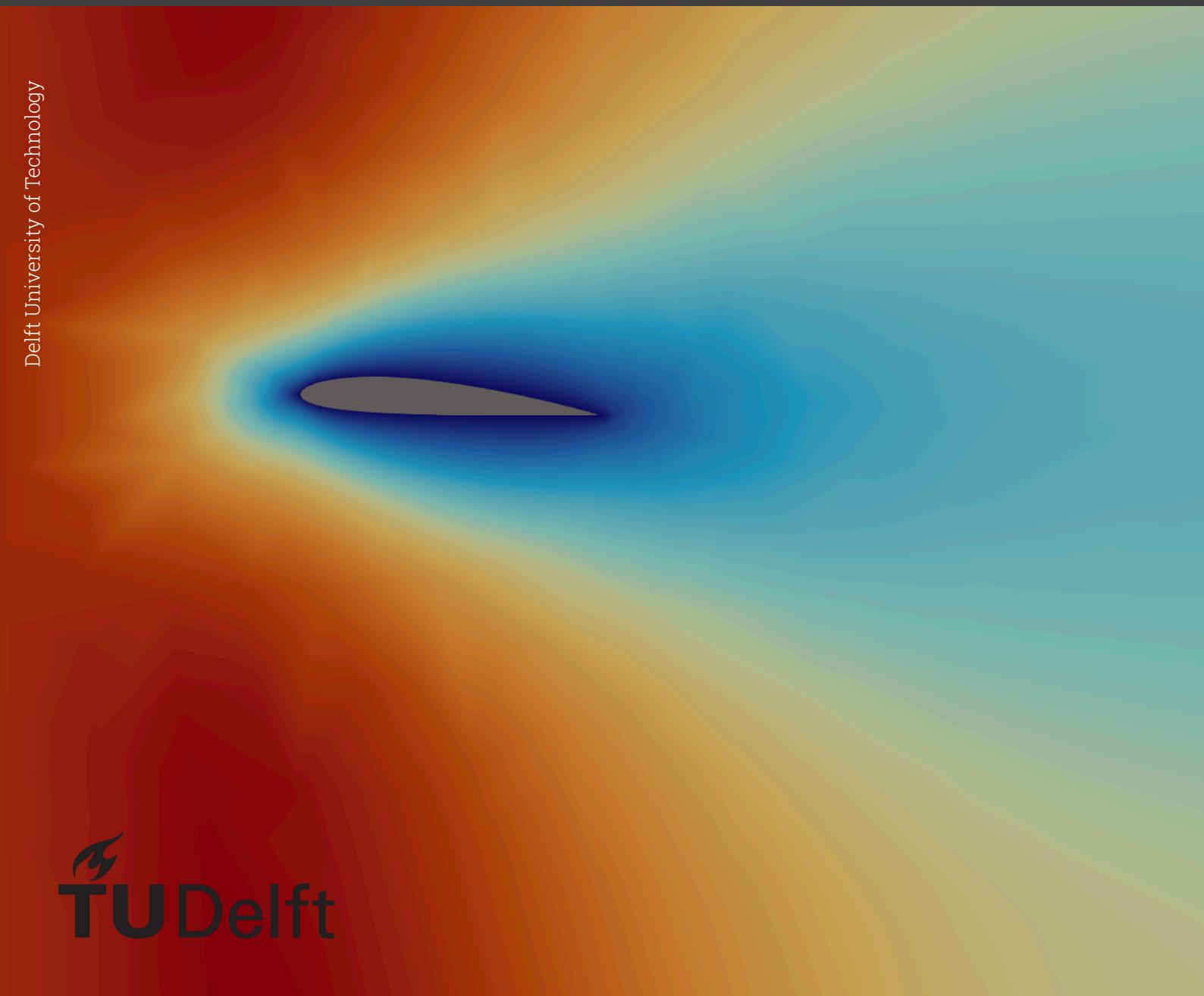
Preconditioned Krylov Solvers under Shared-Memory Parallelism

Evaluating Convergence, Scalability, and Parallel Overhead

H.J.G. Reijersen van Buuren



Delft University of Technology

TU Delft

Preconditioned Krylov Solvers under Shared-Memory Parallelism

Evaluating Convergence, Scalability, and Parallel Overhead

by

H.J.G. Reijersen van Buuren

Supervisor: Dr. A. Heinlein
Project Duration: September, 2025 - December, 2025
Faculty: Faculty of Electrical Engineering, Mathematics and Computer Science, Delft

TU Delft

Preface

This thesis is the final project of my bachelor studies in Applied Mathematics at Delft University of Technology. Over the past months I have explored how preconditioned Krylov subspace methods behave on modern multi-core architectures, using the finite element library NGSolve and the performance-portable framework PyKokkos. The project combines topics from numerical analysis, scientific computing and high-performance computing.

The report is primarily intended for students and researchers with an interest in iterative methods, finite element discretisations and performance-portable implementations. A basic familiarity with linear algebra and partial differential equations is assumed, but the relevant background on Krylov methods, preconditioning and parallelism is reviewed in the early chapters.[1]

I am very grateful for Dr. Heinlein for his insights, guidance and all the helpful discussions throughout this project.

Soli Deo gloria.

H.J.G. Reijersen van Buuren
Delft, December 2025

[1] I declare that I have used generative AI or AI-assisted technologies during my thesis. Specifically, I used such tools in the following ways:

- Create tables in proper LaTeX code.
- Improve the grammar, clarity, and flow of the entire text.
- Assist in writing code.
- Convert sources into the proper referencing format.

Layman's abstract

Many simulations in science and engineering boil down to solving huge systems of linear equations. Think of trying to predict how heat spreads in a metal plate, or how air flows around an kite. These problems are too large to solve with pen and paper. Instead, engineers often use iterative methods: algorithms that start with an initial guess and improve it step by step.

This thesis studies two such iterative methods, called Conjugate Gradient (CG) and Generalized Minimal Residual (GMRES), on modern computers. The questions are: How fast can they obtain a solution? How well can they be improved with more computational power? And how much does their performance depend on a helper tool called a "preconditioner"? Preconditioners change the system into an easier one that the solver can handle more efficiently.

Using the finite element library NGSolve and the performance-portable framework PyKokkos, we solve three test problems: a simple heat equation, a simplified flow problem, and finally a realistic Stokes flow around an aircraft wing profile (a NACA airfoil). For each case we compare different preconditioners and measure how the runtime changes when we add more computing power.

The main findings are that not every preconditioner is equally useful on modern hardware. A method that needs fewer steps to get to the solution can still take more time if it is hard to parallelise. On the other hand, very simple preconditioners may be easy to parallelise, but not strong enough to reduce the amount of steps. For complex flow problems, we find that preconditioners must also respect the coupling block structure of the equations (for example, the relations between velocity and pressure) to be effective.

In short: getting good performance is not just about choosing the right iterative method, but also about matching the preconditioner and implementation to both the mathematics of the problem and the parallel hardware.

Abstract

This thesis investigates how preconditioned Krylov subspace methods perform and scale under shared-memory parallelism. The focus is on the Conjugate Gradient (CG) method for symmetric positive definite systems and the Generalized Minimal Residual (GMRES) method for non-symmetric systems. Both solvers are implemented in PyKokkos and applied to finite element discretisation generated with NGSolve. We look at a scalar Laplace problem, a Stokes-like vector problem, and a steady Stokes flow around a NACA,2412 airfoil.

For CG, we study Jacobi and symmetric Gauss–Seidel (SGS) preconditioning, strong and weak scaling up to 16 threads, and kernel-level timings. As the mesh is refined, the iteration count grows in line with the increasing condition number of the stiffness matrix. Jacobi reduces the iteration count only slightly but is cheap and fully parallel, leading to runtimes similar to, and sometimes slightly better than, unpreconditioned CG. SGS roughly halves the iteration count, but its forward/backward sweeps are largely sequential, which limits speed-up on many cores and can make SGS slower overall despite faster convergence.

For GMRES we analyse the influence of the restart parameter, preconditioning, and polynomial order. Higher-order vector elements lead to more off-diagonal entries in the system matrix, here scalar Jacobi becomes too weak and can even make restarted GMRES slower than using no preconditioner. SGS remains effective in terms of iterations, but this comes with the same parallel limitations as in CG. And overall GMRES shows poor strong and weak scaling on the tested CPUs. For the NACA Stokes system, Jacobi and SGS preconditioning fails, whereas a block preconditioner that respects the velocity–pressure structure shows rapid convergence.

Overall, the results show that good performance on shared-memory architectures requires preconditioners that both respect the block structure of the PDE and are highly parallelizable.

Contents

1 Introduction

In many areas of engineering, such as fluid dynamics, structural mechanics and electromagnetics, simulations often involve solving very large systems of linear equations. These systems typically arise from discretising partial differential equations using methods like the finite element method [1, Ch. 2]. For large problems, especially those with fine meshes or complex geometries, direct solvers quickly become infeasible because of their high computational cost and memory requirements [2].

Iterative solvers provide a more scalable option than direct methods, making it possible to solve problems that would otherwise be too computationally expensive [1, Chs. 6–7]. In practice, Krylov subspace methods such as the Conjugate Gradient (CG) and Generalized Minimal Residual (GMRES) methods are widely used and are often very effective. Their efficiency and robustness, however, depend strongly on the choice of preconditioner, which can dramatically influence convergence rates and overall performance [3, Ch. 1.2]. As multi-core architectures are now widespread, understanding how these solvers behave and scale under shared-memory parallelism is essential for achieving high performance in modern computations.

The primary goal of this thesis is to investigate how preconditioned Krylov subspace methods perform and scale under shared-memory parallelism. The following research questions guide this work:

> **Main question:** How do preconditioned Krylov subspace methods perform and scale under shared-memory parallelism?
>
> **Subquestions.** To answer the main question, we address the following subquestions:
>
> **Q1** How does the Conjugate Gradient method (CG) perform when solving the Laplace equation, and what are the key computational bottlenecks?
>
> **Q2** How do different preconditioners affect the convergence and performance of CG when solving the Laplace equation?
>
> **Q3** How does the performance of a preconditioned Generalized Minimal Residual method (GMRES) compare when solving the Stokes system, and how do the results relate to those observed for the Laplace equation?

Scope and Limitations

This thesis focuses exclusively on shared-memory parallelism. Although distributed-memory implementations are widely used in large-scale high-performance computing, including them here would require substantially more time and complexity than is feasible for this project. The experiments are conducted on CPU architectures only. The study also restricts itself to a subset of solvers and preconditioners, specifically the CG and GMRES methods with Jacobi and Gauss-Seidel preconditioners, in order to keep the analysis focused and the conclusions clear.

Thesis Structure

The remainder of this thesis is organised as follows.

- **Chapter 2** introduces iterative methods for linear systems. It starts with classical stationary schemes and then presents Krylov subspace methods, including Conjugate Gradient (CG) and GMRES, with an emphasis on their convergence properties and computational costs.
- **Chapter 3** focuses on preconditioning. It explains the motivation for preconditioning, discusses several common constructions (such as Jacobi and Gauss–Seidel), and outlines how they can be combined with Krylov methods to improve convergence.
- **Chapter 4** describes the model problem used throughout the thesis. It moves from the Laplace equation in strong form to the weak formulation, introduces the finite element discretisation on the unit square, and explains how the resulting linear systems are assembled with NGSolve.
- **Chapter 5** provides a brief overview of parallel computing on multi-core architectures. It introduces shared- and distributed-memory models, basic parallel programming concepts, and the notions of strong and weak scaling that are used later to interpret the timing results.
- **Chapter 6** details the implementation of the preconditioned CG solver in PyKokkos. It covers the mesh generation and sparse matrix storage, the decomposition of CG into basic kernels, and the implementation of Jacobi and symmetric Gauss–Seidel preconditioners. It also outlines how the same ideas can be extended to GMRES.
- **Chapter 7** presents the numerical results. It compares the convergence behaviour of the different preconditioners, analyses the strong- and weak-scaling behaviour on multi-core CPUs, and reports kernel-level timings to identify the main performance bottlenecks. The chapter concludes with a Stokes case based on a NACA airfoil to illustrate the solver on a more realistic geometry.
- **Chapter 8** summarises the main findings and offers recommendations for future work, including potential extensions of the implementation to other Krylov methods, more advanced preconditioners and more challenging model problems.

2

Iterative Methods for Linear Systems

This chapter establishes the theoretical foundations and presents an analysis of two principal classes of iterative methods for the numerical solution of large and sparse linear systems, $Ax = b$.

The first section introduces *classical stationary methods*, including Jacobi, and Gauss-Seidel. These methods are derived from the concept of a *matrix splitting* and are analysed for their fundamental convergence behaviour. Although no longer competitive as stand-alone solvers, these methods continue to play an important theoretical and practical role as *building blocks* or *smoothers* in multilevel algorithms such as multigrid.

The second section focuses on *Krylov subspace methods*, this includes the Conjugate Gradient method (CG) and the Generalized Minimal Residual method (GMRES). These methods are widely used in large-scale engineering simulations due to their rapid convergence, especially when combined with effective preconditioning strategies.

Finally Section 2.3 gives an overview of additional iterative method found in literature. As these methods fall outside the scope of this research, they will not be examined in further detail.

The goal of this chapter is to provide an overview of the theoretical principles of these methods. We will focus on the mathematical derivation and the resulting algorithmic properties, such as memory use, stability, and speed of convergence, to understand how they work and perform.

2.1. Classical Stationary Iterative Methods

Stationary iterative methods are characterized by a fixed iteration matrix that is applied at every step. As noted by van Gijzen [4], the foundation of these methods is the *matrix splitting* of the system matrix A.

Starting from the linear system

$$Ax = b,$$

we introduce a splitting

$$A = M - R,$$

where the matrix M is chosen to be easily invertible, and R is the remainder. This leads to the general stationary iteration

$$Mx_{k+1} = Rx_k + b.$$

By substituting $R = M - A$, the iteration can be rewritten in the *residual-correction form*

$$x_{k+1} = x_k + M^{-1}(b - Ax_k).$$

The vector $r^{(k)} = b - Ax^{(k)}$ is the residual at iteration k, which measures how far the current estimate is from satisfying the system. The choice of M determines two things: the computational cost (since we

must be able to efficiently apply M^{-1} at each step) and the convergence rate. This rate is controlled by the iteration matrix G, which is defined as

$$G = M^{-1}R = I - M^{-1}A$$

The method is guaranteed to converge if and only if the spectral radius of G, denoted $\rho(G)$, is strictly less than 1 [1, Ch. 4.2]. The spectral radius $\rho(G)$ is the largest absolute value among all eigenvalues of G, and its value determines how quickly the iterative process approaches the exact solution [1, Ch. 1.2, Ch. 4.2].

The next subsections detail classical methods arising from specific, choices of the splitting matrix M.

2.1.1. Richardson Method

The Richardson method corresponds to the simplest non-trivial splitting where the matrix M is proportional to the identity matrix, $M = \alpha^{-1}I$, or the unrelaxed case $M = I$.

The unrelaxed Richardson iteration is

$$x_{k+1} = x_k + (b - Ax_k) = b + (I - A)\, x_k.$$

The introduction of a scalar *relaxation parameter* $\alpha > 0$ yields the general form:

$$x_{k+1} = x_k + \alpha(b - Ax_k).$$

The corresponding iteration matrix is $G_\alpha = I - \alpha A$.

Convergence requires $\rho(I - \alpha A) < 1$. For a symmetric positive definite (SPD) matrix A with eigenvalues $\lambda_i(A)$, convergence is guaranteed if

$$0 < \alpha < \frac{2}{\lambda_{\max}},$$

and the *optimal relaxation parameter* that minimizes the spectral radius of the iteration matrix is

$$\alpha_{\text{opt}} = \frac{2}{\lambda_{\max} + \lambda_{\min}},$$

where $\lambda_{\max}$ and $\lambda_{\min}$ are the extreme eigenvalues of A [1, Ch. 4.2].

Algorithmic Properties

- **Simplicity**: The method involves only one matrix-vector multiplication per step, making it computationally cheap per iteration.
- **Role in Multilevel Methods**: The Richardson structure, particularly in its relaxed form, is frequently used as a smoother in multigrid algorithms due to its low cost and effectiveness in reducing high-frequency error components [1, Ch. 4].
- **Limitation**: Convergence is at best linear, and the method is highly sensitive to the spectral properties of A [1, Ch. 4].

2.1.2. Jacobi Method

The Jacobi method is derived from the decomposition of A into its diagonal (D), strictly lower (L), and strictly upper (U) triangular parts: $A = L + D + U$ [1, Ch. 4.1].

The splitting $M = D$ and $R = -(L + U)$ leads to the iteration

$$Dx_{k+1} = -(L + U)x_k + b,$$

which is equivalent to the residual-correction form

$$x_{k+1} = x_k + D^{-1}(b - Ax_k).$$

Convergence is guaranteed if the matrix A is strictly diagonally dominant [1, Ch. 4.1]. The rate of convergence is determined by the spectral radius $\rho(G_J)$ of the Jacobi iteration matrix

$$G_J = I - D^{-1}A.$$

Algorithmic Properties

Parallelism: The calculation of each component of $x^{(k+1)}$ depends only on the components of $x^{(k)}$ from the previous iteration. This complete independence makes the Jacobi method fully parallelizable [5, Ch. 2].

Application: Despite its typically slow convergence rate, the method is primarily used in advanced schemes, serving as a highly parallel *smoother* in multigrid methods or as a basic preconditioner for more advanced Krylov solvers [1, Ch. 4.1, Ch. 13.2].

2.1.3. Gauss-Seidel Method

The Gauss–Seidel method improves upon the Jacobi method by using the most recently updated components of the solution vector within the same iteration, making it an sequential scheme [1, Ch. 4.1].

The Gauss-Seidel splitting is $M = L + D$ and $R = -U$. The iteration is defined by the lower triangular solve

$$(L + D)x_{k+1} = -Ux_k + b,$$

this sequential dependence is clearly seen in the component form

$$x_{i,k+1} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_{j,k+1} - \sum_{j=i+1}^{n} a_{ij} x_{j,k} \right).$$

[1, Ch. 4.1].

For many matrix classes, the spectral radius of the Gauss-Seidel iteration matrix is significantly smaller than that of the Jacobi matrix, often leading to a convergence speed roughly twice as fast as Jacobi's method [1, Ch. 4.3]**Heath_Solomonik**.

The method is guaranteed to converge when the system matrix A is symmetric positive definite (SPD) [5, Ch. 2.2]. And as before, the rate of convergence is determined by the spectral radius $\rho(G_{GS})$, the Gauss-Seidel iteration matrix is

$$G_{GS} = -(D + L)^{-1}U.$$

Algorithmic Properties

Data Dependency and Parallelism: The core characteristic of Gauss-Seidel is its sequential update mechanism. Because the computation of $x_{j,k+1}$ immediately uses the newly computed components $x_{j,k+1}$ for $j < i$, this introduces a significant data dependency. This dependency limits its parallel efficiency on shared and distributed-memory architectures.

Implementation: Despite its sequential nature, the method is straightforward to implement and is widely used inside more advanced solvers such as multigrid methods, where repeated Gauss-Seidel sweeps effectively damp out oscillatory (high-frequency) components of the error [1, Ch. 13.2].

2.2. Krylov Subspace Methods

Krylov subspace methods are a class of non-stationary iterative solvers that construct a sequence of approximations x_k from a subspace generated by repeated application of the matrix A [1, Ch. 6.1]. Given an initial guess x_0 and residual $r_0 = b - Ax_0$, the *Krylov subspace* of dimension k is defined as

$$\mathcal{K}_k(A; r_0) = \operatorname{span}\{r_0, Ar_0, A^2r_0, \ldots, A^{k-1}r_0\}.$$

At iteration k, the approximate solution is sought in the affine space $x_0 + \mathcal{K}_k(A; r_0)$. The iterate x_k is selected from this space by enforcing a *projection condition* [1, Ch. 5.1], typically

- a **Galerkin condition**, used by methods such as Conjugate Gradient (CG), or
- a **minimal residual condition**, used by methods such as GMRES [1, Ch. 5, 6].

The choice of projection criterion influences key properties of the method, including how optimality is defined on the search space and how memory and computational cost grow with k [1, Ch. 5, 6].

2.2.1. Conjugate Gradient Method

The Conjugate Gradient (CG) method is a Krylov subspace solver for linear systems

$$Ax = b,$$

where A is *symmetric positive definite* (SPD) [1, Ch. 6.7]. This assumption is important for two reasons. First, $x^T Ax > 0$ for all nonzero x, which makes $\sqrt{x^T Ax}$ a proper norm. Second, symmetry ensures the orthogonality relations used by CG are consistent ($x^T Ay = y^T Ax$). Together, these properties make the CG update formulas well-defined (no division by zero in exact arithmetic) and guarantee monotonic convergence in the A-norm [1, Ch. 6.7].

CG constructs approximations x_k in the affine space

$$x_0 + \mathcal{K}_k(A, r_0), \qquad r_0 = b - Ax_0,$$

where the Krylov subspace is

$$\mathcal{K}_k(A, r_0) = \operatorname{span}\{r_0, Ar_0, \ldots, A^{k-1} r_0\}.$$

Optimality and search directions

Let x^* be the exact solution and $e_k = x_k - x^*$ the error. For SPD matrices, the A-norm

$$\|v\|_A = \sqrt{v^T Av}$$

is a true norm. CG chooses x_k such that the error is minimal in this norm over the current search space:

$$\|x_k - x^*\|_A = \min_{x \in x_0 + \mathcal{K}_k(A, r_0)} \|x - x^*\|_A.$$

This is achieved by generating search directions p_k that are mutually A-conjugate,

$$p_i^T Ap_j = 0 \quad \text{for } i \neq j.$$

In practice, CG updates $x_{k+1} = x_k + \alpha_k p_k$, choosing α_k so the new residual is orthogonal to the current Krylov space, and forms p_{k+1} from the new residual plus a correction that preserves A-conjugacy.

A key practical benefit is that conjugacy leads to a *short recurrence*: each new iterate can be computed using only a small number of vectors from the previous step, giving CG low memory use and a predictable cost per iteration [1, Ch. 6.7].

Residual and stopping criterion

An important quantity in CG is the residual

$$r_k = b - Ax_k,$$

which measures how well the current iterate satisfies the linear system. Convergence is measured via the Euclidean residual norm $\|r_k\|_2$. An absolute tolerance ($\|r_k\|_2 < \varepsilon_{\text{abs}}$) can be misleading, because the scale of the right-hand side b may vary between test problems, e.g. a residual norm of 10^{-6} is small if $\|b\|_2 \approx 1$, but relatively large if $\|b\|_2 \approx 10^{-2}$.

In the implementation used in this thesis, the iteration is terminated as soon as

$$\|r_k\|_2 \leq \texttt{tol} \cdot \|b\|_2, \tag{2.1}$$

where `tol` is a user-specified tolerance. inequality (2.1) enforces a bound on the *relative* residual. This makes it possible to compare runs with different problem sizes and right-hand sides [5].

A maximum number of iterations $k_{\max}$ is also enforced to prevent excessive runtimes in problematic cases.

Algorithmic properties

- **Finite termination in exact arithmetic**: In exact arithmetic, CG reaches the exact solution in at most n iterations for an $n \times n$ system [3, Ch. 5.1].
- **Restriction to SPD systems**: If A is nonsymmetric or indefinite, CG may fail, in those cases one typically uses a different Krylov method such as GMRES [3].

2.2.2. Generalized Minimal Residual Method

The Generalized Minimal Residual (GMRES) method, proposed by Saad and Schultz [6], is a widely used Krylov subspace method for solving nonsymmetric linear systems $Ax = b$. GMRES constructs iterates of the form

$$x_k = x_0 + V_k y_k,$$

where $V_k = [v_1, v_2, \ldots, v_k]$ is an orthonormal basis of the Krylov subspace generated by the initial residual $r_0 = b - Ax_0$ [1, Ch. 6.5].

The orthonormal basis vectors $v_1, \ldots, v_k$ are constructed by the *Arnoldi process*, which can be viewed as applying (modified) Gram–Schmidt orthogonalization to the sequence

$$r_0,\ Ar_0,\ A^2 r_0,\ \ldots$$

First, the initial residual is normalized,

$$v_1 = \frac{r_0}{\|r_0\|_2}.$$

Then, at step $j = 1, 2, \ldots$, one computes

$$w = Av_j,$$

and orthogonalizes w against the existing basis vectors:

$$h_{ij} = v_i^T w, \qquad w \leftarrow w - h_{ij} v_i, \qquad i = 1, \ldots, j.$$

The remaining vector is normalized,

$$h_{j+1,j} = \|w\|_2, \qquad v_{j+1} = \frac{w}{h_{j+1,j}},$$

unless $h_{j+1,j} = 0$, in which case the process terminates. This procedure is precisely a Gram–Schmidt orthogonalization of Av_j against the previously computed basis vectors [1, Ch. 6.5].

In matrix form, the Arnoldi process yields the relation

$$AV_k = V_{k+1} \bar{H}_k,$$

where $V_{k+1} = [v_1, \ldots, v_{k+1}]$ and $\bar{H}_k \in \mathbb{R}^{(k+1)\times k}$ is an upper Hessenberg matrix. The j-th column of $\bar{H}_k$ collects the coefficients $\{h_{ij}\}_{i=1}^{j+1}$ that express Av_j in the current basis,

$$Av_j = \sum_{i=1}^{j+1} h_{ij} v_i, \qquad j = 1, \ldots, k.$$

By construction, $\bar{H}_k$ is nearly triangular, with nonzero entries only on and above the first subdiagonal [7].

Optimality and Recurrence

GMRES minimizes the Euclidean norm of the residual $\|b - Ax_k\|_2$ at every iteration. Using the Arnoldi relation, this minimization reduces to a small least-squares problem involving only $\bar{H}_k$:

$$\min_{y_k \in \mathbb{R}^k} \left\| \beta e_1 - \bar{H}_k y_k \right\|_2, \qquad \beta = \|r_0\|_2.$$

Solving this yields the coefficients y_k and hence the GMRES iterate $x_k = x_0 + V_k y_k$.

This least-squares problem can be solved via a QR factorization of $\bar{H}_k$ [1, Ch. 6.5]. Rather than recomputing the factorization from scratch at each step, GMRES updates it incrementally using Givens

rotations that introduce zeros in the subdiagonal entries of $\bar{H}_k$. This allows one to track the residual norm efficiently at every iteration without explicitly forming the full QR factorization. The concrete implementation of this Givens-based update is described in Chapter 6.

Because the residual norm is explicitly minimized at each iteration, GMRES guarantees a *monotonically* nonincreasing residual norm in exact arithmetic. In particular, the method converges in at most n steps for an $n \times n$ system, but the cost of storing and orthogonalising all k basis vectors grows like $\mathcal{O}(k^2)$ after k iterations. This motivates restarted variants such as GMRES(m) [7].

Algorithmic Properties

- **Robustness**: The guaranteed monotonic decrease of the residual norm in exact arithmetic ensures stability, making GMRES highly reliable for challenging nonsymmetric problems [3, Ch. 6.1].
- **Memory and cost growth**: The Arnoldi process requires storing the entire orthonormal basis V_k (of size $n \times k$), and the orthogonalization cost increases quadratically with k. For large systems, this escalating memory and computational cost necessitates the use of a *restarted variant, GMRES(m)*, where the process is truncated and restarted after m iterations [3, Ch. 6.1].

Restarted GMRES(m)

In practice, one typically chooses a restart parameter $m \ll n$. The GMRES(m) algorithm proceeds in *cycles*: starting from an initial guess x_0 and residual $r_0 = b - Ax_0$, one performs m standard GMRES iterations to obtain

$$x_m = x_0 + V_m y_m,$$

which minimizes the residual norm over the Krylov subspace $\mathcal{K}_m(A, r_0)$. At this point, the Arnoldi basis V_m and the Hessenberg matrix $\bar{H}_m$ are discarded, and a new cycle is started with

$$x_0^{\text{new}} = x_m, \qquad r_0^{\text{new}} = b - Ax_m,$$

this process is repeated until a prescribed convergence criterion is satisfied [3, Ch. 6.1].

Restarting controls both the memory footprint and the per-cycle computational cost, but it also comes at a price: GMRES(m) no longer minimizes the residual over the *entire* accumulated Krylov subspace, only over the most recent $\mathcal{K}_m(A, r_0^{\text{current}})$ [3, Ch. 6.1]. As a consequence, convergence may slow down or even stagnate if important spectral information is lost at the restart [1, Ch. 6.5]. In practice, the choice of m represents a trade-off between robustness and efficiency, and is often problem dependent. In the numerical experiments of this thesis, we use GMRES(m) with different values of m to analyse this trade-off.

2.3. Summary of Iterative Method Properties

Table 2.1 provides an overview of common iterative methods found in literature. The table shows when each method is most useful and notes key considerations for their use.

Table 2.1: Comparison of Iterative Methods for Linear Systems: Algorithmic Properties and Requirements. The table combines properties discussed in [1], [3] and [5].

Method	Key Properties and Requirements	Primary Algorithmic Limitations
I. Classical Stationary Methods (Fixed Iteration Matrix)		
Richardson	Requires A to satisfy spectral radius $\rho(I - \alpha A) < 1$; minimal computational cost per iteration.	Highly sensitive to $\lambda_{\max}(A)$; convergence is slow (linear) and parameter-dependent.
Jacobi	Convergence guaranteed for strictly diagonally dominant or certain SPD A; fully parallelizable updates.	Slow linear convergence; iteration matrix $\rho(D^{-1}(L + U))$ is often close to unity.
Gauss-Seidel	Convergence faster than Jacobi for many systems; guaranteed for SPD A.	Sequential nature of updates strictly limits modern parallel efficiency.
SOR	Potential for significant convergence acceleration through parameter ω.	Performance critically dependent on *a priori* unknown optimal parameter ω.
II. Krylov Subspace Methods (Projection Techniques)		
Conjugate Gradient (CG)	**Optimality**: Minimizes the A-norm of the error; requires A to be SPD.	Breaks down or diverges for nonsymmetric or indefinite A.
Conjugate Residual (CR)	**Optimality**: Minimizes the residual norm for SPD A; short recurrence.	Less efficient than CG per iteration; restricted to SPD systems.
GMRES	**Optimality**: Minimizes the Euclidean residual norm; applicable to general nonsymmetric A.	High memory and computational cost due to full orthogonalization; necessitates restarting.
BiCG	Extends short recurrence to nonsymmetric A via bi-orthogonality; low memory.	Non-monotonic, irregular convergence; susceptible to numerical breakdown (division by zero).
BiCGSTAB	Stabilized BiCG; achieves smoother convergence by local residual minimization; low memory.	No global optimality guarantee; prone to stagnation or breakdown in the smoothing step.

3

Preconditioning Techniques

The iterative methods discussed in Chapter 2 depend on the properties of the system matrix for their efficiency. For many algorithms, including the Conjugate Gradient method, the convergence rate is closely related to the *condition number*, $\kappa(A)$, of the system matrix A [1, Ch. 6, 9]. For a symmetric positive definite (SPD) matrix, this number is defined as the ratio of the largest to the smallest eigenvalue

$$\kappa(A) = \frac{\lambda_{\max}}{\lambda_{\min}}.$$

A large condition number indicates an ill-conditioned system, which is more sensitive to numerical errors and often results in slow convergence of iterative solvers. Consequently, such systems may require a large number of iterations. In practice, the sparse matrices that arise from fine discretizations of Partial Differential Equations (PDEs) are frequently ill-conditioned [1, Ch. 9].

Preconditioning is a technique designed to improve the conditioning of a linear system $Ax = b$, making it more suitable for iterative methods [5, Ch. 3]. The idea is to introduce a *preconditioner matrix* M, which approximates A in a way that is easier to invert. For instance if a M approximates the coefficient matrix A in some way, the system is transformed into

$$M^{-1}Ax = M^{-1}b.$$

The iterative solver then operates on this modified system, using $M^{-1}A$ as the coefficient matrix and $M^{-1}b$ as the right-hand side, while still solving for the original unknown x [5, Ch. 3].

An effective preconditioner M should approximate A such that $M^{-1}A$ is well-conditioned, ideally with its eigenvalues clustered around one [1, Ch. 10]. The extreme case $M = A$ yields $M^{-1}A = I$, and the system would be solved in a single iteration. Although trivial, this example illustrates the goal of preconditioning, which is to make the transformed system as close to the identity as possible while keeping the preconditioner cheap to apply.

As noted in [5], ”There is a trade-off between the cost of constructing and applying the preconditioner and the gain in convergence speed.” A complex preconditioner might reduce the iteration count dramatically but be so expensive to apply at each step that the total computational cost increases.

3.1. Classes of Preconditioners

3.1.1. Stationary Methods as Preconditioners

In this section we will go over different classes of preconditioners. The stationary methods introduced in Chapter 2 are based on a matrix splitting $A = M - R$. This same splitting can be used to define a preconditioner, where M serves as an approximation to A.

Jacobi Preconditioner
The **Jacobi preconditioner** sets M equal to the diagonal of A:

$$M_J = \text{diag}(A) = D.$$

The preconditioning step $z = M_J^{-1} r$ then reduces to element-wise division, $z_i = r_i / A_{ii}$. Its main advantage is that each component can be computed independently, which makes the method well suited for parallel computation, though the improvement in conditioning is often limited [1, Ch. 10].

Gauss-Seidel Preconditioning
The **Gauss-Seidel (GS)** method, defined in Section 2.1.3, uses the splitting $M_{GS} = D + L$, where D is the diagonal and L the strictly lower-triangular part of A. While M_{GS} often leads to faster convergence than Jacobi, it is not symmetric, even when A is. This prevents its direct use in the Conjugate Gradient method, which requires both the system matrix and the preconditioner to be symmetric positive definite [1, Ch. 9].

A symmetric alternative is the **Symmetric Gauss-Seidel (SGS)** preconditioner, defined by

$$M_{SGS} = (D + L)D^{-1}(D + U),$$

where U is the strictly upper-triangular part of A. For symmetric problems, $U = L^T$, and M_{SGS} is symmetric by construction. The preconditioning step $z = M_{SGS}^{-1} r$ consists of:

1. Solving $(D + L)y = r$ (forward substitution),
2. Solving $(D + U)z = Dy$ (backward substitution).

Although SGS typically reduces the number of iterations compared to Jacobi, the forward and backward solves introduce sequential dependencies that limit parallel performance [1, Ch. 10].

3.1.2. Incomplete LU Factorization

Incomplete LU factorization (ILU) preconditioners [1, Ch. 10.3] extend the idea of matrix splitting by constructing approximate lower and upper triangular factors of A. Instead of computing a full LU decomposition, which often destroys sparsity, ILU methods retain only selected nonzero entries. The resulting matrices $\tilde{L}$ and $\tilde{U}$ form the preconditioner

$$M = \tilde{L}\tilde{U} \approx A.$$

The simplest version, ILU(0), restricts the sparsity pattern of $\tilde{L}$ and $\tilde{U}$ to match that of A. Any fill-in that would appear during a full factorization is discarded [1, Ch. 10]. This keeps the memory cost and computational effort similar to those of a sparse matrix-vector product.

Applying an ILU preconditioner involves two triangular solves:

1. Solve $\tilde{L}y = r$ (forward substitution),
2. Solve $\tilde{U}z = y$ (backward substitution).

ILU(0) is often effective for a wide range of problems because it balances quality and cost without requiring problem-specific tuning. However, the triangular solves remain sequential, which limits the scalability on parallel architectures.

3.1.3. Multigrid Methods

Multigrid (MG) methods [1, Ch. 13] form a class of algorithms designed to efficiently eliminate errors on all spatial scales. They can be used both as standalone solvers and as preconditioners for Krylov methods such as CG.

The key idea is that different error components behave differently across grid levels: high-frequency errors can be reduced effectively by simple smoothers (like Jacobi or Gauss-Seidel), while low-frequency errors are better handled on coarser grids. Multigrid methods combine these effects through a hierarchy of grids.

A single **V-cycle**—often used as a preconditioner—proceeds as follows:

1. **Pre-smoothing:** Apply a few iterations of a simple smoother (e.g., Jacobi) on the fine grid to reduce high-frequency errors.
2. **Restriction:** Compute the residual $r = b - Ax$ and transfer it to a coarser grid, where low-frequency components appear as higher-frequency ones.
3. **Coarse-grid solve:** Solve the residual equation on the coarser grid. This step can itself call another V-cycle recursively or use a direct solve if the grid is small enough.
4. **Prolongation:** Interpolate the coarse-grid correction back to the fine grid and update the fine-grid approximation.
5. **Post-smoothing:** Apply the smoother again to remove high-frequency errors reintroduced by interpolation.

Each V-cycle acts as an approximate inverse of A and can therefore serve as a powerful preconditioner. In ideal cases, the total computational cost grows linearly with the number of unknowns, $O(N)$, which represents optimal efficiency. However implementing this is beyond the scope of this work.

4

Problem Setup

4.1. From PDE to Weak Formulation

We consider the Laplace equation on the unit square with homogeneous Dirichlet boundary conditions. In strong form, the problem reads

$$-\Delta u = f(\mathbf{x}) \quad \text{for } \mathbf{x} \in \Omega = (0,1)^2, \qquad u = 0 \text{ on } \partial\Omega. \tag{4.1}$$

In this formulation, the differential operator involves second derivatives of u, and the equation is required to hold at every point $\mathbf{x} \in \Omega$. A classical (strong) solution therefore has to be sufficiently smooth (roughly speaking, twice differentiable) and satisfy the PDE pointwise. For finite element methods, which are based on piecewise polynomial functions, this is inconvenient. To avoid these difficulties, we reformulate the problem in a *weak form*. The idea is to test the equation against a family of functions and integrate over the domain. In this way we obtain an identity that only contains first derivatives and that has to hold in an integrated (or averaged) sense rather than pointwise.

We start by multiplying the PDE by a test function v that vanishes on the boundary and integrating over Ω:

$$\int_\Omega \big(-\Delta u\big)\, v\, dx = \int_\Omega f(\mathbf{x})\, v(\mathbf{x})\, dx. \tag{4.2}$$

For the moment, v can be thought of as an arbitrary smooth function with $v = 0$ on $\partial\Omega$, later we will specify the function space more precisely.

Next, we apply integration by parts to the left-hand side. This moves one derivative from u to v and produces a boundary term:

$$\int_\Omega \big(-\Delta u\big)\, v\, dx = \int_\Omega \nabla u \cdot \nabla v\, dx - \int_{\partial\Omega} \nabla u \cdot \mathbf{n}\, v\, ds, \tag{4.3}$$

where $\mathbf{n}$ is the outward unit normal vector on $\partial\Omega$. Because we choose test functions v that vanish on $\partial\Omega$, the boundary integral is zero, and we obtain

$$\int_\Omega \nabla u \cdot \nabla v\, dx = \int_\Omega f(\mathbf{x})\, v(\mathbf{x})\, dx \qquad \text{for all test functions } v. \tag{4.4}$$

The homogeneous Dirichlet boundary condition $u = 0$ on $\partial\Omega$ is now enforced through the choice of function space. Both the unknown u and the test functions v are taken from a space of functions that are weakly differentiable and have zero boundary values on $\partial\Omega$ [8]. More precisely, we work in the Sobolev space $H_0^1(\Omega)$, which consists of functions with square-integrable gradients and vanishing boundary values on $\partial\Omega$.

We can now state the weak formulation:

Find $u \in H_0^1(\Omega)$ such that

$$\int_\Omega \nabla u \cdot \nabla v \, dx = \int_\Omega f(\mathbf{x})\, v(\mathbf{x})\, dx \qquad \forall v \in H_0^1(\Omega). \tag{4.5}$$

Compared to the strong form, this weak formulation has two main advantages for the finite element method. First, it only involves first derivatives of u and v. Second, it is formulated in the Sobolev space $H_0^1(\Omega)$. This space is large enough to contain the relevant solutions of the PDE, but structured enough to be approximated by piecewise polynomial functions on a mesh [8], [9]. We will not go into further detail on Sobolev spaces here. A formal introduction can be found, for example, in [8], [9]. In the next section we construct a finite-dimensional approximation space and derive the corresponding discrete linear system.

4.2. Finite Element Approximation

The weak formulation is still an infinite-dimensional problem, we seek $u \in H_0^1(\Omega)$ such that the equation holds for all $v \in H_0^1(\Omega)$. Computers, however, can only handle finitely many unknowns. The main idea of the finite element method is therefore to approximate $H_0^1(\Omega)$ by a finite-dimensional subspace V_h and to look for an approximate solution in V_h [8], [9].

Triangulation and finite element space. We start by partitioning the domain Ω into smaller pieces. To do this we introduce a mesh (triangulation) $\mathcal{T}_h$ of Ω, where h denotes a mesh size (e.g. the maximal edge length). Each subregion of $\mathcal{T}_h$ is called an *element*, and the points at which elements meet are called *nodes* [10]. On the unit square, we obtain a collection of triangles that cover the domain without overlaps or gaps.

Next, we define a finite element space V_h on this mesh. In this thesis we use the standard space of continuous, piecewise linear functions on $\mathcal{T}_h$ that vanish on the boundary. The space V_h serves as our approximation of the infinite-dimensional space in the weak formulation.

Shape functions and degrees of freedom. The space V_h is spanned by a set of *shape functions*. To each node i we associate a shape function N_i with the property

$$N_i(x_j) = \delta_{ij}.$$

In other words, N_i takes the value 1 at node i, the value 0 at all other nodes, and varies linearly on the elements that touch node i. In Figure 4.1 we illustrate such a “hat-shaped” basis function [11].

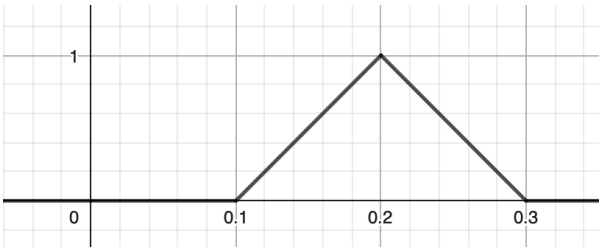


Figure 4.1: Hat function created using Geogebra [12]

The functions $\{N_i\}_{i=1}^n$ form a basis of a finite-dimensional subspace $V_h \subset H_0^1(\Omega)$. Any function $u_h \in V_h$ can therefore be written as

$$u_h(x) = \sum_{i=1}^{n} u_i N_i(x),$$

where the coefficients u_i are the *degrees of freedom*. Thus, the finite element method replaces the problem of finding an unknown (infinite-dimensional) function u by the problem of finding a finite collection of unknown numbers $\{u_i\}_{i=1}^n$, which we can solve using computers.

Discrete weak problem and linear system. To derive the discrete problem, we restrict the weak formulation from equation (4.5) to the finite element space V_h. We then try to find $u_h \in V_h$ such that

$$\int_\Omega \nabla u_h \cdot \nabla v_h \, dx = \int_\Omega f(\mathbf{x})\, v_h(\mathbf{x}) \, dx \qquad \forall v_h \in V_h.$$

And since the basis functions $N_1, \ldots, N_n$ span V_h, it is enough to impose this equation for test functions $v_h = N_i$, $i = 1, \ldots, n$. Substituting the expansion

$$u_h(x) = \sum_{j=1}^{n} u_j N_j(x)$$

into the left-hand side gives

$$\int_\Omega \nabla u_h \cdot \nabla N_i \, dx = \sum_{j=1}^{n} u_j \int_\Omega \nabla N_j \cdot \nabla N_i \, dx.$$

This leads to the linear system

$$\sum_{j=1}^{n} A_{ij} u_j = F_i, \qquad i = 1, \ldots, n,$$

or in matrix form

$$Au = F,$$

where

$$A_{ij} = \int_\Omega \nabla N_i \cdot \nabla N_j \, dx, \qquad F_i = \int_\Omega f N_i \, dx.$$

The matrix A is called the *stiffness matrix*. It is symmetric and positive definite, and it is sparse. Each shape function N_i overlaps only with shape functions associated with adjacent elements, so each row of A contains only a small number of nonzero entries [9]. These properties are crucial for the efficient solution of the linear system with iterative methods.

The problem setup in this chapter focused on the Laplace equation with a scalar unknown u. Later in this thesis we also consider a flow problem where the unknown is a velocity field $u = (u_x, u_y)$ with two components. The underlying ideas remain the same: we write the equations in weak form, choose suitable finite element spaces, and obtain a large sparse linear system that we solve with iterative methods. The details for this Stokes-like model problem are outlined in the next subsection.

4.2.1. Extension to a vector-valued Stokes-like problem

In the GMRES experiments we move from a scalar problem (the Laplace equation) to a flow problem in a two-dimensional channel, derived from an NGSolve example [13]. The unknown is no longer a single scalar u, but a velocity field

$$u = (u_x, u_y) : \Omega \to \mathbb{R}^2,$$

so there are two components at each point in the domain.

The geometry is a rectangular channel of length 2.0 and height 0.41. The left boundary is treated as an inlet, the right boundary as an outlet, and the top and bottom boundaries as solid walls. On the walls we impose a no-slip condition $u = 0$, and at the inlet we prescribe a horizontal parabolic inflow profile

$$u_{\text{in}}(y) = \big(1.5 \cdot 4y(0.41 - y)/0.41^2,\ 0\big),$$

which is zero at the walls and maximal in the centre of the channel. At the outlet we use a natural (do-nothing) boundary condition. Informally, this means that we do not prescribe any additional forces at the outlet and let the flow leave the domain freely. There is no body force (such as gravity) in the domain, so the flow is driven purely by the imposed inflow.

As a model for the velocity we consider a Stokes-like equation of the form

$$-\nu\Delta u + \varepsilon u = f \quad \text{in } \Omega,$$

with viscosity $\nu = 10^{-3}$, a small mass term $\varepsilon = 10^{-6}$ for improved conditioning, and a body force f that we set to zero in the experiments [13]. The boundary conditions are given by the inflow profile and the no-slip walls as described above.

The weak form is obtained in the same way as for the Laplace problem: we multiply the equation by a vector-valued test function $v = (v_x, v_y)$, integrate over the domain, and integrate by parts. This leads to

$$\int_\Omega \nu\, \nabla u : \nabla v \, dx + \varepsilon \int_\Omega u \cdot v \, dx = \int_\Omega f \cdot v \, dx \qquad \text{for all test functions } v,$$

where ":" denotes the Frobenius (component-wise) inner product of the Jacobian matrices, i.e.

$$\nabla u : \nabla v = \sum_{i=1}^{2} \sum_{j=1}^{2} \frac{\partial u_i}{\partial x_j} \frac{\partial v_i}{\partial x_j},$$

so we multiply all corresponding partial derivatives of u and v and sum them up [1, Ch. 1.5].

To discretise this problem we again use a finite element space built from continuous, piecewise polynomial functions on a triangular mesh. As in the Laplace case, the mesh size controls how fine the grid is, and the polynomial order k controls how high the degree of the shape functions is (with $k = 1$ corresponding to piecewise linear functions, $k = 2$ to quadratics, and so on) [8, Ch. 3.2].

For a given polynomial order k we first define a scalar space V_h of order k and then form the vector-valued space

$$V_h^{\text{vec}} = V_h \times V_h,$$

so that each node carries two degrees of freedom, one for each velocity component. The discrete solution $u_h \in V_h^{\text{vec}}$ and test functions $v_h \in V_h^{\text{vec}}$ are expanded in vector-valued shape functions in the same way as before. This leads to a linear system

$$Au = F,$$

where A is a sparse matrix.

In the GMRES chapter we use this Stokes-like system for several polynomial orders (in particular $k = 1$ and $k = 3$) to study how increasing the approximation order, affects preconditioning and parallel performance.

4.2.2. Stokes flow around a NACA airfoil

Besides the model problems on simple geometries, we also consider a more realistic fluid flow example taken from the NGSolve documentation [13]. Here we describe the mathematical setup.

The domain $\Omega \subset \mathbb{R}^2$ represents a two-dimensional channel with a NACA 2412 airfoil placed inside and rotated by 4° to model a small angle of attack. The unknowns are the velocity field $u : \Omega \to \mathbb{R}^2$ and the pressure $p : \Omega \to \mathbb{R}$. We consider the steady Stokes equations

$$-\nu\Delta u + \nabla p = 0, \qquad \nabla \cdot u = 0 \quad \text{in } \Omega,$$

with viscosity $\nu = 10^{-4}$.

The boundary is divided into four parts: an inflow boundary, an outflow boundary, the airfoil surface, and the outer channel walls. On the inflow boundary we prescribe a constant horizontal velocity

$$u = (5, 0)^{\mathsf{T}},$$

while on the airfoil and the outer walls we impose no-slip conditions $u = 0$. And as before, at the outflow boundary we use the natural "do-nothing" condition that comes directly from the weak formulation.

For the finite element discretisation we use a vector-valued H^1 space of order 3 for the velocity and a scalar H^1 space of order 2 for the pressure, this is called a Taylor-Hood pair.

$$V = \mathsf{VectorH1}(\Omega, k = 3), \qquad Q = H^1(\Omega, k = 2), \qquad X = V \times Q.$$

As in the Laplace case, we obtain a weak formulation by multiplying the equations with test functions $(v, q) \in X$ and integrating by parts [8, Ch. 12]. This leads to the following problem: find $(u, p) \in X$ such that

$$a\big((u,p),(v,q)\big) = \int_\Omega \nu\, \nabla u : \nabla v \, dx - \int_\Omega (\nabla \cdot u)\, q \, dx - \int_\Omega (\nabla \cdot v)\, p \, dx = 0 \qquad \text{for all } (v, q) \in X,$$

with the boundary conditions imposed as described above.

Realization in NGSolve. In this thesis, the meshes, finite element spaces, and linear systems are generated with NGSolve/Netgen. NGSolve constructs a triangular mesh and a piecewise linear finite element space H^1. From this, it assembles the stiffness matrix A and load vector F matching the bilinear and linear forms above. Different problem sizes are obtained by varying the global mesh size parameter `maxh`; the resulting meshes and matrix sizes are described in more detail in Chapter 6.

5

Parallel Computing on Multi-Core Architectures

In the previous chapters we introduced the numerical methods that can be used to solve linear systems that arise from the finite element discretization of the Laplace problem. This chapter introduces the concepts needed to understand how the methods from Chapter 2 are optimized on modern multi-core hardware, such as the compute nodes of the TU Delft DelftBlue supercomputer [14]. We first compare serial and parallel execution, then discuss memory architectures and programming interfaces, and finally look at limits to parallel scalability [15].

5.1. From Serial to Parallel Execution

In theory parallelism allows the execution time of suitable algorithms to decrease significantly. For large-scale simulations, such as those involving fine finite element meshes, running computations in serial on a single core quickly becomes impractical. Moreover, memory capacity on a single core is limited, which further increases the need for parallel computation [15].

Figure 5.1 shows the serial execution model: the program is viewed as a sequence of instructions, and a single processor executes them one after another. Each step must wait for the previous one to finish before it can start.

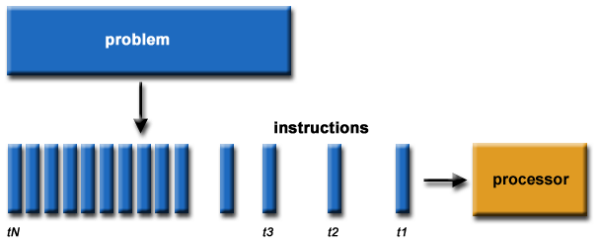


Figure 5.1: Taken from [15]. Serial execution model: a single processor executes one sequence of instructions step by step.

An alternative to this is a parallel execution model, shows in Figure 5.2. Here, the problem is decomposed into multiple parts that can be processed at the same time by different processors or cores. If the work can be divided into equal parts that do not need to communicate, the total runtime can be significantly reduced.

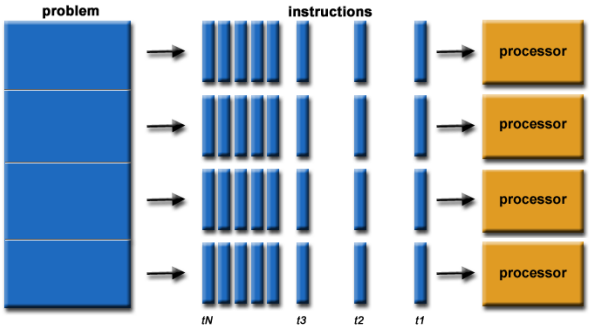


Figure 5.2: Taken from [15]. Parallel execution model: the work is split into four sequences of instructions that are executed simultaneously using four processors.

In this thesis, the main examples of parallelizable work are the vector operations and sparse matrix-vector products in the Conjugate Gradient and GMRES solver. Each entry of the solution and residual vectors can often be updated independently, which makes these operations well suited for parallel execution on multi-core CPUs.

5.2. Memory Architectures

In this section we will look at two common hardware configurations: shared memory and distributed memory. This thesis uses shared-memory parallelism, but both models are briefly introduced here for completeness.

5.2.1. Shared Memory

In a *shared-memory* architecture, all processor cores read and write from the same global memory [15]. This simplifies programming.

Shared-memory machines work well for tasks that can be broken into small parts, like processing rows in sparse matrix–vector multiplication. Single compute nodes in high performance computing (HPC) systems typically have this architecture.

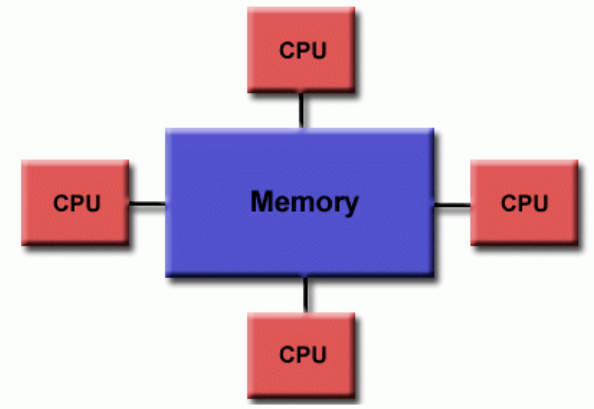


Figure 5.3: Taken from [15]. This illustration shows a shared-memory architecture, where four CPU cores access the same global memory.

5.2.2. Distributed Memory

In a distributed-memory architecture, each processor has its own local memory and cannot directly access the memory of other processors. Processors communicate by sending data over the network. Although programming is more complex, distributed-memory systems can scale to thousands of nodes and form the basis of most large supercomputers. Distributed-memory parallelism is not used in this thesis, but it is common in large-scale finite element software that runs with meshes having hundreds of millions of unknowns [16].

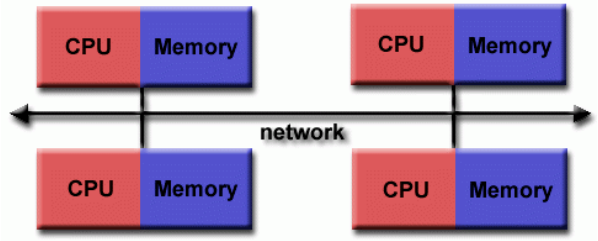


Figure 5.4: Taken from [15].This illustration shows a distributed-memory architecture, where four CPU cores all access there private memory.

5.3. Parallel Programming Interfaces

Parallel computations can be implemented using different programming interfaces, depending on the underlying hardware.

For shared-memory systems, the *OpenMP* API is widely used. OpenMP is often used for multi-threaded programming where multiple CPU cores share the same main memory [17, Ch. 11]. The code written is similar to a serial program and then *marks* certain regions (typically loops) as parallel by adding small OpenMP pragmas before them [17]. These pragmas are compiler hints that tell the compiler how to parallelize a given region of code. A typical example is a loop where each iteration updates a different entry of a vector; such a loop can be turned into a parallel loop by adding an OpenMP pragma [17, Ch. 11.3]. The compiler and OpenMP runtime then create the threads, divide the iterations between them, and synchronize the threads at the end of the loop.

For distributed-memory systems, the *Message Passing Interface* (MPI) is the standard programming model [18]. In MPI, each process has its own private memory and holds a portion of the data. Processes communicate by explicitly sending and receiving messages. This makes it possible to run simulations on large clusters with many nodes.

In this thesis, we focus on shared-memory parallelization. The Conjugate Gradient solver is parallelized using OpenMP-style threading through the PyKokkos library [19], a Python interface to the Kokkos performance-portable programming model [20]. PyKokkos allows the user to express parallel loops in Python, while the library translates these kernels to optimized C++/Kokkos code that can run efficiently on different backends (such as OpenMP on multi-core CPUs) without changing the solver code. In this way, we can take advantage of the multi-core CPU nodes of the TU Delft DelftBlue supercomputer [14] while keeping the implementation close to the mathematical description of the algorithm.

5.4. Limits to Scalability

Adding more threads does not automatically reduce the execution time. How effective parallelization can be depends on the structure of the algorithm and on the overhead introduced by the parallel execution model [15]. In this section we introduce the notions of speedup and parallel efficiency, discuss sources of parallel overhead, and summarize the limits imposed by Amdahl's Law.

5.4.1. Speedup and Parallel Efficiency

Let T_1 denote the execution time of a program on a single core, and T_N the execution time on N cores. The *speedup* is defined as

$$S(N) = \frac{T_1}{T_N}.$$

Ideally, we would like to obtain linear speedup, i.e. $S(N) \approx N$ [21].

The *parallel efficiency* measures how well the available cores are used:

$$E(N) = \frac{S(N)}{N} = \frac{T_1}{NT_N}.$$

An efficiency of $E(N) = 1$ would correspond to perfect scaling. In practice, parallel overhead and serial parts of the algorithm reduce both speedup and efficiency as N increases [15].

When analysing the performance of the Conjugate Gradient solver, we will use speedup and efficiency to interpret the timing results obtained on DelftBlue.

5.4.2. Parallel Overhead

The term *parallel overhead* refers to all work that would not be present in serial execution. Common sources of overhead are:

- synchronization between threads,
- thread creation and scheduling,
- load imbalance, where some threads finish their work earlier than others.

For example, in iterative solvers, computing inner products requires combining results from all threads, which introduces synchronization and adds noticeable overhead. These effects become larger as the number of threads increases [15].

5.4.3. Strong and Weak Scaling and Amdahl's Law

There are two commonly used metrics for scaling:

Strong scaling: measures the speedup obtained when solving a *fixed* problem on increasing numbers of processors. This is relevant when the goal is to reduce the time-to-solution [15].

Weak scaling: measures how well a computation scales as the workload grows together with the number of processors, so that the amount of work per processor remains (approximately) constant. This is relevant when additional resources are used to increase resolution, for example by refining the mesh in a finite element simulation [15].

In practice, both views are important for iterative solvers: strong scaling indicates the performance on a given mesh, while weak scaling shows how well the implementation handles growing problem sizes on larger parallel systems [15].

Amdahl's Law describes the theoretical limit on strong scaling. Let P denote the fraction of the program that can be parallelized, and $S = 1 - P$ the serial fraction. Then the maximum speedup on N threads is

$$Speedup(N) = \frac{1}{S + \frac{P}{N}}.$$

Even a small serial component places a strict upper bound on the achievable speedup. For example, a serial fraction of $S = 0.05$ (i.e. 5%) limits the speedup to at most $20\times$, regardless of how many cores are used.

In the context of the CG solver, the serial fraction includes, for example, sequential setup phases and parts of the algorithm that rely on global reductions. In Chapter 7 we will see that these serial components and parallel overheads explain why the measured speedups on DelftBlue deviate from ideal linear scaling.

6

Implementation

sIn this chapter we describe how the linear systems assembled in Chapter 4 are turned into data structures suitable for solving computationally. And we will look how the Conjugate Gradient (CG) method with different preconditioners is implemented in PyKokkos. First, we will look how the finite element matrices are assembled in NGSolve and how we vary the problem size. Next, a sparse storage format will be introduced. We explain how the CG method from Chapter 2 is used to solve these systems, and how the Jacobi and symmetric Gauss-Seidel preconditioners can be used to mathematically improve the efficiency of the solver. Then we explore how GMRES is implemented. Finally, we discuss how PyKokkos views and memory spaces are used to run these algorithms efficiently.

The source code implementation for the experiments presented in this thesis can be found on the repository [22].

6.1. Discretization with NGSolve and Mesh Generation

The finite element discretisations used throughout this thesis are generated with NGSolve [13]. NGSolve is used to define the geometry, construct a mesh, choose a finite element space, and assemble the corresponding matrices and vectors.

For the model problem in Section 4.1, the domain is the unit square $\Omega = (0,1)^2$. In NGSolve, this geometry is available as `unit_square`, and a mesh is obtained by

$$\texttt{ngmesh} = \texttt{unit\_square.GenerateMesh(maxh = h)},$$

and then passed to `Mesh(ngmesh)`. The argument `maxh` specifies the desired maximal global mesh size: smaller values of `maxh` produce finer meshes with more elements and degrees of freedom [13].

On a given mesh, we use a finite element space of order one (H^1),

$$V_h = H^1(\Omega) \cap \text{(piecewise linear functions)},$$

implemented in NGSolve via

$$V_h = \texttt{H1(mesh, order=1, dirichlet=...)}.$$

The `H1` space in NGSolve consists of continuous, element-wise polynomial functions and is well suited to the Laplace problem with Dirichlet boundary conditions [13]. The stiffness matrix A and load vector F are assembled from the standard Laplace bilinear form and right-hand side linear form, matching the expressions in Section 4.2.

Listing 6.1: Assembly of the Laplace stiffness matrix in NGSolve.

```
1 from ngsolve import *
2 from netgen.geom2d import unit_square
3
```

```
h = 0.01  # global mesh size parameter
mesh = Mesh(unit_square.GenerateMesh(maxh=maxh))
fes = H1(mesh, order=1, dirichlet="top|bottom|left|right")
u, v = fes.TnT()

a = BilinearForm(grad(u) * grad(v) * dx).Assemble()
f = LinearForm(v * dx).Assemble()

A_ng = a.mat      # stiffness matrix in NGSolve format
b_ng = f.vec.FV()      # load vector
```

Listing 6.1 shows a minimal version of the assembly code used in the implementation. The actual code in the repository also extracts the matrix in a SciPy-compatible sparse format and exports the right-hand side as a NumPy array [22].

6.1.1. Mesh Sizes Used in This Thesis

To study both convergence and performance, the problem size is varied by changing the global mesh parameter `maxh`. Throughout the experiments, the following values are used:

$$\texttt{maxh} \in \{0.1,\ 0.01,\ 0.005,\ 0.001\}.$$

For each choice of `maxh`, NGSolve generates a triangular mesh of the unit square with consistent element sizing throughout the domain. This sequence of meshes provides a controlled way to increase the problem size without changing the PDE, the finite element space, or the boundary conditions. It is used in the analysis of the convergence in Chapter 7. The exact numbers of elements, unknowns, and nonzero entries in the stiffness matrix A for each mesh are summarised in Table 6.1.

Table 6.1: Mesh sizes and matrix dimensions used in the implementation.

`maxh`	# elements	# unknowns n	$\mathrm{nnz}(A)$
0.1	230	136	600
0.01	23170	11386	78910
0.005	92504	46653	319379
0.001	2310100	1153051	8063365

6.2. Sparse Storage and Computational Structure

Although A has n^2 possible entries, the number of stored entries $\mathrm{nnz}(A)$ grows only linearly with n. For this reason, we store the matrix in the *Compressed Sparse Row (CSR)* format (see Figure 6.1). In CSR form, the matrix A is stored in three arrays:

- *row pointers*, indicating the starting position of each row within the values array,
- *column indices*, recording the column index of each stored value,
- *data*, containing all nonzero entries.

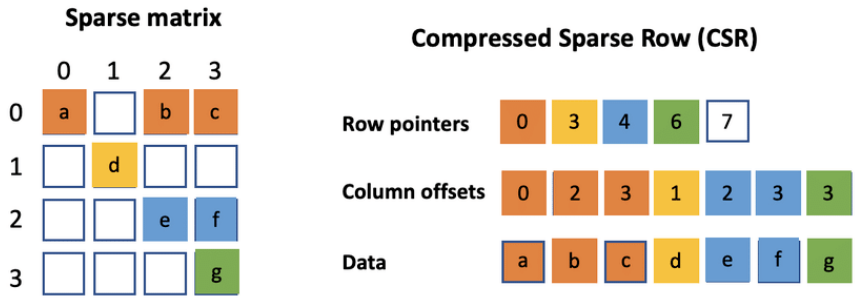


Figure 6.1: Taken from [23]. Compressed Sparse Row (CSR) data structure.

The CSR format is compact and enables efficient sparse matrix–vector multiplication (SpMV), which is the dominant computational kernel in CG, GMRES, and their preconditioned variants. Because the rows of the matrix are independent, SpMV can be parallelised by distributing rows across threads. By contrast, operations such as dot products gather partial results from all threads and require synchronisation before the algorithm can proceed.

In the implementation, the CSR arrays produced by SciPy are copied once into PyKokkos views `A_data_view`, `A_indices_view`, and `A_indptr_view`. All kernels that access the matrix read from these views only.

Listing 6.2: Creating CSR arrays for use in PyKokkos.

```
# 1. NGSolve -> SciPy CSR (simplified, see full code in repository)
a = BilinearForm(fes)
a += grad(u) * grad(v) * dx
a.Assemble()
A_scipy_full = sp.csr_matrix(a.mat.CSR())
A_csr = A_scipy_full[np.ix_(free, free)]

# 2. Allocate CSR data on device
self.A_data = pk.View([A_csr.data.size], pk.double)
self.A_indices = pk.View([A_csr.indices.size], pk.int32)
self.A_indptr = pk.View([A_csr.indptr.size], pk.int32)

# 3. Copy NumPy arrays into views
self.A_data.data[:] = A_csr.data.astype(np.float64)
self.A_indices.data[:] = A_csr.indices.astype(np.int32)
self.A_indptr.data[:] = A_csr.indptr.astype(np.int32)
```

6.3. Preconditioned Conjugate Gradient Solver and Kernels

With the linear system $Ax = b$ assembled and A stored in CSR format, we solve it with the Conjugate Gradient (CG) method. As discussed in Chapter 2, CG is designed for symmetric positive definite matrices, so it is well suited for the systems obtained from the finite element discretisation in Section 6.1. In this implementation, each CG iteration is expressed in terms of a small set of basic operations, and each of these operations is implemented as a PyKokkos kernel. The complete implementation of the solver can be found on the repository [22].

The convergence rate of CG depends strongly on the condition number $\kappa(A)$. To improve the condition number, and with it the convergence, we implement the Jacobi and symmetric Gauss-Seidel preconditioners introduced in Chapter 3. We first discuss the choice of tolerance and maximum iteration count, then describe the kernel-level decomposition and the preconditioner implementations.

6.3.1. Choice of Tolerance and Maximum Iteration Count

The stopping test for CG is based on the *relative residual*, as introduced in Section 2.2.1. Let $r_k = b - Ax_k$ denote the residual after k iterations. To obtain a measure that is independent of the magnitude of b, we monitor the *relative residual*

$$\frac{\|r_k\|_2}{\|b\|_2}.$$

In the implementation, the iteration is terminated as soon as

$$\|r_k\|_2 \leq \texttt{tol} \cdot \|b\|_2, \tag{6.1}$$

where `tol` is a user-specified tolerance. Multiplying by $\|b\|_2$ in (6.1) therefore imposes a bound on the *relative* residual. This makes the stopping criterion comparable across different mesh sizes [5, Ch. 2.3].

Throughout the experiments, we use

$$\texttt{tol} = 10^{-8}.$$

This value is small enough to ensure reliable results; reducing the tolerance further would increase the computational cost without significant improvements in the solution [5, Ch. 2.3].

Besides (6.1), we also limit the maximum number of iterations. In the code [22] we set

$$k_{\max} = 10000.$$

If condition (6.1) is not satisfied within $k_{\max}$ iterations, the solver stops and reports non-convergence. This limit prevents the algorithm from running indefinitely, which can occur if the matrix is ill-conditioned and the solver fails to converge. In practice, $k_{\max}$ is chosen larger than the number of iterations observed for convergence in the experiments.

6.3.2. Decomposition of CG into Basic Operations

In algorithmic form, one preconditioned CG iteration consists of four basic operations on vectors and matrices:

1. a sparse matrix–vector product (SpMV),
2. dot products,
3. vector updates of the form $y \leftarrow \alpha x + y$ (axpy),
4. application of the preconditioner $z = M^{-1}r$.

This can be written in pseudocode as

SpMV:	$q = Ap,$
dot:	$\alpha = \dfrac{(r,z)}{(p,q)},$
axpy:	$x \leftarrow x + \alpha p,$
axpy:	$r \leftarrow r - \alpha q,$
precond.:	$z = M^{-1}r,$
dot:	$\beta = \dfrac{(r,z)}{(r_{\text{old}}, z_{\text{old}})},$
update p:	$p \leftarrow z + \beta p.$

For the unpreconditioned solver, z is simply taken as r.

In the PyKokkos implementation, this translates to a loop of the form

Listing 6.3: Structure of one CG iteration in the implementation.

```
# Ap = A p
pk.parallel_for(
        self.n,
        spmv_kernel,
        y_view=self.Ap,
        x_view=self.p,
        A_data=self.A_data,
        A_indices=self.A_indices,
        A_indptr=self.A_indptr)

# alpha = rsold / (p, Ap)
pAp = self._dot_product(self.p, self.Ap)
if abs(pAp) < 1e-30:
        break
alpha = rsold / pAp

# x = x + alpha p
pk.parallel_for(self.n, axpy_kernel, alpha=alpha, x=self.p, y=self.x)

# r = r - alpha Ap
pk.parallel_for(self.n, axpy_kernel, alpha=-alpha, x=self.Ap, y=self.r)

# Apply preconditioner: z = M^{-1} r
self.apply_preconditioner()

# rsnew = (r, z)
```

```
rsnew = self._dot_product(self.r, self.z)
res_hist.append(np.sqrt(max(rsnew, 0.0)))

# Convergence test
...
#
beta = rsnew / rsold

# p = z + beta p
pk.parallel_for(self.n, scale_kernel, alpha=beta, x=self.p)
pk.parallel_for(self.n, axpy_kernel, alpha=1.0, x=self.z, y=self.p)
rsold = rsnew
```

As the above code chunk shows, each of the basic operations is mapped to a PyKokkos workunit:

- **SpMV:** `spmv_kernel` computes $y = Ax$ using the CSR arrays `A_data`, `A_indices` and `A_indptr` (Figure 6.1). For a given row index i, it loops over the nonzero entries in that row and accumulates the products with the corresponding entries of x:

$$y_i = \sum_{j \in \mathrm{nz}(i)} A_{ij} x_j.$$

- **Dot product:** `reduce_dot_kernel` computes the inner product of two vectors. It takes two one-dimensional views `x_view` and `y_view` and a scalar accumulator `acc` of type `pk.Acc[pk.double]`. For each index i it adds the product $x_i y_i$ to the accumulator, `acc += x_view[i] * y_view[i]`, and PyKokkos performs a reduction over all i to obtain the global dot product.
- **AXPY:** `axpy_kernel` performs the vector update $y \leftarrow \alpha x + y$ component-wise. It is used for both the x-update and the residual update.
- **Scale:** in the search-direction update $p \leftarrow z + \beta p$ we first scale p by β using `scale_kernel` and then add z with a call to `axpy_kernel`. This is the only place where `scale_kernel` is used in the CG iteration.
- **Preconditioner application:** the operation $z = M^{-1} r$ is implemented by separate kernels for each preconditioner: `apply_jacobi` for the Jacobi preconditioner and `apply_symmetric_gauss_seidel` for the symmetric Gauss-Seidel preconditioner (see Section 6.3.3).

Furthermore, two support kernels are used to handle vector initialization and copying: `zero_kernel` sets all entries of a vector to zero, and `copy_kernel` copies one vector into another. All kernels operate on one-dimensional `pk.View1D` objects and are parallelized over the vector index i, except for the symmetric Gauss-Seidel kernel, which is implemented as a sequential sweep over the rows.

In this way the CG solver is expressed entirely in terms of a small set of kernels. This matches the steps of the algorithm and separates the numerical method from the parallel implementation. The concrete python code for these kernels can be found in the repository [22].

6.3.3. Preconditioner Kernels

The Jacobi and symmetric Gauss-Seidel preconditioners are introduced and analysed in Chapter 3. Here we describe how they are implemented as kernels.

Jacobi Kernel

The Jacobi preconditioner uses only the diagonal of A (Chapter 3). The inverse diagonal entries $D_{ii}^{-1} = 1/A_{ii}$ are pre-computed and stored in a vector `d_inv_view`. The kernel `apply_jacobi` then applies the preconditioner by looping over all indices i and computing

$$z_i = D_{ii}^{-1} r_i,$$

implemented as `z_view[i] = d_inv_view[i] * r_view[i]`. The loop runs in parallel over i, and each entry can be updated independently.

Symmetric Gauss-Seidel Kernel

The symmetric Gauss-Seidel (SGS) preconditioner is implemented in the kernel `apply_symmetric_gauss_seidel`. Mathematically it consists of one forward Gauss-Seidel sweep followed by one backward sweep. In the forward sweep the entries are updated in the order $0, 1, \ldots, N-1$,

$$z_i = \frac{1}{A_{ii}} \left(r_i - \sum_{j<i} A_{ij} z_j \right),$$

so z_i uses the *current* values z_j for all $j < i$ that have already been updated in this sweep. The backward sweep runs in the opposite direction and updates the entries in the order $N-1, \ldots, 0$, using the already updated values with $j > i$. Because each new value depends on previously updated entries, the updates along a sweep cannot be carried out independently.

If the rows were updated in parallel, different threads would either have to use only the values from the previous iteration, or they would read and write overlapping entries at the same time. In the first case, the method reduces to a Jacobi-type iteration and is no longer Gauss-Seidel. In the second case, the result would depend on the order in which threads access the data and is not well-defined. For this reason, a standard Gauss-Seidel sweep is inherently sequential.

This dependency is reflected in the kernel design. The workunit `apply_symmetric_gauss_seidel` has an index parameter i, but only the case `i == 0` performs any work. Inside this branch, the kernel executes two explicit loops over the rows:

- **Forward sweep:** for `row = 0, ..., N-1` the kernel computes z_{row} using the entries z_j with $j <$ row, obtained from earlier steps in the same loop. The CSR arrays `A_data`, `A_indices` and `A_indptr` are used to access the nonzero entries in each row.
- **Backward sweep:** for `row = N-1, ..., 0` the kernel updates `z_view[row]` by subtracting the contribution from entries with column index greater than `row`, corresponding to the upper triangular part.

All work is carried out inside this single workunit instance, so the SGS preconditioner processes the rows one after another. Other CG operations (SpMV, dot products, axpy) are executed by parallel kernels, but SGS is kept sequential to preserve the standard Gauss-Seidel update and to make the dependence between rows explicit. The impact of this sequential preconditioner on parallel scaling is visible in Chapter 7.

6.3.4. Serial Fraction and Amdahl's Law

From the perspective of parallel performance, it is important to look at the serial fraction S of the total runtime, i.e. the fraction that cannot be parallelised. As discussed in 5.4.3 Amdahl's law states that the $speedup(N)$ achievable on N processors is bounded by

$$Speedup(N) \leq \frac{1}{S + \dfrac{1-S}{N}}.$$

Even if most of the work is parallel, any non-negligible serial fraction limits the speedup.

In the CG implementation the following components contribute to S:

- the symmetric Gauss-Seidel preconditioner, which must run as a sequential sweep (Section 6.3.3),
- global dot products, which require reductions over all threads and synchronisation between iterations,
- the residual norm evaluations for the stopping criterion.

Jacobi preconditioning does not add serial work beyond the dot products, so its serial fraction is relatively small. By contrast, the SGS preconditioner introduces an explicitly sequential kernel inside every CG iteration. This can be clearly seen in the strong-scaling results. As the number of threads increases, the speedup is well below the ideal linear scaling prediction for a parallel code that is dominated by the SpMV kernel.

6.4. GMRES Implementation, Orthogonality, and Givens Rotations

Now that we can solve symmetric positive definite systems with the Krylov solver CG, we also want a method for nonsymmetric matrices. For this purpose we implement the Generalized Minimal Residual method (GMRES) [6]. The GMRES solver reuses many of the kernels from CG. The main difference lies in the iteration structure. GMRES builds an orthonormal Krylov basis with the Arnoldi process and chooses the next iterate by solving a small least-squares problem.

The mathematical description of GMRES is given in Section 2.2.2. In this section we focus on how we can implement the Arnoldi process, the least-squares problem, and how we can use preconditioners obtain a more stable method.

6.4.1. Arnoldi Process and Krylov Basis

As seen in Section 2.2.2, GMRES constructs iterates of the form

$$x_k = x_0 + V_k y_k,$$

where $V_k = [v_1, \ldots, v_k]$ contains an orthonormal basis produced by Arnoldi, and the basis and the Hessenberg matrix $\bar{H}_k$ satisfy the Arnoldi relation

$$AV_k = V_{k+1}\bar{H}_k,$$

with $\bar{H}_k \in \mathbb{R}^{(k+1)\times k}$ upper Hessenberg [1, Ch. 6.5]. Equivalently, each new vector is expanded in the current basis as

$$Av_j = \sum_{i=1}^{j+1} h_{i,j} v_i,$$

and these coefficients form the entries of $\bar{H}_k$.

In the implementation we do not repeat this derivation, instead we map the Arnoldi step directly to the available kernels. Conceptually, the basis vectors v_j from Section 2.2.2 are stored as an array of PyKokkos views `q[j]`. One Arnoldi step (without considering restarts) then has the form

$$\text{given } v_1, \ldots, v_j \text{ and } \bar{H}_{j-1} = [h_1, \cdots, h_{j-1}], \text{ compute } v_{j+1} \text{ and } \bar{H}_j.$$

We use a right-preconditioning, so in each Arnoldi step we apply the preconditioner first and then the matrix:

$$z = M^{-1}v_j, \qquad w = Az.$$

The operation $z = M^{-1}v_j$ uses exactly the same Jacobi or symmetric Gauss-Seidel kernels as in CG (Section 6.3.3), and $w = Az$ is computed by the SpMV kernel. Orthogonalisation is then performed by a short serial loop that calls dot-product and axpy kernels. In simplified form, one Arnoldi step in the code looks as follows:

Listing 6.4: One Arnoldi step in the GMRES implementation (right-preconditioned).

```
# q[0], ..., q[m] : Krylov basis vectors, stored as PyKokkos View1D
# H : (m+1) x m Hessenberg matrix (NumPy array on host)
# j : current Arnoldi step (0-based index)

# Apply right preconditioner and matrix
apply_preconditioner(q[j], z_view)        # z = M^{-1} v_j
spmv_kernel(A_data_view, A_indices_view,
            A_indptr_view, z_view, w_view)  # w = A z

# Classical Gram-Schmidt orthogonalisation
for i in range(j+1):
    h_ij = dot_kernel(w_view, q[i])       # h_{i,j} = (w, v_i)
    H[i, j] = h_ij
    axpy_kernel(-h_ij, q[i], w_view)      # w -= h_{i,j} v_i

# Normalise and append new basis vector
H[j+1, j] = norm2_kernel(w_view)
if H[j+1, j] != 0.0:
    scale_kernel(1.0 / H[j+1, j], w_view)
    q[j+1] = copy_view(w_view)           # v_{j+1} = w / h_{j+1,j}
```

Listing 6.4 is slightly different from the actual code, but shows the key ingredients:

- the Arnoldi matrix-vector product $AM^{-1}v_j$ is implemented as a preconditioner call followed by the SpMV kernel,
- the coefficients $h_{i,j}$ are computed with the dot-product kernel,
- the orthogonalisation step $w \leftarrow w - h_{i,j}v_i$ uses the axpy kernel,
- the normalisation and storage of v_{j+1} use the norm and scale kernels and a simple copy.

The orthogonality property of the basis vectors and the Hessenberg relation $AV_k = V_{k+1}\bar{H}_k$ remain exactly as stated in Section 2.2.2. The only difference is that, in the implementation, the inner loop over $i = 1, \ldots, j$ appears as a short serial loop over columns of H, while the vector operations inside this loop are executed in parallel by PyKokkos. For more details on Arnoldi and the structure of $\bar{H}_k$, see [1, Ch. 6.5] and [3, Ch. 6.1].

6.4.2. Givens Rotations and Residual Minimisation

We have seen the GMRES iterate $x_m = x_0 + V_m y_m$ is obtained by solving the least-squares problem

$$y_m = \arg min_{y \in \mathbb{R}^m} \left\| \beta e_1 - \bar{H}_m y \right\|_2,$$

where e_1 is the first unit vector [1, Ch. 6.5]. This reduces the large $n \times n$ system $Ax = b$ to a problem of dimension $m \ll n$.

The least-square problem can be solved by computing a QR factorisation of $\bar{H}_m$,

$$\bar{H}_m = Q^\top R,$$

with Q orthogonal and R upper triangular. Using $\|\beta e_1 - \bar{H}_m y\|_2 = \|Q\beta e_1 - Q\bar{H}_m y\|_2$ and $Q\bar{H}_m = R$, the minimisation becomes

$$\min_y \|g - Ry\|_2, \qquad g := Q\beta e_1.$$

This minimum is then obtained by solving the upper-triangular system $Ry = g$ using backward substitution.

Givens rotations

GMRES builds Q as a product of *Givens rotations*. A Givens rotation acts on two rows i and $i+1$ and is defined by the 2×2 sub-matrix

$$G_j = \begin{pmatrix} \cos\theta_j & \sin\theta_j \\ -\sin\theta_j & \cos\theta_j \end{pmatrix},$$

Given two numbers a and b, we can choose $\cos\theta_j$ and $\sin\theta_j$ so that

$$G_j \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} r \\ 0 \end{pmatrix},$$

the important thing to notice here is that the second component zeros out. Setting

$$r = \sqrt{a^2 + b^2}, \qquad \cos\theta_j = \frac{a}{r}, \qquad \sin\theta_j = \frac{b}{r},$$

we obtain

$$\begin{pmatrix} \cos\theta_j & \sin\theta_j \\ -\sin\theta_j & \cos\theta_j \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} r \\ 0 \end{pmatrix}.$$

In the full matrix $\bar{H}_m$, the 2×2 rotation is integrated into an $(m+1) \times (m+1)$ identity matrix so that it only mixes rows i and $i+1$ and leaves all other rows unchanged. When this extended rotation is multiplied from the left with $\bar{H}_m$, it replaces the entries in rows i and $i+1$ by the rotated values and leaves the other rows unchanged. By choosing the parameters $(\cos\theta_j, \sin\theta_j)$ so that the subdiagonal entry becomes zero, each rotation eliminates exactly one entry below the diagonal. Repeating this process for all

subdiagonal entries turns $\bar{H}_m$ into an upper-triangular matrix R. The product of all applied rotations is the orthogonal matrix Q that satisfies $Q\bar{H}_m = R$.

In the implementation we do not store Q explicitly. Instead, for each GMRES step j we store the two scalars $\cos\theta_j$ and $\sin\theta_j$ that define the corresponding Givens rotation. These values are kept in small arrays `cs[j]` and `sn[j]`. Every time a new subdiagonal entry $\bar{H}_{j+1,j}$ is produced in column j of $\bar{H}_m$, a new pair $(\cos\theta_j, \sin\theta_j)$ is computed to eliminate it, and this rotation is then applied to both the matrix H and the vector g.

Incremental update of H and g

The theory from above is implemented in the following steps. At Arnoldi step j we have computed the column $H(:, j)$ of the Hessenberg matrix and the basis vector v_{j+1}. The update with Givens rotations proceeds as follows:

1. **Apply previous rotations.** For $i = 0, \ldots, j-1$, apply the stored rotation $(\cos\theta_i, \sin\theta_i)$ to the pair $(H_{i,j}, H_{i+1,j})$. This keeps the already processed part of H upper triangular.
2. **Compute new rotation.** Use the current diagonal/subdiagonal pair $(H_{j,j}, H_{j+1,j})$ as (a, b) and form a new Givens rotation $(\cos\theta_j, \sin\theta_j)$ such that $H_{j+1,j}$ becomes zero.
3. **Update H and g.** Apply the same rotation (c_j, s_j) to $(H_{j,j}, H_{j+1,j})$ and to (g_j, g_{j+1}). This corresponds to multiplying the j-th column of $\bar{H}_m$ and the right-hand side βe_1 by the same orthogonal matrix Q.

One thing that comes with this GMRES QR update is that the current residual norm is now given by

$$\|r_j\|_2 = |g_{j+1}|.$$

In the implementation, the stopping criterion is therefore based on the ratio $|g_{j+1}|/\beta$.

The arrays H, g, $\cos\theta_j$ and $\sin\theta_j$ all have a size proportional to m and are updated in serial outside the Pykokkos kernels. The main GMRES operations (Arnoldi, preconditioners application, sparse matrix-vector multiplication and vector updates) still act on vectors of size n and on the $n \times n$ matrix A. And since $n \gg m$, the extra cost from the Givens updates are relatively small. However, since these updates are sequential in each GMRES iteration, they do contribute to the serial fraction of the code that limits the parallel speedup, this is further discussed in Chapter 7.

6.5. PyKokkos Views, Memory Spaces, and Data Movement

An important part of the implementation is how data is stored and accessed inside PyKokkos kernels. PyKokkos is based on Kokkos, a C++ framework designed for portable, high-performance computing. Kokkos manages parallel execution and data handling across both CPUs and GPUs [19]. In Kokkos, data is stored in `Kokkos::View` objects, which represent multidimensional arrays. Such a Kokkos view carries information about where the data lives and how it is laid out in memory. PyKokkos brings this concept to Python through the `pk.View` type, for example `pk.View1D[pk.double]` for a one-dimensional array of doubles. These Python objects are bindings to the underlying Kokkos implementation [24].

Kokkos separates **execution spaces** from **memory spaces**. Execution spaces describe where code runs (in this thesis that is `OpenMP`). Memory spaces describe where the data is stored (for example `HostSpace` or `CudaSpace`). For most code, the default memory space is chosen to match the default execution space. In practice, this means that when a CPU backend is selected, `pk.View` objects are allocated in host memory, and when a GPU backend is selected, the same views are allocated in device memory, without changing the kernel code [24].

In the implementation, we try to minimize data movement by loading matrix and vector data into `pk.View` objects once, and reusing these views across iterations. Operations such as sparse matrix–vector products, preconditioner applications, and vector updates are implemented as PyKokkos kernels that run entirely on these views in the chosen execution space. The data is only copied back to standard Python or NumPy arrays when required for analysis or output.

In early versions of the code, data was transferred between NumPy arrays and `pk.View` objects more often than necessary, and the first SpMV kernel used untyped Python scalars instead of the `pk.double` and `pk.int32` types used in the final implementation. In practice this led to a much higher runtime. It seems like this has to do with workunits that use typed views and scalar types that are translated to compiled Kokkos kernels, and additional data movement reduces the benefit of this translation [19]. The final implementation [22] therefore keeps all main loops inside PyKokkos workunits and uses `pk.double` and `pk.int32` consistently inside those kernels.

7

Results

In this chapter we analyse the performance of the Krylov solvers implemented in PyKokkos. The focus is on two methods:

- the Conjugate Gradient (CG) method for symmetric positive definite systems, applied to a Laplace problem, and
- the Generalized Minimal Residual (GMRES) method for nonsymmetric systems, applied to a Stokes-like problem, and finally in a more realistic setting, a flow around a NACA airfoil.

For both solvers we investigate (i) convergence and preconditioning and (ii) parallel scalability. For CG we additionally analyse the kernel performance of the underlying PyKokkos kernels.

The first part of the chapter is devoted to CG for the Laplace problem. We analyse the effect of Jacobi and symmetric Gauss-Seidel (SGS) preconditioning, the strong and weak scaling behaviour, and the kernel-level performance. In the second part we look at GMRES. We use the same PyKokkos kernels, but change the model to a Stokes-like problem. We analyse the convergence for different restart parameters and preconditioners, and we look at how the GMRES results compare to the CG experiments. Finally, we apply GMRES to a more realistic Stokes flow around a NACA-like geometry to illustrate how these methods can be used in practice.

7.1. Conjugate Gradient for a Scalar Laplace Problem

7.1.1. Model problem and solver parameters

For the CG experiments we consider the Laplace equation on the unit square, discretized with linear finite elements as described in Chapter 4. The problem size is varied by changing the global mesh parameter `maxh` as discussed in 6. We use $\texttt{maxh} \in [0.1, 0.01, 0.005, 0.001]$. For the largest mesh ($\texttt{maxh} = 0.001$) the system contains $1{,}153{,}051$ unknowns and $8{,}063{,}365$ non-zero entries.

All solvers use the relative residual stopping criterion

$$\frac{\|r_k\|_2}{\|b\|_2} \leq \texttt{tol}, \tag{7.1}$$

with tolerance $\texttt{tol} = 10^{-8}$ and a maximum of $k_{\max} = 10{,}000$ iterations. Unless stated otherwise, all reported runs satisfy the stopping criterion before reaching $k_{\max}$.

The experiments are performed on a shared-memory machine with OpenMP parallelism. For each configuration we run with

$$N_{\text{threads}} \in \{1, 2, 4, 8, 16\}.$$

The reported time-to-solution is the total runtime of the full CG solve, this includes the cost of applying the preconditioner for the preconditioned cases. The speedups and parallel efficiency in Section 7.1.5 are measured relative to the single-thread runtime.

7.1.2. Convergence and Effect of Preconditioning

Table 7.1 shows the iteration counts and total runtimes for the three CG variants on all meshes using 4 threads.

Preconditioner	Iterations	Time (s), 4 threads
maxh = 0.1, DOFs = 96		
None	31	0.047
Jacobi	31	0.039
Gauss-Seidel	15	0.019
maxh = 0.01, DOFs = 11 386		
None	273	0.315
Jacobi	262	0.288
Gauss-Seidel	115	0.200
maxh = 0.005, DOFs = 45 853		
None	494	0.683
Jacobi	476	0.624
Gauss-Seidel	221	0.660
maxh = 0.001, DOFs = 1 153 051		
None	1976	16.891
Jacobi	1808	16.227
Gauss-Seidel	973	56.255

Table 7.1: Performance of the CG solver with different preconditioners using 4 threads. All runs reached the stopping criterion $\texttt{tol} = 10^{-8}$ before hitting the iteration limit $k_{\max} = 10{,}000$.

The following things stand out:

- For all three methods the iteration count increases strongly as the mesh is refined.
- The SGS preconditioner consistently reduces the iteration count by roughly a factor of two compared to unpreconditioned CG.
- The Jacobi preconditioner only slightly decreases the amount of iterations. On the finest mesh we see a reduction from 1976 to 1808 iterations. On the coarser meshes the iteration counts of Jacobi and the unpreconditioned method are almost identical.
- For the coarser meshes, SGS not only reduces the iteration count, but also has the fastest runtime. On the finer meshes however ($maxh \leq 0.005$), SGS still converges in fewer iterations, but the total runtime is much larger than for the other methods.

Figure 7.1 shows the relative residual $\|r_k\|_2/\|b\|_2$ versus iteration count for the three methods on the finest mesh with 4 threads. The SGS curve decreases much faster than the other two, whereas the Jacobi curve is very close to the unpreconditioned one. This shows that SGS is much more effective at improving the spectral properties of the system.

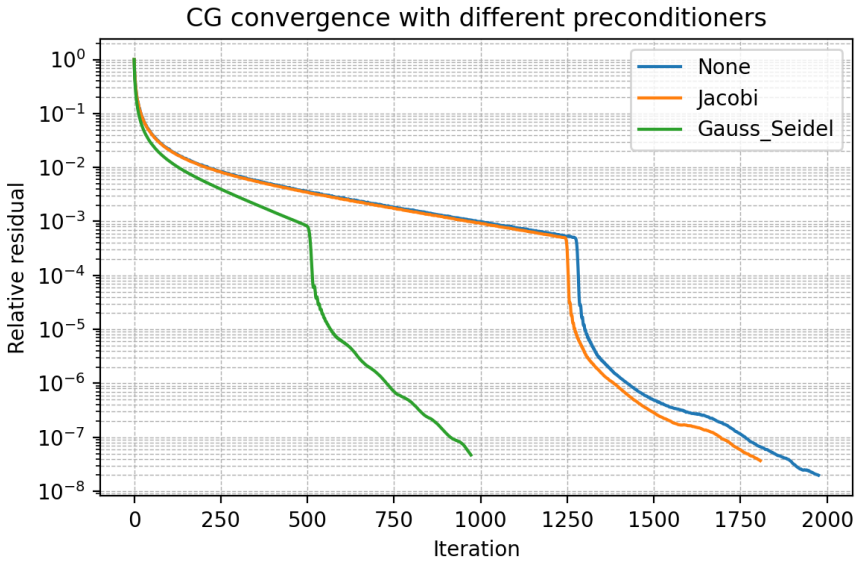


Figure 7.1: Convergence of the CG solver with different preconditioners for `maxh` = 0.001 and $1{,}153{,}051$ DOFs, using 4 threads.

Relation to CG theory. For a symmetric positive definite system $Ax = b$, the CG error satisfies the bound

$$\frac{\|x - x_k\|_A}{\|x - x_0\|_A} \leq 2\left(\frac{\sqrt{\kappa(A)} - 1}{\sqrt{\kappa(A)} + 1}\right)^k, \qquad \kappa(A) = \frac{\lambda_{\max}(A)}{\lambda_{\min}(A)}, \tag{7.2}$$

Here $\kappa(A)$ is the condition number of A, $x - x_k$ is the error after k iterations and $\|x\|_A^2 = x^T Ax$. For a preconditioned system $M^{-1}Ax = M^{-1}b$ the same estimate holds with $\kappa(M^{-1}A)$ [1, Thm. 6.29].

Preconditioning aims to reduce $\kappa(M^{-1}A)$ and thereby the iteration count. In the ideal situation, the CG iteration count would become independent of the mesh size. In practice, Jacobi preconditioning is known to only slightly reduce the condition number. This is why in our results, the iteration count with Jacobi preconditioning remains similar to CG without preconditioning. The SGS preconditioner, on the other hand, is much more effective at clustering the eigenvalues and therefore at reducing the condition number [1]. Which is exactly the reason why SGS significantly reduce the iteration count in Table 7.1. The convergence curves in Figure 7.1 show this as well. The SGS residual decays much faster, while the Jacobi curve stays close to the unpreconditioned one.

The number of CG iterations required to reach a fixed tolerance grows proportionally to $\sqrt{\kappa(A)}$ [1, Thm. 6.29]. For the Laplace model problem considered here, the condition number of matrix A increases when the mesh is refined, so we expect the CG iteration count to grow as we decrease the maximum element size `maxh`. This is exactly what we see in Table 7.1: the number of iterations for unpreconditioned CG increases from 31 on the coarsest mesh to 1976 on the finest mesh.

7.1.3. Sequential SGS preconditioner

As discussed in Section 6.3.3, SGS is implemented as one forward sweep followed by one backward sweep over all rows, corresponding to a symmetric Gauss-Seidel step. In each sweep, the update for row i depends on previously updated entries from earlier rows (in the forward sweep) or from later rows (in the backward sweep). This prevents parallelisation over the rows. In the PyKokkos implementation the kernel `apply_symmetric_gauss_seidel` executes both sweeps inside a single workunit instance. Even when more threads are available, only one logical thread performs the SGS sweeps, while the other CG operations (SpMV, dot, axpy) run in parallel.

On the coarser meshes, the overhead of launching kernels dominates the runtime. This is because there is relatively little work to be done, using more threads brings only a small benefit. So here reducing the number of iterations is the main advantage. This makes SGS the fastest method for coarse meshes. On the finest mesh, the downside of SGS being more sequential dominates. Also the cost of applying

SGS in each iteration becomes large, this outweighs the benefit of fewer iterations. Which results in a total runtime that is much higher than for the other methods, even on a single thread as we will see in the next section.

7.1.4. Small Differences in Iteration Counts Across Threads

The iteration counts in Table 7.1 are reported for 4 threads. When repeating the same experiment with different thread counts, the number of iterations can differ by a few steps, even though the algorithm and tolerance are the same. This behaviour is expected for Krylov methods in parallel arithmetic. Dot products and norms are computed via parallel reductions, and the order in which floating-point numbers are summed depends on the number of threads and the scheduling [25]. Different summation orders lead to slightly different round-off errors, which can be just enough so that the stopping criterion is met a few iterations earlier or later. These differences are small and the effect is negligible for the computed solutions.

7.1.5. Parallel Scalability and Time-to-Solution

In Chapter 5 we made a distinction between *strong* and *weak* scaling. Strong scaling refers to changing the number of threads for a fixed problem size, while weak scaling means increasing the problem size together with the number of threads so that the work per thread stays approximately constant. In this section we examine both aspects for the CG solver.

Strong scaling

For strong scaling we fix the finest mesh (`maxh` = 0.001, $1{,}153{,}051$ DOFs) and vary the number of OpenMP threads. The time-to-solution for all three preconditioners is shown in Table 7.2.

Table 7.2: CG solver time-to-solution (in seconds) for `maxh` = 0.001 ($1{,}153{,}051$ DOFs) as a function of the number of threads.

Method	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads
None	54.34	31.19	16.89	11.90	9.34
Jacobi	48.38	27.59	16.23	9.78	9.69
Gauss-Seidel	79.36	67.91	56.25	50.90	50.60

To relate these results to the theory of Chapter 5 we define, for each method, the strong-scaling speedup

$$\text{Speedup}(N) = \frac{T_1}{T_N}, \qquad E_N = \frac{\text{Speedup}(N)}{N},$$

where T_1 and T_N denote the runtimes of that method with 1 and N threads, respectively, and E_N is the parallel efficiency [21]. Table 7.3 summarises Speedup(N) and E_N for the finest mesh. Here we see the parallel efficiency drop as we increase the number of threads. For SGS the speedups are small for all thread counts, this is because of the dominant serial portion of each iteration.

Table 7.3: Strong-scaling speedup Speedup(N) and parallel efficiency E_N on the finest mesh (`maxh` = 0.001).

Method	Speedup(2) (E_2)	Speedup(4) (E_4)	Speedup(8) (E_8)	Speedup(16) (E_{16})
None	1.74 (0.87)	3.22 (0.80)	4.57 (0.57)	5.82 (0.36)
Jacobi	1.75 (0.88)	2.98 (0.75)	4.95 (0.62)	4.99 (0.31)
SGS	1.17 (0.58)	1.41 (0.35)	1.56 (0.19)	1.57 (0.10)

For the unpreconditioned solver the runtime decreases from 54.34 seconds on 1 thread to 9.34 seconds on 16 threads, corresponding to a speedup of about $5.82\times$. Hence the parallel efficiency,

$$E_{16} = \frac{5.82}{16} \approx 0.36,$$

is far from ideal (an efficiency of 1 would correspond to perfect linear scaling), but the solver still clearly benefits from additional threads. The Jacobi-preconditioned solver shows similar behaviour: its runtime

decreases from 48.38 seconds to 9.69 seconds, with almost identical speedups. For a small number of threads, the total runtime is slightly lower because the iteration count is somewhat reduced and the Jacobi kernel itself is cheap and fully parallel. When running with 16 threads the efficiency of Jacobi goes down and the difference in runtime disappears. Hence, for this problem the Jacobi preconditioner is not strong enough to give a substantial speedup, but it also does not introduce much overhead.

The SGS-preconditioned solver behaves quite differently. Although it has the smallest iteration count on this mesh, its runtime on a single thread (79.36 seconds) is already larger than for the other methods. As the number of threads increases, the SGS runtime only decreases slightly and then stagnates around 50 seconds. This poor strong scaling is a consequence of the sequential SGS sweeps (see Section 7.1.3). According to Amdahl's law, the presence of such a large serial fraction S in each iteration limits the achievable speedup

$$\text{Speedup}(N) = \frac{1}{S + \frac{P}{N}},$$

even if the parallel part P (SpMV and vector kernels) scale well. Unpreconditioned CG and Jacobi preconditioning have a relatively larger parallel portion and therefore benefit more from additional threads.

It is therefore not surprising that, on the finest mesh and for many threads, unpreconditioned CG becomes the fastest method in total runtime, despite having the largest iteration count: it avoids the extra serial work inside each iteration.

Weak scaling

For weak scaling we aim to keep the workload per thread approximately constant while increasing both the problem size and the number of threads. Table 7.4 shows such a sequence, when doubling the number of threads, the total number of DOFs is roughly doubled as well, so that the number of DOFs per thread stays close to 4.6×10^3.

Table 7.4: Weak scaling results for CG: approximately constant DOFs per thread while increasing the total problem size and the number of threads.

Threads	`maxh`	DOFs	Prec.	Time (s)	Iterations
1	0.005	45 853	None	0.80	494
1	0.005	45 853	Jacobi	0.80	476
1	0.005	45 853	Gauss-Seidel	0.57	221
2	0.00354	92 000	None	1.11	668
2	0.00354	92 000	Jacobi	1.10	644
2	0.00354	92 000	Gauss-Seidel	1.05	300
4	0.0025	184 004	None	1.67	951
4	0.0025	184 004	Jacobi	1.59	880
4	0.0025	184 004	Gauss-Seidel	2.18	384

In an ideal weak-scaling scenario the runtime would remain roughly constant as the number of threads and the problem size increase, because each thread handles the same amount of work and the iteration count stays fixed. In our experiments the runtimes grow noticeably: for the unpreconditioned solver the time increases from 0.80 seconds (1 thread) to 1.11 seconds (2 threads) and 1.67 seconds (4 threads). The Jacobi-preconditioned solver shows almost identical results.

This growth has two main causes. First, even though the number of DOFs per thread is kept approximately constant, the total number of iterations increases with the global problem size, from 494 to 668 and 951 iterations for the unpreconditioned solver. Second, additional threads introduce parallel overheads such as synchronisation and contention for memory bandwidth, which further increase the runtime.

The SGS-preconditioned solver shows an even stronger increase in runtime: from 0.57 seconds on 1 thread to 1.05 seconds on 2 threads and 2.18 seconds on 4 threads. Here the combination of a growing iteration count and the largely sequential preconditioner reduce the weak-scaling performance even

further. As the mesh is refined, the SGS sweeps become more expensive, and the serial work per iteration dominates the runtime.

Overall, the weak-scaling results show that the current CG implementation does not achieve ideal weak scaling. The total time increases with the number of threads and the problem size, primarily because the iteration count grows with mesh refinement, and for SGS also because a substantial serial component and additional overhead is introduced. The unpreconditioned and Jacobi-preconditioned solvers degrade more gracefully than SGS, but none of the variants maintain a constant time-to-solution under weak-scaling refinement.

7.1.6. Kernel-Level Performance

To understand where the speedups come from, we measure the average execution time of the main kernels on the finest mesh. Tables 7.5–7.7 report the times for the SpMV, dot, axpy and scale kernels for each method.

Table 7.5: Average per-iteration kernel time (in ms) for the unpreconditioned CG solver on the finest mesh (`maxh` = 0.001, $1{,}153{,}051$ DOFs).

Threads	SpMV	Dot	AXPY	Scale
1	12.52	4.75	6.48	1.67
2	7.02	2.80	3.80	0.96
4	3.67	1.58	2.07	0.54
8	2.55	1.11	1.47	0.38
16	1.88	0.92	1.17	0.32

Table 7.6: Average per-iteration kernel time (in ms) for the Jacobi-preconditioned CG solver on the finest mesh.

Threads	SpMV	Dot	AXPY	Scale
1	11.90	4.56	6.07	1.57
2	6.33	2.78	3.62	0.95
4	3.66	1.67	2.10	0.56
8	2.15	0.99	1.28	0.36
16	1.98	1.02	1.30	0.38

Table 7.7: Average per-iteration kernel time (in ms) for the Gauss-Seidel-preconditioned CG solver on the finest mesh.

Threads	SpMV	Dot	AXPY	Scale
1	19.66	7.04	9.58	2.74
2	13.31	4.87	6.34	1.87
4	7.22	3.21	3.70	1.19
8	4.12	2.21	2.17	0.76
16	3.14	1.93	1.65	0.57

From these tables we can draw several conclusions:

- **SpMV dominates the cost.** For all methods and thread counts, the SpMV kernel is the most expensive operation per iteration. It also shows the largest time reduction when the number of threads increases, so most of the speedup in the total runtime comes from parallelising SpMV.
- **AXPY and vector kernels also scale well.** The axpy kernel benefits from parallelism in a similar way to SpMV, but its absolute cost is smaller. The dot and scale kernels are even cheaper, so their contribution to the total runtime remains limited, even though they also parallelise well.
- **SGS increases per-iteration cost.** For the SGS-preconditioned solver, the SpMV, dot and axpy kernels are noticeably more expensive per iteration than in the other two methods. This is mainly because the sequential kernel of SGS dominates the total per-iteration cost and does not benefit from additional threads.

Overall, most of the parallel speedup comes from the SpMV and axpy kernels, which are both well suited to OpenMP parallelism. The preconditioners behave quite differently, Jacobi adds only a small amount of overhead and gives only a modest improvement in convergence, while SGS reduces the iteration count much more strongly but introduces a sequential part that limits scalability for larger problems and higher thread counts.

7.2. GMRES Results for a Stokes-like Vector Problem

7.2.1. Model problem and solver parameters

For the GMRES experiments we switch from the scalar Laplace problem to the Stokes-like channel flow. The PDE and boundary conditions are as in Section 4.2.1. The velocity is discretised in a vector-valued H^1 finite element space of order k (with $k = 1$ and $k = 3$ in the experiments), with no-slip conditions on the walls, a parabolic inflow profile at the inlet, and a natural outflow condition at the outlet.

As before, the problem size is varied by changing the global mesh parameter `maxh`. We use $\texttt{maxh} \in [0.05, 0.01, 0.005, 0.0025]$. For the largest mesh ($\texttt{maxh} = 0.0025$) the system contains $301{,}086$ unknowns. Apart from switching from CG to GMRES, the solver settings are kept as close as possible to the Laplace experiments. We use restarted GMRES with the relative residual stopping criterion

$$\frac{\|r_k\|_2}{\|b\|_2} \le \texttt{tol}, \qquad \texttt{tol} = 10^{-8},$$

and a maximum of $k_{\max} = 10{,}000$ iterations. The experiments are run with varying restart parameter $m \in \{20, 50, 75\}$ and the same three preconditioners as before (none, Jacobi and SGS). All runs are performed on the same shared-memory machine as in the CG experiments, with

$$N_{\text{threads}} \in \{1, 2, 4, 8, 16\},$$

so that the parallel behaviour can be compared directly.

7.2.2. Convergence, preconditioning and restart parameter (with $k = 1$)

We first analyse the convergence behaviour for the lowest-order vector discretisation. The iteration counts and total runtimes are summarised in Table 7.8.

	$m = 20$		$m = 50$		$m = 75$	
Preconditioner	**Iters**	**Time (s)**	**Iters**	**Time (s)**	**Iters**	**Time (s)**
maxh = 0.05, DOFs = 672						
None	58	0.234	58	0.447	54	0.487
Jacobi	60	0.249	75	0.552	62	0.639
Gauss-Seidel	34	0.140	29	0.163	29	0.166
maxh = 0.01, DOFs = 18 470						
None	400	1.913	280	2.623	263	3.408
Jacobi	400	1.938	300	2.960	300	4.187
Gauss-Seidel	120	0.680	149	1.560	146	2.100
maxh = 0.005, DOFs = 74 590						
None	1191	8.570	648	9.130	557	10.863
Jacobi	1180	8.698	650	9.347	600	13.452
Gauss-Seidel	280	3.017	250	4.484	225	5.420
maxh = 0.0025, DOFs = 301 086						
None	4158	65.099	1865	53.958	1416	58.056
Jacobi	4000	65.657	1850	55.759	1425	59.206
Gauss-Seidel	740	20.647	500	21.052	450	24.308

Table 7.8: GMRES convergence for the Stokes-like problem with mesh sizes ($\texttt{maxh} \in \{0.1, 0.05, 0.01, 0.005\}$) for different preconditioners and restart parameters, using 1 thread, for the lowest-order vector discretisation ($k = 1$).

Influence of the restart parameter m. The restart parameter m controls the dimension of the Krylov subspace between restarts. In each GMRES(m) cycle the method builds the m-dimensional Krylov space

$$\mathcal{K}_m(A, r_0) = \mathrm{span}\{r_0, Ar_0, \ldots, A^{m-1}r_0\},$$

larger values of m allow GMRES to approximate the solution using higher-degree polynomials of A, which can reduce the total number of iterations. At the same time, the per-cycle cost grows roughly with m because orthogonalisation and the least-squares solve operate on an m-dimensional basis [1].

In the present experiments larger values of m always reduce the iteration count, but they do not lead to shorter runtimes. On the coarsest mesh ($\texttt{maxh} = 0.05$) the unpreconditioned solver reduces the iteration count from 58 to 54 when m is increased from 20 to 75, but the runtime increases from 0.23 s to 0.49 s. We see this as well with Jacobi and SGS, for this small problem the extra orthogonalisation work for $m = 50$ and $m = 75$ only adds overhead. The problem is too small to draw more conclusions. We see the same pattern on the intermediate mesh ($\texttt{maxh} = 0.01$), Jacobi and SGS both gain some noticeable reduction in iteration count when m is increased, but the gains are too small to compensate for the quadratic growth in orthogonalisation cost.

As the mesh is refined, the dependence on m becomes stronger. For example, for the unpreconditioned solver on the mesh with $\texttt{maxh} = 0.005$ the iteration count drops from 1191 ($m = 20$) to 648 ($m = 50$) and 557 ($m = 75$). However, the runtimes still increase from 8.57 s to 9.13 s and 10.86 s. The same pattern appears for Jacobi and SGS: increasing m gives a noticeable reduction in iterations, but the cost per cycle grows roughly like $\mathcal{O}(m^2)$ (see 2.2.2) due to orthogonalisation and the small least-squares problem, so the total time does not improve [1].

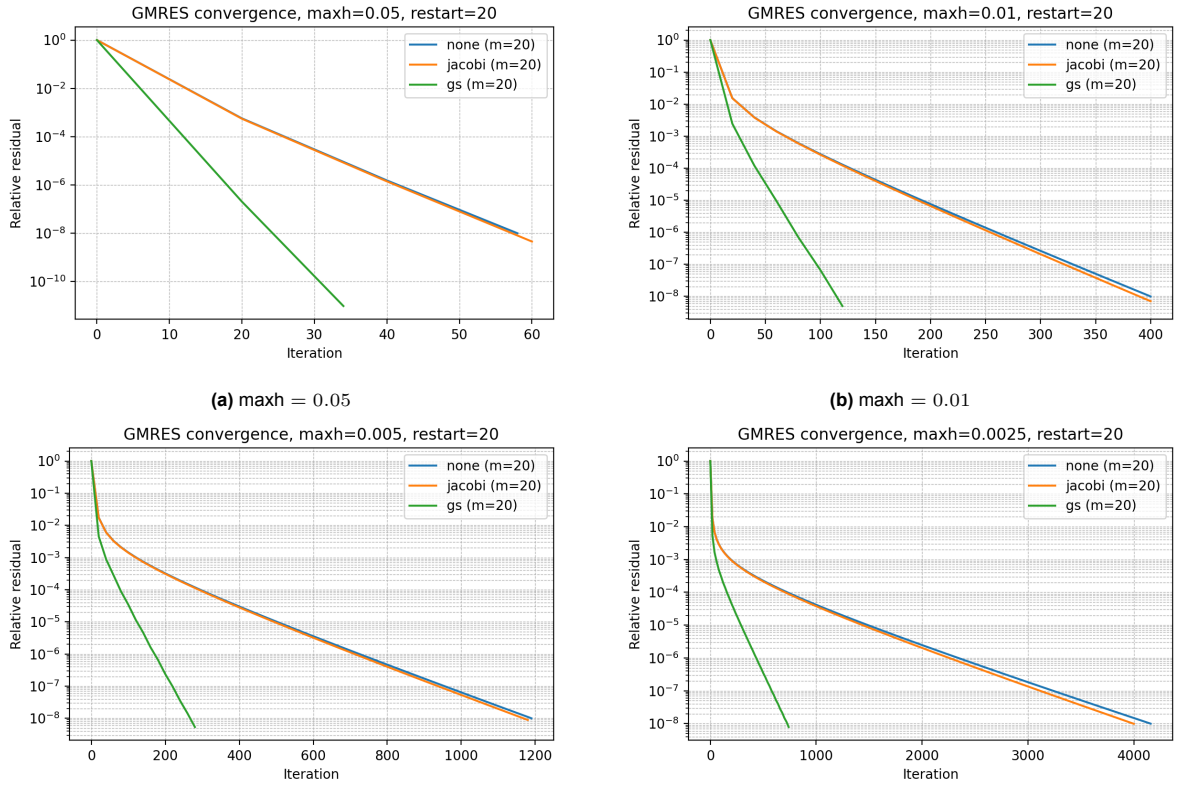
On the finest mesh ($\texttt{maxh} = 0.0025$, $301{,}086$ DOFs) the reduction in iteration count can outweigh the additional orthogonalisation work for the first time. Without preconditioning the iteration count drops from 4158 ($m = 20$) to 1865 ($m = 50$) and 1416 ($m = 75$). In this case $m = 50$ gives the shortest runtime (65.10 s $\rightarrow$ 53.96 s), while $m = 75$ yields even fewer iterations but becomes slower again (58.06 s) because the cost per cycle grows. Jacobi shows a very similar pattern. The iteration count decreases from 4000 to 1850 and 1425 when m is increased from 20 to 50 and 75, respectively, with $m = 50$ again giving the best time (65.66 s $\rightarrow$ 55.76 s, then up to 59.21 s for $m = 75$). In contrast, Gauss-Seidel is already a much stronger preconditioner. The iterations drop from 740 to 500 and 450, but the runtimes increase from 20.65 s to 21.05 s and 24.31 s. Here $m = 20$ remains the most efficient choice. These results indicate that for weak or no preconditioning a moderate increase of m (e.g. to $m = 50$) can be beneficial on large problems, while for a strong smoother such as Gauss-Seidel small restart values are sufficient and larger m primarily increases the Krylov overhead.

Effect of preconditioning (order $k = 1$). We now compare the three preconditioners, focusing on $m = 20$, which gives the best runtimes in all cases. For this lowest-order discretisation the symmetric Gauss-Seidel preconditioner is clearly the most effective. On the finest mesh, SGS reduces the iteration count from 1191 (no preconditioner) to 280 and reduces the runtime from 8.57 s to 3.02 s. Similar gains are observed on the other meshes.

In contrast, the Jacobi preconditioner is rather weak for $k = 1$. On the coarse meshes ($\texttt{maxh} = 0.05$ and 0.01) it produces essentially the same iteration counts and runtime compared to the unpreconditioned solver (400 iterations for $m = 20$ in both cases with $\texttt{maxh} = 0.01$). Even on the finest mesh the differences between Jacobi and no preconditioner are small: for $m = 20$, Jacobi converges in 1180 iterations and 8.70 s, compared to 1191 iterations and 8.57 s without preconditioning.

This behaviour is similar to what we observed for CG in Section 7.1.2. The convergence plots in Figure 7.2 show that the curves for the unpreconditioned solver and Jacobi are almost indistinguishable. For this lowest-order discretisation ($k = 1$), the impact of Jacobi on GMRES is small, and it is never clearly better than the unpreconditioned solver in terms of runtime.

Overall, the $k = 1$ results show a clear ordering: Gauss-Seidel consistently performs best, while Jacobi and the unpreconditioned solver behave very similarly. In the next subsection we increase the polynomial degree to $k = 3$ and observe how the behaviour of Jacobi changes once the system matrix becomes more ill-conditioned.



(a) maxh $= 0.05$ **(b)** maxh $= 0.01$

(c) maxh $= 0.005$ **(d)** maxh $= 0.0025$

Figure 7.2: GMRES convergence for different mesh sizes with $k = 1$ and $N_{\text{threads}} = 1$.

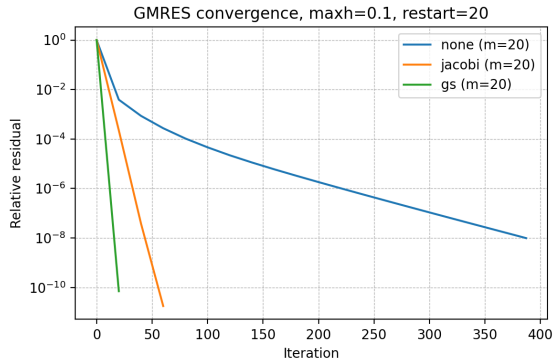
Numerical results for $k = 3$. Table 7.9 summarises the GMRES results for $k = 3$ and one thread. On the coarsest meshes Jacobi still performs well. For example, for $\texttt{maxh} = 0.1$ and $m = 20$ the iteration count drops from 387 (no preconditioner) to 60 (Jacobi), and the runtime decreases from 2.36 s to 0.39 s. SGS remains the most effective method: it converges in only 20 iterations and solves the system in about 0.17 s, essentially independent of m. For the second mesh ($\texttt{maxh} = 0.05$, 6528 DOFs) we see similar behaviour, Jacobi reduces the iteration count from 438 to around 80-100, while SGS brings it down to about 34-36 iterations.

The convergence curves in Figure 7.3 show the same trend. For the coarse meshes the SGS residual decays much faster than the other methods, Jacobi gives a clear improvement over the unpreconditioned solver, and the unpreconditioned curve shows the slowest decay.

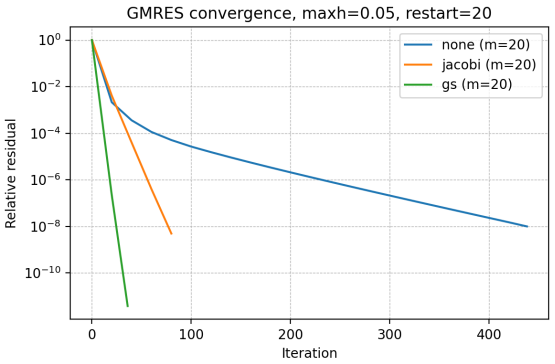
As the mesh is refined and the problem size grows, Jacobi becomes less effective. On the finest mesh ($\texttt{maxh} = 0.005$) the Jacobi-preconditioned solver even needs *more* iterations and more time than the unpreconditioned solver. For $m = 20$ the iteration count increases from 1 202 (no preconditioner) to 1 700 (Jacobi), and the runtime increases from 72.3 s to 105.6 s. SGS still reduces the iteration count to 280 and the runtime to 38.5 s.

At first this might seem surprising: we start from an SPD system and add a preconditioner, yet GMRES becomes slower. The explanation is not simply “preconditioning versus no preconditioning”, but how well the chosen preconditioner matches the structure of the matrix. A detailed analysis of this can be found in Appendix A, higher-order elements increase the number of neighbours each degree of freedom is coupled to.

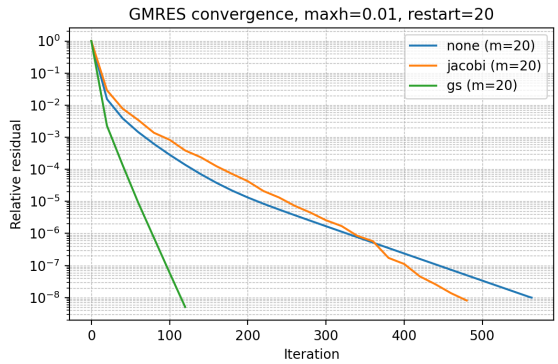
In our case the Jacobi preconditioner M_J keeps only the diagonal of $A^{(3)}$ and discards all couplings between neighbouring degrees of freedom. For the high-order discretisation this diagonal is a very poor approximation of the full operator, most of the important interaction terms live off the diagonal and



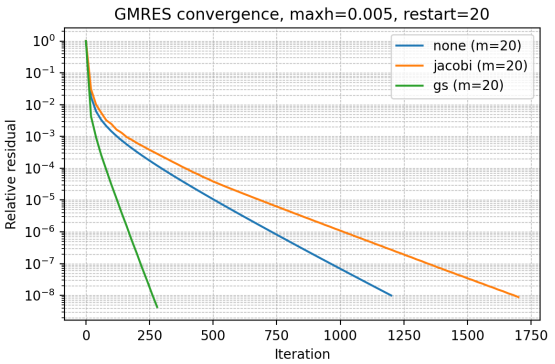
(a) `maxh` $= 0.1$



(b) `maxh` $= 0.05$



(c) `maxh` $= 0.01$



(d) `maxh` $= 0.005$

Figure 7.3: GMRES convergence for different mesh sizes with $k = 3$ and $N_{\text{threads}} = 1$.

are completely ignored. The left-preconditioned system

$$M_J^{-1} A^{(3)} x = M_J^{-1} b,$$

inherits almost all of the ill-conditioning of $A^{(3)}$ and has unfavourable spectral properties for restarted GMRES. For $m = 20$ the degree-m GMRES polynomials cannot approximate the inverse of $M_J^{-1} A^{(3)}$ as well as they do for $A^{(3)}$ itself, so each GMRES(m) cycle with Jacobi reduces the residual by a smaller factor than in the unpreconditioned case, and the total number of iterations increases.

The symmetric Gauss-Seidel preconditioner does not have this problem. In each forward and backward sweep, SGS processes all nonzero entries in a row and updates the unknowns sequentially, so the preconditioned vector contains information from both the diagonal and the off-diagonal couplings. This explains why SGS remains effective for $k = 3$, whereas Jacobi can become worse than no preconditioning.

These observations highlight an important limitation of scalar Jacobi: even in our uncoupled model problem it already discards all couplings between neighbouring unknowns. For genuinely coupled vector PDEs (such as the full Stokes or Navier-Stokes systems), the situation is even worse, because scalar Jacobi would also drop the couplings between the velocity and pressure components. In other words, it loses an even larger and more important part of the operator.

For such systems a more appropriate choice is a block Jacobi preconditioner, which inverts small blocks associated with all vector components at a node (or a small group of nodes) instead of only the scalar diagonal entries. Block preconditioners of this type are discussed, for example, in [26], [27]. We do not implement them here, but they align better with the block structure of the discretised operators and are a good candidate for future work.

	$m = 20$		$m = 50$		$m = 75$	
Preconditioner	**Iters**	**Time (s)**	**Iters**	**Time (s)**	**Iters**	**Time (s)**
maxh = 0.1, DOFs = 1 410						
None	387	2.359	230	2.708	188	3.039
Jacobi	60	0.393	50	0.610	63	0.972
Gauss-Seidel	20	0.166	23	0.165	23	0.166
maxh = 0.05, DOFs = 6 528						
None	438	2.793	293	3.736	264	4.592
Jacobi	80	0.525	100	1.297	75	1.410
Gauss-Seidel	36	0.249	34	0.350	34	0.348
maxh = 0.01, DOFs = 168 630						
None	563	10.960	354	11.670	324	13.874
Jacobi	480	9.433	400	13.341	375	16.838
Gauss-Seidel	120	4.659	150	7.786	146	9.127
maxh = 0.005, DOFs = 676 110						
None	1 202	72.339	652	58.969	560	65.509
Jacobi	1 700	105.596	950	89.882	825	96.692
Gauss-Seidel	280	38.509	250	41.483	225	43.394

Table 7.9: GMRES convergence for the Stokes-like problem with $k = 3$ for different mesh sizes (`maxh` $\in \{0.1, 0.05, 0.01, 0.005\}$), preconditioners and restart parameters, using 1 thread.

7.2.3. Parallel Scalability and Time-to-Solution

For the Stokes-like benchmark we now investigate the parallel behaviour of the GMRES solver. Based on the parameter study in Section 7.2.2, we fix the restart parameter to $m = 20$, as this gave the best trade-off between iteration count and per-iteration cost. The polynomial order is fixed at $k = 3$. All GMRES runs use the same relative tolerance and maximum iteration count as in the CG experiments of Section 7.1.5.

As before, we distinguish between strong and weak scaling. For strong scaling we vary the number of OpenMP threads for a fixed problem size. For weak scaling we increase the problem size together with the number of threads so that the work per thread remains approximately constant. The same notions of speedup and efficiency as in Section 7.1.5 are used.

Strong scaling

For strong scaling we fix the finest GMRES mesh (`maxh` = 0.005, $676{,}110$ DOFs) and vary the number of threads $N_{\text{threads}} \in \{1, 2, 4\}$. The time-to-solution and iteration counts for $m = 20$ and the three preconditioners are shown in Table 7.10. Here we see that increasing the number of threads actually *slows down* the computation.

Table 7.10: GMRES time-to-solution and iteration counts for the finest mesh (`maxh` = 0.005, $676{,}110$ DOFs), restart parameter $m = 20$, and different numbers of OpenMP threads.

Prec.	**1 Thread**	**2 Threads**	**4 Threads**
None	72.34 s (1202 iters)	81.21 s (1202)	111.68 s (1201)
Jacobi	105.60 s (1700 iters)	119.01 s (1700)	160.08 s (1700)
GS	38.51 s (280 iters)	63.14 s (280)	106.84 s (280)

For the unpreconditioned GMRES solver the runtime increases from about 72.3 seconds on 1 thread to 81.2 seconds on 2 threads and 111.7 seconds on 4 threads, even though the iteration count remains constant. The corresponding “speedups” relative to the 1-thread case are therefore smaller than 1 (see 7.11).

Table 7.11: Strong-scaling speedup Speedup(N) and parallel efficiency E_N for GMRES with $m = 20$ on the finest mesh (`maxh` = 0.005, $676{,}110$ DOFs). The reference time is the 1-thread run.

Method	Speedup(2) (E_2)	Speedup(4) (E_4)
None	0.89 (0.45)	0.65 (0.16)
Jacobi	0.89 (0.44)	0.66 (0.16)
GS	0.61 (0.30)	0.36 (0.09)

In other words, adding threads introduces overhead (synchronisation, scheduling) which more than cancels out any parallel benefit.

The Jacobi-preconditioned solver shows similar behaviour to the unpreconditioned case. Its iteration count on the finest mesh is around 1700 for all thread counts, so the total work per solve is fixed. Nevertheless, the runtime grows from approximately 105.6 seconds (1 thread) to 119.0 seconds (2 threads) and 160.1 seconds (4 threads). The corresponding speedups and efficiencies, summarised in Table 7.11, confirm that Jacobi GMRES also experiences a clear slowdown when increasing the number of threads.

The Gauss-Seidel (GS) preconditioner is the most effective in terms of iteration reduction, requiring only 280 iterations on this mesh. On a single thread it is the fastest configuration overall, with a runtime of about 38.5 seconds. However, its parallel behaviour is even worse: the time increases to 63.1 seconds on 2 threads and 106.8 seconds on 4 threads. This corresponds to Speedup$(2) \approx 0.61$ and Speedup$(4) \approx 0.36$, with very low efficiencies. Since the GS sweeps are sequential there is additional parallel overhead from thread management and synchronisation dominates as soon as multiple threads are used.

Compared to the CG results in Section 7.1.5, the GMRES implementation exhibits noticeably poorer strong scaling.

Overall, the strong-scaling results for GMRES show that, on this Stokes-like problem and with the current PyKokkos/NGSolve implementation, the most reliable way to minimise the time-to-solution on the finest mesh is to use a proper preconditioner on a single thread. Increasing the thread count up to 4 does not provide any benefit and in fact significantly degrades performance for all three preconditioners. This negative strong scaling is the reason we did not extend the GMRES experiments to 8 and 16 threads.

Weak scaling

For weak scaling we again aim to keep the workload per thread approximately constant while increasing both the problem size and the number of threads. For GMRES we consider the sequence in Table 7.12, where the mesh is refined and the number of threads is doubled at each step. The total number of DOFs grows from $41{,}196$ to $168{,}630$, so that the number of DOFs per thread stays close to 4.1×10^4.

Table 7.12: Weak scaling results for GMRES with $m = 20$: approximately constant DOFs per thread while increasing the total problem size and the number of threads.

Threads	`maxh`	**DOFs**	**Prec.**	**Time (s)**	**Iterations**
1	0.02	41 196	None	4.27	428
1	0.02	41 196	Jacobi	2.06	200
1	0.02	41 196	GS	0.92	60
2	0.0141	83 340	None	11.38	534
2	0.0141	83 340	Jacobi	6.49	300
2	0.0141	83 340	GS	3.31	80
4	0.01	168 630	None	25.84	563
4	0.01	168 630	Jacobi	22.40	480
4	0.01	168 630	GS	15.20	120

In an ideal weak-scaling scenario the runtime would remain roughly constant as the number of threads and the global problem size increase, because each thread handles the same amount of work. This is clearly not the case here. For the unpreconditioned GMRES solver the runtime grows from 4.27 seconds (1 thread) to 11.38 seconds (2 threads) and 25.84 seconds (4 threads), roughly a factor 6 increase when going from 1 to 4 threads while the DOFs per thread remain almost constant. The iteration count also increases with the global problem size, from 428 to 534 and 563 iterations, which already implies additional work per thread even in an ideal parallel scenario.

The Jacobi-preconditioned solver shows a similar trend, but the growth in runtime is even larger. The iteration count increases with a greater margin, resulting in a greater increase in total runtime. The total runtime increases from about 2.06 seconds on 1 thread to 6.49 seconds on 2 threads and 22.40 seconds on 4 threads. This is consistent with the discussion in Appendix A: Jacobi keeps only the diagonal and ignores the strong couplings between neighbouring degrees of freedom, so for the high-order discretisation the preconditioned system remains difficult to solve and the extra work per iteration does not translate into faster overall convergence.

For the GS-preconditioned solver the relative increase is strongest: the runtime goes from 0.92 seconds (1 thread) to 3.31 seconds (2 threads) and 15.20 seconds (4 threads). The iteration counts increase from 60 to 80 and 120, and the sequential components inside each GS sweep become more expensive on the finer meshes. As a result, the serial fraction of the computation grows, and according to Amdahl's law this limits the achievable weak-scaling performance even more severely than for the other methods.

Overall, the weak-scaling results confirm that the current GMRES implementation does not scale well. Even though the DOFs per thread are kept approximately constant, the total runtime increases substantially with the number of threads and the global problem size. The main reasons are the growth in iteration counts with mesh refinement, the higher per-iteration cost of GMRES (orthogonalisation and global reductions), and, for GS, the relatively large sequential part. Of these three preconditioners, GS remains the most effective in terms of iteration reduction and time-to-solution.

7.3. Stokes flow around a NACA airfoil

7.3.1. Model problem and solver parameters

Finally, to demonstrate that the PyKokkos–NGSolve solver can also be applied to a more realistic Computational Fluid Dynamics (CFD) problem, we consider a two–dimensional Stokes flow around a NACA 2412 airfoil [13], as introduced in Section 4.2.2.

Table 7.13: GMRES performance for the NACA Stokes problem on the coarse mesh (`maxh` = 0.5, $4\,865$ DOFs), restart parameter $m = 20$. An asterisk indicates that the maximum of 10 000 iterations was reached.

Prec.	Time (s)	Iterations	Rel. residual
None	65.07	10 000*	5.72×10^{-4}
Jacobi	0.75	116	5.00×10^{-1}
GS	72.49	10 000*	8.44×10^{-4}

Table 7.13 shows that on the coarse NACA mesh the preconditioners from before are no longer effective. Neither the unpreconditioned run nor the Gauss-Seidel preconditioner reaches the desired tolerance within 10 000 iterations. The Jacobi-preconditioned run stops after 116 iterations with a relative residual of about 5×10^{-1}, which is far above the target accuracy.

This seems to happen because the residual norm that GMRES estimates internally drops below the tolerance, even though the true residual (recomputed afterwards) is still large. However, to be certain about this more analysis would be needed. What we can say is that the Stokes system is strongly coupled between velocity and pressure, while the Jacobi preconditioner implemented here ignores this block structure. In such a setting the preconditioned system can behave poorly, and the residual estimate may become less reliable.

So GMRES appears to have "converged" while the solution is in fact still far from the exact one. We therefore regard the Jacobi preconditioner as failing for this test case.

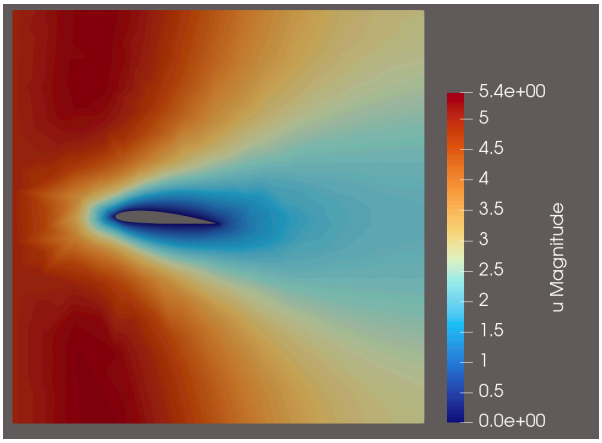


Figure 7.4: Velocity magnitude $|u|$ for the Stokes flow around a NACA 2412 airfoil with 4° angle of attack (`maxh` = 0.5). Red corresponds to high speed, blue to low speed.

Figure 7.4 shows the magnitude of the velocity $|u|$ for the Stokes flow around the NACA airfoil. The colours can be read as speed. The warm colours (yellow–red) indicate fast flow, and the cool colours (blue) indicate slow flow. Far away from the airfoil the flow is close to the imposed inflow, so the speed is almost constant, which appears as the large yellow–red region. On the airfoil itself we prescribe a no-slip condition, which means that the fluid sticks to the surface and the velocity is zero. This creates the thin dark-blue band of very low speed around the airfoil. Behind the trailing edge there is an extended blue region where the flow is noticeably slower than in front of the airfoil. This is called *wake*, a region where the fluid has been slowed down by the airfoil and only gradually speeds up again as we move further downstream.

For the NACA test case the preconditioners we considered before (Jacobi and Gauss-Seidel) are no longer sufficient. As shown in Table 7.13, for maxh $= 0.5$ none of the unpreconditioned, Jacobi or Gauss-Seidel preconditioned GMRES runs converge to the target tolerance. In contrast, a block preconditioner that respects the velocity-pressure structure of the Stokes system reduces the residual to 10^{-11} in only 35 iterations for a restart parameter $m = 50$. On finer meshes with up to $\mathcal{O}(10^5)$ unknowns the iteration counts with this preconditioner remain modest (about 50-60 iterations for $m = 50$), with runtimes below 15 seconds. This shows how important it is to use a preconditioner that takes the block structure of the equations into account.

preconditioners. However, such an analysis would require an extension of the present work and could not be included here due to time constraints. We therefore restrict ourselves to a brief description of the approach and refer the reader to [27], [28], [26] for further details. A more systematic investigation of the block preconditioner used here is good direction for future work.

8

Conclusions and Future

The goal of this thesis was to investigate how preconditioned Krylov subspace methods perform and scale under shared-memory parallelism. To this end, we implemented a preconditioned Conjugate Gradient (CG) solver in PyKokkos for a finite element Laplace model problem and extended the same kernel infrastructure to a restarted GMRES solver for a Stokes-like vector problem. The solvers were evaluated in terms of convergence behaviour, the effect of preconditioning, strong and weak scaling, and kernel-level performance on a multi-core CPU.

This chapter summarises the main findings, answers the research questions from Chapter 1, and outlines directions for future work.

8.1. Summary of Main Findings

8.1.1. Behaviour of CG for the Laplace problem

For the scalar Laplace model problem, the PyKokkos implementation of CG behaves in line with the theory.

- **Convergence and mesh refinement.** The iteration count increases significantly as the mesh is refined, from a few dozen iterations on the coarsest mesh to almost two thousand on the finest one. This is consistent with the dependence of CG on the condition number of the stiffness matrix, which becomes worse as the mesh size decreases.
- **Effectiveness of preconditioning.** The symmetric Gauss-Seidel (SGS) preconditioner consistently reduces the iteration count by roughly a factor of two compared to unpreconditioned CG. The Jacobi preconditioner has a much smaller effect. It's iteration count stays almost identical to the unpreconditioned method.
- **Time-to-solution versus iteration count.** On coarse meshes, SGS is the fastest method in total runtime. The problem is small and kernel launch overheads dominate. Therefore reducing the iteration count is the main advantage here. On the finest mesh, however, SGS becomes significantly slower than both Jacobi and the unpreconditioned solver, despite its lower iteration count. The sequential forward/backward sweeps inside SGS make for a large serial fraction per iteration, which limits parallel speedup and eventually outweighs the gains from fewer iterations.
- **Parallel scalability.** For the largest Laplace system (about 1.15×10^6 unknowns), CG with no preconditioning or with Jacobi scales reasonably well up to 16 threads. The speedup is clearly not linear, but the time-to-solution still improves from tens of seconds on a single thread to roughly one sixth of that on 16 threads. The parallel efficiency is far from ideal, yet there is a clear benefit of adding cores. In contrast, SGS exhibits very poor strong scaling. The runtime decreases only slightly with more threads. Weak-scaling experiments show a similar trend: none of the CG variants achieve ideal weak scaling, also here the degradation is much more pronounced for SGS.

Taken together, the CG experiments show that:

1. the iteration count is strongly influenced by the choice and quality of the preconditioner,
2. the amount of sequential work inside the preconditioner has a large effect on parallel scalability, and
3. larger problems gain more from shared-memory parallelism, because the parallel kernels (in particular SpMV) dominate the runtime while overheads and serial work become relatively less important.

8.1.2. Behaviour of GMRES for a Stokes-like vector problem

For GMRES, the results are less straightforward. We see that the structure of the problem, between preconditioning, problem structure, and parallel performance.

- **Restart parameter.** Increasing the restart parameter m always reduces the iteration count, however it doesn't not improve, and often worsens, the runtime. The additional work in orthogonalisation and the small least-squares solve grows roughly like $\mathcal{O}(m^2)$, while the time gained from having to perform fewer iterations is comparatively small. Over all, $m = 20$ provided the best balance between convergence and per-iteration cost.
- **Preconditioning for low-order ($k = 1$) discretisations.** For the lowest-order vector discretisation, SGS is clearly the most effective preconditioner. It reduces both iteration counts and runtimes by large factors compared to unpreconditioned GMRES. Jacobi behaves similarly to the CG case on the scalar Laplace problem: it produces iteration counts and runtimes that are very close to the unpreconditioned solver and therefore offers little benefit.
- **Influence of polynomial order and neighbour couplings.** For the higher-order vector discretisation ($k = 3$), each scalar basis function interacts with more neighbours and the assembled matrix contains more off-diagonal entries, as analysed in Appendix A. On coarse meshes the simple scalar Jacobi preconditioner still performs reasonably well, but on finer meshes the convergence even degrades, for $k = 3$ and the finest mesh the Jacobi-preconditioned GMRES solver requires more iterations and more time than the unpreconditioned solver. The key reason is that scalar Jacobi keeps only the diagonal entries and discards all off-diagonal couplings between neighbouring degrees of freedom, so for high-order elements the preconditioned operator $M_J^{-1}A$ keeps most of the ill-conditioning of A while changing its spectral properties in a way that is unfavourable for restarted GMRES. In contrast, SGS remains effective because its forward and backward sweeps process all nonzero entries in each row and therefore incorporate both diagonal and off-diagonal neighbour couplings.
- **Parallel scalability.** In contrast to CG, the GMRES implementation shows poor strong and weak scaling. On the finest mesh, increasing the number of threads up to four does not reduce the runtime. Instead, it even leads to a slowdown. The orthogonalisation and global reductions in GMRES introduce additional synchronisation overhead on top of the kernels. And although SGS shows the lowest iteration counts, the preconditioner itself is largely sequential, which makes the limits the overall speedup even more, this makes that the best time-to-solution is obtained on a single thread.

These findings show that for GMRES-like methods, simply adding more threads doesn't help. Both the preconditioner and the implementation of the Krylov subspace method need to be parallelizable, otherwise, the additional cores only lead to memory overhead.

8.1.3. Kernel-level observations

The kernel-level timings show consistent results for both CG and GMRES:

- **SpMV dominates.** Sparse matrix-vector products are the most expensive kernels in all experiments and they account for a large fraction of the total runtime per iteration. They benefit most clearly from increasing the thread count.
- **Vector operations scale reasonably.** The axpy, scaling and dot-product kernels also gain from multi-threading, but their contribution to the runtime is smaller.

- **Preconditioners shift the balance.** Stronger preconditioners reduce the number of outer iterations but introduce additional work per iteration. For SGS, this extra work is largely sequential and therefore limits the fraction of the code that can be accelerated by shared-memory parallelism.

Overall, the experiments confirm that, within the current PyKokkos design, the main performance gain on CPUs is the combination of a parallelizable preconditioner and an SpMV-dominated workload large enough to outweigh parallel overheads.

8.2. Answers to the Research Questions

Q1. How does the Conjugate Gradient method perform when solving the Laplace equation, and what are the key computational bottlenecks?

For the Laplace equation on the unit square, discretised with linear finite elements, the CG method behaves largely as the theory predicts. As the mesh is refined, the number of iterations grows noticeably, which reflects the increase in the condition number of the stiffness matrix. In terms of runtime, CG performs well once the systems become large. On the finest mesh the strong-scaling experiments show clear, although less than ideal, speedups up to 16 threads.

The main computational bottleneck is the sparse matrix–vector product, followed by the vector updates and global reductions. These operations are memory-bandwidth bound and dominate the cost of each iteration. In the PyKokkos implementation these kernels are parallelised with OpenMP, and for sufficiently large problems the speedup in SpMV translates directly into a shorter time-to-solution.

Q2. How do different preconditioners affect the convergence and performance of CG when solving the Laplace equation?

Both Jacobi and symmetric Gauss–Seidel (SGS) preconditioners reduce the CG iteration count, but in very different ways and with different consequences for runtime.

Jacobi preconditioning only modestly reduces the number of iterations, especially on the finer meshes, but it is cheap to apply and fully parallel. In practice this means that the Jacobi-preconditioned solver has runtimes that are comparable to, and sometimes slightly better than, unpreconditioned CG, with similar scaling behaviour.

SGS, on the other hand, often cuts the number of iterations roughly in half. On coarse meshes, where launch overheads dominate and the total amount of work is small, this makes SGS the fastest option. On fine meshes and for many threads the picture changes: the forward and backward SGS sweeps are essentially sequential, so they limit the speedup that can be achieved. As a result, the total runtime becomes much worse than for the Jacobi and unpreconditioned solvers, despite the smaller iteration count. This shows a key trade-off: a useful preconditioner must not only improve convergence, it also has to fit the parallel execution model, otherwise the extra work can cancel out its benefits.

Q3. How does the performance of a preconditioned GMRES method compare when solving the Stokes system, and how do the results relate to those observed for the Laplace equation?

For the Stokes-like vector problem, preconditioned GMRES shows a similar *ordering* of preconditioners as CG. SGS is again the most effective in terms of reducing iterations, Jacobi is relatively weak for low-order discretisations, and for higher-order elements it becomes clear that the coupling between neighbour of the system matrix should not be ignored.

In terms of raw performance and scalability for a SPD problem, GMRES is clearly less favourable than CG. This is not very surprising as CG is designed for symmetric positive definite systems and make use of that structure, whereas GMRES is designed for more general (in particular nonsymmetric or block-structured) linear systems such as the Stokes problem. The restarted GMRES implementation has a higher per-iteration cost because of orthogonalisation and global reductions, and these parts are difficult to parallelise efficiently on shared-memory architectures. On the finest mesh, the strong-scaling experiments show that increasing the number of threads up to four does not reduce the runtime and can even slow the solver down for all three preconditioners. The weak-scaling experiments point in the same direction: growing iteration counts, more expensive iterations, and sequential work inside the

preconditioners together prevent GMRES from achieving good parallel efficiency. In other words, the extra generality of GMRES comes with a noticeable performance penalty on this hardware and with the simple preconditioners used here.

Compared to the Laplace/CG case, the Stokes-like/GMRES results indicate that:

- the quality and structure of the preconditioner are even more important for nonsymmetric or block-structured systems, and
- the choice of Krylov method and its implementation details (restart strategy, orthogonalisation scheme, communication pattern) play a crucial role in determining scalability.

In short, the Stokes experiments show that simply switching from CG to GMRES on a more complicated system, while reusing the same basic preconditioners, is not enough to obtain good performance on multi-core hardware.

8.3. Recommendations and Future Work

The results in this thesis suggest several practical recommendations for designing Krylov solvers on shared-memory architectures, and point to a number of promising directions for further work. The main themes are the role of preconditioning, the interaction with parallelism, and the choice of architectures.

8.3.1. Preconditioners and Solver Design

A recurring observation throughout the experiments is the large effect of preconditioning for both CG and GMRES. For the Laplace problem, we saw that symmetric Gauss–Seidel can roughly halve the number of CG iterations compared to the unpreconditioned solver, while scalar Jacobi only gives a modest improvement. For the Stokes-like GMRES tests, SGS again clearly reduced the iteration counts, whereas Jacobi often behaved similarly to no preconditioning for low-order discretisations and even degraded convergence for higher-order vector problems. This underlines that a “good” preconditioner must be judged in context: it has to improve convergence for the specific discretisation, and at the same time it must fit the parallel execution model.

These observations motivate two concrete recommendations. First, for vector-valued and higher-order finite element spaces, preconditioners should respect the local coupling between neighbouring degrees of freedom and, in genuinely coupled systems, the natural block structure of the equations. As discussed in Appendix A, higher-order elements make each unknown interact with many neighbours, so a purely diagonal preconditioner like scalar Jacobi discards almost all of the important off-diagonal couplings and quickly becomes ineffective. The Stokes and NACA experiments further show that for mixed velocity–pressure systems simple pointwise preconditioners can fail completely, whereas block preconditioners that treat the coupled fields together can give mesh-robust convergence [26], [27].

Second, the strong reduction in iteration counts obtained with SGS indicates that preconditioners which incorporate the full row-wise couplings can be very effective, but their sequential sweeps are a bottleneck on many cores. This points towards preconditioners that use Gauss–Seidel or Jacobi-type updates as local smoothers inside more parallel-friendly multilevel methods, such as multigrid, where only a few sequential smoothing steps are combined with coarse-grid correction and most of the work can be distributed efficiently across threads.

Beyond this, it might be good to look at more advanced preconditioning strategies that are standard in the literature but not yet implemented here. Incomplete factorisation methods (ILU) and multigrid are known to yield iteration counts that are essentially independent of the mesh size. Given that the current PyKokkos implementation already provides parallel SpMV and vector kernels, integrating such preconditioners would be a way to reduce the overall cost, especially for large problems. For true Stokes or Navier–Stokes systems, block preconditioners that approximate the Schur complement of the pressure block are a particularly promising direction, as they explicitly target the saddle-point structure that GMRES has to deal with.

On the Krylov side, the GMRES experiments show that the choice of restart parameter and orthogonalisation strategy has a large impact on both robustness and scalability. In the present work, a standard restarted GMRES with fixed m and classical orthogonalisation was used. Communication-avoiding

or pipelined variants, as well as flexible GMRES with varying preconditioners may be better suited to modern multi-core architectures. Exploring such variants would help to clarify whether the poor parallel scalability observed here is intrinsic to GMRES in this setting, or mostly a consequence of the simplest possible implementation.

8.3.2. Parallel Implementation and Architectures

From a performance point of view, the experiments show that problem size and parallelism cannot be considered separately. For the Laplace/CG case, small meshes are dominated by kernel launch overheads and serial work, and scaling is poor. As the mesh is refined, the sparse matrix–vector product and associated vector operations become the main cost per iteration, and the CG solver shows clear, though not optimal, speedups up to 16 threads. For GMRES on the Stokes-like problem, however, the additional orthogonalisation and global reductions largely cancel the gains from parallel SpMV, and increasing the number of threads can even slow the solver down.

A practical recommendation is to treat parallelism as part of the algorithmic design, rather than as an afterthought. Preconditioners and Krylov variants should be chosen not only for their convergence properties, but also for how well their internal operations can be parallelised. In particular, reducing the serial fraction of the computation, such as sequential sweeps in SGS, is important. The current PyKokkos implementation already makes it possible to parallelise the core kernels (SpMV, axpy, dot, scale). The main limitations observed in this thesis are therefore due to the chosen algorithms and preconditioners, rather than to PyKokkos itself. This suggest that more advanced preconditioners and parallel-friendly Krylov variants could be integrated without changing the overall design.

Another possible extension is to test the developed solvers on other architectures. All experiments here were carried out on a shared-memory CPU, but PyKokkos is designed for performance portability. Repeating the study on GPU backends and in hybrid MPI+OpenMP settings would give a more complete picture of the strengths and weaknesses of the current approach. On GPUs, for example, the balance between computation and memory bandwidth is different, and simple preconditioners such as Jacobi may be more attractive, for instance as smoothers inside multigrid methods.

In summary, this thesis shows that preconditioned Krylov methods implemented in PyKokkos can achieve good performance and reasonable parallel scalability on multi-core CPUs for large finite element problems, provided that the preconditioner and problem size are chosen carefully. At the same time, the results highlight that there is still room for improvement, in particular through preconditioners that exploit block structure and multilevel ideas, and through Krylov variants that are designed explicitly with modern parallel hardware in mind.

References

[1] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd. Philadelphia, PA: SIAM, 2003.

[2] W. F. Claude Pommerell, "Memory aspects and performance of iterative solvers," *SIAM*, 1994.

[3] H. A. van der Vorst, *Iterative krylov methods for large linear systems*. [Online]. Available: https://www.cambridge.org/core/books/iterative-krylov-methods-for-large-linear-systems/FFB93854B3C47699F045AC396C0A208F.

[4] M. van Gijzen, *Iterative methods for linear systems of equations*, PhD-course, 2008. [Online]. Available: https://diamhomes.ewi.tudelft.nl/~mvangijzen/PhDCourse_DTU/LES1/TRANSPARANTEN/les1.pdf.

[5] R. Barrett, M. Berry, T. Chan, *et al.*, *Templates for the solution of linear systems: Building blocks for iterative methods*. SIAM, 1994.

[6] Y. Saad and M. H. Schultz, "GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems," *SIAM Journal on Scientific and Statistical Computing*, vol. 7, no. 3, pp. 856–869, Jul. 1986, ISSN: 0196-5204. DOI: 10.1137/0907058. [Online]. Available: http://dx.doi.org/10.1137/0907058.

[7] M. van Gijzen, *Iterative methods for linear systems of equations*, PhD-course, 2008. [Online]. Available: https://diamhomes.ewi.tudelft.nl/~mvangijzen/PhDCourse_DTU/LES3/TRANSPARANTEN/les3.pdf.

[8] S. C. Brenner and L. R. Scott, *The mathematical theory of finite element methods*. Springer, 2011.

[9] T. J. R. Hughes, *Finite element method: Linear static and dynamic finite element analysis*. Dover Publications, 2012.

[10] C. Pechstein, *Finite element basics*. [Online]. Available: https://www.numa.uni-linz.ac.at/Teaching/LVA/2012s/CEM/finiteelements.pdf.

[11] Simulate, Jul. 2024. [Online]. Available: https://youtu.be/1wSE6iQiScg?si=KVqUfX_8Ozz1nfy-.

[12] [Online]. Available: https://www.geogebra.org/calculator.

[13] N. team., *Ngsolve*. [Online]. Available: https://docu.ngsolve.org/v6.2.2203/index.html.

[14] D. H. P. C. C. (DHPC), *DelftBlue Supercomputer (Phase 2)*, https://www.tudelft.nl/dhpc/ark:/44463/DelftBluePhase2, 2024.

[15] B. Barney, L. Computing (retired), D. Frederick, and LLNL, *Introduction to parallel computing tutorial*. [Online]. Available: https://hpc.llnl.gov/documentation/tutorials/introduction-parallel-computing-tutorial.

[16] T. Heister, M. Kronbichler, and W. Bangerth, "Massively parallel finite element programming," vol. 6305, Jul. 2010, pp. 122–131, ISBN: 978-3-642-15645-8. DOI: 10.1007/978-3-642-15646-5_13.

[17] S. Weiss, *Cuny*. [Online]. Available: https://www.cs.hunter.cuny.edu/~sweiss/course_materials/csci493.65/lecture_notes/chapter11.pdf.

[18] M. P. I. Forum, Nov. 2003. [Online]. Available: https://www.mpi-forum.org/docs/mpi-4.1/mpi41-report.pdf.

[19] P. Team, *Pykokkos documentation*. [Online]. Available: https://kokkos.org/pykokkos/index.html.

[20] C. to the Kokkos project, *Kokkos: The programming model¶*, 2025. [Online]. Available: https://kokkos.org/kokkos-core-wiki/index.html.

[21] A. Grama, A. Gupta, G. Karypis, and V. Kumar, Jan. 2003. [Online]. Available: https://github.com/vineethshankar/pagerank/blob/master/Introduction%20to%20Parallel%20Computing,%20Second%20Edition-Ananth%20Grama,%20Anshul%20Gupta,%20George%20Karypis,%20Vipin%20Kumar.pdf.

[22] H. J. Reijersen van Buuren, *Hugoreijersen/krylov-subspace-methods: Analyzing the performance of preconditioned krylov subspace methods.* Dec. 2025. [Online]. Available: https://github.com/Hugoreijersen/Krylov-Subspace-Methods.git.

[23] K. Hunter, L. Spracklen, and S. Ahmad, *Two sparsities are better than one: Unlocking the performance benefits of sparse-sparse networks*, Dec. 2021. DOI: 10.48550/arXiv.2112.13896.

[24] C. to the Kokkos project, *Kokkos: The programming model*. [Online]. Available: https://kokkos.org/kokkos-core-wiki/.

[25] O. Villa, D. Chavarrıa-Miranda, V. Gurumoorthi, A. Marquez, and S. Krishnamoorthy. [Online]. Available: https://www.sci.utah.edu/~beiwang/teaching/cs6210-fall-2016/nonassociativity.pdf.

[26] M. Benzi. [Online]. Available: https://wsc.project.cwi.nl/woudschoten-conferences/2012-woudschoten-conference/Benzi1.pdf.

[27] M. Murphy, G. Golub, and A. Wathen, Jun. 1999. [Online]. Available: https://epubs.siam.org/doi/10.1137/S1064827599355153.

[28] C. Vuik, Aug. 2009. [Online]. Available: https://diamhomes.ewi.tudelft.nl/~kvuik/talks/preco_2009.pdf.

A

Influence of polynomial order

To investigate how the polynomial order impacts the behaviour of the preconditioners, we repeat the GMRES experiments with a higher-order vector space

$$V_h^{(3)} = \mathrm{VectorH1}(\mathsf{order} = 3),$$

which uses cubic vector-valued basis functions. As a reference, we compare to the linear space $V_h^{(1)} = \mathrm{VectorH1}(\mathsf{order} = 1)$. Increasing the polynomial order raises the number of degrees of freedom per element and changes both the sparsity pattern and the conditioning of the system matrix.

Local degrees of freedom and element matrices. We conside the Stokes-like model as introduced in Section 4.2.1, where the unknown is the velocity

$$u = (u_x, u_y),$$

and both components satisfy a scalar reaction-diffusion equation

$$-\nu\Delta u_x + \varepsilon u_x = f_x \quad \text{in } \Omega, \qquad u_x = 0 \text{ on } \partial\Omega,$$

$$-\nu\Delta u_y + \varepsilon u_y = f_y \quad \text{in } \Omega, \qquad u_y = 0 \text{ on } \partial\Omega.$$

Let V_h be the scalar finite element space of continuous piecewise polynomials of degree k on a shape-regular triangulation $\mathcal{T}_h$ of Ω. On a single triangle $K \in \mathcal{T}_h$ the local scalar space is the polynomial space $\mathbb{P}_k(K)$, which has

$$n_{\mathsf{loc}} = \frac{(k+1)(k+2)}{2}$$

local basis functions. For $k = 1, 2, 3$ this gives $n_{\mathsf{loc}} = 3, 6, 10$, which matches the vertex, edge and interior nodes shown in Figure A.1.

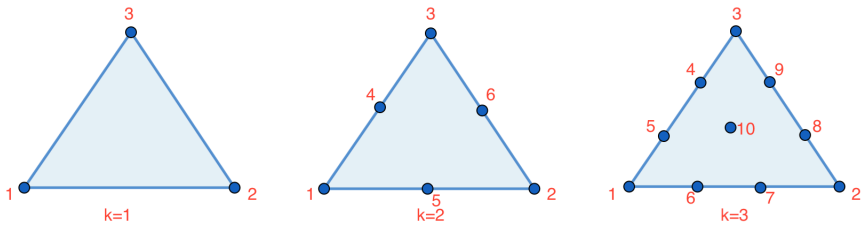


Figure A.1: Distribution of nodes (degrees of freedom) on the reference triangle for polynomial orders $k = 1$, $k = 2$, and $k = 3$, created using Geogebra [12].

As in Section 4.2, let $\{N_i\}_{i=1}^{n_{\text{loc}}}$ be the scalar basis functions on K. For one velocity component u_x or u_y the local matrix $A^{K,(k)}$ has entries

$$A_{ij}^{K,(k)} = \nu \int_K \nabla N_i \cdot \nabla N_j \, dx + \varepsilon \int_K N_i N_j \, dx, \qquad 1 \le i, j \le n_{\text{loc}}.$$

The vector-valued space we use for the velocity is the product

$$V_h^{\text{vec}} = V_h \times V_h = \{(v_x, v_y) : v_x, v_y \in V_h\}.$$

With test functions $v = (v_x, v_y)$, the weak form of the problem splits into two parts,

$$a(u, v) = a_x(u_x, v_x) + a_y(u_y, v_y),$$

with

$$a_x(u_x, v_x) = \nu \int_\Omega \nabla u_x \cdot \nabla v_x \, dx + \varepsilon \int_\Omega u_x v_x \, dx,$$

and

$$a_y(u_y, v_y) = \nu \int_\Omega \nabla u_y \cdot \nabla v_y \, dx + \varepsilon \int_\Omega u_y v_y \, dx.$$

We see the discrete system consists of two independent scalar problems with the same operator.

On each element K we have a matrix $A^{K,(k)}$ with entries $A_{ij}^{K,(k)}$. This matrix couples the unknowns from that triangle. To get the matrix for the whole problem, we first number all unknowns from 1 to n. Then, for each triangle K, we take its matrix $A^{K,(k)}$ and add its entries to the large matrix at the positions corresponding to the numbers of the involved unknowns.

After this assembly step over all elements in $\mathcal{T}_h$ we obtain two matrices $A_x^{(k)}, A_y^{(k)} \in \mathbb{R}^{n \times n}$ one for each component, where n is the number of degrees of freedom in V_h. These matrices are assembled from the same local matrices $A^{K,(k)}$ and are therefore identical up to possible differences in boundary rows, both are $n \times n$ symmetric positive definite matrices. With the ordering

$$u_h = \begin{pmatrix} u_{x,1} \\ \vdots \\ u_{x,n} \\ u_{y,1} \\ \vdots \\ u_{y,n} \end{pmatrix},$$

the global matrix has the block-diagonal form

$$A^{(k)} = \begin{pmatrix} A_x^{(k)} & 0 \\ 0 & A_y^{(k)} \end{pmatrix}.$$

Consequences for Jacobi preconditioning. The Jacobi preconditioner for the vector-valued system is defined as the diagonal of $A^{(k)}$,

$$M_J = \operatorname{diag}\big(A^{(k)}\big) = \begin{pmatrix} \operatorname{diag}\big(A_x^{(k)}\big) & 0 \\ 0 & \operatorname{diag}\big(A_y^{(k)}\big) \end{pmatrix},$$

so each degree of freedom is scaled only by its diagonal entry and all off-diagonal entries are ignored.

For $k = 1$ each row of $A_x^{(k)}$ and $A_y^{(k)}$ contains relatively few off-diagonal entries, and the diagonal still represents a reasonable fraction of the local stiffness-plus-mass operator. In this case Jacobi provides a slight improvement over no preconditioning, as seen in the GMRES iteration counts for $V_h^{(1)}$ (see Figure 7.2).

For $k = 3$ the situation is different. Each row now couples to many more neighbouring unknowns with entries of comparable magnitude, while M_J still uses only the diagonal. The preconditioned matrix $M_J^{-1}A^{(k)}$ therefore remains strongly ill-conditioned and its eigenvalues are poorly clustered. This is reflected in the experiments: for $V_h^{(3)}$, the Jacobi-preconditioned GMRES method requires a similar or even larger number of iterations than the unpreconditioned method (see Figure 7.3). This behaviour is consistent with the theory that GMRES(m) convergence depends on the spectral properties of the preconditioned operator $M_J^{-1}A^{(k)}$ rather than on $A^{(k)}$ alone [1], and with the fact that higher-order elements lead to worse conditioning and weaker diagonal dominance.