Children of a node j denote the columns which require to be eliminated prior to column j.

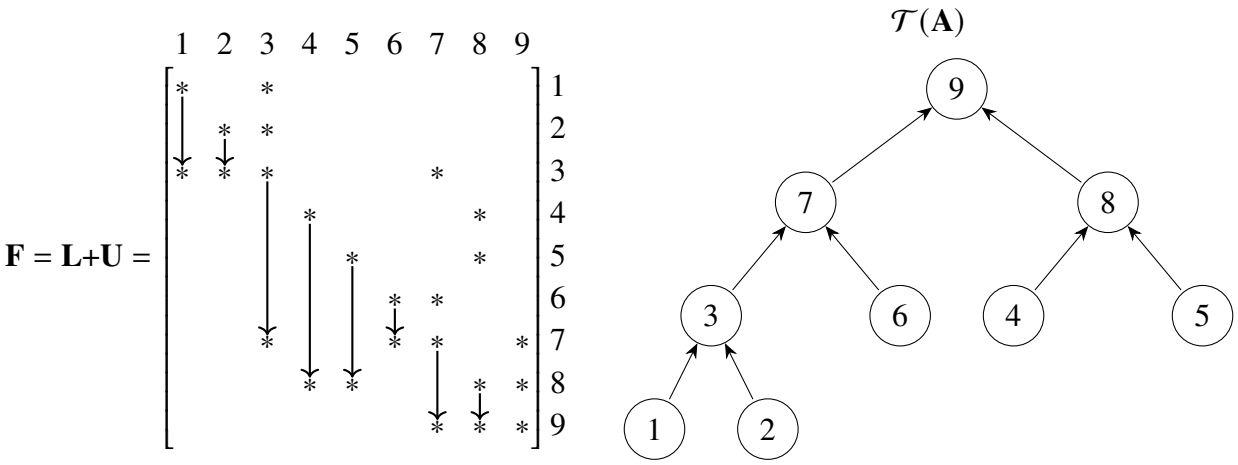


Figure 3.3: Matrix with fill-in $\mathbf{F} = \mathbf{L} + \mathbf{U}$ with the elimination tree $\mathcal{T}(\mathbf{A})$ resulting from the factorization of a matrix $\mathbf{A}$. Arrows represent data dependencies between the different elimination steps required to factorize $\mathbf{A}$ for both the matrix with fill-in $\mathbf{F}$ and the elimination tree: from the elimination of children of node j, elimination step at column j can be performed.

Consider the matrix with fill-in $\mathbf{F} = \mathbf{L} + \mathbf{U}$ in Figure 3.3. Performing the elimination step at column 1 will affect the third row and thus column 3. Since column 3 requires first column 1 to be eliminated, the node 1 is a child of node 3. Similarly, the elimination step at column 2 affects column 3, making node 2 a child of node 3. The same applies for all the elimination steps to build the elimination tree $\mathcal{T}(\mathbf{A}, 9)$.

Once the elimination tree of a matrix has been determined, the *numerical factorization* step can start to compute the actual values of the sparse structure. This step is the most time consuming task in a sparse direct solver and is also the one that can benefit the most from parallelization [15]. Indeed, since the elimination tree represents the data dependency between the elimination steps, the elimination tree can be used to factorize independent subtrees of $\mathcal{T}(\mathbf{A})$ in parallel, like in Figure 3.3 for instance with $\mathcal{T}(\mathbf{A}, 7)$ and $\mathcal{T}(\mathbf{A}, 8)$ which are the subtrees of $\mathcal{T}(\mathbf{A})$ with respective parents 7 and 8. This is known as large-grain parallelism [15, 25].

In a sparse direct solver, there are three levels of parallelization [15, 25]. *Fine-grain parallelism*, in which each task is a single floating point operation. *Medium-grain parallelism* means treating each column operation as a separate task that can be done in parallel. *Large-grain parallelism* refers to each task being the factorization of an independent subtree of the elimination tree. For dense direct solvers, BLAS-3 allows fine-grain parallelism when factorizing a matrix. However, for direct sparse solvers based on Gaussian elimination, a combination of both large grain and fine-grain parallelism is used [15, 26].

3.2.7 Supernodal methods

As the factorization proceeds, the columns of the factorized matrices generally become denser as was explained in Subsection 3.2.5. This density can be exploited in order to enhance significantly

the numerical factorization, both in computation time and in memory requirements [10]. This would enable the use of high-performance BLAS-3 routines. For this purpose, the notion of *supernodes* is useful. A supernode is a set of contiguous columns that share the same sparsity structure below the diagonal, for which it is possible to reassemble them into a dense submatrix [10].
With the usage of supernodes, instead of eliminating one column separately at each elimination step, it is possible to eliminate multiple columns by aggregating them into dense submatrices. This allows floating point tasks to be executed in parallel, which reduces the level of parallel granularity one could obtain by simply considering large-grain parallelism with the elimination trees [10].

If the linear system is very sparse and that its factorization incurs very little fill-in, then identifying supernodes might not offer a computation speed advantage. In this case, some algorithms may merge the small supernodes together that have a similar sparsity structure. This results in an increase of fill-in for the sake of efficient dense operations. Such merge of supernodes is known as a relaxed supernode [10].

3.2.8 Resolution steps in a sparse direct solver

Two direct solvers have been presented which are based on LU and QR factorizations. Several methods to optimize these solvers have also been discussed. The resolution of a sparse direct solver can be summarized in 4 steps, which both solvers follow [10]:

1. **Ordering:** a permutation of rows and columns of the matrix $\mathbf{A}$ is performed in order to minimize fill-in.
2. **Symbolic factorization:** determination of the nonzero structure of the factorization matrices to allocate the necessary memory for the factorization, it is in this step that the elimination tree is built.
3. **Numerical factorization:** computation of the numerical values of the factorization matrices, it is the most time-consuming task and it is parallelized on the basis of the elimination tree.
4. **Solve:** resolution of the simpler triangular linear systems through forward and/or backward substitution.

3.3 Iterative sparse solvers

Contrary to direct sparse solvers, iterative sparse solvers do not differ significantly from their dense versions since they do not suffer the problem of fill-in and refine their solutions iteratively. The difference lies only in the matrix storage and basic matrix operations. This section presents the general ideas behind the most used iterative solvers.

3.3.1 Krylov-subspace methods

Most of iterative solvers rely on projection methods, which aim to search an approximate solution $\tilde{\mathbf{x}}$ of Equation 3.1 in a simple subspace of dimensions $m \times n$ with $m < n$ [18]. This subspace is

called the *search space*, and the approximate solution must respect constraints by having a residual vector $\mathbf{r} = \mathbf{b} - \mathbf{A}\tilde{\mathbf{x}}$ which is orthogonal to another subspace called the *constraint space*. A projection method can be thus defined as

$$\text{Find } \tilde{\mathbf{x}} \in \mathbf{x}_0 + \mathcal{K}, \quad \text{such that } \mathbf{b} - \mathbf{A}\tilde{\mathbf{x}} \perp \mathcal{L}, \tag{3.11}$$

where $\mathbf{x}_0 + \mathcal{K} \subset \mathbb{R}^n$ is the search space, $\mathcal{L} \subset \mathbb{R}^n$ the constraint space and $\mathbf{x}_0$ an initial guess.

For an iterative solver, these spaces are constructed iteratively until the residual norm is lower than a specified tolerance. Krylov subspace methods represent a widely used category of projection methods, where the search space is constructed at each iteration m such that

$$\mathcal{K}_m(\mathbf{A}, \mathbf{r}_0) = \text{span}\{\mathbf{r}_0, \mathbf{A}\mathbf{r}_0, \mathbf{A}^2\mathbf{r}_0, ..., \mathbf{A}^{m-1}\mathbf{r}_0\} \tag{3.12}$$

where $\mathbf{r}_0$ is the initial residual.

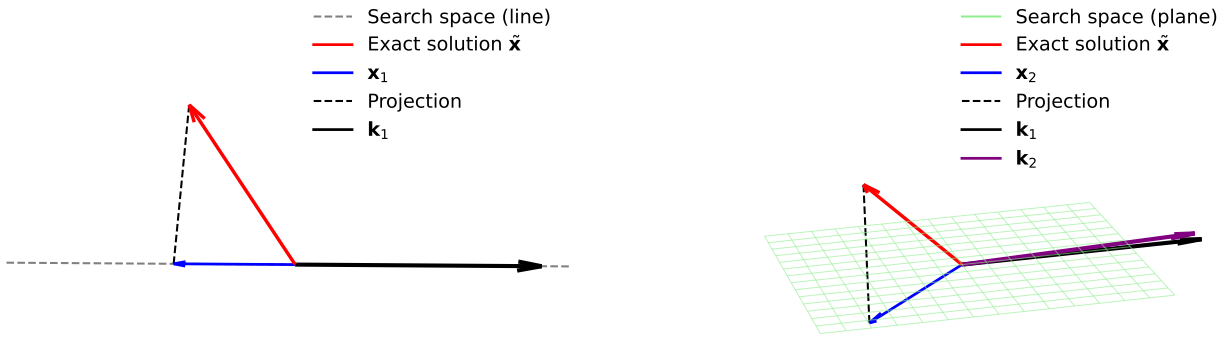


Figure 3.4: Resolution of a simple linear system of solution $\tilde{\mathbf{x}}$ using the Generalized Minimal Residual method (GMRES), which is a Krylov-subspace method. Vectors $\mathbf{k}_1 = \mathbf{r}_0$ and $\mathbf{k}_2 = \mathbf{r}_0 + \mathbf{A}\mathbf{r}_0$ generate the search space on which the approximation of $\mathbf{x}$ is obtained. The approximation corresponds to the projection of $\mathbf{x}$ onto $\mathbf{x}_0 + \mathbf{K}$.

Figure 3.4 illustrates the behavior of a Krylov-subspace method. The approximate solution in blue is found in the search-space which corresponds to a simple line at the first iteration, and a plane at the second iteration. The solution at the second iteration $\mathbf{x}_2$ is closer to the actual solution $\tilde{\mathbf{x}}$ than the solution at the first iteration $\mathbf{x}_1$.
Krylov subspace methods can differ in their constraint space $\mathcal{L}$ as well as how the Krylov subspace is constructed algorithmically or how the constraints are enforced.

The first Krylov-subspace method was the Conjugate Gradient (CG) method, which was developed in 1952 by Hestenes and Stiefel [27]. This method is only applicable to symmetric positive definite (SPD) matrices [18]. Since then, many other Krylov-subspace methods have been developed in order to solve more general problems such as the Generalized Minimum Residual Method and the Biconjugate Gradient method. These two methods are presented in the following subsections.

3.3.2 GMRES

The Generalized Minimum Residual Method (GMRES) is a Krylov-subspace method for which $\mathcal{L}_m = \mathbf{A}\mathcal{K}_m$. In this case, the orthogonality condition of Equation (3.11) is equivalent to minimizing the norm of the residual over the search space. The orthogonality condition is difficult to enforce algorithmically contrary to solving simple minimization problems. The latter is the approach adopted by GMRES. [18]

In GMRES, the Krylov subspace is not directly constructed from Equation (3.12). As one can see in Figure 3.4, the vectors obtained using the above definition do not form an orthogonal basis. This can cause numerical instabilities as well as resulting in ill-conditioned least square problems [28]. Therefore, an orthogonal basis $\mathbf{V}_{m+1}$ is constructed at each iteration m using the Arnoldi process [18]. This leads to the recurrence relation:

$$\mathbf{A}\mathbf{V}_m = \mathbf{V}_{m+1}\mathbf{H}_m \tag{3.13}$$

where $\mathbf{V}_m \in \mathbb{R}^{n\times m}$ contains the first m basis vectors, and $\mathbf{H}_m \in \mathbb{R}^{(m+1)\times m}$ is an *upper Hessenberg matrix*. That is, all entries of $\mathbf{H}_m$ below the first subdiagonal are zero.

This decomposition allows us to transform the original high-dimensional residual minimization problem into a small least squares problem of dimension m, given by:

$$\mathbf{y}_m = \text{argmin}_{\mathbf{y}\in\mathbb{R}^m} \|\beta\mathbf{e}_1 - \mathbf{H}_m\mathbf{y}\|_2 \,,$$

where $\mathbf{y}$ is the solution in the Krylov space, $\beta = \|\mathbf{r}_0\|_2$ and $\mathbf{e}_1 \in \mathbb{R}^{m+1}$ is the first canonical basis vector. Least squares problems can be solved using QR factorization as described in Subsection 3.2.4. The approximate solution is then retrieved with

$$\tilde{\mathbf{x}}_m = \mathbf{x}_0 + \mathbf{V}_m\mathbf{y}_m. \tag{3.14}$$

An advantage of the GMRES algorithm is that its convergence is guaranteed [18]. However, building the orthonormal basis of the Krylov subspace becomes increasingly expensive both in memory and in computation as iterations grow.

As a consequence, a *restarted GMRES* (GMRES(k)) can be used to solve these issues [18, 29]. This algorithm consists in discarding the already built Krylov subspace to restart it from scratch once iteration $m = k$ is reached, where k is the maximum iteration parameter. Therefore, the algorithm is restarted such that $m \leftarrow 0$ with a new initial guess $\mathbf{x}_0 \leftarrow \tilde{\mathbf{x}}_k$ and residual $\mathbf{r}_0 \leftarrow \mathbf{b} - \mathbf{A}\mathbf{x}_k$. A potential issue with GMRES(k) is that convergence is no longer guaranteed [29].

3.3.3 BiConjugate Gradient

The Biconjugate Gradient (BiCG) method is a Krylov-subspace method that was proposed for the first time by Lanczos [30]. It aims to solve nonsymmetric and indefinite linear systems by generalizing the Conjugate Gradient method [31]. What distinguishes BiCG is that it constructs

two Krylov subspaces simultaneously: one associated with the matrix $\mathbf{A}$, and another with its transpose $\mathbf{A}^{\mathrm{T}}$, given respectively by

$$\mathcal{K}_m(\mathbf{A}, \mathbf{r}_0) = \mathrm{span}\{\mathbf{r}_0, \mathbf{A}\mathbf{r}_0, \mathbf{A}^2\mathbf{r}_0, \ldots, \mathbf{A}^{m-1}\mathbf{r}_0\}, \tag{3.15}$$

$$\mathcal{K}_m(\mathbf{A}^{\mathrm{T}}, \tilde{\mathbf{r}}_0) = \mathrm{span}\{\tilde{\mathbf{r}}_0, \mathbf{A}^{\mathrm{T}}\tilde{\mathbf{r}}_0, (\mathbf{A}^{\mathrm{T}})^2\tilde{r}_0, \ldots, (\mathbf{A}^{\mathrm{T}})^{m-1}\tilde{\mathbf{r}}_0\}, \tag{3.16}$$

where $\mathbf{r}_0 = \mathbf{b} - \mathbf{A}\mathbf{x}_0$ is the initial residual, and $\tilde{\mathbf{r}}_0$ is a shadow residual such that $\tilde{\mathbf{r}}_0 \cdot \mathbf{r}_0 \neq 0$.

At each iteration, BiCG constructs a pair of direction vectors $\mathbf{p}_m$ and $\tilde{\mathbf{p}}_m$, and updates both the solution and the residual using scalar coefficients derived from biorthogonality conditions. These ensure that residuals and search directions remain mutually orthogonal between the two spaces:

$$\tilde{\mathbf{r}}_i \cdot \mathbf{r}_j = 0, \quad \tilde{\mathbf{p}}_i \cdot \mathbf{A}\mathbf{p}_j = 0 \quad \text{for } i \neq j.$$

This approach allows BiCG to use short recurrence relations and low memory, since it does not require storage of the full Krylov basis contrary to GMRES. The approximate solution is updated as

$$\tilde{\mathbf{x}}_{m+1} = \tilde{\mathbf{x}}_m + \alpha_m \mathbf{p}_m$$

where $\alpha_m = \frac{\tilde{\mathbf{r}}_m \cdot \mathbf{r}_m}{\tilde{\mathbf{p}}_m \cdot \mathbf{A}\mathbf{p}_m}$ is the step size ensuring the biorthogonality conditions are maintained.

While efficient, BiCG may suffer from breakdowns if the required inner products vanish. For this reason, more stable variants such as BiCGStab have been introduced.

3.3.4 BiConjugate Gradient Stabilized

The BiConjugate Gradient Stabilized method (BiCGStab) is an evolution of the BiCG algorithm that aims to improve numerical stability and convergence behavior. The idea is to eliminate the irregular convergence patterns and possible breakdowns that BiCG might suffer from, while preserving low memory.

The method retains the biorthogonal spaces of BiCG, but introduces a smoothing mechanism through a residual minimization step. At each iteration, the residual is corrected using a scalar ω_m chosen such that the new residual is orthogonal to $\mathbf{A}\mathbf{p}_m$. This additional step prevents the erratic oscillations typically encountered with BiCG and leads to smoother convergence [18].

The update for the approximate solution now involves two scalars α_m and ω_m, computed such that:

$$\mathbf{r}_{m+1} = \mathbf{r}_m - \alpha_m \mathbf{A}\mathbf{p}_m - \omega_m \mathbf{A}\mathbf{s}_m,$$

where $\mathbf{s}_m$ is an intermediate residual vector. The approximate solution is then updated using:

$$\tilde{\mathbf{x}}_{m+1} = \tilde{\mathbf{x}}_m + \alpha_m \mathbf{p}_m + \omega_m \mathbf{s}_m.$$

Compared to BiCG, BiCGStab avoids the need to maintain a second Krylov subspace for $\mathbf{A}^{\mathrm{T}}$, reducing the number of matrix-vector products per iteration and making the implementation simpler and often more robust in practice [32]. However, this comes at the cost of additional vector operations per iteration and a more complex recurrence scheme.

3.3.5 Iterative refinement

For large linear systems solved with direct methods, round-off amplification is still present even if it was previously minimized by the use of a pivoting technique. To improve the accuracy of the solution, iterative refinement algorithms can be used. The *simple iterative refinement* algorithm is an iterative method which can improve the accuracy of an already obtained solution which presents an error $\delta\mathbf{x}$ [16, 17].

Numerically, the actual linear system that was solved is therefore

$$\mathbf{A}(\mathbf{x} + \delta\mathbf{x}) = \mathbf{b} + \delta\mathbf{b}, \tag{3.17}$$

with $\mathbf{b}$ an unknown error on the right-hand side of Equation 3.1 due to round-off errors. Subtracting Equation 3.1 from 3.17 yields

$$\mathbf{A}\delta\mathbf{x} = \delta\mathbf{b} \tag{3.18}$$

which can be substituted into 3.17 to yield the following linear system

$$\mathbf{A}\delta\mathbf{x} = \mathbf{A}(\mathbf{x} + \delta\mathbf{x}) - \mathbf{b}. \tag{3.19}$$

The system of Equation 3.19 can be efficiently solved for $\delta\mathbf{x}$ using a direct method since $\mathbf{A}$ has already been factorized, requiring therefore only forward and backward substitution which are computationally inexpensive compared to the factorization [10, 15].

From the obtained correction $\delta\mathbf{x}$, the solution $\mathbf{x}$ is updated and the system of Equation 3.19 is solved again until a satisfying precision is reached [17].

As the *simple refinement* algorithm can be a good candidate to correct solutions obtained with a direct solver, other iterative methods can be used as well such as the Krylov-subspace methods [33]. In this case, the already computed factorized matrices are used as a preconditioner [10].

3.3.6 Preconditioning

The iterative solvers that were seen until now are likely to suffer from slow convergence for fluid dynamics problems. This can arise especially if the system is ill-conditioned. Contrary to direct solvers, iterative methods cannot manage ill-conditioned problems through pivoting, such that they have been considered impractical until the usage of *preconditioners* [10, 18, 34].

A *preconditioner* is a linear operator which can be applied to a linear system in order to speed up its convergence. Originally, attempts to improve convergence were done via developing linear operators that bound the condition number, hence the name of a "preconditioner". Although reducing the condition number is sufficient to ensure fast convergence only for SPD matrices, many of these attempts still resulted in efficient preconditioners for general matrices [10].

Preconditioning exploits ideas borrowed from sparse direct solvers. In practice, the quality of the preconditioner contributes much more to the robustness of iterative solvers than the specific Krylov-subspace method used [18].

There exist two main types of preconditioning. *Left preconditioning* transforms the linear system by applying the preconditioner from the left:

$$\mathbf{M}^{-1}\mathbf{A}\mathbf{x} = \mathbf{M}^{-1}\mathbf{b}. \tag{3.20}$$

This formulation ensures that the preconditioned residual $\mathbf{M}^{-1}(\mathbf{b}-\mathbf{A}\mathbf{x}_m)$ is minimized by the Krylov method. However, the true residual $\mathbf{b} - \mathbf{A}\mathbf{x}_m$ is not directly available at each iteration, making it less convenient as a stopping criterion [18].

Right preconditioning applies the preconditioner from the right, introducing an auxiliary variable $\mathbf{y}$:

$$\mathbf{A}\mathbf{M}^{-1}\mathbf{y} = \mathbf{b}, \quad \mathbf{x} = \mathbf{M}^{-1}\mathbf{y}. \tag{3.21}$$

This formulation preserves the true residual $\mathbf{r}_m = \mathbf{b} - \mathbf{A}\mathbf{x}_m$, which can be used directly as a convergence indicator. For this reason, right preconditioning is often preferred in the context of Krylov-subspace methods [18].

3.3.7 Jacobi Preconditioner

The Jacobi preconditioner, also called the diagonal preconditioner, is the simplest preconditioner available [10, 18]. The preconditioner matrix $\mathbf{M}$ is taken as the diagonal of $\mathbf{A}$:

$$\mathbf{M} = \mathrm{diag}(a_{11}, a_{22}, \ldots, a_{nn}), \tag{3.22}$$

so that applying $\mathbf{M}^{-1}$ amounts to scaling each equation by the inverse of its diagonal entry. This rescaling reduces the heterogeneity in row norms, which is a common source of ill-conditioning. The Jacobi preconditioner is extremely cheap to construct and apply, requiring only n divisions per iteration. However, it generally provides only moderate convergence improvement, and can fail entirely when diagonal entries are small or zero.

It is worth noting that the optimal preconditioner depends on the structure of the matrix $\mathbf{A}$ and the iterative method being used. Therefore, in this work, the impact of different preconditioning strategies will be investigated in conjunction with the iterative solvers presented earlier.

3.3.8 Flexible GMRES

Flexible GMRES (FGMRES) is an extension of the right-preconditioned GMRES algorithm in which the preconditioning operator is allowed to vary from one iteration to another. It originates from the classical right-preconditioned GMRES formulation, where a fixed preconditioner $\mathbf{M}$ is applied to transform the original linear system [18].

Right preconditioning is fundamental to FGMRES for two reasons. First, it preserves the true residual $\mathbf{r}_m = \mathbf{b} - \mathbf{A}\mathbf{x}_m$, which serves as a consistent convergence indicator across iterations. Second, and more critically, FGMRES allows the preconditioner to vary between iterations. If a left preconditioner were used instead, the minimized quantity at each step would be $\|\mathbf{M}_k^{-1}(\mathbf{b} - \mathbf{A}\mathbf{x}_m)\|$,

which changes with $\mathbf{M}_k$, leaving no consistent norm to minimize across iterations and causing the optimality property inherited from GMRES to collapse.

The main overhead of FGMRES compared to standard GMRES is that it requires additional memory to store more vectors. Indeed, the basis vectors no longer lie in the same Krylov subspace, therefore they cannot be expressed as linear combinations of the previous ones. Contrary to classical GMRES, FGMRES does not guarantee convergence [18].

3.3.9 ILU Preconditioners

The ILU(0) preconditioner, or Incomplete LU factorization with zero fill-in, is a widely used and simple preconditioning technique for sparse linear systems. It corresponds to the LU factorization of the matrix $\mathbf{A}$ which discards all fill-in elements that do not belong to the original sparsity pattern of $\mathbf{A}$ [18].

It is worth noting that ILU(0) is not a special case of ILUT: ILU(0) drops fill-in entries based purely on the sparsity structure of $\mathbf{A}$, with no numerical criterion involved, whereas ILUT retains entries according to both a fill parameter p, which selects the largest entries per row and therefore depends on both structure and numerical values, and a drop tolerance τ, which is an entirely numerical criterion.

Alternatively, the ILUT preconditioner aims to build a better approximation of the LU factors by selectively retaining additional nonzero entries during the factorization.

Two parameters control ILUT. The first is a drop tolerance τ, below which elements are discarded to preserve numerical sparsity. The second is a maximum number of nonzeros per row p to limit memory usage.
At each step of the factorization, small entries are dropped if their magnitude is less than τ, and the largest p entries per row are retained based on absolute value. The resulting preconditioner $\mathbf{M} \approx \mathbf{A}$ generally offers better convergence than ILU(0), at the cost of increased memory and setup time [18].

3.4 Presentation of Sparse Linear Solver Libraries

This section presents the sparse linear solver libraries that are interfaced in PFEM3D. Subsection 3.4.1 describes the ordering libraries. Then, Subsections 3.4.2, 3.4.3, 3.4.4, 3.4.5, 3.4.6, and 3.4.7 present the Eigen, MKL, PaStiX, AMGCL, SuperLU, and Panua PARDISO solver libraries along with their respective algorithms and parallelization strategies.

3.4.1 Ordering libraries

For sparse direct solvers, finding an optimal ordering which maximizes parallel efficiency in the factorization step is an NP-hard problem, meaning that no algorithm is known to solve it in polynomial time. Thus, many algorithms such as nested dissection or minimum degree have been developed in order to find approximate suitable solutions. Such approximation algorithms are called *heuristics*

[10].

It has been recognized in the literature that the nested dissection and its variants are the most suitable algorithms used to reduce fill-in and factorization computation time in structurally symmetric matrices. In addition to providing good approximate solutions, the nested dissection algorithms are efficiently parallelized [10, 35, 36, 37].

The PFEM software is compiled with two libraries being METIS and SCOTCH, both implement their own variant of the nested dissection. These libraries are also used for the ordering of the incomplete factorization preconditioner (ILU(0) and ILUT) [38, 39].

3.4.2 Eigen

Eigen is a C++ library for linear algebra which implements several sparse linear solvers including LU, QR, GMRES, and BiCGStab [40].

LU

The sparse LU solver of Eigen is a sequential supernodal LU factorization with fill-in reduction [41]. It uses BLAS-3 operations for the supernodes as described earlier, unless the supernodes are too small in order to enable a better optimization of the compiler. Since it is sequential, it does not take advantage of the elimination trees during the numerical factorization. Therefore, only the dense matrix operations used in BLAS-3 are parallelized in this implementation [42].

QR

Similarly, the sparse QR algorithm of Eigen is also strictly sequential [43]. It corresponds to the simple QR algorithm with Householder reflectors as described in Section 3.2.4. Contrary to the LU solver, supernodes are ignored and the QR solver does not leverage BLAS-3 routines [44].

GMRES

The implementation of this algorithm is the classical GMRES with restarting as described in Section 3.3.2 [45]. It is entirely sequential, as Eigen does not parallelize this solver. In PFEM3D, it is used with Eigen's built-in IncompleteLUT preconditioner, which is also sequential [42].

BiCGStab

The BiCGStab algorithm of Eigen implements the method described in Section 3.3.4. It is the only sparse linear solver of the Eigen library for nonsymmetric systems that is parallelized using OpenMP [42, 46]. In PFEM3D, it is used with an ILU(0) preconditioner.

3.4.3 MKL

The Intel Math Kernel Library (MKL) is a highly optimized numerical library developed by Intel for high-performance computing applications. It provides a wide range of routines for linear algebra,

including dense and sparse solvers, Fast Fourier Transforms (FFT), vector mathematics, and random number generation. MKL is specifically optimized for Intel architectures, offering multithreaded and vectorized implementations that take advantage of modern CPU features [20].

Direct Sparse Solver

The direct solver in MKL is based on the PARDISO library, which was originally developed by Panua [47]. It can be used through the Pardiso interface defined by Panua or by using the Direct Sparse Solver (DSS) interface as in the PFEM3D software. The DSS simplifies the Pardiso routines interface by providing default values that were selected by Intel for their observed performances [20].

This library selects the most suited algorithm based on the type of the matrix. For the sparse and structurally symmetric matrix of Equation 3.1, it is first solved by computing a fill-in reducing permutation using the nested dissection algorithm from the METIS library, before performing the numerical factorization with partial dynamic pivoting [20]. Therefore, the factorization considered is

$$\mathbf{PAP}^{\mathrm{T}} = \mathbf{QLU}, \tag{3.23}$$

where $\mathbf{P}$ is the fill-in reduction matrix, and $\mathbf{Q}$ the supernode pivoting matrix. The elimination tree offered by nested dissection algorithm and the use of supernodes allow both large and fine grain parallelism [15]. These make the DSS of MKL a highly parallelized method [20].

As previously explained, dynamic pivoting is impractical for sparse linear systems as swapping the rows or the columns during the numerical factorization leads to a change of the sparsity structure of the factorized matrices. Hence, the elimination tree built is no longer valid. In order to keep the constant elimination tree which was optimized during the ordering step, MKL uses partial pivoting only within the supernodes and eliminates its variable based on a supernodal elimination tree [20].

The pivoting strategy corresponds to applying partial pivoting for all supernodes separately, i.e., the candidate pivots are searched in the current supernode instead of looking for all candidate pivots in the entire matrix. The solution $\mathbf{x}$ is then found through backward and forward substitution, before applying the simple refinement algorithm in order to cancel the growing rounding errors of the substitutions [20].

GMRES

The MKL library implements the Flexible GMRES (FGMRES) algorithm, described in Section 3.3.8 [20, 48]. The main overhead of FGMRES compared to standard GMRES is the additional memory required to store extra basis vectors, since the preconditioned vectors no longer lie in the same Krylov subspace.

The MKL GMRES implementation includes a restarting mechanism and provides built-in ILU(0) and ILUT preconditioners. Both the solver and the preconditioner construction are internally parallelized through the MKL threading layer [20].

3.4.4 PaStiX

PaStiX (Parallel Sparse matriX package) is a high-performance parallel direct solver for sparse linear systems developed at Inria Bordeaux [49, 50]. It adopts a supernodal approach and leverages the elimination tree for parallelism, grouping contiguous columns with identical sparsity patterns into dense submatrices and exploiting the tree structure of the factorization for parallel execution. PaStiX is compatible with the METIS and SCOTCH ordering libraries [49].

LU

The LU factorization in PaStiX uses static pivoting, which is a key differentiator from MKL's dynamic partial pivoting within supernodes [49, 50]. With static pivoting, the permutation is entirely determined during the ordering step, and small diagonal entries are replaced by a fixed perturbation value rather than performing row or column exchanges during the numerical factorization. The factorization is therefore

$$\mathbf{PAP}^{\mathrm{T}} = \mathbf{LU}, \tag{3.24}$$

where $\mathbf{P}$ is the fill-in reduction permutation matrix. Unlike MKL, no additional pivoting matrix $\mathbf{Q}$ appears, since no pivoting is performed during the numerical factorization. This has the important consequence that the elimination tree and the sparsity pattern of the factors remain constant throughout the factorization.

Dense blocks within supernodes are factorized using BLAS-3 operations. Parallelism is achieved at two levels: large-grain parallelism through the elimination tree, where independent subtrees are processed concurrently, and fine-grain parallelism through the threaded BLAS routines applied to the dense supernodal blocks [50].

After the solve step, PaStiX applies iterative refinement in order to reduce rounding errors that may have accumulated during the forward and backward substitution steps [49].

3.4.5 AMGCL

AMGCL is an open-source, header-only C++ library developed by Denis Demidov for solving large sparse linear systems [51, 52]. It provides a modular framework that combines iterative solvers with preconditioners . Unlike MKL or PaStiX, AMGCL is designed exclusively for iterative methods and does not include any direct solver. In PFEM3D, AMGCL is configured with OpenMP for shared-memory parallelization.

GMRES

AMGCL implements the standard restarted GMRES algorithm as described in Section 3.3.2 [51]. Unlike MKL which implements the Flexible GMRES variant, the AMGCL GMRES is a classical GMRES where the preconditioner remains fixed throughout the iterations [18]. In practice, this is not a limitation when combined with ILU-type preconditioners since their application is deterministic.

In PFEM3D, the AMGCL GMRES solver is used with either an ILU(0) or an ILUT preconditioner. The ILUT preconditioner in AMGCL uses a fill factor p_{fill} which specifies the allowed fill relative to the original number of nonzeros per row in $\mathbf{A}$. This differs from MKL's ILUT, which uses a parameter specifying the absolute maximum number of nonzero entries per row in the incomplete factors. Both the solver iterations and the preconditioner construction are parallelized using OpenMP [18, 20, 53].

BiCGStab

AMGCL implements the BiCGStab algorithm described in Section 3.3.4 [51]. Like GMRES, all matrix-vector products and vector operations are parallelized using OpenMP.

A key advantage of BiCGStab over GMRES is its use of short recurrence relations. While GMRES must store the entire orthonormal basis of the Krylov subspace, which grows with the number of iterations until restart, BiCGStab maintains only a fixed number of vectors regardless of the iteration count [18]. This results in significantly lower memory requirements, particularly for problems where a large number of iterations is needed before convergence [18].

The same preconditioner options are available as for the GMRES solver, including ILU(0) and ILUT. In PFEM3D, the AMGCL BiCGStab solver is interfaced with an ILU(0) preconditioner.

3.4.6 SuperLU

SuperLU is a sparse direct solver library developed by Demmel, Gilbert, and Li in 1999 at Lawrence Berkeley National Laboratory, providing a high-performance implementation of the LU factorization for sparse unsymmetric matrices [54]. Its original sequential version has been extended into two parallel variants: SuperLU MT, for shared-memory architectures, and SuperLU Dist, which is designed for distributed-memory systems [55]. Although the Vaillant platform is a shared-memory machine, SuperLU Dist was selected over SuperLU MT, as the latter receives less frequent maintenance, and SuperLU Dist can be configured to operate in shared-memory mode through the MPI library. Like PaStiX, SuperLU Dist uses static pivoting, yielding the factorization of Equation (3.24).

3.4.7 Panua PARDISO

PARDISO is a high-performance direct sparse solver library originally developed by Olaf Schenk and Klaus Gärtner. Intel licensed this solver in 2006 for inclusion in the Math Kernel Library (MKL). The original developers subsequently founded the company *Panua* to continue independent development of PARDISO, while Intel maintained its own fork optimized specifically for Intel architectures. Therefore, like Intel MKL, Panua PARDISO uses dynamic pivoting and the nested dissection ordering algorithm from Metis will be used by default [47, 56]. The description of MKL DSS in Section 3.4.3 applies in full to Panua PARDISO, with the factorization taking the form of Equation (3.24).

Chapter 4

Comparison of Linear Solvers

This chapter presents an empirical comparison of several linear solvers, both direct and iterative, conducted through simulations performed with PFEM3D. The objective is to identify the fastest solvers and their associated optimal parameters, with a view to enabling large-scale simulations involving hundreds of thousands of particles to be carried out within a reasonable computational time. Throughout this chapter, practical recommendations are provided to guide the reader in selecting an appropriate solver depending on the size of the linear systems to be solved and the computational resources available.

Section 4.1 describes the experimental framework employed throughout this chapter, including the computational platform, the benchmark linear systems derived from PFEM3D simulations of the three-dimensional dam break problem, and the performance metrics used to compare the solvers. Section 4.2 presents the evaluation of four direct sparse LU solvers, namely SuperLU Dist, Intel MKL DSS, Panua PARDISO, and PaStiX LU, with respect to their computational time, parallel scalability, and memory requirements. The performance of iterative solvers is subsequently investigated, covering the influence of preconditioning strategies, convergence tolerance, and solver types including BiCGStab and GMRES. The chapter closes with practical recommendations for solver selection as a function of problem size and available computational resources.

4.1 Methodology

This section describes the framework employed to evaluate and compare the performance of linear solvers in the context of PFEM3D simulations. First, the computational platform used for all benchmarks is presented in Section 4.1.1, along with the hardware characteristics relevant to parallel performance. The benchmark linear systems are then introduced in Section 4.1.2, specifying the simulation scenario, the mesh sizes, and the time instants selected for the comparative study. The performance metrics used throughout this chapter are subsequently defined in Section 4.1.3, namely the speedup, the parallel efficiency, and the Performance Increase Factor. The set of linear solvers retained for the comparison is then established in Section 4.1.4 based on a preliminary assessment of their multi-threading capabilities. Finally, the performance of the Eigen solvers is assessed in Section 4.1.5 to establish a reference baseline.

4.1.1 Computer Specifications

All simulations and benchmarks presented in this work were executed on a single computational platform dedicated to this study, in order to ensure reproducibility. The hardware specifications of this platform are summarized in Table 4.1.

Component	**Specification**
Processor	AMD Ryzen Threadripper PRO 3995WX
Cores	64
Threads per core	1
RAM	256 GiB
Architecture	x86_64

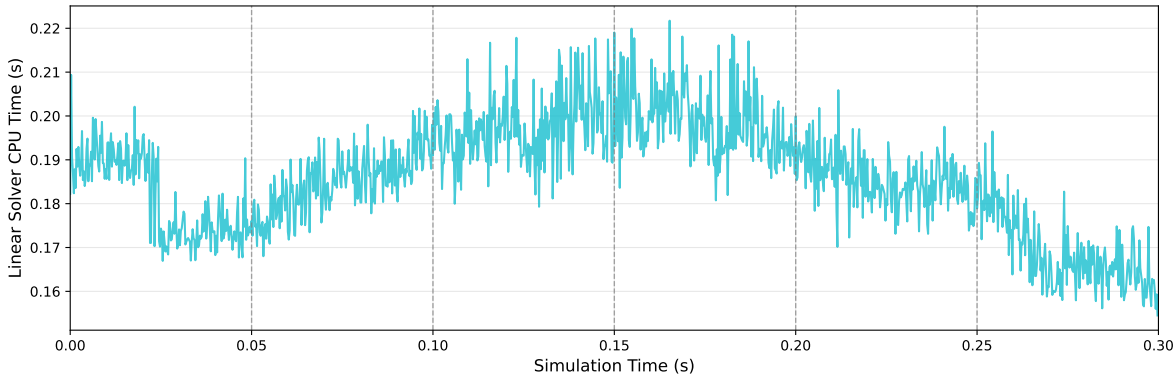
Table 4.1: Hardware specifications of the computational platform.

The AMD Ryzen Threadripper PRO 3995WX features 64 physical cores, each capable of executing up to two simultaneous threads [57]. Simultaneous Multithreading (SMT) is disabled on this platform so that each core executes a single thread, as reflected in Table 4.1. This configuration is preferred for numerical computations, where interference between threads sharing cache resources can reduce performance [58]. Furthermore, to prevent benchmark results from being skewed by other processes sharing the cores, most simulations are performed using up to 32 isolated cores. All linear systems that arose in the simulations presented in this chapter fit in the 256 GiB of RAM to avoid relying on disk-based storage, which would significantly degrade performance [59].

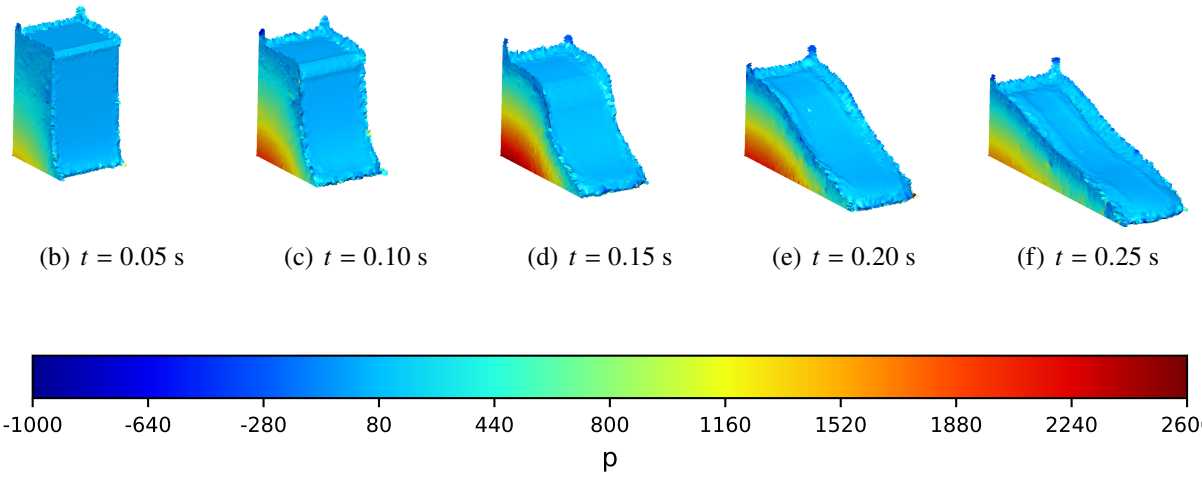
4.1.2 Benchmark Linear Systems

In this section, five representative time instants are selected from the full simulation on which the solvers will be compared. This will allow to assess the influence of both the simulation time, which affects the structure of the matrices due to remeshing, and the nonlinear iterations, which determine the number of linear systems to be solved at each time step. The computational cost measured on these representative instants is then extrapolated to estimate the total solver cost over the complete simulation duration.

The time instants selected for the comparison are illustrated in Figure 4.1. The computation time dedicated to the linear solver using MKL DSS throughout the simulation for a mesh with $n = 10{,}000$ particles is displayed. The five time instants at $t = 0.05$, 0.10, 0.15, 0.20, and 0.25 seconds are indicated by vertical dashed lines and correspond to different stages of the dam break evolution. These instants capture the fluid at various configurations: from the initial collapse phase through the spreading stage until the impact with the opposite wall, as shown in the corresponding snapshots.



(a) Linear solver computation time as a function of simulation time for n = 10,000 initial particles using MKL DSS 2019 with the Newton-Raphson nonlinear algorithm. Vertical dashed lines indicate the selected time instants for detailed analysis.



(b) t = 0.05 s (c) t = 0.10 s (d) t = 0.15 s (e) t = 0.20 s (f) t = 0.25 s

(g) Pressure colorbar (Pa)

Figure 4.1: Selection of time instants for linear solver comparison. (a) Evolution of the linear solver computation time throughout the simulation for a mesh with n = 10,000 particles. (b)–(f) Snapshots for a mesh with n = 50,000 particles at the five selected time instants. (g) Pressure colorbar, where the color is proportional to pressure intensity (high pressure in red, low in blue).

The analysis is conducted for four different mesh sizes corresponding to n = 50,000, 90,000, 150,000, and 300,000 particles. Each time step involves a varying number of nonlinear iterations, with each iteration requiring the solution of a linear system. The Newton-Raphson algorithm will be used in this chapter to solve the different time instants. The different computation times presented will be averaged accross these time instants.

4.1.3 Speedup, Parallel Efficiency and Performance Increase Factor

The speedup $S(j)$ represents the performance improvement obtained by an algorithm using j threads compared to a single-threaded execution. It is defined as:

$$S(j) = \frac{T(1)}{T(j)}, \tag{4.1}$$

where $T(1)$ represents the execution time with one thread and $T(j)$ the execution time with j threads. Ideally, one would expect $S(j) = j$, meaning that doubling the number of threads halves the execution time. However, not all operations in an algorithm can be parallelized, thus limiting the maximum achievable speedup [60].

Since the speedup is bounded, using too many threads may not be beneficial if the speedup does not increase accordingly. To quantify how effectively an algorithm utilizes the available threads, the parallel efficiency $E(j)$ is introduced. It is defined as:

$$E(j) = \frac{S(j)}{j} = \frac{T(1)}{j \cdot T(j)}. \tag{4.2}$$

The parallel efficiency represents the fraction of ideal speedup achieved and is bounded by $E(j) \in (0, 1]$, with $E(j) = 1$ corresponding to perfect linear scaling [60]. This metric will be employed in subsequent sections to compare the parallelization capabilities of different solvers.

To compare the performance of two solvers, the Performance Increase Factor (PIF) is introduced [47]. It is defined as:

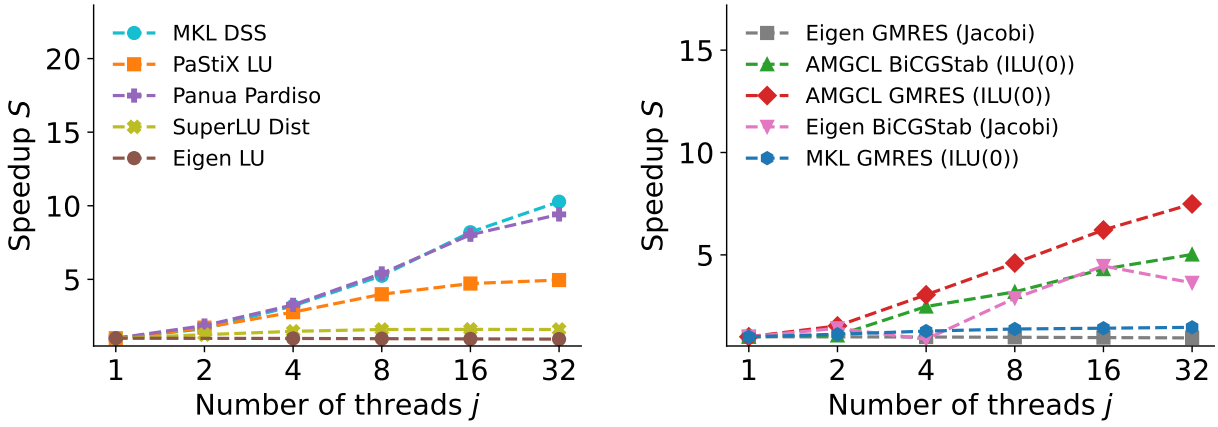
$$\text{PIF}(\text{solver}_1, \text{solver}_2) = \frac{\bar{T}_{\text{solver}_1}}{\bar{T}_{\text{solver}_2}}, \tag{4.3}$$

where $\bar{T}$ denotes the average computation time per time step. A PIF greater than 1 indicates that solver_2 outperforms solver_1, while a PIF less than 1 indicates that it is slower.

4.1.4 Linear Solvers Investigated

Among the linear solver libraries presented in the previous chapter, focus is placed on parallelized solvers in order to fully exploit the computational resources available. Parallelization tests are conducted over 5 time instants at a mesh size of $n = 50{,}000$ particles to assess the multi-threading capabilities of each solver.

Figure 4.2 reports the speedup $S(j)$ as a function of the number of threads. As one can see, MKL DSS, Panua Pardiso, PaStiX LU, Eigen BiCGStab, AMGCL BiCGStab, and AMGCL GMRES solvers exhibit a significant speedup with increasing thread count, whereas both Eigen LU and Eigen GMRES remain at a speedup of unity, confirming their sequential nature as outlined in Eigen's documentation [42]. Regarding SuperLU and MKL GMRES, both solvers display only slight improvements when parallelized.



(a) Direct solvers.

(b) Iterative solvers.

Figure 4.2: Speedup S as a function of the number of threads j (n = 50,000 particles, 5 time instants).

Throughout this study, a choice is made to consider only highly parallelized solvers or newly interfaced solvers, excluding both Eigen GMRES and MKL GMRES. SuperLU, as a newly interfaced solver, is thus retained. Although Eigen LU is neither new nor parallelized, it is used as a baseline and its performance is assessed on simple systems to evaluate the performance increase factors of the other solvers relative to it.

The comparison conducted will therefore focus on eight linear solvers, which are Eigen LU, Eigen BiCGStab, MKL DSS, PaStiX LU, SuperLU Dist, Panua Pardiso, AMGCL BiCGStab, and AMGCL GMRES.

4.1.5 Eigen Solvers as Performance Baseline

When PFEM3D was first developed, the Eigen library was the only available option for solving the linear systems arising from the discretized incompressible Navier-Stokes equations. Although the LTAS-MN2L lab interfaced several additional solver libraries, no systematic benchmarking study had been conducted to evaluate and compare their relative performances. To start this benchmarking, the initial performances of PFEM3D using the Eigen linear solvers are first assessed in order to establish a reference baseline against which the subsequently investigated solvers are compared.

Figure 4.3 presents the computational time breakdown for two simulations performed over the first 10 time instants (from 0 s to $2.5 \cdot 10^{-3}$ s) of the dam break benchmark described in Section 4.1.2, one employing the Eigen LU direct solver and one employing the Eigen BiCGStab iterative solver with a Jacobi preconditioner.

A complete simulation with Eigen LU would require approximately 10 days of computation, while Eigen BiCGStab reduces this estimate to approximately 1 day. Larger meshes would require even longer computation times, rendering large-scale simulations impractical with the Eigen solvers. Furthermore, as evidenced by the breakdown of the computation time, the linear solver constitutes

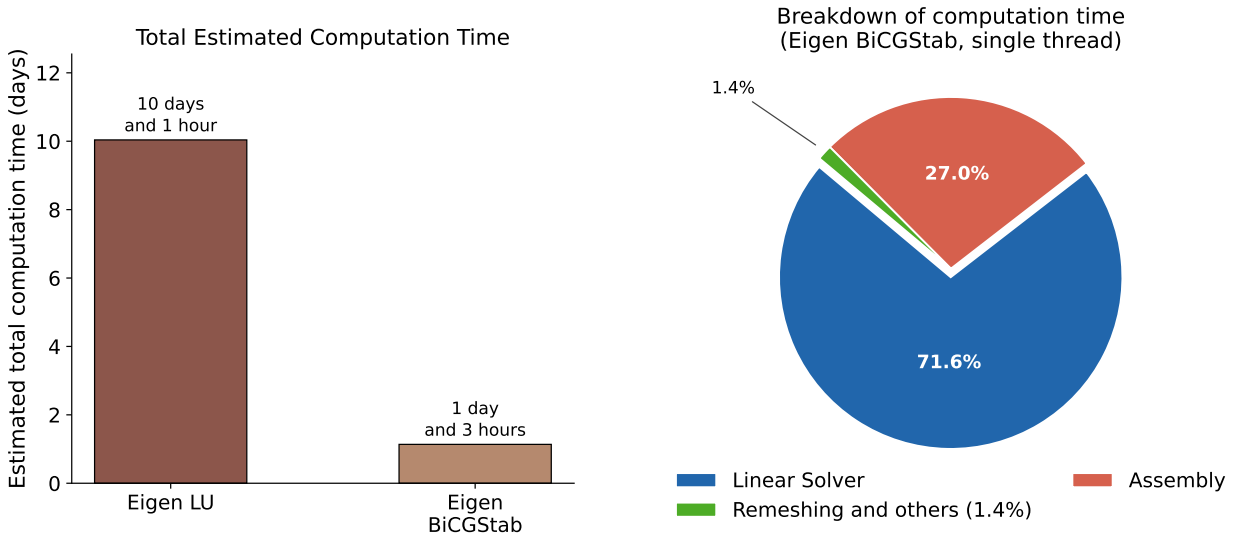


Figure 4.3: Computational cost of PFEM3D using the Eigen linear solvers on a dam break benchmark with 50 000 particles over 10 time steps. *Left*: estimated total computation time, obtained by scaling the 10-step duration by a factor of 120. *Right*: breakdown of computation time for Eigen BiCGStab, with computation time decomposed into LU factorization and solve phases.

the dominant source of computational cost in PFEM3D. The linear solver being the bottleneck, optimizing it would enhance significantly the rate at which incompressible Navier-Stokes simulations can be performed when using the PFEM method. Moreover, the results already reveal a significant performance gain when transitioning from Eigen LU to Eigen BiCGStab, with a speedup of 9, highlighting the critical influence of the linear solver choice on the overall computational efficiency. Throughout the remainder of this chapter, the Eigen LU solver serves as the reference point against which the performance of alternative solvers is evaluated.

4.2 Direct Solvers

This section presents a comparison of the different direct sparse solvers interfaced in PFEM3D, namely Eigen LU, SuperLU Dist, Intel MKL DSS, Panua PARDISO, and PaStiX LU, with Eigen LU serving as the performance baseline.

Section 4.2.1 first evaluates SuperLU Dist against Eigen LU. Section 4.2.2 then compares the two PARDISO-based solvers, one being Panua PARDISO and the other MKL DSS. After evaluating the different groups of direct solvers, Sections 4.2.3 and 4.2.4 analyze their collective empirical time complexity and parallel efficiency across all mesh sizes considered. Finally, the memory requirements of all direct solvers are examined and compared.

4.2.1 SuperLU based solvers

According to Eigen's documentation, the sparse LU solver provided by the library is based on the main algorithmic techniques of the SuperLU package [41]. Both solvers are compared in this

section to determine the most performant SuperLU-based solver for PFEM3D simulations.

The solver was evaluated across all mesh sizes and time instants described in the preceding sections, with thread counts ranging from 1 to 32. Contrary to the simulations launched with Eigen, the ordering algorithm used with SuperLU was not COLAMD but the multilevel nested dissection from the Metis package. This difference will result in a difference of structural factorization and should thus have an influence on the fill-in and the performances. Figure 4.4 presents the Performance Increase Factor (PIF) relative to Eigen LU for n = 50,000 particles, along with the peak memory required to factorize the different matrices encountered.

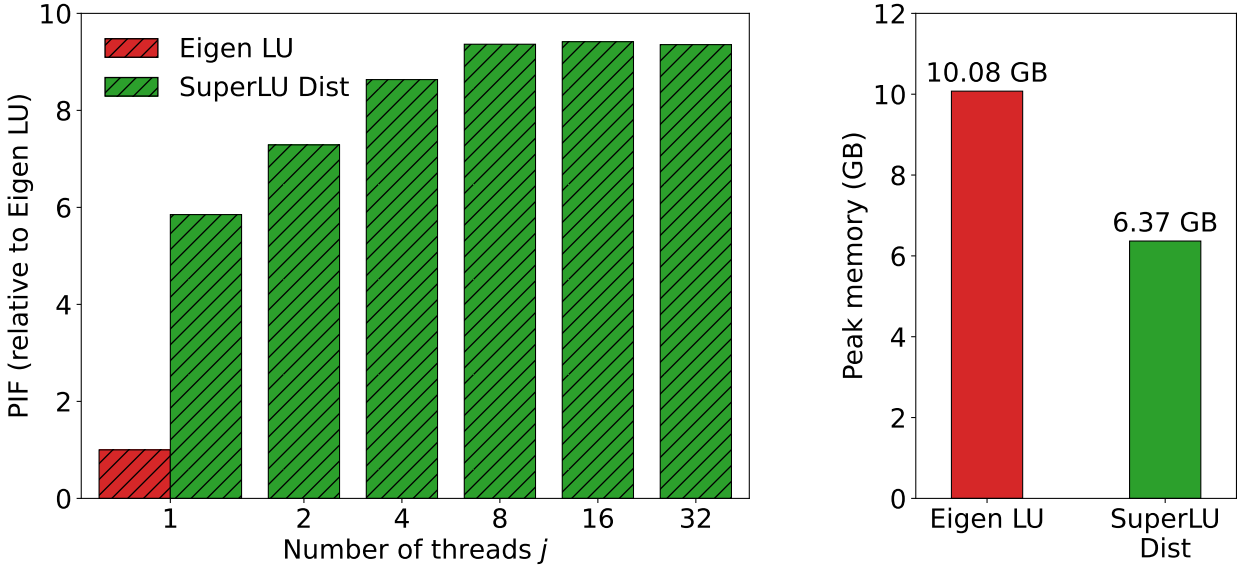


Figure 4.4: Performance Increase Factor of SuperLU Dist relative to Eigen LU as a function of the number of threads (left) and peak memory requirement (right), for n = 50,000 particles.

A substantial performance advantage over Eigen LU is observed at all thread counts. With a single thread, a PIF of approximately 5.9 is already achieved. The superiority of SuperLU over Eigen LU can be seen in both the computation times and the required memory, which results from different ordering algorithm.

COLAMD is a greedy local heuristic that reorders columns by iteratively eliminating the one producing the fewest new non-zeros, making decisions based solely on local information. Multilevel nested dissection, as implemented in the Metis package, takes a global approach: it recursively partitions the sparsity graph into subgraphs separated by small vertex separators, ordering the separator vertices last. For matrices arising from 3D unstructured meshes, this global strategy yields significantly less fill-in, reducing both memory consumption and the number of floating-point operations during factorization, which explains the performance gap already visible at a single thread.

Extrapolating these results to the full-scale simulation, the estimated computation time of 11 days with Eigen LU would be reduced to approximately 28 hours with SuperLU Dist when using at least 8 threads.

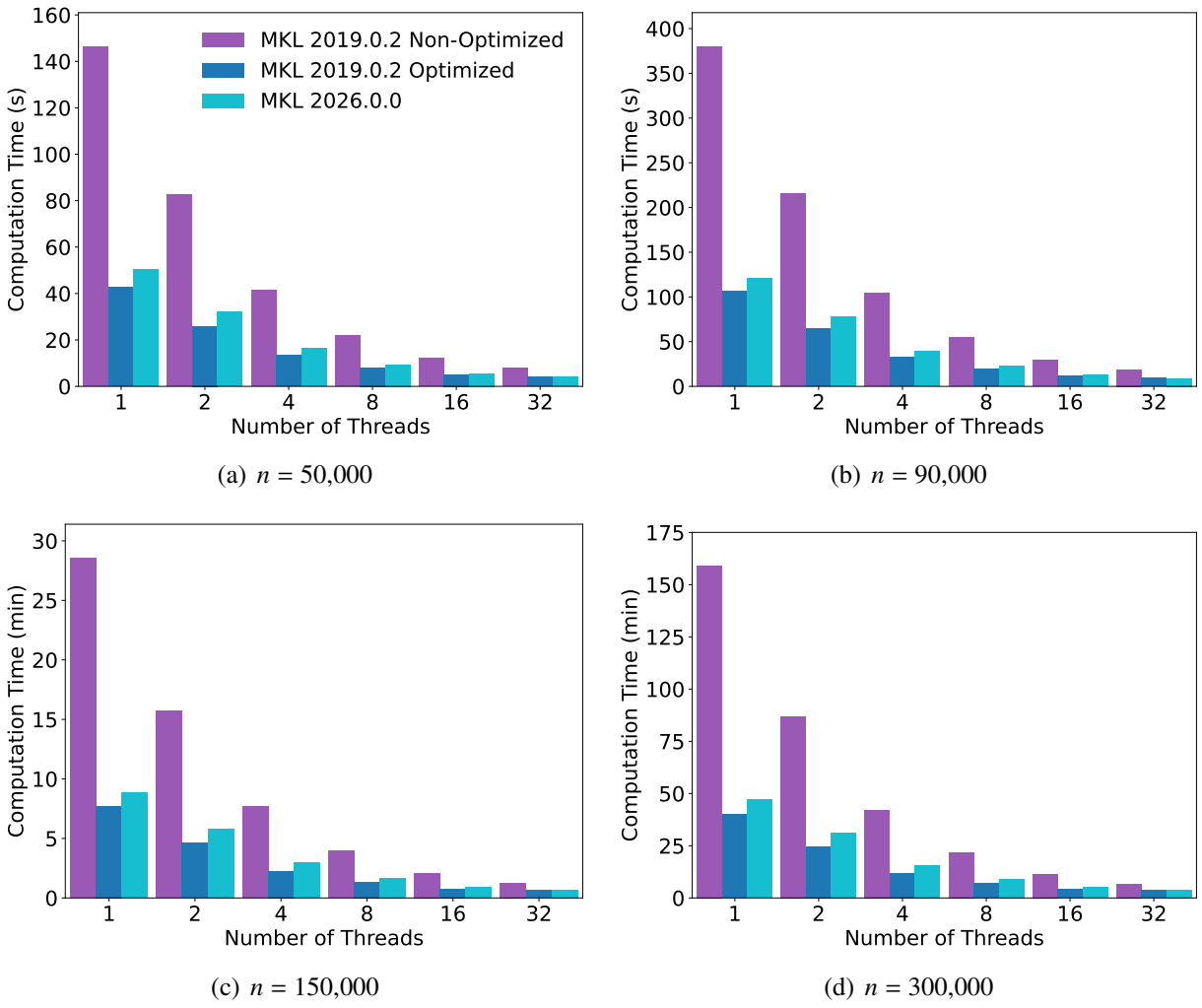
4.2.2 PARDISO-based Solvers

Intel has a history of optimizing the MKL specifically for Intel processors. In particular, it has been observed that MKL employs CPU vendor detection to selectively enable optimized code paths [61, 62, 63]. Prior to MKL version 2020.1, it was possible to override this behavior on non-Intel processors by setting the environment variable `MKL_DEBUG_CPU_TYPE=5`, which forced the library to use optimized code paths regardless of the detected CPU vendor [64]. Starting from version 2020.1, this workaround was disabled. Whether this vendor-specific behavior persists in the latest versions of MKL is therefore investigated in this section. Since the original PARDISO solver on which MKL DSS is based remains available as a standalone library through Panua, it was interfaced in this work and is also evaluated as a vendor-neutral alternative.

In this section, the performance of the DSS solver is compared for the 2026.0.0 version, in which setting this environment variable is no longer possible, and the 2019.0.2 version which presents two configurations: the default non-optimized mode, where MKL detects the AMD processor and uses generic code paths, and the optimized mode, where the environment variable enables the use of Intel-optimized routines. All simulations were performed using the five time steps of the three-dimensional dam break test case, as described earlier in Section 4.1.2. The ordering algorithm used is the default one of MKL, which is the nested dissection algorithm from the Metis package [38].

As one can see in Figure 4.5, the 2019 optimized configuration consistently outperforms the non-optimized configuration across all tested mesh sizes. In the single-threaded case, the optimized version achieves a PIF of approximately 3.1 to 3.5 relative to the non-optimized version. This relative performance improvement remains consistent across all mesh sizes, confirming that the optimization benefits are independent of the problem size. Regarding the 2026 version, performances are almost identical to the 2019 optimized version. Even if a vendor-specific penalty is still present, its impact is negligible, with at most 19% overhead at a single thread and within 6% at 32 threads. The optimized MKL 2019 configuration is nonetheless retained for subsequent comparisons, as it yields better or equivalent performance across all configurations and guarantees that the CPU vendor detection is bypassed.

Since MKL is an Intel-maintained library with no guarantee of long-term support for AMD architectures, Panua PARDISO represents a vendor-neutral option worth evaluating. Despite being a subscription-based solver, benchmarks on the Panua website claim to outperform MKL DSS in most scenarios [47]. Its performance is therefore compared against MKL DSS using the linear systems arising from the PFEM3D dam break described in Section 4.1.2, for the four mesh sizes considered.



(a) $n = 50{,}000$

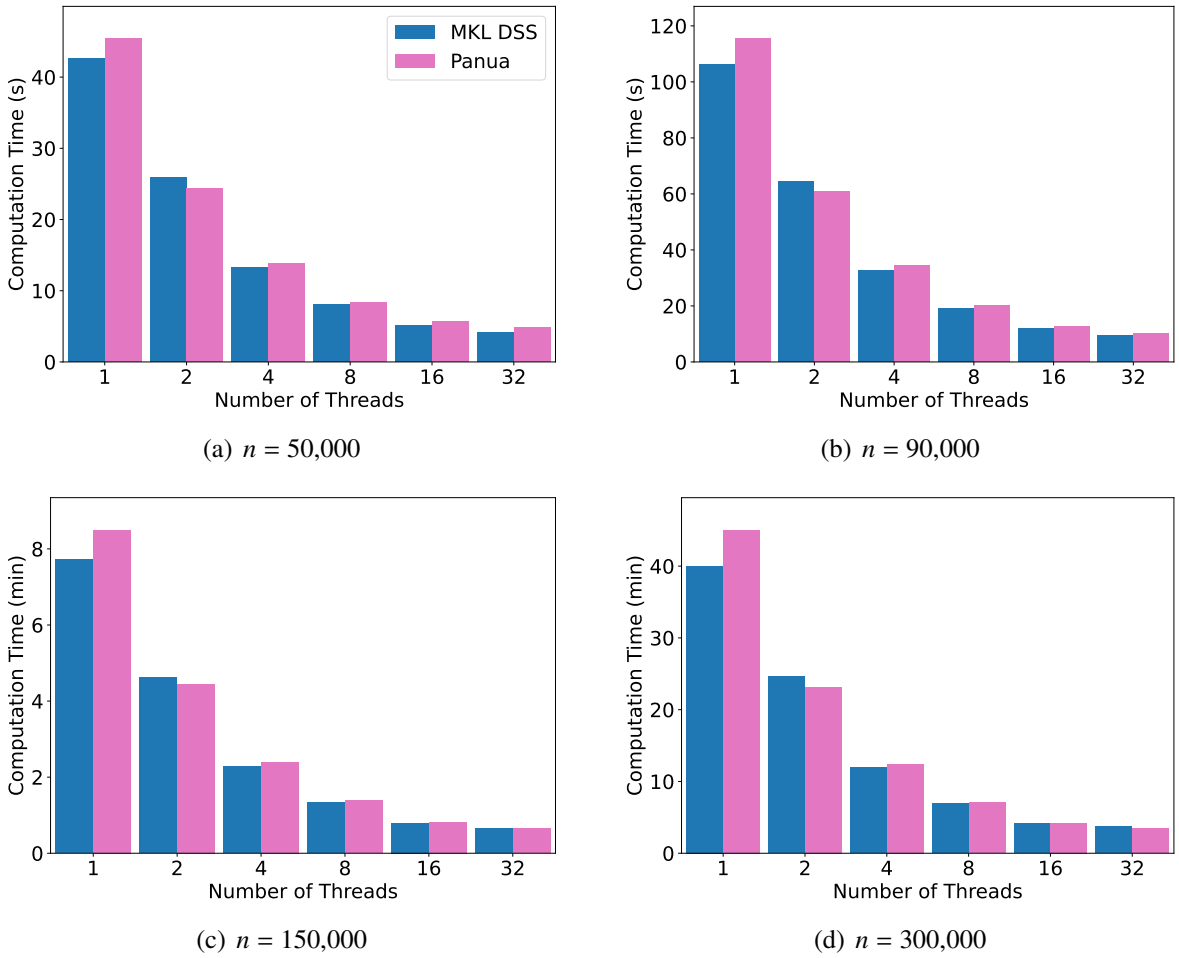
(b) $n = 90{,}000$

(c) $n = 150{,}000$

(d) $n = 300{,}000$

Figure 4.5: Total computation time of the DSS solver as a function of the number of threads for MKL in non-optimized and optimized configurations. Each subfigure corresponds to a different mesh size.

Figure 4.6 compares the computation time of both solvers. Panua PARDISO and MKL DSS display similar performance across all mesh sizes and thread counts, with Panua PARDISO underperforming slightly in most configurations. Given that Panua PARDISO is a subscription-based solver while MKL DSS is freely available, these results do not justify its adoption for the linear systems arising from the PFEM3D dam break test case. To investigate the cases where Panua PARDISO can beat MKL DSS, the corresponding PIF are reported in Table 4.2.



(a) $n = 50{,}000$ (b) $n = 90{,}000$

(c) $n = 150{,}000$ (d) $n = 300{,}000$

Figure 4.6: Computation time of MKL DSS and Panua PARDISO as a function of the number of threads for linear systems arising from the dam break test case in PFEM3D.

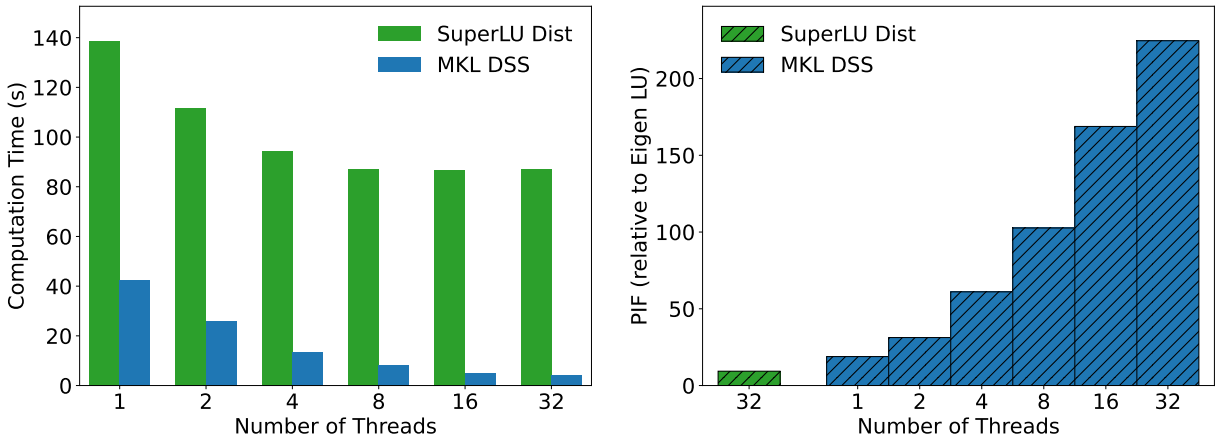
	$j = 1$	$j = 2$	$j = 4$	$j = 8$	$j = 16$	$j = 32$
$n = 50{,}000$	0.94	1.06	0.96	0.97	0.92	0.86
$n = 90{,}000$	0.92	1.06	0.95	0.96	0.94	0.93
$n = 150{,}000$	0.91	1.04	0.95	0.96	0.96	1.00
$n = 300{,}000$	0.89	1.06	0.96	0.98	0.98	1.08

Table 4.2: Ratio of Pardiso PIF to MKL DSS PIF across mesh sizes and thread counts. Values above 1 indicate Panua PARDISO is faster; values below 1 indicate MKL DSS is faster.

According to both Figure 4.6 and Table 4.2, Panua PARDISO outperforms MKL DSS when using 2 threads across all mesh sizes, and it also outperforms MKL DSS with 32 threads for the largest mesh size of $n = 300{,}000$ particles. By looking at the first column, one can see that as the mesh size increases, Panua's relative performance decreases. For the last column, it is the opposite: the

relative performance keeps increasing with the mesh size until it finally outperforms MKL DSS at n = 300,000 particles. Since MKL DSS 2019 optimized is the fastest, freely available, and guarantees that CPU vendor detection is bypassed, it is retained as the best PARDISO-based solver.

The performance of MKL DSS 2019 is finally compared against SuperLU Dist and Eigen LU for n = 50,000 particles in Figure 4.7.



(a) Computation time

(b) Performance Increase Factor relative to Eigen LU

Figure 4.7: Comparison of MKL DSS (2019 optimized) and SuperLU Dist against Eigen LU for n = 50,000 particles, as a function of the number of threads j.

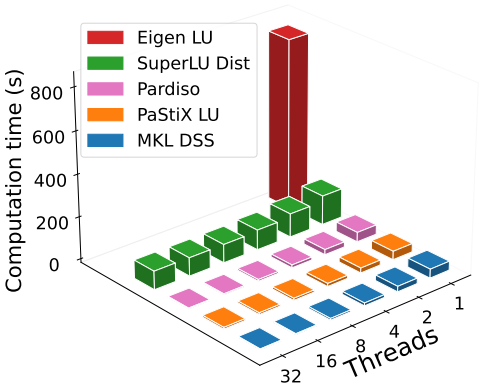
At all thread counts, MKL DSS outperforms SuperLU Dist. While SuperLU Dist reaches a PIF of approximately 9.4 at 32 threads, MKL DSS achieves a PIF exceeding 220 at the same thread count, representing a more than 20× improvement over SuperLU Dist. The simulation that would take 11 days with Eigen LU would now only take approximately 1.2 hours with MKL DSS using 32 threads.

4.2.3 Time Complexities

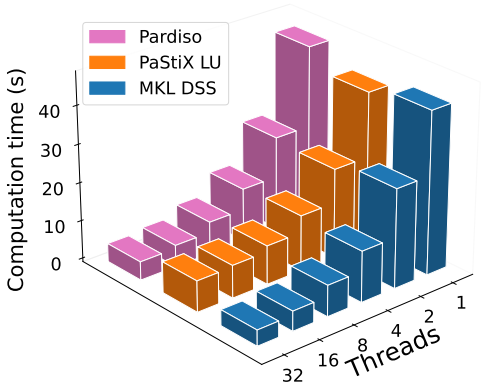
This section investigates how performances of the sparse direct solvers evolve with respect to the mesh size and how this scaling can be compared between the solvers in order to understand cases in which one should prefer a solver over an other. In addition to SuperLU-based and Pardiso-based solvers, PaStiX is now included in the comparison. The main differences with Pardiso-based solvers is that PaStiX relies on static pivoting and uses the Scotch library for nested dissection ordering, contrary to MKL DSS and Panua Pardiso which use METIS [20, 49, 50, 56, 65].

PaStiX LU was run with a static pivoting threshold $\delta = 10^{-16}$ and an iterative refinement tolerance $\varepsilon = 10^{-16}$, on the same mesh sizes and thread counts as the other solvers. Figure 4.8 presents the total computation time of all evaluated direct solvers as a function of the number of threads, for each mesh size considered in this study. As shown in Figure 4.8(a), the computation time of Eigen LU on the smallest mesh (n = 50,000) with a single thread exceeds that of all other solvers, and comparing the results with the remaining subfigures shows that simulations using this solver with

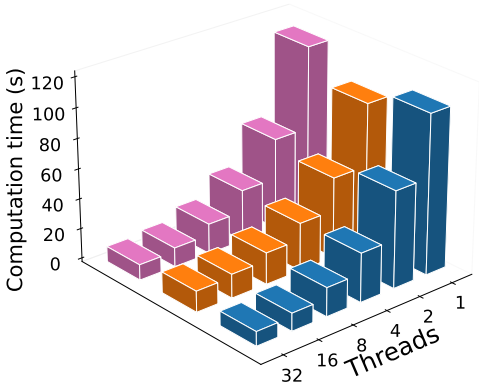
50,000 nodes is still slower than using another solver on higher mesh sizes below $n = 300{,}000$ and for every thread count. The direct solvers PaStiX, MKL DSS, and Panua Pardiso remain competitive for all mesh sizes and thread counts, with PaStiX LU being highly performant on the largest problem sizes.



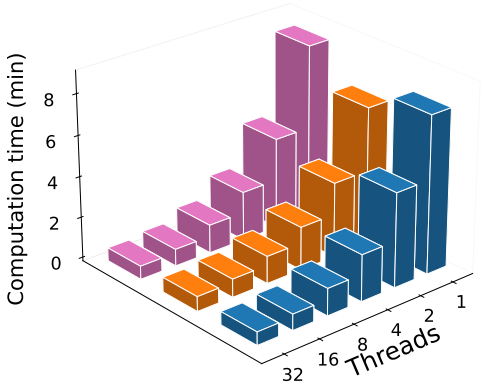
(a) $n = 50{,}000$ (all solvers)



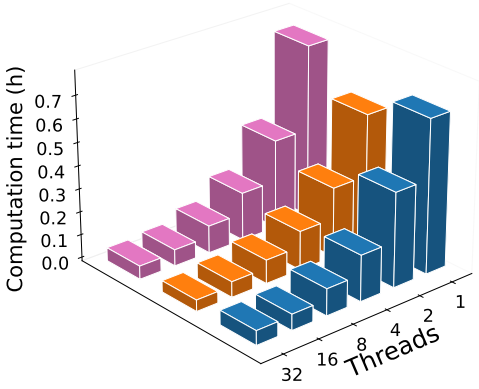
(b) $n = 50{,}000$



(c) $n = 90{,}000$



(d) $n = 150{,}000$



(e) $n = 300{,}000$

Figure 4.8: Total computation time of all direct solvers as a function of the number of threads. Subfigure (a) includes Eigen LU and SuperLU Dist, which are only evaluated at $n = 50{,}000$; subfigures (b)–(e) show PaStiX LU, MKL DSS, and Panua Pardiso across all mesh sizes.

To assess how these solvers behave asymptotically with respect to problem size, and whether this behaviour depends on the degree of parallelization, Figure 4.9 presents the total computation time as a function of the number of non-zeros (NNZ) on a log-log scale, for each thread count $j \in \{1, 2, 4, 8, 16, 32\}$. As one can see, all three solvers exhibit approximately linear trends in the log-log representation for every thread count. This observation suggests that the empirical time complexity can be modeled following a power-law relationship of the form

$$\log T(\text{nnz}) \approx \alpha \log(\text{nnz}) + C \quad \Longrightarrow \quad T(\text{nnz}) = O(\text{nnz}^{\alpha}), \tag{4.4}$$

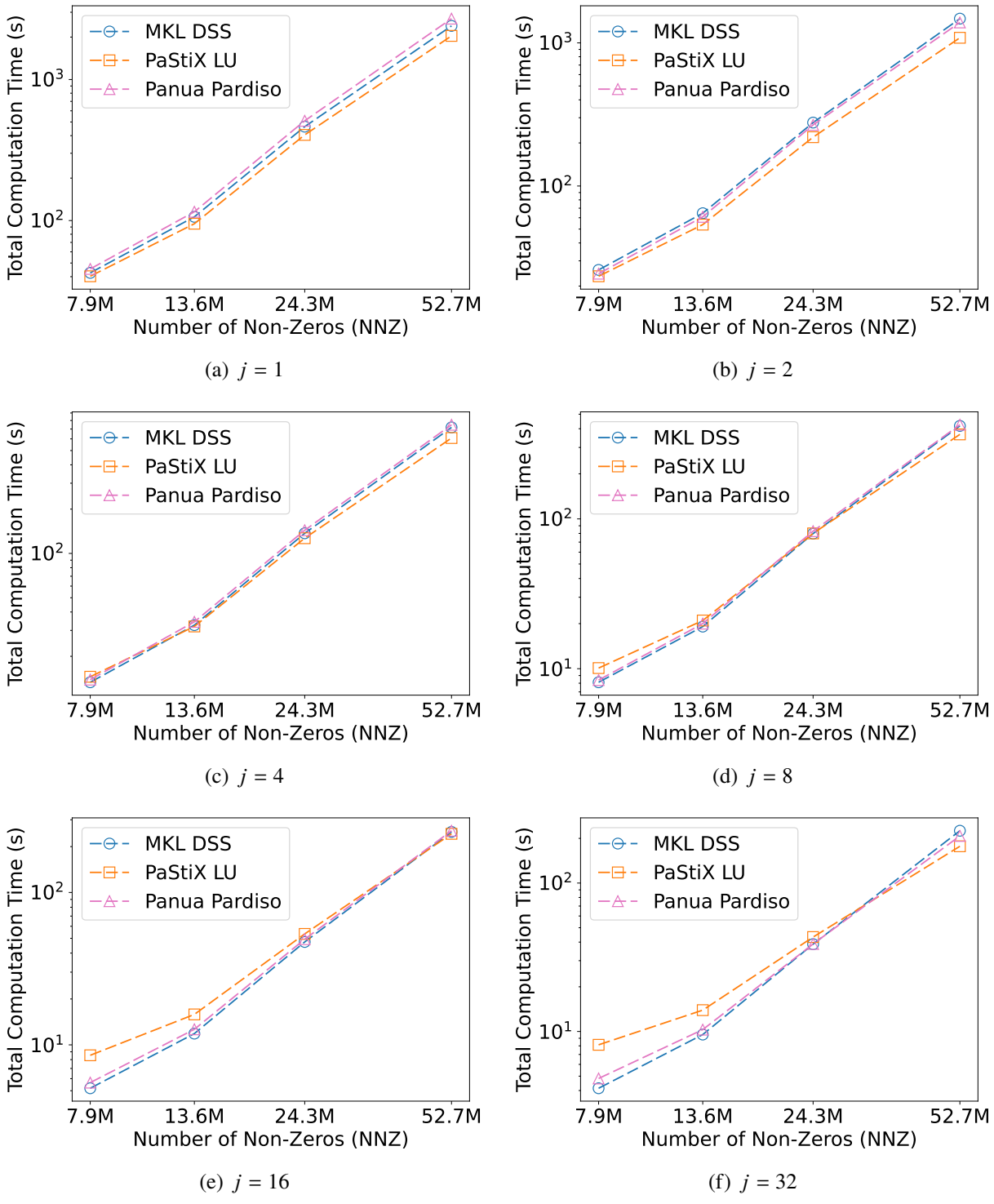
where α denotes the slope of the linear regression in log-log space and C is a constant term. The exponent α can therefore be estimated from a least-squares fit of $\log T$ versus $\log(\text{nnz})$ across all problem sizes. Applying this methodology across all thread counts yields the empirical time complexity exponents presented in Table 4.3.

Solver	1	2	4	8	16	32	**Avg**
MKL DSS	2.16	2.17	2.14	2.11	2.08	2.14	**2.13**
PaStiX LU	2.11	2.06	2.01	1.93	1.80	1.66	**1.93**
Panua Pardiso	2.19	2.17	2.14	2.10	2.04	2.02	**2.11**

Table 4.3: Empirical time complexity exponent α such that $T(\text{nnz}) \propto \text{nnz}^{\alpha}$, estimated from the slope of a linear regression of $\log T$ vs $\log(\text{nnz})$ across all problem sizes (n = 50 000, 90 000, 150 000, 300 000). Columns indicate the number of threads; the last column is the average across all thread counts. Cells are colored from green (lowest exponent, best) to red (highest exponent, worst).

According to the results, the empirical time complexity of all three solvers is around $O(\text{nnz}^2)$ across all thread counts. For performant sparse direct solvers, factorization is the task that dominates the time complexity and is known to scale as $O(\text{nnz}^2)$ for 3D PDE-derived problems which use the nested dissection algorithm [66, 67].

Since both PaStiX and MKL DSS are free, contrary to Panua Pardiso, one wants to determine in which cases one should be preferred over the other. Table 4.4 reports the ratio of the PaStiX LU PIF to the MKL DSS PIF across all mesh sizes and thread counts, where a ratio greater than one indicates that PaStiX is faster than MKL DSS. As shown, PaStiX LU outperforms MKL DSS at low thread counts across all mesh sizes, but its parallel efficiency degrades more rapidly as the thread count increases for small problem sizes. As the mesh size grows, PaStiX LU's parallel performance improves, and for the largest mesh (n = 300,000), it outperforms MKL DSS across all thread counts. Based on these observations, PaStiX LU is the preferred free direct solver for large problem sizes, while MKL DSS is recommended for smaller problems at high thread counts.



(a) $j = 1$

(b) $j = 2$

(c) $j = 4$

(d) $j = 8$

(e) $j = 16$

(f) $j = 32$

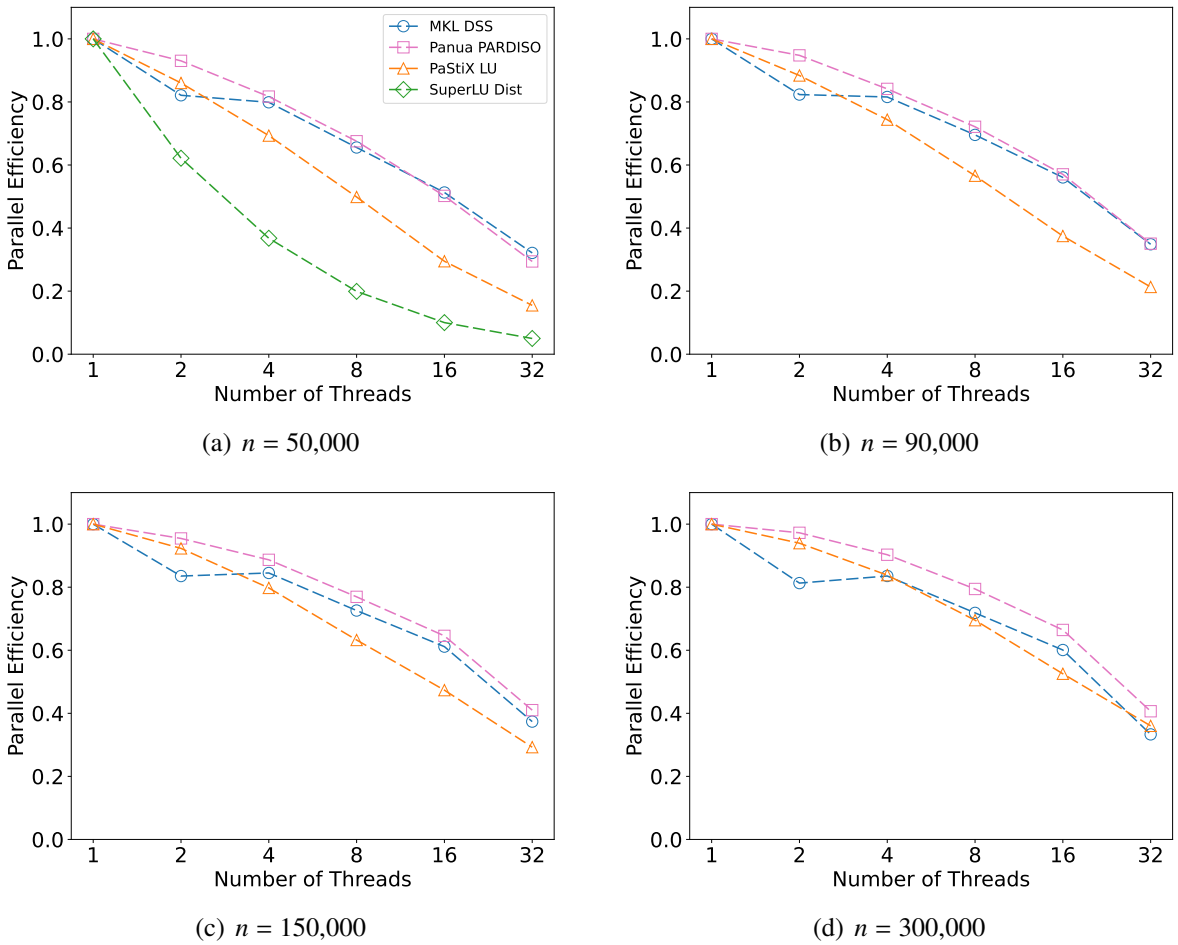
Figure 4.9: Total computation time of MKL DSS, PaStiX LU, and Panua Pardiso as a function of the number of non-zeros (NNZ), plotted on a log-log scale for each thread count j. Dashed lines indicate the fitted power-law model.

	$j = 1$	$j = 2$	$j = 4$	$j = 8$	$j = 16$	$j = 32$
$n = 50{,}000$	1.06	1.11	0.92	0.80	0.61	0.51
$n = 90{,}000$	1.12	1.20	1.02	0.91	0.75	0.68
$n = 150{,}000$	1.14	1.26	1.08	1.00	0.89	0.90
$n = 300{,}000$	1.18	1.36	1.18	1.14	1.03	1.27

Table 4.4: Ratio of PaStiX LU PIF to MKL DSS PIF across mesh sizes and thread counts. Green: PaStiX faster (ratio ≥ 1). Red: MKL DSS faster (ratio < 1).

4.2.4 Parallel efficiency

To compare the parallelization capabilities of the different solvers, the parallel efficiency $E(j)$ introduced in Equation (4.2) is employed.



(a) $n = 50{,}000$ (b) $n = 90{,}000$

(c) $n = 150{,}000$ (d) $n = 300{,}000$

Figure 4.10: Parallel efficiency as a function of the number of threads for different linear solvers. Each subfigure corresponds to a different mesh size.

Figure 4.10 presents the parallel efficiency as a function of the number of threads for each solver, evaluated across the four mesh sizes. All solvers exhibit decreasing parallel efficiency with increasing thread count, though the rate of decline depends on both the solver and the problem size. For the largest mesh (n = 300,000) at j = 32, Panua PARDISO achieves the highest efficiency (E = 0.41), followed by PaStiX LU (E = 0.36) and MKL DSS (E = 0.33). Panua PARDISO in particular demonstrates very strong parallel scaling, achieving E = 0.93 at j = 2 for n = 300,000 and maintaining the highest efficiency at every thread count among the tested solvers.

For the $n = 50,000$ particles mesh size, SuperLU parallel efficiency is convex, showing a bad parallel efficiency contrary to MKL DSS and Panua PARDISO, whereas PaStiX is in between, showing that PaStiX doesn't benefit completely from parallelization at this size. When the mesh becomes finer, the parallel efficiency of PaStiX becomes higher and more concave, the DSS remains the same and Panua PARDISO efficiencies slightly increase.

4.2.5 Memory Requirements

As was previously discussed in Section 3.2.5 and 4.2.1, the memory requirements of direct sparse LU solvers can become significantly high due to differences in the ordering algorithm used. In this section, the memory usage and the memory complexity of the different direct solvers are compared across the different mesh sizes.

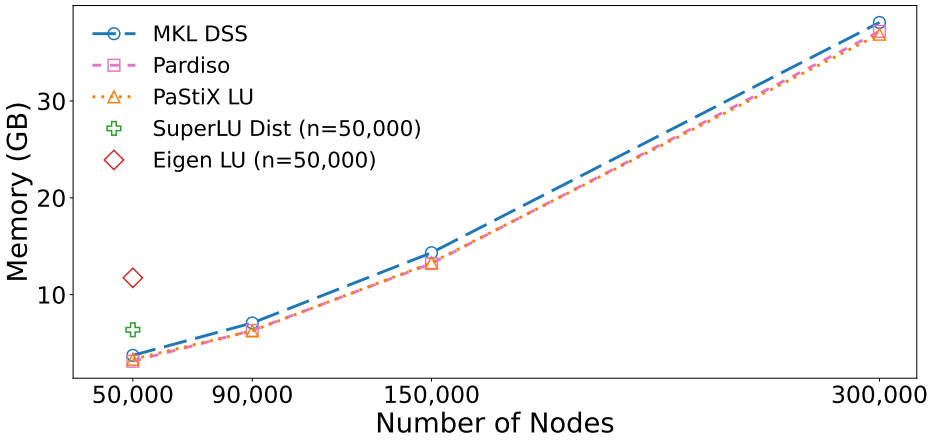


Figure 4.11: Memory consumption of the direct solvers as a function of the mesh size.

Figure 4.11 presents the peak memory consumption of the different solvers as a function of the mesh size. All three high performance direct solvers (MKL DSS, PaStiX LU, and Panua PARDISO) display very similar memory requirements and tendencies as a function of the mesh size, this is consistent with the fact that they all use the same ordering algorithm being the nested dissection. On the other hand, with a mesh size of $n = 50,000$ particles, Eigen LU and SuperLU Dist require significantly more memory than the other solvers, with Eigen using the COLAMD ordering and SuperLU using also the multilevel nested dissection from Metis. Since memory usage in direct solvers is largely dictated by the number of fill-ins, the ordering algorithm is of crucial importance when memory is limited [68].

The DSS of MKL requires slightly more memory than PaStiX and Panua PARDISO, nevertheless, all three solvers have similar asymptotic growth. Sparse direct solvers using the nested dissection algorithm should have a memory complexity of $O(nnz^{4/3})$ for 3D problems [69]. To investigate if this theoretical complexity is observed in practice, a power-law model similar to Section 4.2.3 was used to determine the exponent α of the empirical memory complexity $O(nnz^{\alpha})$. Figure 4.12 presents the peak memory consumption on a log-log scale, with the fitted power-law curves and an extrapolation to 256 GB.

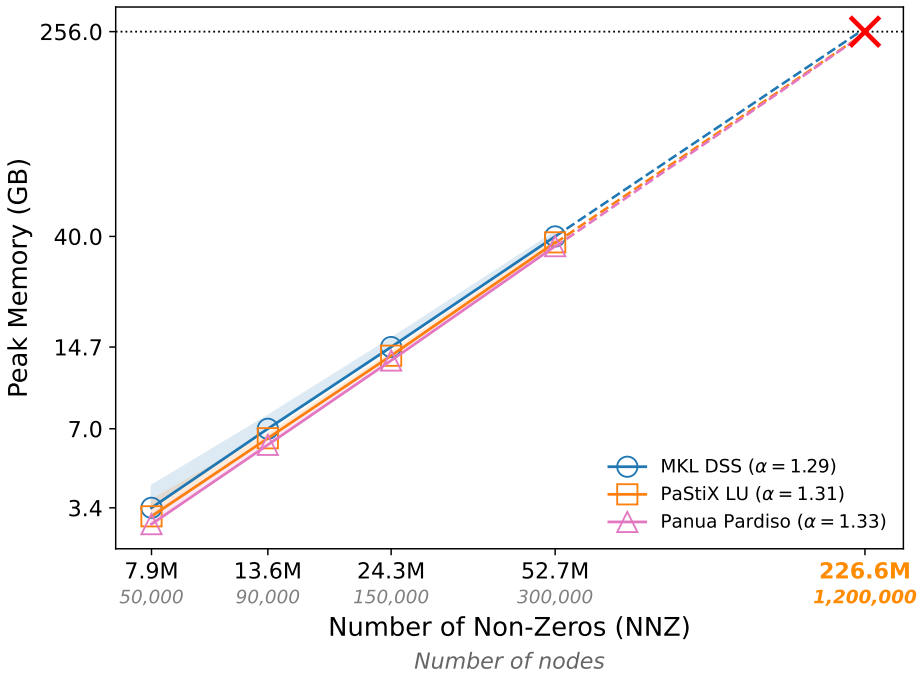


Figure 4.12: Log-log plot of peak memory consumption as a function of NNZ for the three direct solvers. Solid lines show measured data ($j = 1$), shaded bands span the min/max across all thread counts, dashed lines extrapolate the fitted power law, and the orange tick marks the estimated NNZ at which memory reaches 256 GB.

In Figure 4.12, all three solvers display linear trends in the log-log plot, indicating that the power-law model is a good fit for the empirical memory complexity. The estimated exponents α for MKL DSS, PaStiX LU, and Panua PARDISO are 1.29, 1.31 and 1.33. Using more threads, the memory increases slightly. Extrapolating these trends to the 256 GB memory limit of the computation platform used, the maximum problem size that can be solved is estimated to be around $n = 1,200,000$ particles for all three solvers, which would correspond to approximately an average of 226 million non-zeros in the linear system matrix. This mesh size was tested and the solvers were not able to factorize the matrix due to a lack of memory, confirming the validity of this extrapolation. As for the memory complexities, these empirical correlations were computed on all thread counts and are presented in Table 4.5.

Solver	1	2	4	8	16	32	Average
MKL DSS	1.29	1.29	1.28	1.27	1.25	1.21	1.27
PaStiX LU	1.31	1.27	1.29	1.28	1.28	1.25	1.28
Panua Pardiso	1.33	1.32	1.32	1.32	1.31	1.29	1.31

Table 4.5: Empirical memory complexity exponent α such that $M(\text{nnz}) \propto \text{nnz}^{\alpha}$, estimated from the slope of a linear regression of $\log M$ vs $\log(\text{nnz})$ across all problem sizes (n = 50 000, 90 000, 150 000, 300 000). Columns indicate the number of threads and the rightmost column is the average across all thread counts.

The memory complexities of all three solvers are very stable and similar, and they are around $O(nnz^{1.3}) \in O(nnz^{4/3})$, confirming the theoretical complexity for 3D problems. To derive a practically usable law for PFEM3D problems, an empirical upper bound $M_{\text{max,direct}}$ is fitted as a function of the number of particles n since the number of non-zeros is unavailable prior to factorization. From the maximum memory observations in Figure 4.12:

$$M_{\text{max,direct}}(n) \leq C \cdot n^{1.33}, \tag{4.5}$$

where C is the fitted prefactor, with the 256 GB limit reached at approximately $n = 1.2 \times 10^6$ particles.

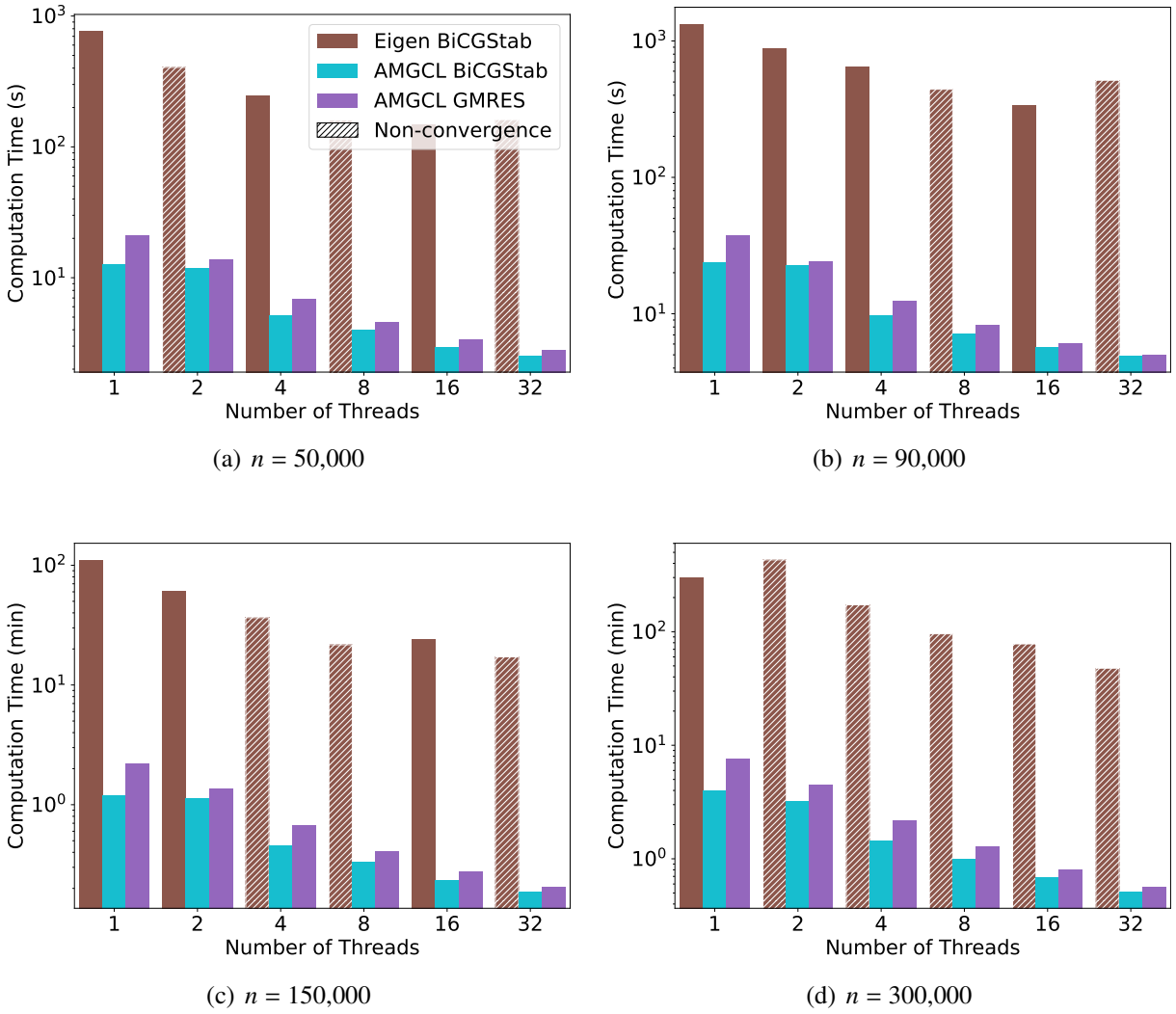
4.3 Iterative Solvers

This section investigates the performance of iterative solvers as alternatives to the direct solvers presented in the previous section. Subsection 4.3.1 first establishes a baseline comparison of the three iterative solvers interfaced in PFEM3D with minimal parameter tuning. Subsection 4.3.2 then investigates the influence of the convergence tolerance on computation time and nonlinear iteration count. Subsection 4.3.3 examines the effect of the ILUT preconditioner parameters on convergence. Subsection 4.3.4 compares unrestarted GMRES against its restarted version. Finally, the memory requirements are examined in Subsection 4.3.5.

4.3.1 Loose Tolerance with Native Preconditioners

In contrast to direct solvers, which have few user-facing parameters, iterative solvers expose a wide range of tunable settings, including the solver algorithm, the preconditioner type, and the convergence tolerance. Before investigating the effect of each of these parameters individually, this section first establishes a baseline comparison of the three iterative solvers interfaced in PFEM3D: Eigen BiCGStab with Jacobi preconditioning, AMGCL BiCGStab with ILU(0) preconditioning, and AMGCL GMRES with ILU(0) preconditioning. These preconditioners are the simplest available in PFEM3D, as they require no parameter tuning and are used directly as provided by each library. The convergence tolerance of 10^{-16} will be fixed in this section in order to compare direct solvers with the worst case for iterative solvers.

Figure 4.13 presents the total computation time of the three iterative solvers as a function of the number of threads for each mesh size. Hatched bars indicate that the solver failed to converge during at least one of the time steps.



(a) n = 50,000

(b) n = 90,000

(c) n = 150,000

(d) n = 300,000

Figure 4.13: Total computation time of the three iterative solvers as a function of the number of threads. Hatched bars indicate that at least one time step encountered a non-convergence. Each subfigure corresponds to a different mesh size.

As shown in Figure 4.13, the AMGCL solvers with ILU(0) preconditioning are consistently one to two orders of magnitude faster than Eigen BiCGStab with Jacobi preconditioning across all mesh sizes and thread counts. Furthermore, Eigen BiCGStab fails to converge for at least one time step at most thread counts for every mesh size. When a non-convergence occurs, Eigen returns an error indicating that the solver wasn't able to satisfy the convergence tolerance. PFEM3D then subdivides the time step and retries, which further increases the total computation time. This poor convergence behavior is attributed to the Jacobi preconditioner, which consists only of the diagonal of $\mathbf{A}$ and is too weak to effectively cluster the eigenvalues of the preconditioned system. The ILU(0)

preconditioner used by the AMGCL solvers retains the full nonzero structure of $\mathbf{A}$, providing a much better approximation of the inverse and dramatically reducing the iteration count.

To quantify the performance gap between iterative and direct solvers, Figure 4.14 presents the Performance Increase Factor (PIF) of MKL DSS, AMGCL BiCGStab, and AMGCL GMRES relative to Eigen BiCGStab at a single thread for n = 300,000 particles, along with the peak memory required by each solver.

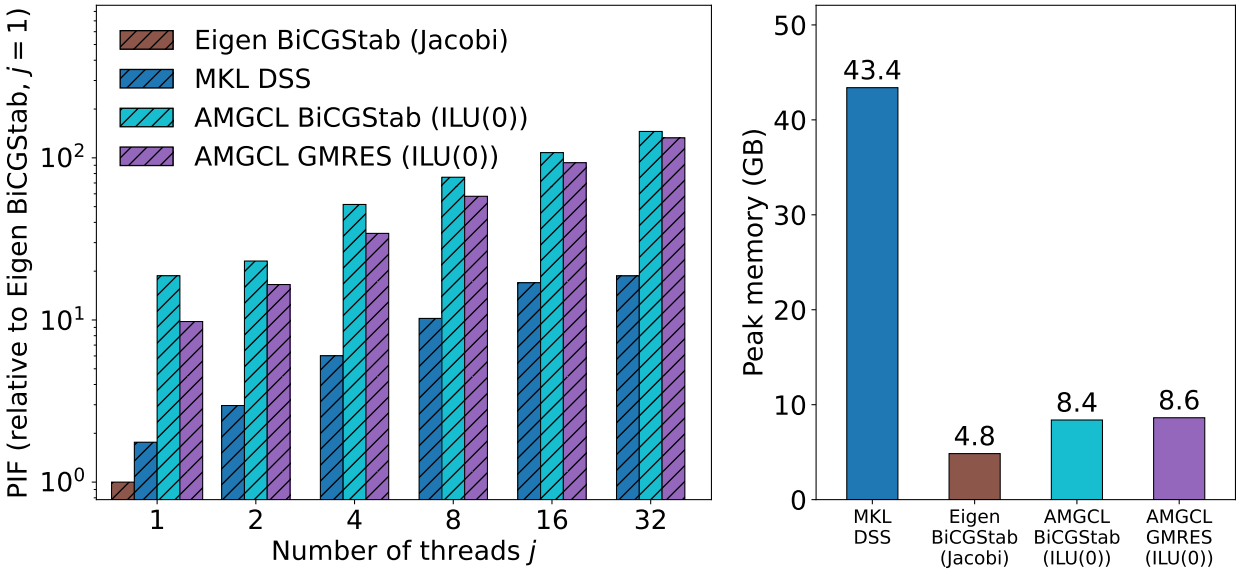


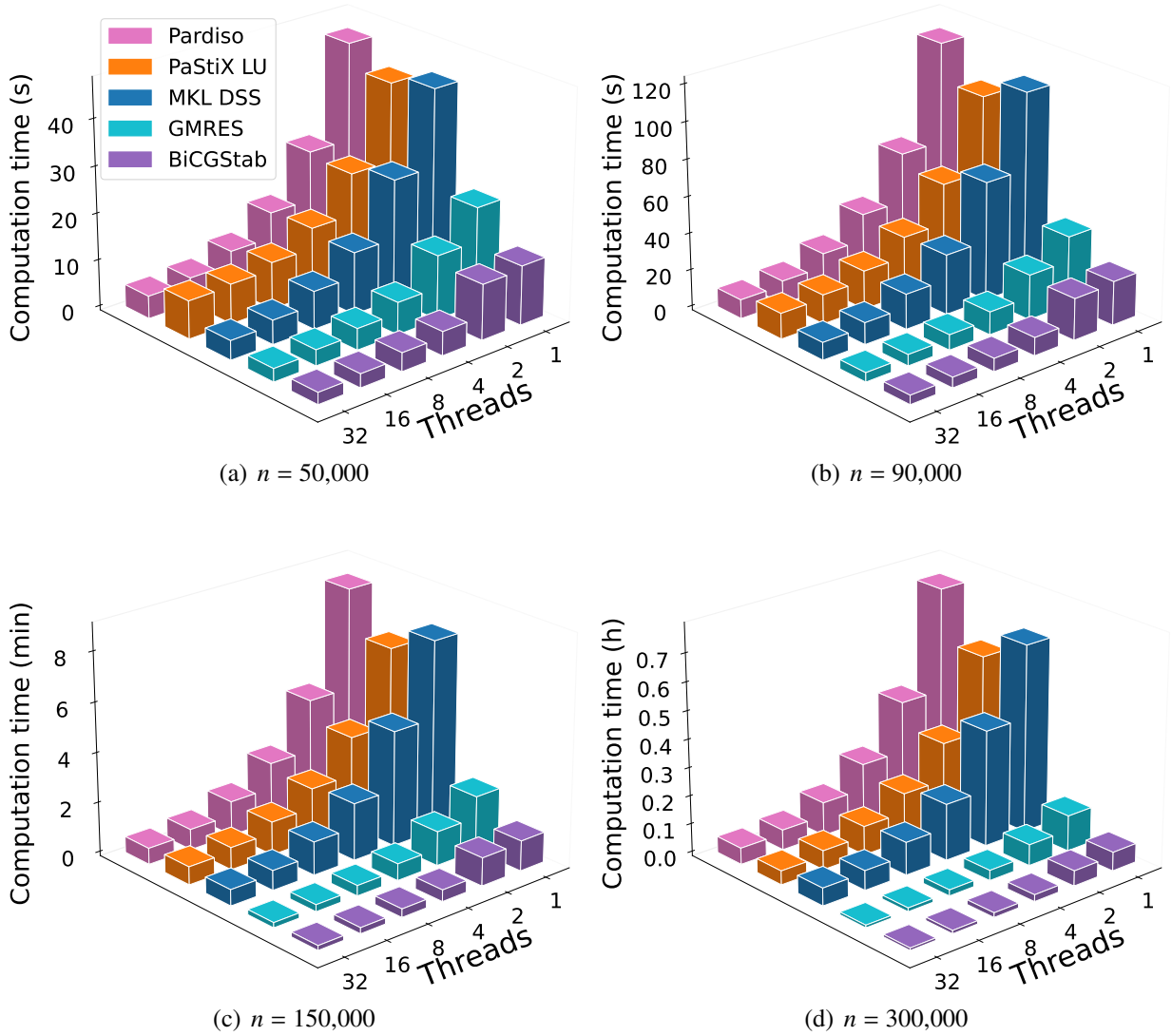
Figure 4.14: Performance Increase Factor relative to Eigen BiCGStab (j = 1) as a function of the number of threads (left) and peak memory requirement (right), for n = 300,000 particles.

As shown in Figure 4.14, the AMGCL solvers vastly outperform both Eigen BiCGStab and MKL DSS. At 32 threads, AMGCL BiCGStab achieves a PIF of approximately 146 and AMGCL GMRES reaches approximately 133 relative to single-threaded Eigen BiCGStab, whereas MKL DSS only reaches a PIF of approximately 19. Even at a single thread, AMGCL BiCGStab already achieves a PIF of approximately 19, matching the performance of MKL DSS at 32 threads while using a single core.

The right panel of Figure 4.14 reveals an equally significant advantage in memory consumption. MKL DSS requires 43.4 GB of peak memory at n = 300,000, whereas the AMGCL solvers require only 8.4 GB and 8.6 GB for BiCGStab and GMRES respectively, representing a reduction by a factor of approximately 5. Eigen BiCGStab uses even less memory at 4.8 GB due to its diagonal Jacobi preconditioner, but as demonstrated above, this comes at the cost of unacceptable computational performance and robustness. These results confirm that the ILU(0)-preconditioned AMGCL solvers provide the best trade-off between computation time, memory usage, and convergence reliability for the linear systems arising in PFEM3D.

To further contextualise this advantage against the direct solvers evaluated in Section 4.2, Figure 4.15 presents the total computation time of all evaluated solvers, both direct and iterative, as a function

of the number of threads for each mesh size.



(a) $n = 50{,}000$ (b) $n = 90{,}000$

(c) $n = 150{,}000$ (d) $n = 300{,}000$

Figure 4.15: Total computation time of all direct and iterative solvers as a function of the number of threads for each mesh size. Each subfigure corresponds to a different mesh size.

As shown in Figure 4.15, the AMGCL solvers with ILU(0) preconditioning consistently outperform the direct solvers at every thread count and mesh size. The performance gap widens as the number of particles increases: at $n = 300{,}000$, the iterative solvers are faster than the best direct solver by approximately one order of magnitude, confirming the advantage of iterative methods for the linear systems arising in PFEM3D.

4.3.2 Tolerance Setting

Until now, all iterative solver results have been obtained using a convergence tolerance of 10^{-14} to ensure the best possible robustness. For direct solvers, each nonlinear iteration requires a new

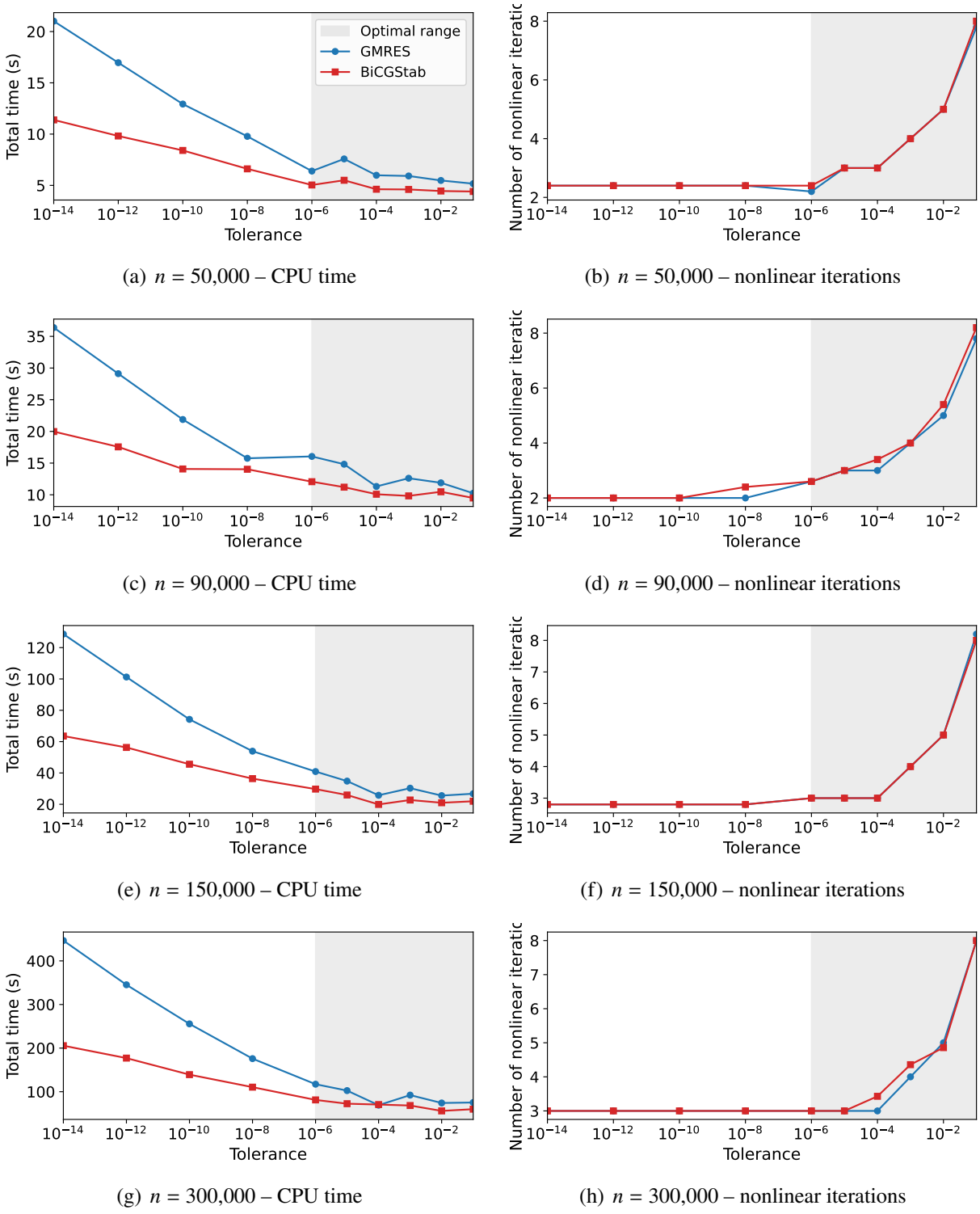
matrix factorization, so that increasing the number of nonlinear iterations leads to very high computation times without any benefits. However, for iterative solvers, factorization is not performed, and the main cost corresponds to the number of iterations required to reach convergence [18, 70]. This section investigates whether using higher tolerance can benefit positively the performance of the simulation without compromising on the robustness.

Indeed, the selection of an appropriate tolerance involves a trade-off between accuracy and convergence. If the tolerance is set too small, iterative solvers may fail to converge [71]. Conversely, a tolerance that is excessively large yields imprecise solutions, which in turn requires additional nonlinear iterations to satisfy the convergence criterion of the outer Newton-Raphson algorithm. Since each nonlinear iteration requires the solution of a linear system, a loose tolerance effectively increases the total number of linear systems to be solved [72].

On the five time steps and four mesh sizes considered, the tolerance was varied from 10^{-14} to 10^{-1} using AMGCL BiCGStab and unrestarted GMRES with ILU(0) preconditioners, with a single thread. The resulting computation times and numbers of nonlinear iterations are presented in Figure 4.16.

As can be seen in Figure 4.16, as the convergence tolerance is relaxed from 10^{-14}, the total CPU time decreases monotonically while the number of nonlinear iterations remains constant at approximately 10 to 14 iterations per time step. This plateau extends from 10^{-14} to 10^{-6} for all mesh sizes considered (n = 50,000, 90,000, 150,000, and 300,000), showing that, within this range, the prescribed linear tolerance is unnecessarily tight: the linear solver performs far more work than required to maintain the nonlinear convergence behavior.

Beyond 10^{-6}, the nonlinear iteration count starts to increase for all mesh sizes, reaching around 35 to 40 iterations at 0.1. For n = 300,000, this increase begins slightly earlier, at 10^{-5}. Despite the growing number of nonlinear iterations, the total CPU time continues to decrease slightly or remains stable within $[10^{-6}, 0.1]$, as the reduction in linear work per system more than compensates for the extra nonlinear iterations. The optimal tolerance range is therefore $[10^{-6}, 0.1]$, where the computation time is low and the nonlinear convergence is not significantly affected. This behaviour is illustrated in Figure 4.17.



(a) n = 50,000 – CPU time

(b) n = 50,000 – nonlinear iterations

(c) n = 90,000 – CPU time

(d) n = 90,000 – nonlinear iterations

(e) n = 150,000 – CPU time

(f) n = 150,000 – nonlinear iterations

(g) n = 300,000 – CPU time

(h) n = 300,000 – nonlinear iterations

Figure 4.16: Total CPU time (left column) and number of nonlinear iterations (right column, with CPU time shown in grey on the secondary axis) as a function of the convergence tolerance for BiCGStab and GMRES with ILU(0) preconditioner (j = 1 thread). Rows correspond to mesh sizes of n = 50,000, 90,000, 150,000, and 300,000 particles. The shaded region indicates the identified optimal tolerance range $[10^{-6}, 10^{-1}]$.

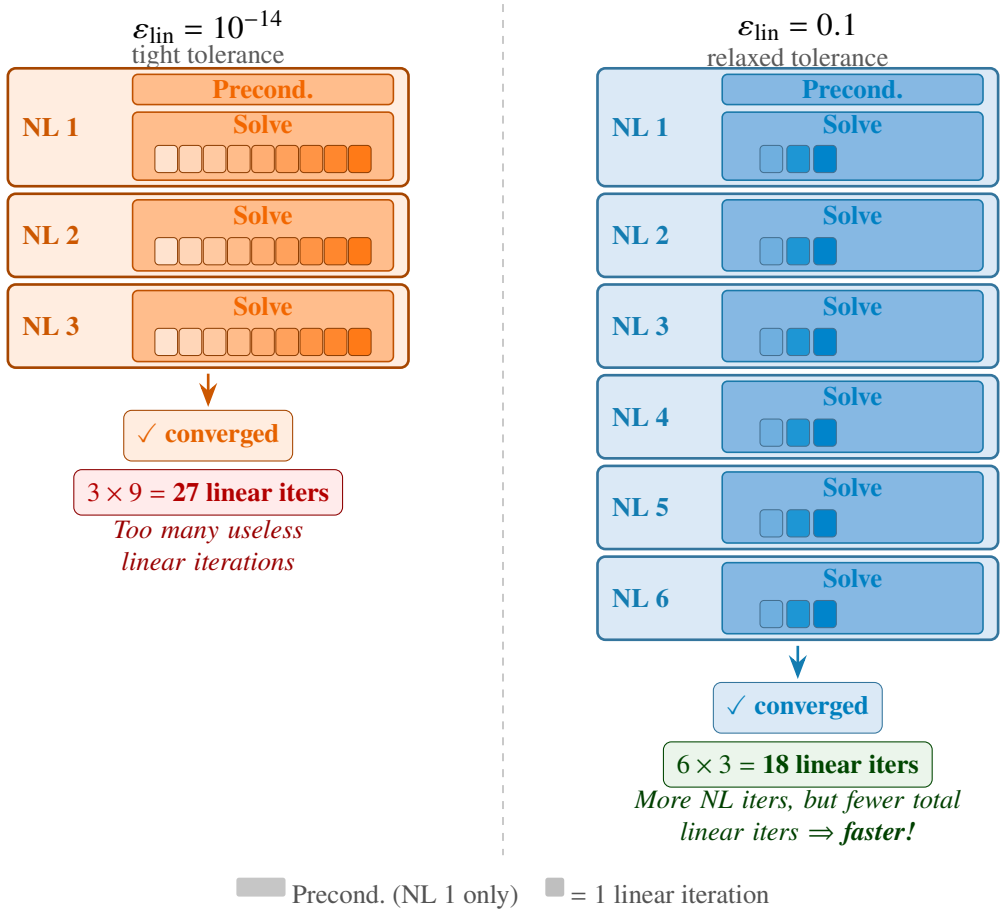
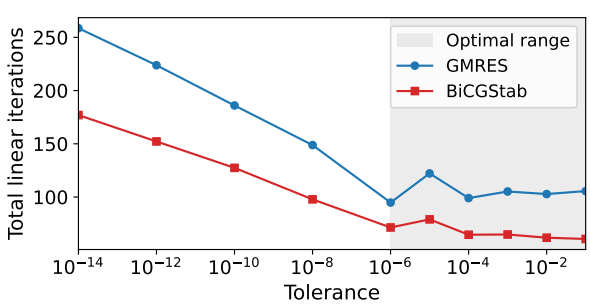


Figure 4.17: Schematic illustration of the effect of linear solver tolerance on Newton–Raphson convergence. Each row represents one nonlinear iteration; coloured squares denote linear iterations required to solve the corresponding linear system. A tight tolerance ($\varepsilon_{\text{lin}} = 10^{-14}$) yields few nonlinear iterations but many linear iterations per system, while a relaxed tolerance ($\varepsilon_{\text{lin}} = 0.1$) increases the nonlinear iteration count but reduces the number of linear iterations per system.

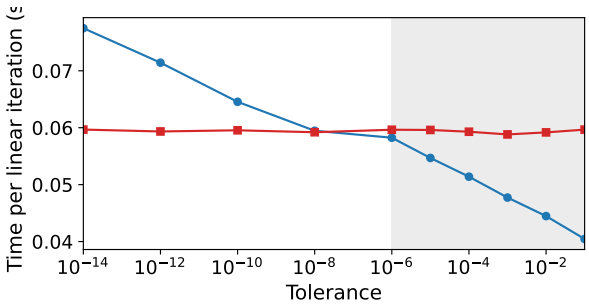
As observed in Figure 4.16, AMGCL BiCGStab consistently outperforms the unrestarted GMRES across all mesh sizes considered. To understand the origin of this performance difference, the total solve time T is decomposed into two contributions: the number of linear iterations required to solve the i-th system, k_i, and the computational cost per linear iteration, c_i. The total solve time is thus expressed as

$$T = \sum_i k_i \cdot c_i, \tag{4.6}$$

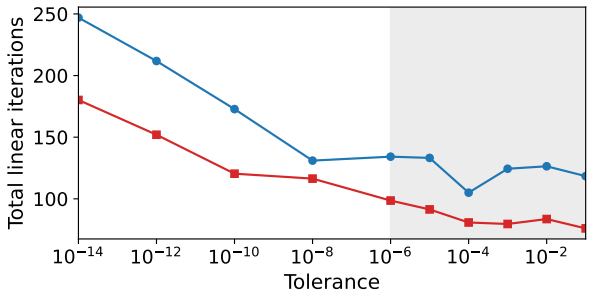
following the cost model of [70]. Figure 4.18 presents both the average number of linear iterations per system $\bar{k}$ and the average cost per linear iteration $\bar{c}$ as a function of the convergence tolerance for BiCGStab and GMRES with ILU(0) preconditioner.



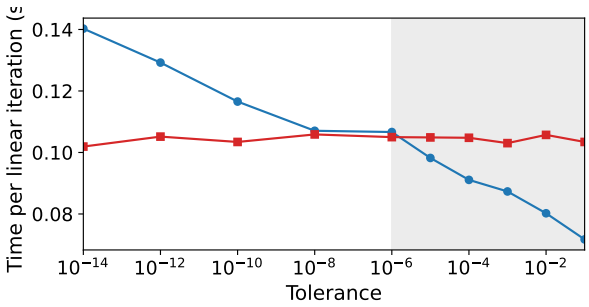
(a) n = 50,000 – avg. linear iterations per system



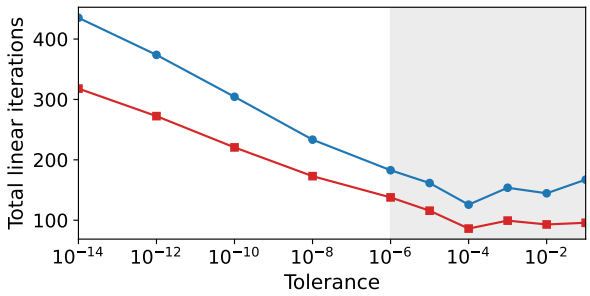
(b) n = 50,000 – time per linear iteration



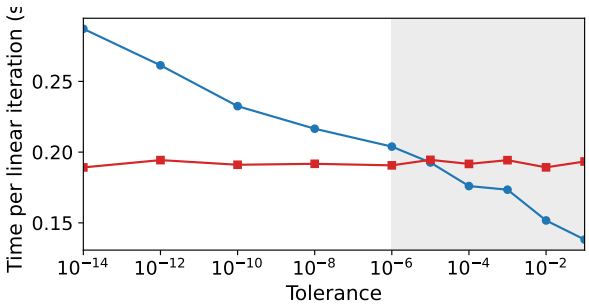
(c) n = 90,000 – avg. linear iterations per system



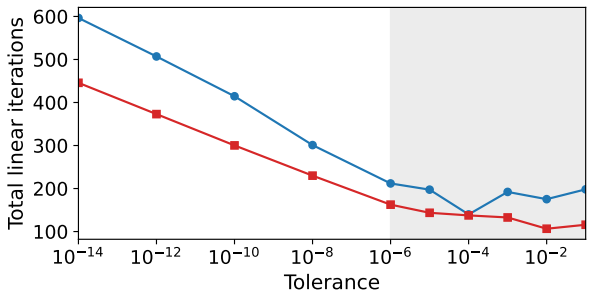
(d) n = 90,000 – time per linear iteration



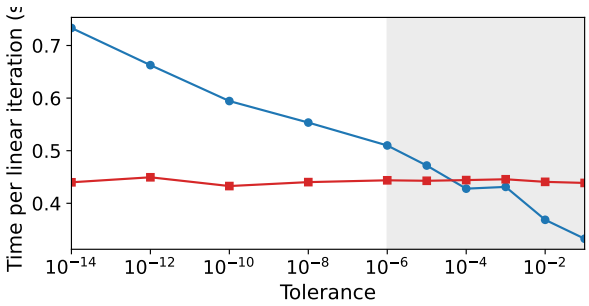
(e) n = 150,000 – avg. linear iterations per system



(f) n = 150,000 – time per linear iteration



(g) n = 300,000 – avg. linear iterations per system



(h) n = 300,000 – time per linear iteration

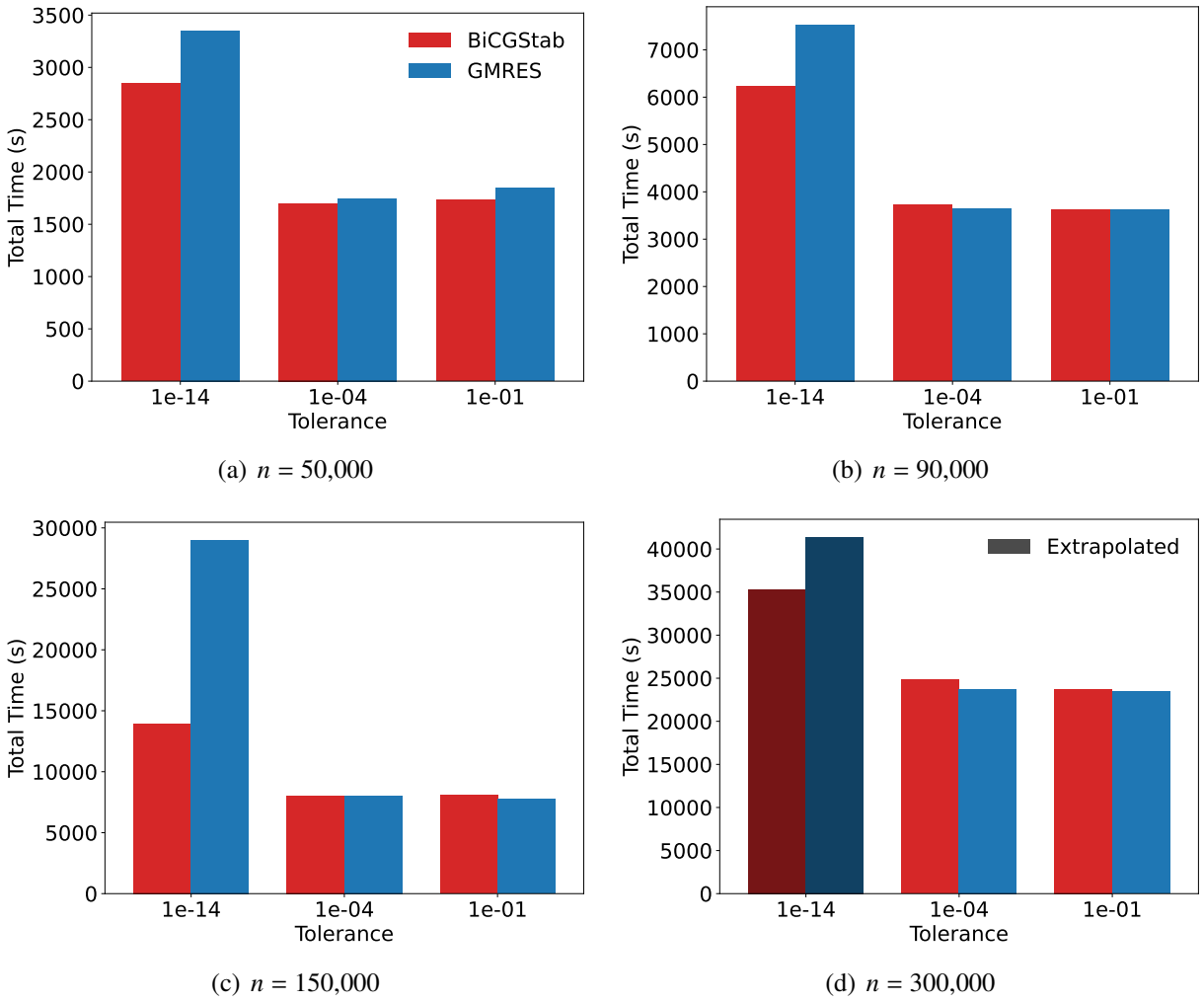
Figure 4.18: Average number of linear iterations per system (left column) and time per linear iteration (right column) as a function of the convergence tolerance for BiCGStab and GMRES with ILU(0) preconditioner (j = 1 thread). Rows correspond to mesh sizes of n = 50,000, 90,000, 150,000, and 300,000 particles. The shaded region indicates the identified optimal tolerance range $[10^{-6}, 10^{-1}]$.

As one can see in Figure 4.18, the cost per linear iteration is constant for BiCGStab but increases with the iteration count for unrestarted GMRES. This comes from the Arnoldi process used in GMRES: at iteration k, the new basis vector must be orthogonalised against all $k-1$ previously stored vectors, so that the cost per iteration grows as $O(\text{nnz} + kn)$, where nnz is the number of nonzero entries of $\mathbf{A}$ and n the system size [73]. On the other hand, BiCGStab relies on short recurrences: each iteration updates the solution from only the vectors of the previous step, requiring exactly two sparse matrix-vector products with $\mathbf{A}$, so that the cost per iteration is a constant $O(\text{nnz})$ regardless of the iteration count [18]. At relaxed tolerances, where few iterations are needed, the per-iteration cost of GMRES is low. However, as the tolerance is tightened and the iteration count increases, the GMRES cost grows until it exceeds that of BiCGStab, which explains in part its lower performance observed in Figure 4.16.

Nonetheless, as it can be seen on the left side of the figure, the total number of linear iterations achieved by the BiCGStab algorithm is always lower than that of GMRES. Thus, it means that the superiority of the BiCGStab algorithm is here due to the convergence rate of the method which is better for this test case. The selected tolerances for both solvers will therefore be fixed to 10^{-4}, the reason being that it is contained in the optimal range to minimize computation time, and also that we want to avoid to distribute too much the work of the linear iterations across the different nonlinear iterations, which would make it difficult to optimize even further the preconditioner or other parameters such as the restart of GMRES.

While the five-time-step experiments suggest that a tolerance of 10^{-4} is appropriate, iterative solvers are susceptible to breakdown over the course of a full simulation like the BiCGStab with the Jacobi preconditioner [18, 71, 74]. Complete simulations are therefore required to verify that the selected tolerance yields robust convergence across all time steps.

Figure 4.19 presents the total CPU time for full simulations at tolerances of 10^{-14}, 10^{-4}, and 10^{-1} across all four mesh sizes. One can see that all simulations converged across all mesh sizes and both solvers. The best computation times are consistently achieved at tolerances within the optimal range, while the computation times at 10^{-14} are significantly higher. Although BiCGStab demonstrated superior performance over unrestarted GMRES in the five-time-step experiments, both solvers yield comparable computation times at the relaxed tolerances. However, a notable discrepancy is visible for $n = 150{,}000$ particles where the unrestarted GMRES at the 10^{-14} tolerance reaches approximately twice the computation time of BiCGStab, contrary to other tolerances where both solvers do not struggle to converge.



(a) $n = 50{,}000$

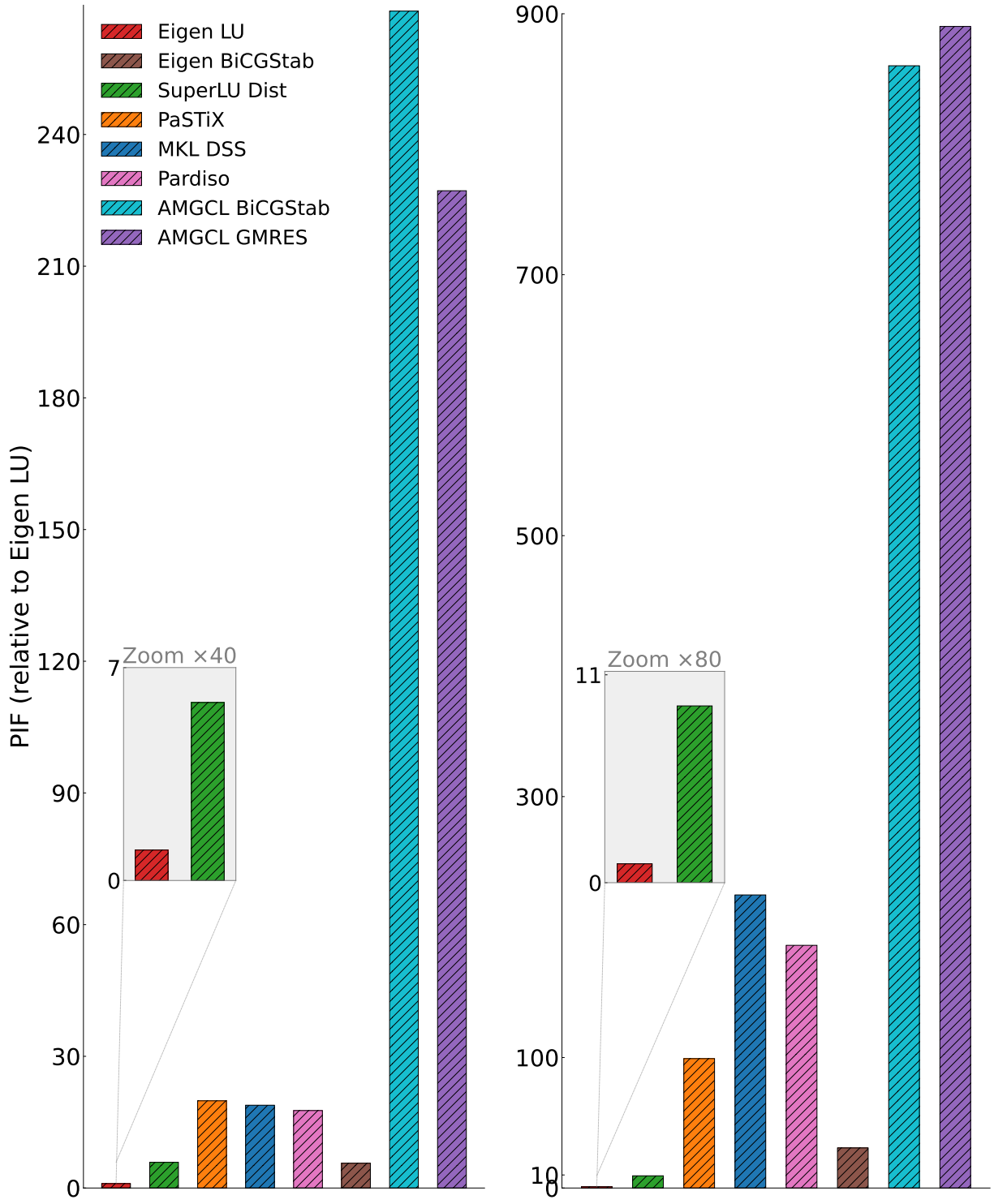
(b) $n = 90{,}000$

(c) $n = 150{,}000$

(d) $n = 300{,}000$

Figure 4.19: Simulation time of complete PFEM3D simulations for the selected tolerance values across all mesh sizes (n = 50,000, 90,000, 150,000, and 300,000 particles). The shaded region indicates the tolerance range $[10^{-6}, 10^{-1}]$. Darker bars are extrapolated from five specific time instants.

The improvement on the computation time thanks to the selected tolerance can be compared to the time required by other linear solvers. Figure 4.20 presents the Performance Improvement Factor (PIF) relative to Eigen LU for all linear solver libraries considered in this study, evaluated at $j = 1$ and $j = 32$ threads.



(a) $j = 1$ thread

(b) $j = 32$ threads

Figure 4.20: Performance Improvement Factor (PIF) relative to Eigen LU for all solver libraries at $j = 1$ thread (a) and $j = 32$ threads (b). Hatched bars denote iterative solvers. An inset provides a zoomed view of the direct solver range for readability.

At $j = 1$ thread, AMGCL BiCGStab achieves the highest PIF of approximately 260, followed by AMGCL GMRES at approximately 225, while all direct solvers remain below a PIF of 20. At $j = 32$ threads, both AMGCL solvers scale substantially further, reaching PIFs of approximately 870 and 890 for BiCGStab and GMRES, respectively, whereas the parallel direct solvers yield comparatively modest improvements. The original simulation that would take 10 days to be run, took only 28 minutes when using the GMRES of AMGCL with a tolerance of 10^{-4}.

4.3.3 ILUT Preconditioners

As introduced in Subsection 3.3.9, the ILUT preconditioner is characterized by two parameters that control the trade-off between preconditioner quality and construction cost. The first parameter is the drop tolerance τ: an element a_{ij} is discarded during the incomplete factorization if

$$\frac{|a_{ij}|}{\|\mathbf{a}_i\|_2} < \tau, \tag{4.7}$$

so that a larger τ yields sparser factors, faster construction, and slower convergence. The second parameter p is the maximum number of fill-in elements retained per row in both $\mathbf{L}$ and $\mathbf{U}$, with the smallest entries discarded first; a larger p permits denser factors and potentially faster convergence at the cost of a more expensive construction. [18]

With AMGCL, the drop tolerance τ can be directly specified, while the maximum number of fill-in is controlled through the fill-factor ρ [53]. For each row i, the maximum number of retained elements in each triangular factor is:

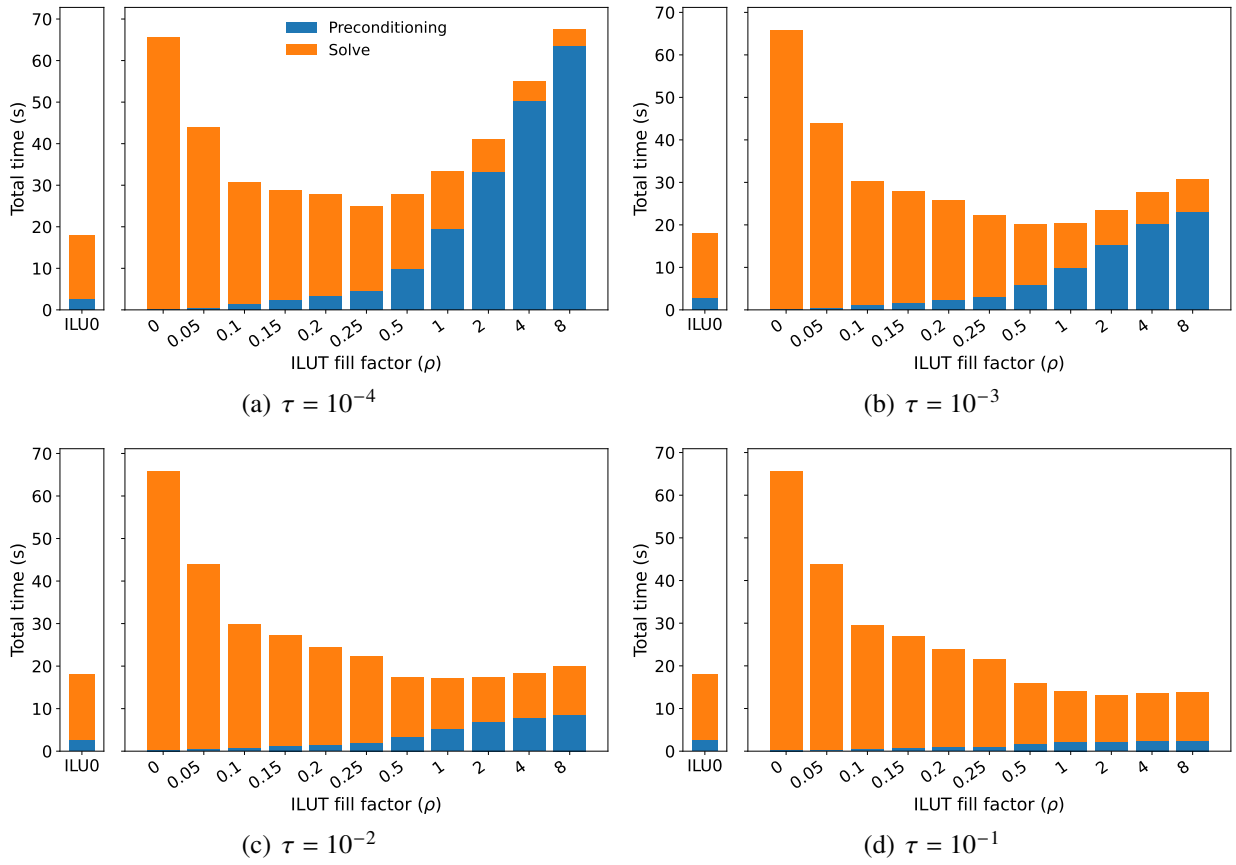
$$p_l = \rho \cdot l_i, \qquad p_u = \rho \cdot u_i, \tag{4.8}$$

where l_i and u_i denote the number of nonzero elements below and above the diagonal in the i-th row of $\mathbf{A}$. The limiting case $\rho = 0$ retains only diagonal elements, which is equivalent to a Jacobi preconditioner, while $\rho = 1$ permits at most the same number of nonzeros per row as $\mathbf{A}$ itself.

It should be noted that ILUT($\tau = 0, \rho = 1$) is not equivalent to ILU(0). ILU(0) preserves the exact sparsity pattern of $\mathbf{A}$, discarding all fill-in regardless of magnitude. ILUT instead retains the p largest entries per row: a large fill-in element may displace a small original entry, resulting in a different sparsity pattern from $\mathbf{A}$ even at $\rho = 1$.

To determine optimal values for τ and ρ, simulations were performed with unrestarted GMRES at tolerance $\varepsilon = 10^{-4}$ using $j = 1$ thread. The results presented in Figure 4.21 show that reducing τ and increasing ρ yields denser incomplete factors, which improves preconditioner quality and reduces the solve time. However, denser factors also require a more expensive factorization, so the preconditioning cost increases significantly with ρ, as evidenced by the growing blue portion of the bars. Beyond an optimal value of ρ, the gains in solve time are outweighed by the construction overhead, and the total computation time increases. ILU(0), shown as a reference, already achieves approximately 90 s, which is close to the best ILUT configuration observed at $\tau = 10^{-2}$, $\rho = 2$ (≈ 65 s). The choice of τ and ρ can dramatically affect the total computation time: at $\rho = 8$, for instance, a change in τ from 10^{-2} to 10^{-3} increases the total time from approximately 70 s to 335 s.

Unfortunately, no general rule exists to determine optimal values of τ and ρ a priori [18], and selecting these parameters requires empirical tuning for each problem. Given that ILU(0) delivers near-optimal performance without requiring any parameter tuning, using this preconditioner instead of ILUT for PFEM3D simulations is preferred, as it minimizes the risk of inadvertently selecting detrimental values of τ and ρ.

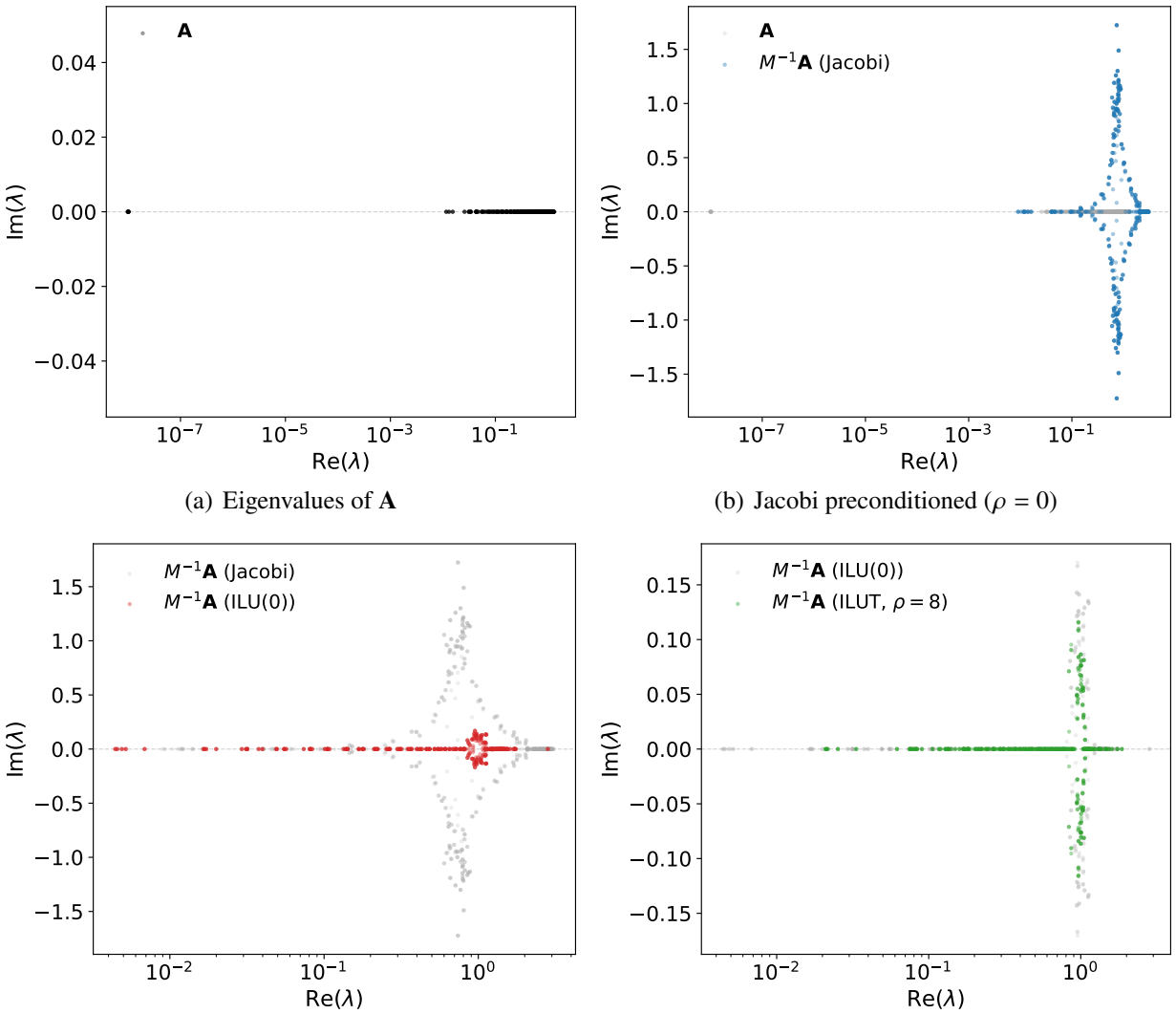


(a) $\tau = 10^{-4}$ (b) $\tau = 10^{-3}$

(c) $\tau = 10^{-2}$ (d) $\tau = 10^{-1}$

Figure 4.21: Total computation time as a function of the ILUT fill factor ρ for four values of the drop tolerance τ (n = 50,000 particles, j = 1 thread). The ILU(0) result is shown as a reference on the left of each panel.

A natural question arising from these observations is how the choice of preconditioner affects the number of iterations required to reach convergence, for which the answer lies in the eigenvalue distribution of the preconditioned matrix $\mathbf{M}^{-1}\mathbf{A}$ [18]. The eigenvalue distributions obtained with the three preconditioners considered are presented in Figure 4.22.



(a) Eigenvalues of $\mathbf{A}$

(b) Jacobi preconditioned ($\rho = 0$)

(c) ILU(0) preconditioned

(d) ILUT preconditioned ($\rho = 8$)

Figure 4.22: Eigenvalue distribution of the system matrix $\mathbf{A}$ and the preconditioned matrices $\mathbf{M}^{-1}\mathbf{A}$ for three preconditioners: Jacobi ($\rho = 0$), ILU(0), and ILUT with fill factor $\rho = 8$. All eigenvalues are pooled across multiple checkpoints ($n = 50{,}000$ particles, $j = 32$ threads).

The eigenvalue distribution of the unpreconditioned system matrix $\mathbf{A}$ and preconditioned matrices $\mathbf{M}^{-1}\mathbf{A}$ is presented in Figure 4.22. The eigenvalues of $\mathbf{A}$ are purely real. In contrast, the preconditioned matrices $\mathbf{M}^{-1}\mathbf{A}$ exhibit complex eigenvalues. The resulting matrix $\mathbf{M}^{-1}\mathbf{A}$ is a generic non-symmetric real matrix, whose eigenvalues are complex in general, appearing in conjugate pairs since $\mathbf{M}^{-1}\mathbf{A} \in \mathbb{R}^{n\times n}$. It can be observed that the range of the x-axis decreases as the preconditioner becomes denser, showing the progressive clustering of the eigenvalues. Preconditioning transforms $\mathbf{A}$ such that $\mathbf{M}^{-1}\mathbf{A}$ approximates the identity matrix, whose eigenvalues are all equal to unity. This is evidenced by the progressive convergence of the eigenvalue distribution toward $(1, 0) \in \mathbb{C}$, from the Jacobi preconditioner to ILU(0) and finally ILUT with $\rho = 8$. As $\mathbf{M}^{-1}\mathbf{A}$ approaches the identity, the preconditioned system $\mathbf{M}^{-1}\mathbf{A}\mathbf{x} = M^{-1}\mathbf{b}$ approaches $\mathbf{x} = M^{-1}\mathbf{b}$, meaning the solution is directly available from the right-hand side without any iteration. Clustering of eigenvalues around $(1, 0)$

therefore directly reduces the number of iterations required to reach convergence.

This observation draws a parallel with Gaussian elimination, presented in Subsection 3.2.2: row elimination transforms $\mathbf{A}$ by applying linear combinations of its rows, while the same operations are applied to the right-hand side $\mathbf{b}$. Carried to completion, this process transforms $\mathbf{A}$ into the identity matrix, making the solution directly readable from the right-hand side. Preconditioning can therefore be seen as computing an approximation of $\mathbf{A}^{-1}$ prior to solving, which is then refined iteratively until convergence. This is precisely what sparse direct solver libraries do: they compute an exact factorization of $\mathbf{A}$ before applying iterative refinement to correct for numerical errors introduced during the factorization.

In this light, a direct solver can be seen as a preconditioner so effective that the solution is obtained in very few iterations. This is illustrated in Figure 4.23.

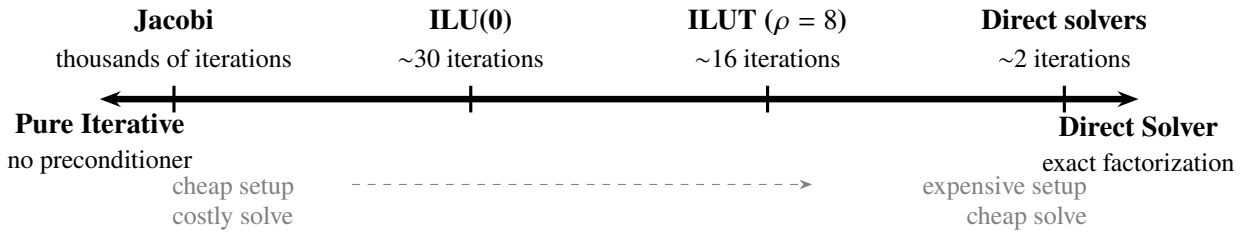


Figure 4.23: The direct-iterative continuum: as the quality of the preconditioner improves, the number of iterations required to reach convergence decreases. The iteration counts are approximate values observed at n = 50,000 particles.

Figure 4.23 illustrates the direct-iterative continuum: as the preconditioner quality improves, the eigenvalues of $\mathbf{M}^{-1}\mathbf{A}$ cluster more tightly around $(1, 0)$ and fewer iterations are required to reach convergence, until the preconditioner is so effective that the solution is obtained in a single factorization step. Solvers such as PaStiX and SuperLU employ static pivoting, which replaces near-zero diagonal entries by a perturbed value, giving resulting factors $\mathbf{L}$ and $\mathbf{U}$ that only approximate the original matrix, requiring additional iterative refinement steps to reach the solution [49, 50]. In this framework, these solvers can therefore be seen as less effective preconditioners than an exact factorization such as MKL DSS. It should however be noted that, in the context of PFEM3D simulations, the nonlinearity of the system implies that the matrix $\mathbf{A}$ changes at every Newton-Raphson iteration, preventing direct solvers from reusing their factorization. Iterative solvers, on the other hand, can reuse the preconditioner from the previous iteration as an initial approximation, which contributes to their superior performance in this setting.

4.3.4 Restarted GMRES

As established in Section 4.3.2, the cost per iteration of unrestarted GMRES grows with the iteration count. The objective of this section is to investigate whether restarted GMRES can outperform BiCGStab by simultaneously achieving a low total number of linear iterations, as observed for unrestarted GMRES at $\varepsilon = 10^{-4}$ (see Figure 4.18), while maintaining a low cost per iteration, as

obtained at tolerances $\varepsilon \gg 10^{-4}$.

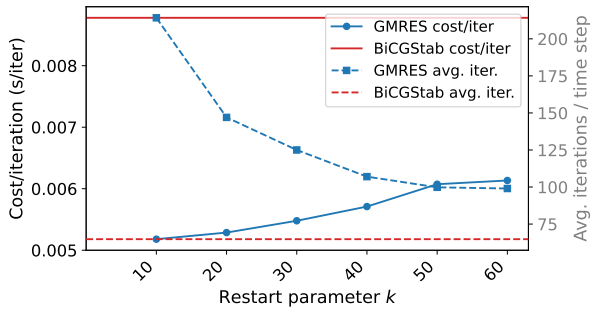
The GMRES algorithm can be restarted after a fixed number of iterations in order to limit both the cost per iteration and the memory requirements associated with storing the orthonormal basis of the Krylov subspace as discussed in Section 3. The restarted variant, denoted GMRES(k), discards the accumulated basis vectors after k iterations and restarts the algorithm with the current approximate solution as the new initial guess. Simulations are conducted for the four mesh sizes with the restart parameter k varied starting from 10 until it reaches a value equivalent in which no restart is done (unrestarted).

The left column of Figure 4.24 presents, for each mesh size, the cost per linear iteration and the average iteration count per time step as a function of the restart parameter k. Two opposing effects are observed as k is reduced. First, the cost per iteration decreases, since the Arnoldi orthogonalisation is performed against a basis of at most k vectors, reducing the associated $O(\text{nnz} + kn)$ cost per step. Second, the average number of iterations per time step increases sharply, as each restart discards the accumulated Krylov subspace information, forcing the algorithm to rebuild approximation quality from a zero initial correction at each cycle.

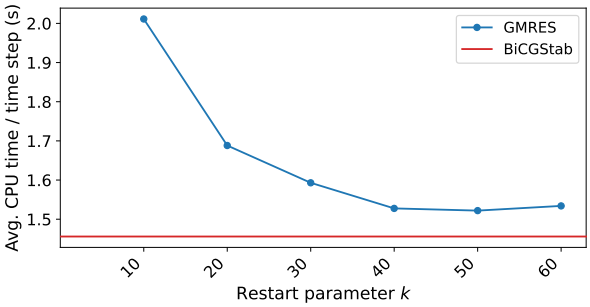
Therefore, restarting GMRES leads to worse convergence in exchange for a lower cost per iteration. The right column of Figure 4.24 shows that these two effects are dominated by the increase in iteration count, leading to a higher average CPU time per time step as k is reduced. Therefore, restarting GMRES does not yield a net performance gain over BiCGStab for the 50,000, 90,000, and 150,000 mesh sizes.

However, as n increases, the gap between GMRES and BiCGStab narrows until GMRES is able to outperform BiCGStab with a restart around 70, suggesting that the lower cost per iteration of GMRES may become more advantageous at larger scales. Indeed, as mesh size increases, the cost per iteration of BiCGStab grows more rapidly than the iteration count due to the increasing cost of the matrix-vector product. This trend suggests that at even larger mesh sizes, the performance advantage of GMRES over BiCGStab may become more pronounced.

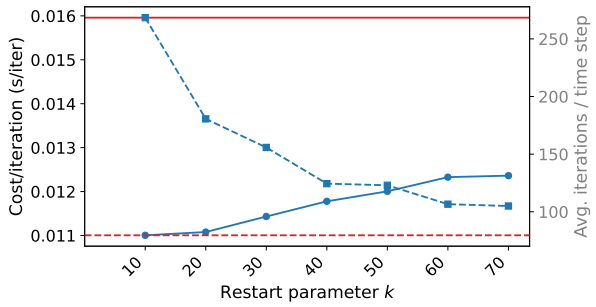
Another objective of the restarted GMRES is to decrease the memory requirements of the solver. The influence of the restart parameter k on the maximum memory consumption is presented in Figure 4.25.



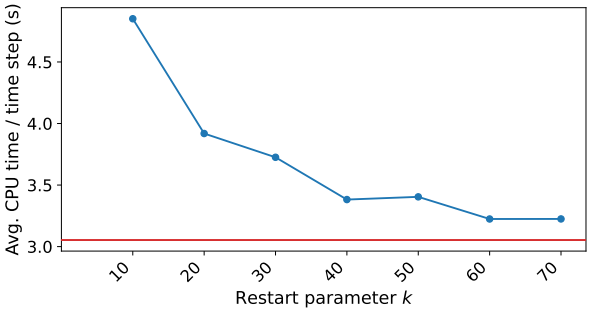
(a) n = 50,000 – cost/iter & avg. iterations per time step



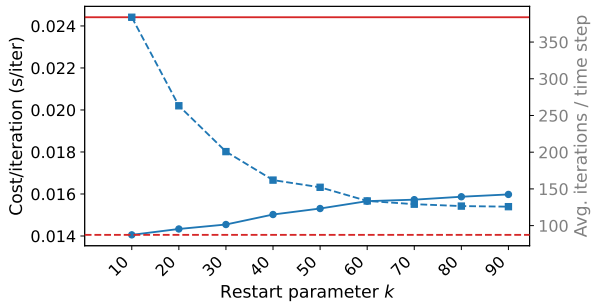
(b) n = 50,000 – avg. CPU time per time step



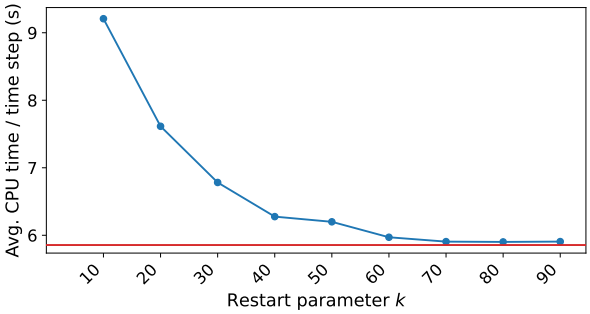
(c) n = 90,000 – cost/iter & total iterations



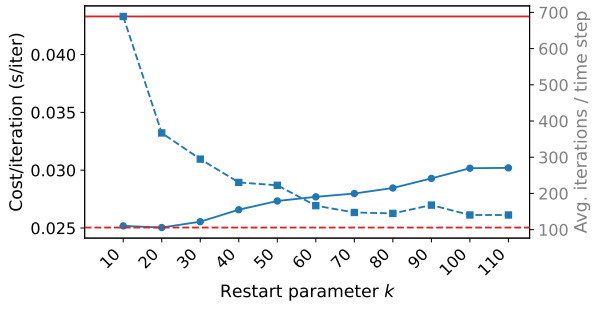
(d) n = 90,000 – avg. CPU time per time step



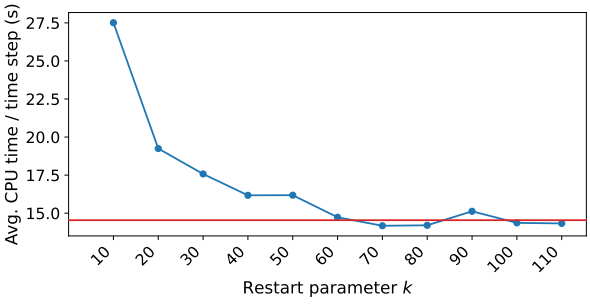
(e) n = 150,000 – cost/iter & total iterations



(f) n = 150,000 – avg. CPU time per time step

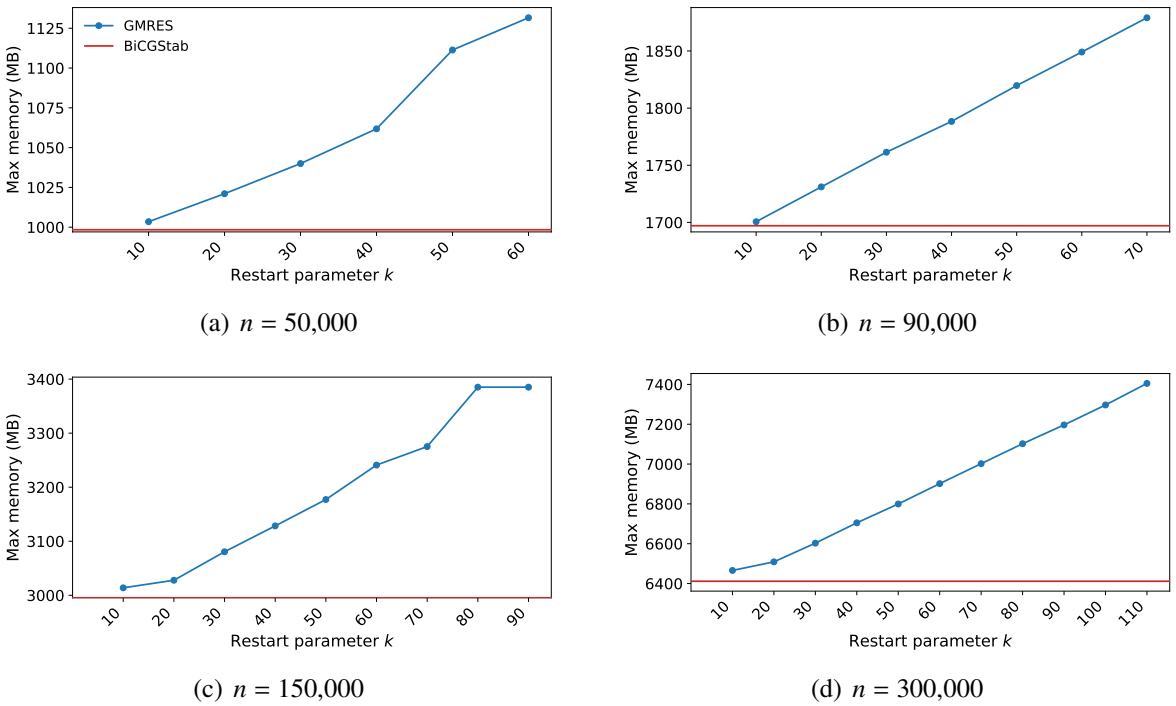


(g) n = 300,000 – cost/iter & total iterations



(h) n = 300,000 – avg. CPU time per time step

Figure 4.24: GMRES(k) with ILU(0) preconditioner, j = 32 threads. Left: cost per linear iteration and solver iteration count. Right: average CPU time per time step. Horizontal lines: BiCGStab reference ($\varepsilon = 10^{-4}$). Dotted line: boundary of *Unrestarted* regime. Rows: n = 50,000, 90,000, 150,000, 300,000 particles.



(a) $n = 50{,}000$

(b) $n = 90{,}000$

(c) $n = 150{,}000$

(d) $n = 300{,}000$

Figure 4.25: Maximum memory usage as a function of the restart parameter for the GMRES implementations of MKL and AMGCL. Each subfigure corresponds to a different mesh size.

As shown in Figure 4.25, memory consumption increases monotonically with the restart parameter k, since a larger orthonormal basis must be stored to support the Arnoldi orthogonalisation process. Although the memory is reduced when using a restarted GMRES, this doesn't justify the cost of increased CPU time as the reduction in memory is negligible.

4.3.5 Memory Requirements

Contrary to direct solvers, iterative solvers do not require the computation of an LU factorization and can therefore avoid the fill-in problem inherent to direct methods. In this section, the memory usage and the memory complexity of the iterative solvers are compared across the different mesh sizes and against the direct solvers.

Figure 4.26 presents the peak memory consumption of all solvers as a function of the mesh size. The memory consumption of the AMGCL solvers with ILU(0) preconditioning is substantially lower than that of the direct solvers, and only slightly higher than that of the Eigen BiCGStab solver with Jacobi preconditioning. This is due to the ILU(0) preconditioner storing no fill-in entries, as it retains only the nonzero structure of the original matrix $\mathbf{A}$, whereas direct solvers must store the additional fill-in entries generated during the factorization. Furthermore, the GMRES solver requires slightly more memory than BiCGStab, as it must store the orthonormal basis vectors of the Krylov subspace accumulated over the iterations, as described in Section 3.3.2.

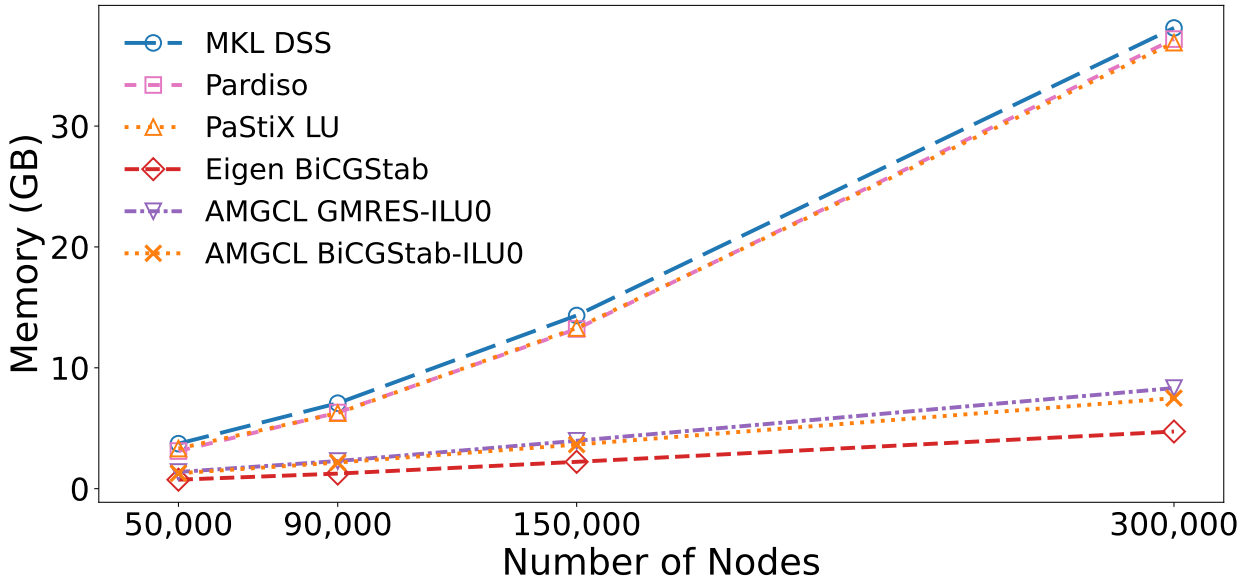


Figure 4.26: Peak memory consumption as a function of mesh size for direct solvers, Eigen BiCGStab, and AMGCL iterative solvers with ILU(0) preconditioning.

To investigate whether the empirical memory complexity confirms the expected linear scaling, the same power-law methodology as in Section 4.2.5 is applied.
Figure 4.27 presents the peak memory consumption of BiCGStab and GMRES with ILU(0) preconditioning on a log-log scale as a function of the number of non-zeros, with the fitted power-law curves and an extrapolation to the 256 GB memory limit of the computation platform. One can see that both solvers display linear trends in the log-log plot, confirming that the power-law model is a good fit for the empirical memory complexity. The estimated exponents α for BiCGStab and GMRES are 0.98 and 0.95 respectively, both very close to 1.0, indicating near-linear memory scaling with the number of non-zeros. The thread count has virtually no effect on memory consumption, as shown by the very narrow shaded bands. Extrapolating these trends to the 256 GB memory limit, the maximum problem size that can be solved is estimated to be around n = 8,000,000 particles, corresponding to approximately 1.8 billion non-zeros. This represents a significant improvement over the direct solvers, which were estimated to reach the same memory limit at approximately n = 1,200,000 particles in Section 4.2.5.

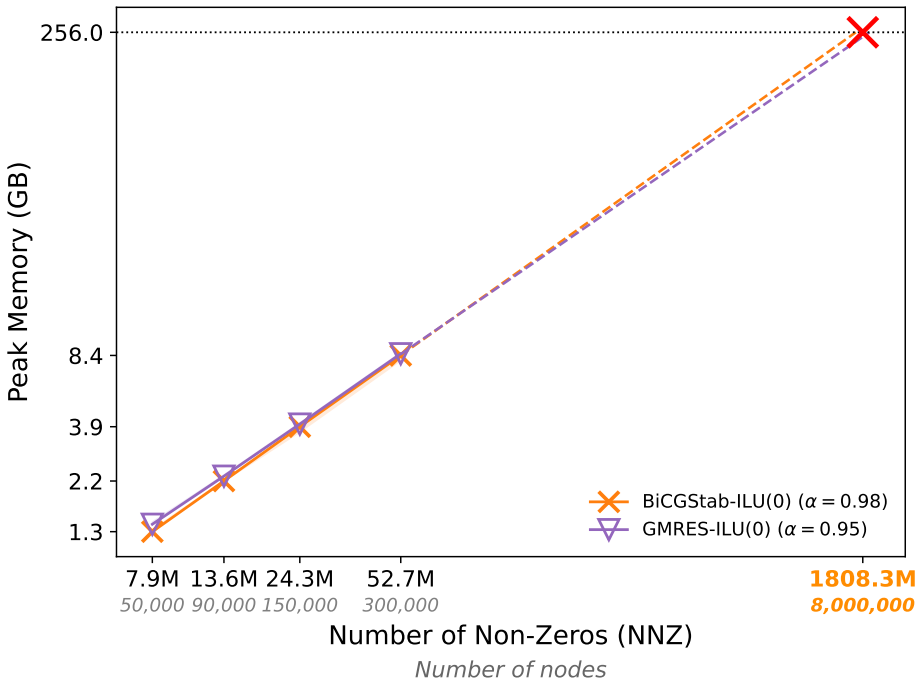


Figure 4.27: Log-log plot of peak memory consumption as a function of NNZ for the two AMGCL iterative solvers. Solid lines show measured data ($j = 1$), shaded bands span the min/max across all thread counts, dashed lines extrapolate the fitted power law, and the red cross marks the estimated NNZ at which memory reaches 256 GB.

The empirical memory complexity exponents computed across all thread counts are presented in Table 4.6.

Solver	1	2	4	8	16	32	**Avg**
BiCGStab	0.98	0.95	0.95	0.95	0.94	0.94	**0.95**
GMRES	0.95	0.95	0.95	0.95	0.95	0.94	**0.95**

Table 4.6: Empirical memory complexity exponent α such that $M(\text{nnz}) \propto \text{nnz}^{\alpha}$, estimated from the slope of a linear regression of $\log M$ vs $\log(\text{nnz})$ across all problem sizes ($n = 50\,000$, $90\,000$, $150\,000$, $300\,000$). Columns indicate the number of threads and the rightmost column is the average across all thread counts. Cells are colored from green (closest to linear) to red (furthest from linear).

The memory complexities of both solvers are very stable across thread counts and close to $O(\text{nnz}^{0.95}) \approx O(\text{nnz})$, confirming the expected linear memory scaling. This is consistent with the fact that neither BiCGStab nor GMRES with ILU(0) preconditioning requires additional fill-in storage beyond the original matrix and its incomplete factorization. To derive a practically usable law for PFEM3D problems, an empirical upper bound $M_{\text{max,iter}}$ is fitted as a function of the number of particles n from the maximum memory observations in Figure 4.27:

$$M_{\text{max,iter}}(n) \leq C \cdot n, \tag{4.9}$$

where C is the fitted prefactor, with the 256 GB limit reached at approximately $n = 8 \times 10^6$ particles. Compared to the direct solver bound $M_{\text{max,direct}}(n) \leq C \cdot n^{1.33}$ from Equation (4.5), the iterative solvers offer a substantially more favorable memory scaling, enabling the solution of problems approximately 6.7 times larger in terms of particle count within the same memory budget.

4.4 Conclusions

This chapter presented an empirical comparison of direct and iterative linear solvers in the context of incompressible fluid PFEM3D simulations on the dam break benchmark. The main findings are summarised below.

Among the direct solvers, MKL DSS, PaStiX LU, and Panua PARDISO all demonstrated competitive performance when using the nested dissection ordering algorithm, with empirical time complexities of approximately $O(\text{nnz}^2)$. PaStiX LU exhibited a decreasing time complexity exponent with increasing thread count, reaching $O(\text{nnz}^{1.66})$ at $j = 32$ threads, which suggests superior asymptotic scaling for large problem sizes. Panua PARDISO achieved the highest parallel efficiency among all direct solvers, maintaining $E = 0.41$ at $j = 32$ threads for $n = 300{,}000$ particles. However, all direct solvers share a common limitation: their memory consumption grows as $O(\text{nnz}^{4/3})$ due to fill-in in the LU factors, which restricts their applicability to problems below approximately $n = 1{,}200{,}000$ particles on the computational platform used in this study.

Iterative solvers with ILU(0) preconditioning, implemented through the AMGCL library, were shown to outperform all direct solvers by a significant margin. At $j = 32$ threads, BiCGStab and unrestarted GMRES with ILU(0) achieved Performance Improvement Factors of approximately 870 and 890 relative to Eigen LU, respectively, while maintaining a memory footprint that grows only linearly with the number of particles. The ILU(0) preconditioner was recommended as the default choice, as it delivers near-optimal performance without requiring any parameter tuning, unlike the ILUT preconditioner whose drop tolerance τ and fill factor ρ must be calibrated empirically for each problem.

The convergence tolerance was identified as a critical parameter for iterative solvers. An optimal range of $[10^{-6}, 10^{-1}]$ was established, within which the computation time is minimised while the nonlinear convergence behaviour remains unaffected. A tolerance of 10^{-4} was selected as the recommended default, and its robustness was confirmed through full-length simulations across all mesh sizes.

Regarding the choice between BiCGStab and unrestarted GMRES, BiCGStab demonstrated superior performance for mesh sizes up to $n = 150{,}000$ particles, owing to its constant $O(\text{nnz})$ cost per iteration and faster convergence rate on this benchmark. However, at $n = 300{,}000$ particles, unrestarted GMRES became competitive, suggesting that it may offer advantages at larger scales where the per-iteration cost of BiCGStab grows more rapidly than the iteration count of GMRES.

Figure 4.28 summarises these findings as a practical decision tree for selecting a linear solver in the context of incompressible fluid PFEM3D simulations.

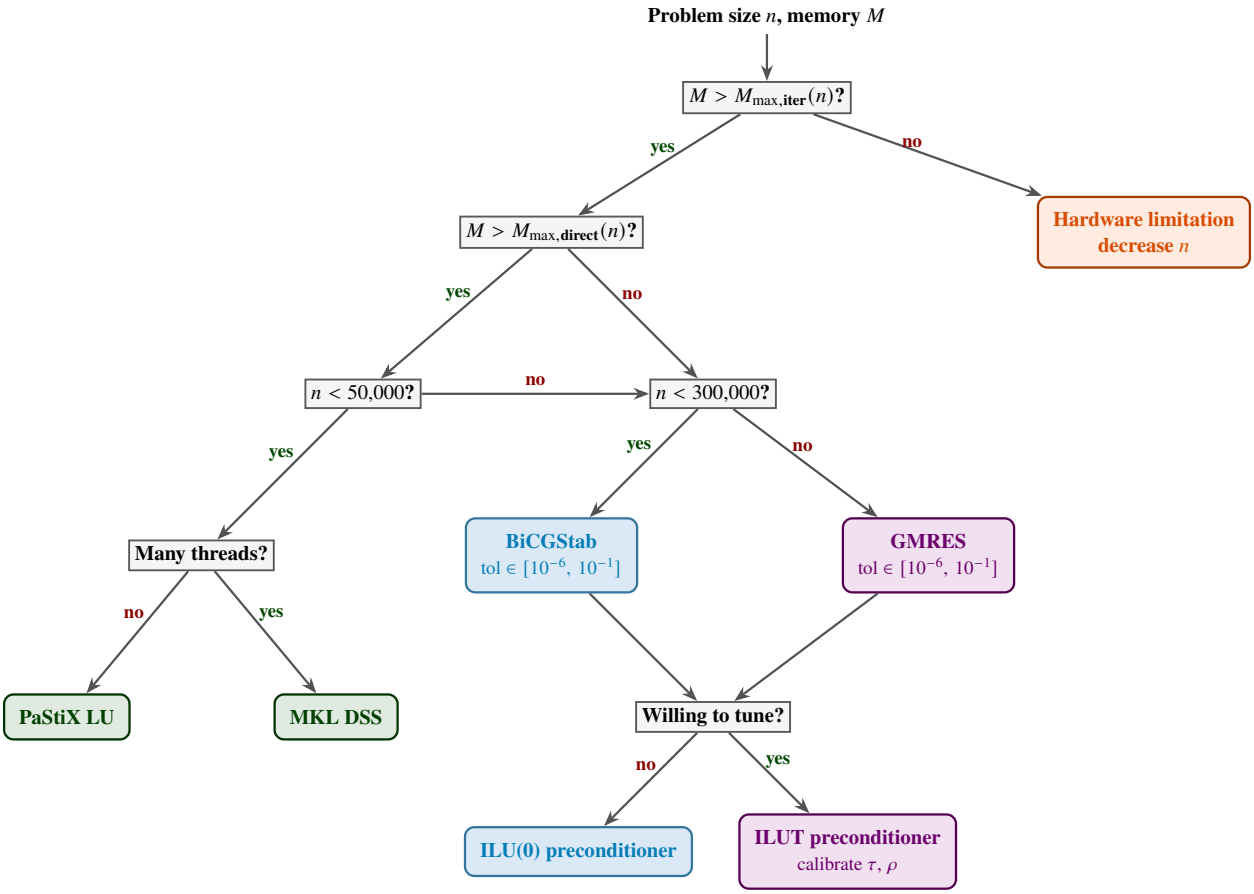


Figure 4.28: Decision tree for selecting a linear solver for incompressible fluid PFEM3D simulations, based on the empirical findings of this chapter.

Chapter 5

Laser track benchmark

The dam break example presented in Chapter 2 involves an incompressible fluid and serves as a baseline for comparing linear solvers. However, in practical applications of PFEM3D, the problems of interest are significantly more complex. To assess the performance of the linear solvers in a more physically demanding context, a second benchmark is considered: the 3D laser track test case from Bogucki *et al.* [75]. In this example, a laser beam scans a bare substrate of 316L stainless steel, inducing melting, phase change and thermo-mechanical coupling. The initial configuration and boundary conditions are shown in Figure 5.1, and the key numerical and physical parameters of the simulation are given in Table 5.1.

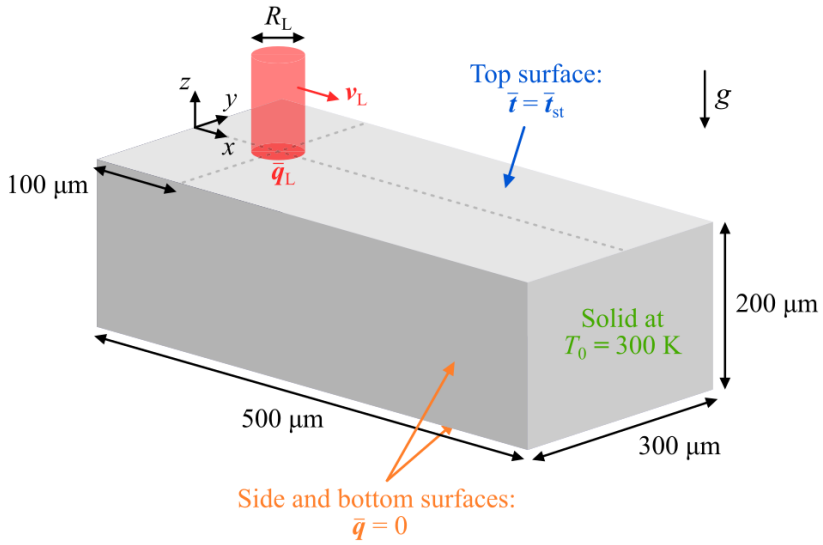


Figure 5.1: Initial configuration and boundary conditions of the laser track test. Figure adapted from Bogucki *et al.* [75]

The computational domain is a $500 \times 300 \times 200$ μm^3 box representing a bare substrate of 316L stainless steel, initially at a uniform temperature $T_0 = 300$ K.

Property	**Value**
Laser power (P_L)	45 W
Laser radius (R_L)	$1.699/2 \times 60$ μm
Scanning speed (v_L)	0.3 m/s
Absorptivity (a)	0.3
Surface tension coefficient	1.7 N/m
Solidus temperature (T_s)	1670 K
Liquidus temperature (T_l)	1730 K
Latent heat (L_m)	267 776 J/kg
Maximum time step (Δt)	$5 \cdot 10^{-2}$ μs
Total simulation time	1 ms

Table 5.1: Key simulation properties for the laser track test case.

In these simulations, remeshing is performed at every time step. In addition to the fluid part, the solid part is also remeshed to avoid excessive element distortion as the material softens near the melt pool. Initially, the total number of nodes is $n = 17,849$. Figure 5.2, adapted directly from Bogucki *et al.* [75], shows the temperature, Von Mises stress and pressure distributions inside the substrate at the end of the simulation ($t = 1$ ms).

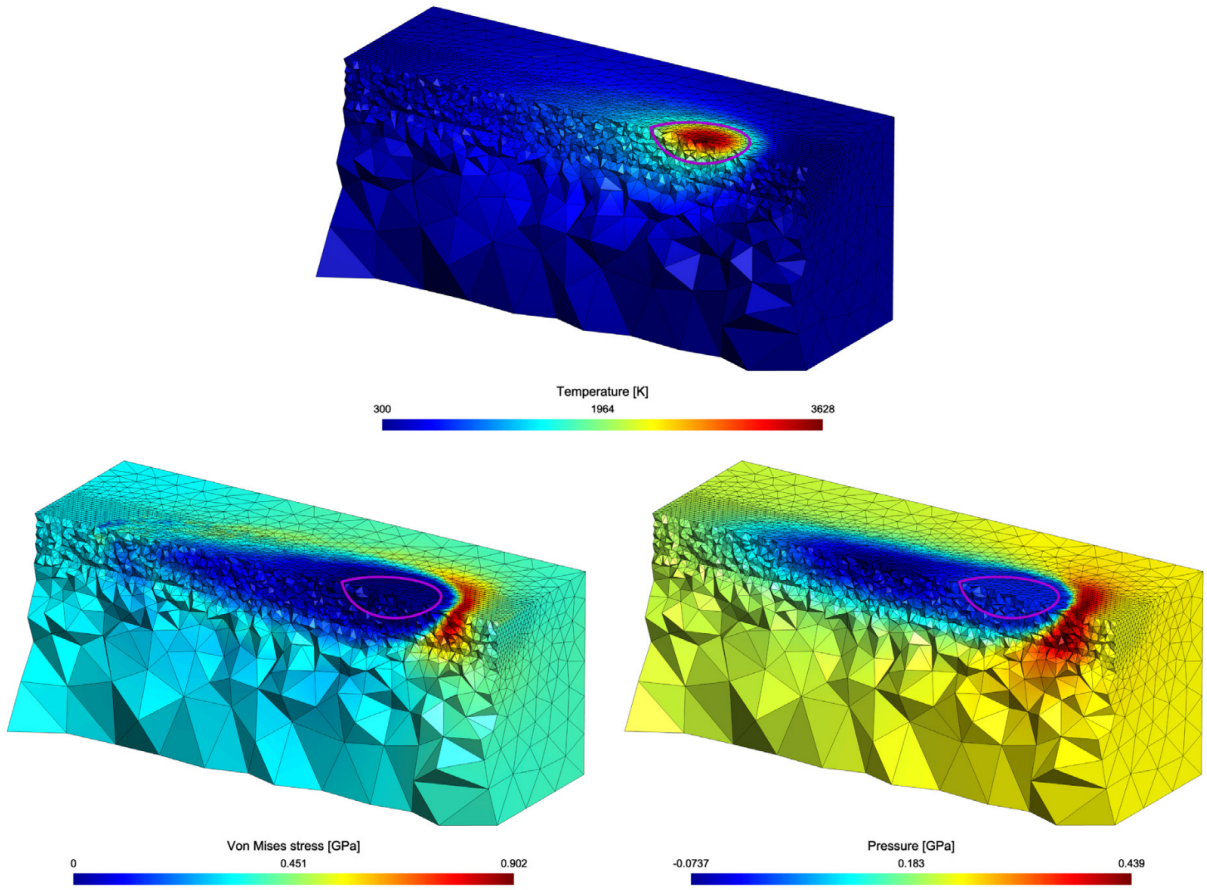


Figure 5.2: Temperature (top), Von Mises stress (left) and pressure (right) distributions at $t = 1$ ms of the laser track test. Figure adapted from Bogucki *et al.* [75].

Five solver configurations are compared on this test case: two direct solvers, MKL DSS and PaStiX, both performing LU factorization, and three iterative solvers from the AMGCL library preconditioned with ILU(0), namely GMRES with a relative tolerance of 10^{-2} and BiCGStab with relative tolerances of 10^{-2} and 10^{-4}. The cumulative CPU time over the course of the simulation is shown in Figure 5.3.

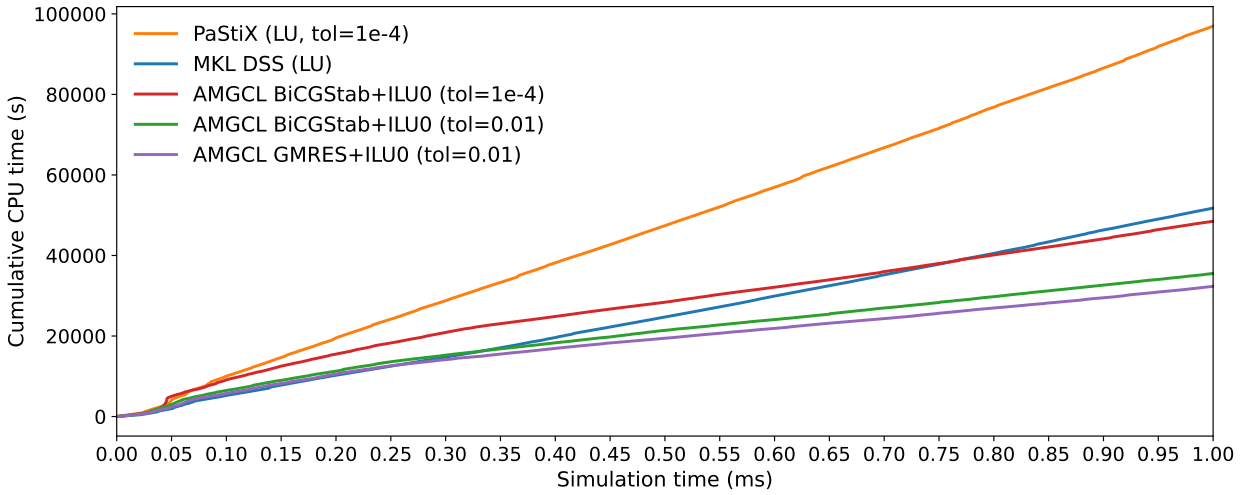


Figure 5.3: Cumulative CPU time as a function of the simulation time for five solver configurations for the laser track test.

During the early stages of the simulation (up to approximately $t = 0.15$ ms), all solvers exhibit a similar cumulative cost. As the simulation progresses, the performances diverge significantly. By the end of the simulation at $t = 1$ ms, GMRES achieves the lowest cumulative CPU time at approximately 31 000 s, followed by BiCGStab with tolerance 10^{-2} at approximately 35 000 s. BiCGStab with the tighter tolerance 10^{-4} and MKL DSS yield comparable costs at approximately 48 000 s and 51 000 s, respectively. PaStiX is by far the most expensive solver, reaching approximately 96 000 s at the end of the simulation, roughly three times the cost of GMRES.

A period of convergence difficulty is observed for the iterative solvers during the early stages of the simulation, approximately between $t = 0.05$ ms and $t = 0.10$ ms, during which all solvers exhibit a similar cumulative cost. Beyond this phase, iterative solvers with a loose tolerance exhibit a consistently lower rate of cumulative CPU time growth than the direct solvers. Notably, BiCGStab with the tighter tolerance 10^{-4} converges toward the cost of MKL DSS, suggesting that the stricter convergence criterion eliminates the computational benefit of the iterative approach. PaStiX is outperformed by all other solvers, including MKL DSS. This is consistent with the results of the dam break benchmark, where MKL DSS already outperformed PaStiX at mesh sizes as low as 50 000 nodes. Since the present simulation involves only 17 849 nodes, it is not surprising that PaStiX performs even worse in relative terms. These observations are consistent with the solver selection guidelines presented in Chapter 4, where iterative solvers configured with a low relative tolerance are expected to outperform direct solvers for large mesh sizes and long simulation times.

Chapter 6

Conclusion

This work presented a systematic comparison of sparse linear solvers within the PFEM3D software, with the objective of identifying the most efficient solver configurations for large-scale simulations. The study was motivated by the observation that the linear solver constitutes the dominant source of computational cost in PFEM3D, with the initial Eigen LU solver requiring approximately 10 days of computation for a dam break simulation with 50,000 particles.

The theoretical foundations were first established through a review of the incompressible Navier-Stokes equations and their discretization within the PFEM framework. The PSPG-stabilized formulation yields, at each nonlinear iteration, a sparse linear system $\mathbf{Ax} = \mathbf{b}$ whose coefficient matrix is nonsymmetric but structurally symmetric. The properties of this system, including its sparsity structure and its variation across time steps due to the Lagrangian remeshing procedure, were shown to have direct implications on the choice of linear solver. A presentation of both direct and iterative solving strategies was then provided, covering LU factorization, ordering algorithms, Krylov subspace methods, and preconditioning techniques.

The empirical comparison on the 3D dam break benchmark constituted the core of this work. Four direct solvers were evaluated: SuperLU Dist, Intel MKL DSS, Panua PARDISO, and PaStiX LU. Among these, MKL DSS, PaStiX LU, and Panua PARDISO demonstrated competitive performance when using the nested dissection ordering algorithm, with empirical time complexities of approximately $O(\text{nnz}^2)$ for 3D problems. PaStiX LU exhibited a decreasing complexity exponent with increasing thread count, reaching $O(\text{nnz}^{1.66})$ at $j = 32$ threads, while Panua PARDISO achieved the highest parallel efficiency at $E = 0.41$ for $n = 300{,}000$ particles. However, all direct solvers share a fundamental limitation: their memory consumption grows as $O(\text{nnz}^{4/3})$ due to fill-in, restricting their applicability to problems below approximately 1,200,000 particles on the computational platform used in this study.

Iterative solvers with ILU(0) preconditioning from the AMGCL library were shown to outperform all direct solvers by a significant margin, achieving Performance Improvement Factors of approximately 870 and 890 for BiCGStab and unrestarted GMRES, respectively, relative to Eigen LU at $j = 32$ threads. Their memory footprint grows only linearly with the number of particles, removing the memory barrier encountered by direct solvers. The ILU(0) preconditioner was recommended as the default choice over ILUT, as it delivers near-optimal performance without requiring parameter

tuning. The convergence tolerance was identified as a critical parameter, with an optimal range of $[10^{-6}, 10^{-1}]$ established for the dam break benchmark. A tolerance of 10^{-4} was selected as the recommended default, and its robustness was confirmed through full-length simulations. The relationship between preconditioner quality and iteration count was further illustrated through the direct-iterative continuum, which positions direct solvers and iterative methods on a unified spectrum ranging from cheap setup with costly solve to expensive setup with cheap solve.

Regarding the choice of Krylov method, BiCGStab demonstrated superior performance for mesh sizes up to $n = 150{,}000$ particles, owing to its constant $O(\text{nnz})$ cost per iteration. At $n = 300{,}000$ particles, unrestarted GMRES became competitive, suggesting that it may offer advantages at larger scales. These findings were synthesised into a practical decision tree to guide solver selection for incompressible fluid PFEM3D simulations.

The applicability of these guidelines was then assessed on the laser track benchmark, a thermo-mechanical problem involving phase change, surface tension, and laser-induced melting. In this more physically demanding context, the iterative solvers with ILU(0) preconditioning remained faster than the MKL direct solver, with GMRES achieving a speedup of approximately 1.7. However, the performance gap was considerably narrower than on the dam break benchmark, and a period of convergence difficulty was observed during the early phase of the simulation. These results confirmed that the solver selection guidelines established on the dam break benchmark extend to more complex physics, while highlighting that the convergence tolerance must be adapted to the problem at hand, with a value of 10^{-2} proving appropriate for the laser track test case compared to 10^{-4} for the dam break.

Bibliography

[1] M. L. Cerquaglia. "Development of a Fully-Partitioned PFEM-FEM Approach for Fluid-Structure Interaction Problems Characterized by Free Surfaces, Large Solid Deformations, and Strong Added-Mass Effects". Doctoral Thesis. Liège, Belgium: Université de Liège, Feb. 2019. URL: https://hdl.handle.net/2268/233166.

[2] V. John. *Finite Element Methods for Incompressible Flow Problems*. Springer Series in Computational Mathematics. Cham, Switzerland: Springer, 2016. ISBN: 978-3-319-45749-9. DOI: 10.1007/978-3-319-45750-5.

[3] Koen Hillewaert. "Development of the discontinuous Galerkin method for high-resolution, large scale CFD and acoustics in industrial geometries". PhD thesis. Louvain-la-Neuve, Belgium: Université catholique de Louvain, 2013.

[4] Georg May and Antony Jameson. "A Spectral Difference Method for the Euler and Navier-Stokes Equations on Unstructured Meshes". In: *Proceedings of the 47th AIAA Aerospace Sciences Meeting*. AIAA, 2006. URL: https://arc.aiaa.org/doi/10.2514/6.2006-304.

[5] M. Rakhsha, C. E. Kees, and D. Negrut. "Lagrangian vs. Eulerian: An Analysis of Two Solution Methods for Free-Surface Flows and Fluid Solid Interaction Problems". In: *Fluids* (2021). DOI: 10.3390/fluids6120460.

[6] S. H. Lo, R. A. Figueroa, and J. Donea. "Smoothed Particle Hydrodynamics for Complex Fluid–Structure Interaction Problems". In: *Computers & Structures* 173 (2016), pp. 1–19. DOI: 10.1016/j.compstruc.2016.07.004.

[7] S. Février. "Development of a Compressible Flow Solver for PFEM Fluid Simulations". Master Thesis. Université de Liège, 2020. URL: http://hdl.handle.net/2268.2/9010.

[8] T. J. R. Hughes and L. P. Franca. "A New Finite Element Formulation for Computational Fluid Dynamics: VII. The Stokes Problem with Various Well-Posed Boundary Conditions". In: *Computer Methods in Applied Mechanics and Engineering* (1987). DOI: 10.1016/0045-7825(87)90184-8.

[9] R. L. Sani et al. "The Cause and Cure (!) of the Spurious Pressures Generated by Certain FEM Solutions of the Incompressible Navier-Stokes Equations: Part 2". In: *International Journal for Numerical Methods in Fluids* (1981). DOI: 10.1002/fld.1650010206.

[10] J. Scott and M. Tůma. *Algorithms for Sparse Linear Systems*. Nečas Center Series. Cham: Birkhäuser, 2023. ISBN: 978-3-031-25819-0. DOI: 10.1007/978-3-031-25820-6.

[11] Claudio Paniconi and Mario Putti. "A comparison of Picard and Newton iteration in the numerical solution of multidimensional variably saturated flow problems". In: *Water Resources Research* 30.12 (1994), pp. 3357–3374. DOI: 10.1029/94WR02046.

[12] C. Geuzaine and J.-F. Remacle. "Gmsh: A 3-D Finite Element Mesh Generator with Built-in Pre- and Post-Processing Facilities". In: *International Journal for Numerical Methods in Engineering* (2009). DOI: 10.1002/nme.2579.

[13] S. Koshizuka and Y. Oka. "Moving-Particle Semi-Implicit Method for Fragmentation of Incompressible Fluid". In: *Nuclear Science and Engineering* (1996). DOI: 10.13182/NSE96-A24205.

[14] S. Meduri, M. Cremonesi, and U. Perego. "An Efficient Runtime Mesh Smoothing Technique for 3D Explicit Lagrangian Free-Surface Fluid Flow Simulations". In: *International Journal for Numerical Methods in Engineering* 172 (2018), pp. 145–160. DOI: 10.1002/nme.5962.

[15] O. Schenk. "Scalable Parallel Sparse LU Factorization Methods on Shared Memory Multiprocessors". Doctoral Thesis. Swiss Federal Institute of Technology Zurich (ETH Zurich), 2000. DOI: 10.3929/ethz-a-003876213.

[16] G. H. Golub and C. F. Van Loan. *Matrix Computations*. 4th. Baltimore: Johns Hopkins University Press, 2013. ISBN: 978-1-4214-0794-4. DOI: 10.56021/9781421407944.

[17] W. H. Press et al. *Numerical Recipes: The Art of Scientific Computing*. 3rd. New York: Cambridge University Press, 2007. ISBN: 978-0-521-88068-8.

[18] Y. Saad. *Iterative Methods for Sparse Linear Systems*. 2nd. Philadelphia, PA: Society for Industrial and Applied Mathematics, 2003. ISBN: 978-0-89871-534-7. DOI: 10.1137/1.9780898718003.

[19] BLAST Forum. *Basic Linear Algebra Subprograms Technical (BLAST) Forum Standard*. Technical Report. Knoxville, TN: University of Tennessee, Netlib, Aug. 2001. URL: https://www.netlib.org/blas/blast-forum/blas-report.pdf.

[20] Intel Corporation. *Intel® oneAPI Math Kernel Library Developer Reference C*. Version 2025.2. Accessed: 2025-06-30. Intel Corporation. Santa Clara, CA, June 2025. URL: https://www.intel.com/content/www/us/en/docs/onemkl/developer-reference-c/2025-2/overview.html.

[21] T. A. Davis. *Direct Methods for Sparse Linear Systems*. Philadelphia, PA: Society for Industrial and Applied Mathematics, 2006. ISBN: 978-0-89871-613-9. DOI: 10.1137/1.9780898718881.

[22] M. Faverge et al. "Designing LU-QR Hybrid Solvers for Performance and Stability". In: *Proceedings of the International Parallel and Distributed Processing Symposium* (2014). DOI: 10.1109/IPDPS.2014.108.

[23] SciPy Developers. *scipy.sparse.linalg.splu*. 2024. URL: https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.linalg.splu.html.

[24] Thomas H. Cormen et al. *Introduction to Algorithms*. 3rd. Cambridge, MA: MIT Press, 2009. ISBN: 978-0-262-03384-8.

[25] M. T. Heath, E. Ng, and B. W. Peyton. “Parallel Algorithms for Sparse Linear Systems”. In: *SIAM Review* (1991). DOI: 10.1137/1033099.

[26] I. S. Duff and H. A. van der Vorst. “Developments and Trends in the Parallel Solution of Linear Systems”. In: *Parallel Computing* (1999). DOI: 10.1016/S0167-8191(99)00077-0.

[27] Magnus R. Hestenes and Eduard Stiefel. “Methods of Conjugate Gradients for Solving Linear Systems”. In: *Journal of Research of the National Bureau of Standards* 49.6 (1952), pp. 409–436. DOI: 10.6028/jres.049.044. URL: https://nvlpubs.nist.gov/nistpubs/jres/049/jresv49n6p409_a1b.pdf.

[28] A. Bultheel, M. Van Barel, and P. Van Gucht. “Orthogonal Basis Functions in Discrete Least-Squares Rational Approximation”. In: *Journal of Computational and Applied Mathematics* (2004). DOI: 10.1016/S0377-0427(03)00497-7.

[29] R. Cuevas Núñez, C. E. Schaerer, and A. Bhaya. “A Proportional-Derivative Control Strategy for Restarting the GMRES(m) Algorithm”. In: *Journal of Computational and Applied Mathematics* 337 (2018), pp. 209–224. DOI: 10.1016/j.cam.2018.01.009.

[30] C. Lanczos. “Solution of Systems of Linear Equations by Minimized Iterations”. In: *Journal of Research of the National Bureau of Standards* 49.1 (1952), pp. 33–53. DOI: 10.6028/jres.049.033.

[31] Randolph E. Bank and Tony F. Chan. “A Composite Step Bi-Conjugate Gradient Algorithm for Nonsymmetric Linear Systems”. In: *Numerical Algorithms* 7.1 (1994), pp. 1–16. DOI: 10.1007/BF02141258.

[32] G. L. G. Sleijpen and H. A. van der Vorst. “An Overview of Approaches for the Stable Computation of Hybrid BiCG Methods”. In: *Applied Numerical Mathematics* 19 (1995), pp. 235–254. DOI: 10.1016/0168-9274(95)00085-2.

[33] Hartwig Anzt, Vincent Heuveline, and Björn Rocker. “Mixed Precision Iterative Refinement Methods for Linear Systems: Convergence Analysis Based on Krylov Subspace Methods”. In: *Parallel Computing: Accelerating Computational Science and Engineering (CSE)*. Ed. by Friedhelm Schwenker et al. Springer, 2010, pp. 237–244. DOI: 10.1007/978-3-642-28145-7_24.

[34] V. V. S. Prakash and Raj Mittra. “Multi-frontal preconditioners for iterative solvers”. In: *Proceedings of the IEEE Antennas and Propagation Society (APS) Symposium*. Vol. 3. July 2001, pp. 12–15. DOI: 10.1109/APS.2001.960019.

[35] Claudson Ferreira Bornstein. “Parallelizing and de-parallelizing elimination orders”. In: *Proceedings of the International Symposium on Sparse Matrix Computations*. 1998.

[36] Laura Grigori et al. “Hypergraph-Based Unsymmetric Nested Dissection Ordering for Sparse LU Factorization”. In: *SIAM Journal on Scientific Computing* 32.6 (2010), pp. 3426–3446. DOI: 10.1137/080720395.

[37] P. J. de Jonge. “A comparative study of algorithms for reducing the fill-in during Cholesky factorization”. In: *Journal of Geodesy* 66.10 (1992), pp. 685–694. DOI: 10.1007/BF02033190.

[38] George Karypis. *Family of Graph and Hypergraph Partitioning Software : METIS – Serial Graph Partitioning and Fill-reducing Matrix Ordering*. https://karypis.github.io/glaros/software/metis/overview.html. Consulté le 10 août 2025.

[39] LaBRI (F. Pellegrini). *SCOTCH: Static Mapping, Graph, Mesh and Hypergraph Partitioning, and Parallel and Sequential Sparse Matrix Ordering Package*. https://www.labri.fr/perso/pelegrin/scotch/. Consulté le 10 août 2025.

[40] Eigen Developers. *Eigen C++ Library for Linear Algebra*. 2025. URL: https://eigen.tuxfamily.org/index.php?title=Main_Page.

[41] Eigen Developers. *Eigen: SparseLU Class Reference*. 2025. URL: https://eigen.tuxfamily.org/dox/classEigen_1_1SparseLU.html.

[42] Eigen Developers. *Eigen: Topic Multi-Threading*. 2025. URL: https://eigen.tuxfamily.org/dox/TopicMultiThreading.html.

[43] Eigen Developers. *Eigen: SparseQR Class Reference*. 2025. URL: https://eigen.tuxfamily.org/dox/classEigen_1_1SparseQR.html.

[44] Eigen Developers. *Eigen: Topic Sparse Systems*. 2021. URL: https://eigen.tuxfamily.org/dox/group__TopicSparseSystems.html.

[45] Eigen Developers. *Eigen: GMRES Class Reference*. 2025. URL: https://eigen.tuxfamily.org/dox/unsupported/classEigen_1_1GMRES.html.

[46] Eigen Developers. *Eigen: BiCGSTAB Class Reference*. 2025. URL: https://eigen.tuxfamily.org/dox/classEigen_1_1BiCGSTAB.html.

[47] Panua GmbH. *PARDISO Solver Project*. https://www.panua.ch/pardiso/. Accessed: 2025-09-28. 2025.

[48] Y. Saad. "A Flexible Inner-Outer Preconditioned GMRES Algorithm". In: *SIAM Journal on Scientific Computing* 14.2 (1993), pp. 461–469. DOI: 10.1137/0914028.

[49] Solverstack@Inria Bordeaux. *PaStiX: A sparse direct solver*. 2025. URL: https://solverstack.gitlabpages.inria.fr/pastix/index.html.

[50] P. Hénon, P. Ramet, and J. Roman. "PaStiX: A High-Performance Parallel Direct Solver for Sparse Symmetric Positive Definite Systems". In: *Parallel Computing* 28.2 (2002), pp. 301–321. DOI: 10.1016/S0167-8191(01)00141-7.

[51] D. Demidov. "AMGCL: An Efficient, Flexible, and Extensible Algebraic Multigrid Implementation". In: *Lobachevskii Journal of Mathematics* 40.5 (2019), pp. 535–546. DOI: 10.1134/S1995080219050056.

[52] D. Demidov. *AMGCL: C++ Library for Solving Large Sparse Linear Systems with Algebraic Multigrid*. https://github.com/ddemidov/amgcl. Accessed: 2026-01-10. 2020.

[53] D. Demidov. *AMGCL Documentation: Relaxation*. https://amgcl.readthedocs.io/en/latest/components/relaxation.html. Accessed: 2026-01-10. 2020.

[54] X. S. Li. "An Overview of SuperLU: Algorithms, Implementation, and User Interface". In: *ACM Transactions on Mathematical Software* 31.3 (2005), pp. 302–325. DOI: 10.1145/1089014.1089017.

[55] X. S. Li and J. W. Demmel. "SuperLU_DIST: A Scalable Distributed-Memory Sparse Direct Solver for Unsymmetric Linear Systems". In: *ACM Transactions on Mathematical Software* 29.2 (2003), pp. 110–140. DOI: 10.1145/779359.779361.

[56] O. Schenk and K. Gärtner. “Solving Unsymmetric Sparse Systems of Linear Equations with PARDISO”. In: *Future Generation Computer Systems* 20.3 (2004). This paper is recommended by Intel for citing the PARDISO solver in MKL, pp. 475–487. DOI: 10.1016/j.future.2003.07.011.

[57] Advanced Micro Devices, Inc. *AMD Ryzen Threadripper PRO Processors: Riding the Third Wave of Workstation Computing*. White Paper. Accessed: 2025-12-28. AMD, 2020. URL: https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/white-papers/threadripperpro-white-paper-third-wave-workstation-computing.pdf.

[58] S. Saini et al. “The Impact of Hyper-Threading on Processor Resource Utilization in Production Applications”. In: *18th International Conference on High Performance Computing (HiPC)*. Bengaluru, India: IEEE, 2011, pp. 1–10. DOI: 10.1109/HiPC.2011.6152743.

[59] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. 6th. Cambridge, MA: Morgan Kaufmann, 2017. ISBN: 978-0-12-811905-1.

[60] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts*. 10th ed. John Wiley & Sons, 2018. ISBN: 978-1-118-06333-0.

[61] Daniël de Kok. *Intel MKL on AMD Zen*. https://danieldk.eu/Posts/2020-08-31-MKL-Zen.html. Accessed: 2026-01-08. 2020.

[62] maboroshi81. *How to force MATLAB to use a fast codepath on AMD Ryzen/TR CPUs*. Reddit r/matlab. Accessed: 2026-01-08. 2019. URL: https://old.reddit.com/r/matlab/comments/dxn38s/howto_force_matlab_to_use_a_fast_codepath_on_amd/fm2j83e/.

[63] Mingru Yang. *MKL has bad performance on an AMD CPU*. https://sites.google.com/a/uci.edu/mingru-yang/programming/mkl-has-bad-performance-on-an-amd-cpu. Accessed: 2026-01-08. 2019.

[64] Puget Systems. *How To Use MKL with AMD Ryzen and Threadripper CPUs Effectively for Python, NumPy, and Other Applications*. https://www.pugetsystems.com/labs/hpc/How-To-Use-MKL-with-AMD-Ryzen-and-Threadripper-CPU-s-Effectively-for-Python-Numpy-And-Other-Applications-1637/. Accessed: 2026-01-08. 2020.

[65] Panua GmbH. *PARDISO User Guide*. Version 8.2. Academic trial licenses are limited to two threads. Accessed: 2026-01-05. Panua GmbH. 2025. URL: https://panua.ch/static/manual/manual.pdf.

[66] P. Amestoy et al. “On the Complexity of the Block Low-Rank Multifrontal Factorization”. In: *SIAM Journal on Scientific Computing* 39.4 (2017), A1710–A1740. DOI: 10.1137/16M1077192.

[67] P.-G. Martinsson. *Fast Direct Solvers for Elliptic PDEs*. Vol. 96. CBMS-NSF Regional Conference Series in Applied Mathematics. Philadelphia, PA: Society for Industrial and Applied Mathematics, 2019. DOI: 10.1137/1.9781611976045.

[68] T. A. Davis, S. Rajamanickam, and W. M. Sid-Lakhdar. “A Survey of Direct Methods for Sparse Linear Systems”. In: *Acta Numerica* 25 (2016), pp. 383–566. DOI: 10.1017/S0962492916000076.

[69] M. Woźniak et al. “Computational Cost Estimates for Parallel Shared Memory Isogeometric Multi-Frontal Solvers”. In: *Computers & Mathematics with Applications* 67.10 (2014), pp. 1864–1883. DOI: 10.1016/j.camwa.2014.03.017.

[70] E. Carson and Z. Strakoš. “On the Cost of Iterative Computations”. In: *Philosophical Transactions of the Royal Society A* 378.2166 (2020), p. 20190050. DOI: 10.1098/rsta.2019.0050.

[71] R. Barrett et al. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. 2nd. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1994. ISBN: 978-0-89871-328-2. DOI: 10.1137/1.9781611971538.

[72] C. T. Kelley. *Iterative Methods for Linear and Nonlinear Equations*. Frontiers in Applied Mathematics 16. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1995. ISBN: 978-0-898713-52-8. DOI: 10.1137/1.9781611970944.

[73] Y. Saad and M. H. Schultz. “GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems”. In: *SIAM Journal on Scientific and Statistical Computing* 7.3 (1986), pp. 856–869. DOI: 10.1137/0907058.

[74] Z. Leyk. “Breakdowns and Stagnation in Iterative Methods”. In: *BIT Numerical Mathematics* 37 (1997), pp. 377–403. DOI: 10.1007/BF02510219.

[75] D. Bogucki et al. “A Unified Fluid–Solid Formulation for Thermo-Mechanical Problems Using the Particle Finite Element Method: Application to Metal Additive Manufacturing”. In: *Computer Methods in Applied Mechanics and Engineering* 459 (2026), p. 119074. DOI: 10.1016/j.cma.2026.119074.