Parallel Geometric Multigrid for Poisson Problems

Authors:
Francesco VIRGULTI
Laura Gioanna PAXTON

Supervisor:
Emile PAROLIN

Project Report
at Sorbonne University

May 17, 2025

Contents

1 Mathematical Introduction

The solution of large sparse systems of linear equations is a fundamental task in scientific computing, particularly when addressing discretized elliptic problems such as the Poisson equation. Multigrid methods stand out as some of the fastest numerical algorithms for this purpose. Their efficiency is attributed to two core strategies: utilizing coarse grids to construct approximate solutions, which are then interpolated onto finer grids, and smoothing high-frequency error components before restriction to coarser levels; techniques that will be introduced in this paper.

The project focuses on implementing a multigrid method for solving a Poisson problem over a rectangular domain with homogeneous Dirichlet boundary conditions. Initially, a sequential algorithm based on a finite-difference discretization is developed, incorporating several iterative solvers such as Jacobi, Gauss-Seidel, Steepest Decent or the Conjugate Gradient method. Subsequently, a parallelized version is created to exploit distributed computing architectures.

Verification against direct solvers with known analytic solutions, convergence studies, and performance scaling tests form essential components of this work.

We begin with the definition and discretization of the presented problem to solve, finding the solution to the Poisson problem in a rectangular domain $\Omega = [0, a] \times [0, a]$ with homogeneous Dirichlet boundary conditions where $a \in \mathbb{R}$:

$$-\Delta u = f \quad \text{in } \Omega, \tag{1}$$

$$u = 0 \quad \text{on } \partial\Omega. \tag{2}$$

To verify the correctness of the later implementation, it is crucial to select a right-hand side f corresponding to an analytical solution u that satisfies the boundary conditions and is easily computable.

We take

$$u(x, y) = \sin\left(p\pi\frac{x}{a}\right)\sin\left(q\pi\frac{y}{a}\right), \ (x, y) \in \Omega \tag{3}$$

with $p, q \in \mathbb{Z}$. This is a well known solution to the Poisson equation with which we can verify our results. This function vanishes on the boundary $\partial\Omega$, satisfying the homogeneous Dirichlet conditions naturally. Therefore, the corresponding right-hand side f is the Laplacian of the above equation

$$f(x, y) = \left(\left(\frac{p\pi}{a}\right)^2 + \left(\frac{q\pi}{a}\right)^2\right)\sin\left(p\pi\frac{x}{a}\right)\sin\left(q\pi\frac{y}{a}\right), \ (x, y) \in \Omega \tag{4}$$

We consider our two-dimensional square domain $[0, 1] \times [0, 1]$ and discretize it uniformly with N points in each direction and grid spacing $h = \frac{1}{N-1}$. To discretize the Laplacian operator $\Delta u = \nabla^2 u$, we use a second-order accurate 5-point stencil finite difference approximation. At each interior grid point (x_i, y_j) with $x_i = iah$ and $y_j = jah$ for $i, j \in \{0, 1, ..., N-1\}$, the Laplacian is approximated by:

$$\Delta u \approx \frac{1}{h^2}\left(u_{(i-1)\cdot N+j} + u_{(i+1)\cdot N+j} + u_{i\cdot N+(j-1)} + u_{i\cdot N+(j+1)} - 4u_{i\cdot N+j}\right).$$

The resulting linear system that arises from the discretization is

$$Au_h = f$$

where $u_h = [u_{i,j}]_{i\cdot N+j}$ is the discrete approximation of the solution $u(x, y)$ at the N^2 uniformly distributed interior grid points with $u_{i,j}$ the approximation of $u(x_i, y_j)$. The same holds for $f = [f_{i,j}]_{i\cdot N+j}$ with $f_{i,j} \approx f(x_i, y_j)$. Both are vectors of dimension $1 \times N^2$.

The matrix A inherits the structure of the 5-point stencil. This structure is block tri-diagonal, where each block itself is tri-diagonal. The entries -1 represent the coupling between neighbouring nodes in the x and y directions, while the 4 on the diagonal reflects the sum of contributions from all four neighbouring points. The dimension of the matrix is $N^2 \times N^2$ and will look approximately like the following

$$A = \frac{a}{h^2}\begin{pmatrix} 4 & -1 & 0 & \cdots & -1 & 0 & \cdots \\ -1 & 4 & -1 & \ddots & 0 & \ddots & \ddots \\ 0 & -1 & 4 & \ddots & \ddots & \ddots & -1 \\ \vdots & \ddots & \ddots & \ddots & -1 & 0 & \vdots \\ -1 & 0 & \ddots & -1 & 4 & -1 & 0 \\ 0 & \ddots & \ddots & 0 & -1 & 4 & -1 \\ \vdots & \ddots & -1 & \cdots & 0 & -1 & 4 \end{pmatrix}$$

2 Analysis of Stationary Iterative Methods

The most commonly used iterative methods for solving the Poisson equation include Jacobi, Gauss-Seidel, Steepest Descent, and the Conjugate Gradient method. As these are well-established techniques in numerical linear algebra, we refrain from providing detailed individual introductions here. Instead, we refer the reader to the comprehensive treatment in Demmel's book [3]. In particular, Chapter 6.5 discusses classical basic iterative methods, which are often derived via matrix splitting strategies.

These approaches decompose the system matrix into easily manageable components to iteratively approximate the solution. Chapter 6.6 then introduces Krylov subspace methods, including the Conjugate Gradient method, which leverage orthogonal projections onto subspaces to accelerate convergence for symmetric positive definite systems.

Given that our primary goal is to evaluate the correctness and efficiency of the implemented algorithms, we now turn to an error convergence analysis. Specifically, we will assess the convergence behaviour of each method with respect to the residual. This residual provides a practical measure of how accurately the current iterate satisfies the discrete system. In addition to residual-based convergence, we also examine the time-to-convergence and various forms of numerical error, which offer insight into how closely the computed solution approximates the true solution of the continuous problem. These analyses collectively provide a deeper understanding of each method's performance and reliability when applied to the Poisson equation.

2.1 Residual Convergence Analysis

Figure 1 shows the residual norm defined as $||f - Au_h^{(k)}||$, where h denotes the discretization step size and k the iteration for the four iterative methods: Jacobi, Gauss-Seidel, Steepest Descent, and Conjugate Gradient. This residual provides a quantitative measure of how well each intermediate solution satisfies the discretized linear system at each step.

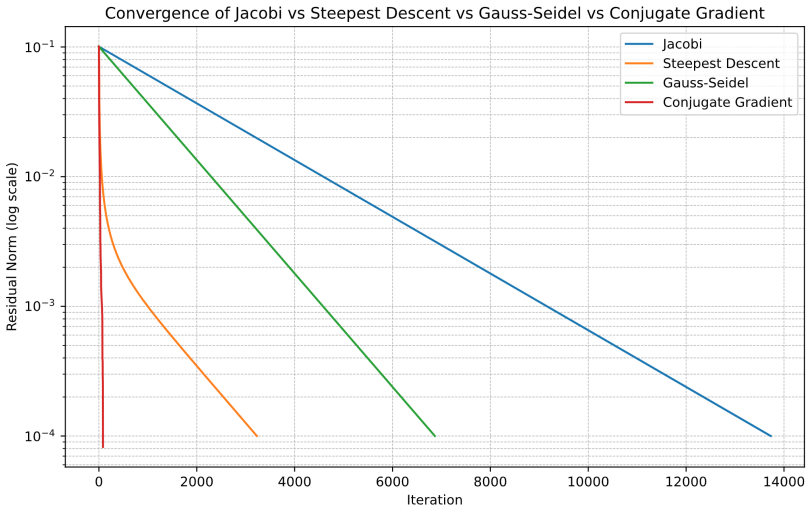

Figure 1: Residual Convergence of Methods vs Iterations with $N = 64$

From the plot on the figure 1, we observe that the Conjugate Gradient (CG) method exhibits the fastest convergence in terms of residual reduction. This aligns with its theoretical foundation as a Krylov subspace method tailored for symmetric positive definite (SPD) systems—such as those arising from the discretization of the Poisson equation.

The Steepest Descent method, while also a gradient-based approach, converges more slowly due to its reliance on a single direction of descent at each iteration. It does not benefit from the orthogonalization and conjugacy properties that accelerate CG. Gauss-Seidel, a classic stationary method based on matrix splitting, outperforms Jacobi due to its use of updated values within the iteration, which effectively enhances information propagation across the grid. Its convergence is guaranteed for SPD matrices but can still be slow for large systems. The Jacobi method demonstrates the slowest convergence, primarily due to its reliance on a simple diagonal splitting of the matrix and complete decoupling of updates.

Notably, the residual curves stop near a certain threshold, namely 10^{-4}. This is a direct result of the stopping criterion defined in the implementation: the iterations terminate once the residual norm falls below this

predefined tolerance. As such, the abrupt discontinuation of the residual curves does not imply stagnation of the method, but rather successful convergence within the prescribed numerical accuracy.

2.2 Time Convergence Analysis

Figure 2 shows the computation time as a function of the problem size N on a log-log scale for four classical iterative methods. Reference slopes of $\mathcal{O}(N^2)$ and $\mathcal{O}(N^3)$ are included for comparison, providing theoretical complexity benchmarks.

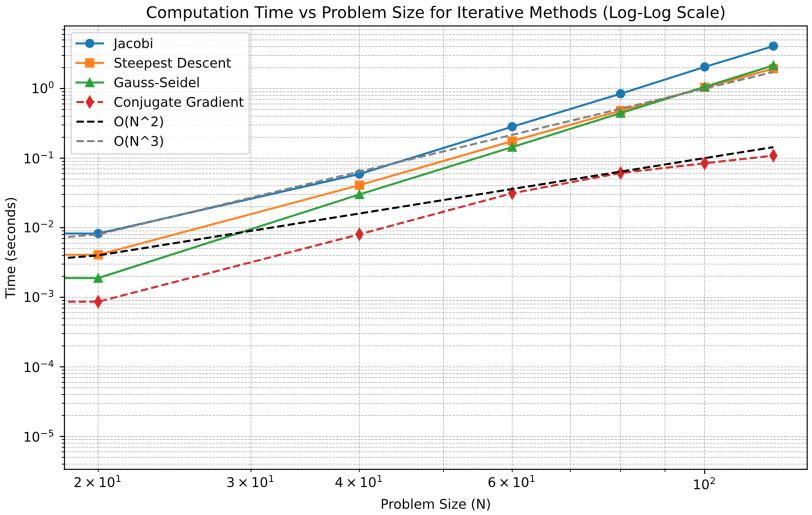

Figure 2: Computation time of different stationary iterative methods

We observe that the Conjugate Gradient (CG) method consistently yields the lowest computation time and follows a trend close to the $\mathcal{O}(N^2)$ line. This agrees with theoretical expectations as seen it the analysis of the iterative methods in [3]. The Steepest Descent method shows a steeper growth curve, indicating slower convergence. Gauss-Seidel and Jacobi, both matrix-splitting methods, display the steepest curves, with Jacobi in particular approaching or exceeding $\mathcal{O}(N^3)$ for larger problems. This is consistent with theory: for a discretized 2D Poisson problem, these methods typically require $\mathcal{O}(N^2)$ iterations (due to the condition number), each costing $\mathcal{O}(N^2)$, resulting in an overall cost near $\mathcal{O}(N^4)$ — though simplified models often approximate this as $\mathcal{O}(N^3)$ in practice.

2.3 Error Convergence Analysis

To evaluate the accuracy of an iterative numerical solution, we analyse the total error between the true solution and the approximate result after k iterations on a grid with spacing h. This error is given by:

$$\|u - u_h^{(k)}\| = \underbrace{\|u - u_h\|}_{\text{discretization error}} + \underbrace{\|u_h - u_h^{(k)}\|}_{\text{iteration error}}$$

Each component captures a distinct source of inaccuracy. Note that $u = u_{true}(x_i, y_j)$ for $0 \leq i, j \leq N^2$ the discretisation coordinates.
The discretization error $\|u - u_h\|$, or $\|u - u_h^{(\infty)}\|$ taking the method when it converges, arises from approximating the continuous problem using a discrete grid. This error is dependent on the mesh size h and gives an error curve following $\mathcal{O}(h^2)$).

The iteration error $\|u_h - u_h^{(k)}\|$ measures how far the numerical solution at the current iteration k is from the converged solution of the discrete problem.

Figure 3 illustrates the convergence of the total error $\|u - u_h^{(k)}\|$ for a grid size of $N = 64$, comparing several iterative methods. As expected, and consistent with the residual convergence shown in Figure 1, the Jacobi method converges the slowest, followed by Gauss-Seidel. The Conjugate Gradient (CG) and Steepest Descent methods achieve rapid convergence, with a significant reduction in error within just a few iterations. The reason for the Steepest Descent method to show a very good error convergence, despite having a time convergence closer to that of the Jacobi method (visible in Figure 2), comes from different costs per iteration and different

convergence behaviour per iteration. The steepest descent requires few iterations, yet they are costlier than the CG method.

A horizontal reference line at $\frac{1}{N^2}$ is shown, representing the theoretical bound of the discretization error, the error introduced by approximating the continuous PDE on a discrete grid. When the iteration error becomes sufficiently small, the total error levels off near this value. This behaviour highlights the differences of the two aforementioned errors. In order to take full advantage of iterative solvers (as the Multigrid method that is yet to be discussed) it is important to understand how to differentiate the errors.

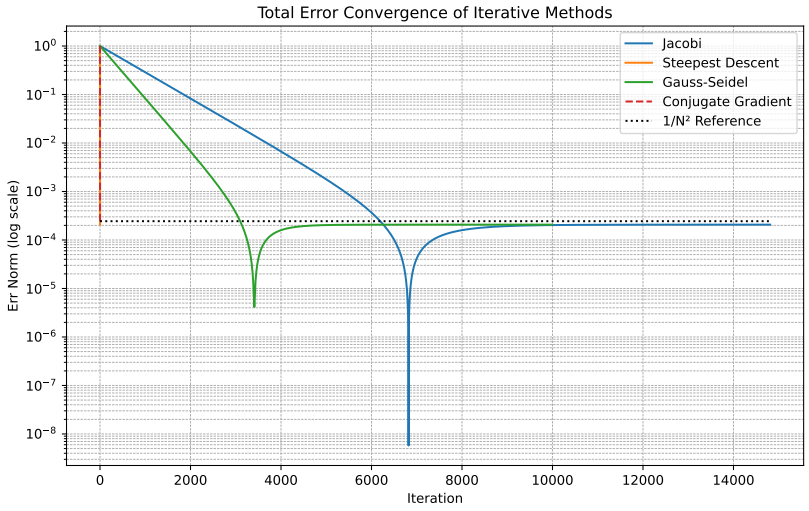

Figure 3: Error convergence using log scale ($N = 64$)

Discretization Error The discretization error represents a lower bound on the total error, as it cannot be reduced by further iterations. This becomes evident when solving the linear system resulting from the discretised Poisson equation using a direct solver. In that case, the solution u_h satisfies the discrete system $Au_h = f$ exactly by inverting the matrix A and solving $u_h = A^{-1}f$, so any remaining error is purely due to discretisation. This is illustrated in Figure 4, which shows the expected $\mathcal{O}(\frac{1}{N^2})$ convergence of the error as N increases. The discretisation error, also known as truncation error, can be traced back to the approximation of the Laplacian operator on the solution u. When Taylor expanding the function u around (x_i, y_j), one can find that the local truncation error is $\tau_{i,j} = \Delta u(x_i, y_j) - \Delta_h u$ which comes from the leading second derivatives in the Taylor series cancelling exactly, and the next term (fourth derivatives) being scaled by h^2. A full proof can be found in [2].

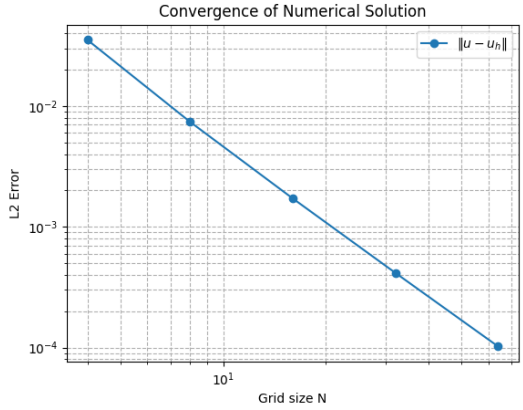

Figure 4: Discretization error demonstrated using a Direct Solver

The plot in Figure 5 shows the total error $\|u - u_h^{(k)}\|$ across multiple grid sizes N for the iterative methods. All methods converge toward the same error trend as N increases, closely following the $\mathcal{O}(\frac{1}{N^2})$ reference line.

The fact that all methods, regardless of their convergence speed, eventually aligning with the $\frac{1}{N^2}$ slope confirms that they are correctly solving the discrete system up to the limit imposed by the spatial discretisation. This reinforces the interpretation of the discretisation error as a lower bound on the achievable accuracy for a given grid size.

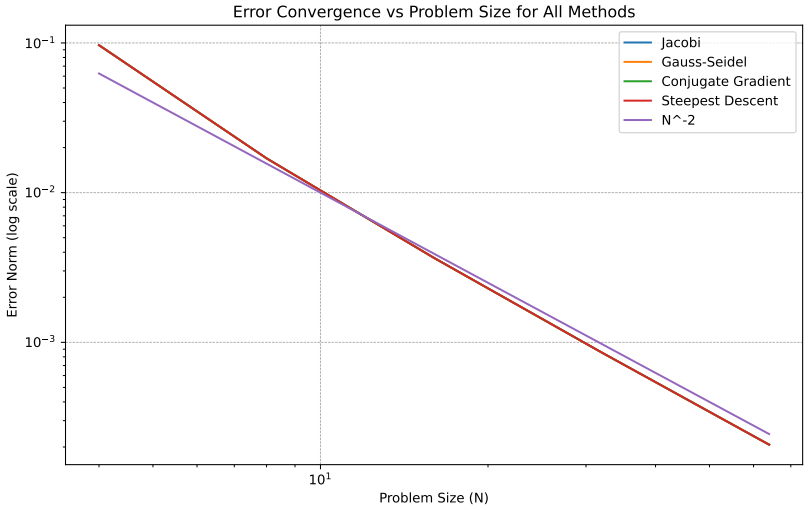

Figure 5: Discretization error convergence of different methods

A previous comparison was made to the residual convergence. It's important to note that, unlike the total error, the residual does not reflect the discretization error. This is because both the right-hand side f and the term $Au_h^{(k)}$ in the residual expression $\|f - Au_h^{(k)}\|$ are derived from the same discretized system. As a result, the residual can converge to zero even though the true solution u has not been reached—since the system itself is an approximation of the continuous problem.

For this reason, the residual is commonly used as a convergence criterion when implementing iterative methods: once the residual is sufficiently small, the algorithm is considered to have converged. In contrast, the total error $\|u - u_h^{(k)}\|$ always includes the discretisation error, because it compares the numerical solution to the true continuous solution u. As such, it would not prove useful to use this total error as a stopping criterion in numerical solvers. Anyhow, the total error cannot be used as a stopping criteria, as the true solution u is unknown, hence making it impossible to use as a stopping criteria.

Iteration Error The remaining error, the iteration error $e^{(k)} = u_h - u_h^{(k)}$, which measures the difference between the exact solution u_h of the discretized system $Au_h = f$ and the current approximation $u_h^{(k)}$ after k iterations of an iterative method, can be analysed in terms of its frequency components. Frequency components can be seen as the amount each error oscillates and can be categorized into high-frequency errors (or oscillatory components), which oscillate rapidly between neighbouring grid points and low-frequency errors (or smooth components), which vary smoothly and change gradually across the domain. An in-depth characterisation will be presented in section 3.1.

3 Preliminary Components

In order to introduce the Multigrid method, some preliminary components of the method will be displayed as motivation for its success.

3.1 Exploitation of the High-Frequency Error

The frequency-based understanding of the iteration error is essential for appreciating the effectiveness of the Multigrid method, which explicitly targets the slow converging low-frequency components by transferring them to coarser grids, where they appear as higher frequencies. However, one must first analyse what makes a low-frequency error slow to converge.

Spectral Properties of Iterative Solvers Consider a stationary linear iterative method of the form

$$x_h^{(k+1)} = Bx_h^{(k)} + g,$$

where B is the iteration matrix, eg. in our Jacobi implementation $B = D^{-1}(L+U)$ and $g = D^{-1}f$ with $Au_h = f \Leftrightarrow (D+L+U)u_h = f$ being the decomposition of the matrix. In Chapter 6 of [3], one is provided with more details on the decomposition of the matrix for different stationary iterative solvers. The convergence behaviour of the method is governed by the spectral radius $\rho(B)$, defined as

$$\rho(B) = \max\{|\lambda_1|, |\lambda_2|, \ldots, |\lambda_{N^2}|\}$$

The method converges for any initial guess if and only if

$$\lim_{k\to\infty} B^k = 0 \quad \Longleftrightarrow \quad \rho(B) < 1.$$

Accordingly, the iteration error $e_h^{(k)} = x_h - x_h^{(k)}$ satisfies

$$\lim_{k\to\infty} e_h^{(k)} = 0 \quad \text{if and only if} \quad \rho(B) < 1.$$

Mode-Wise Error Damping Let $x_h^{(0)}$ be an initial guess, with the corresponding error given by $e_h^{(0)} = x_h - x_h^{(0)}$. Assume that the initial error can be expressed as a linear combination of the eigenvectors w_j of the system matrix B:

$$e_h^{(0)} = \sum_{j=1}^{n} c_j w_j.$$

After k iterations of the method, the error becomes

$$e_h^{(k)} = B^k e_h^{(0)} = \sum_{j=1}^{n} c_j \lambda_j^k(B) w_j, \tag{5}$$

where $\lambda_j(B)$ are the eigenvalues of the iteration matrix B, and w_j represent the frequency components of the error.

A mode of oscillation refers to a specific pattern or shape that the error can take, which repeats regularly across the domain. These patterns are mathematically described by the eigenvectors and eigenvalues of the iteration matrix A.

Each eigenmode decays at a rate proportional to $\lambda_j^k(B)$. We call w_j from 5 the jth frequency component of the error e_h^k and respectively the eigenvalues $\lambda_j^k(B_w)$ determine how fast each frequency component goes to zero.

Typically, for smoothers like the Jacobi method, the eigenvalues associated with high-frequency modes are significantly smaller than those of low-frequency modes. This means that high-frequency error components are damped rapidly, while low-frequency components persist. Thus, the iteration error after several steps is dominated by smooth modes that converge very slowly.

Numerical Example Figure 3 illustrates this behaviour, showing that high-frequency components of the error decay quickly, while smooth, low-frequency components are reduced at a much slower rate.

In Figure 6, we analyse the change in error over 50 iterations:

$$\Delta e_h = e_h^{(50)} - e_h^{(0)}.$$

The plot reveals that high-frequency components are significantly diminished, while low-frequency components remain nearly unchanged. This observation reinforces the motivation for multigrid methods, which address these persistent low-frequency errors by transferring them to coarser grids, where they appear as high-frequency modes and can be efficiently smoothed. A more detailed explanation of this process is provided in Section 4.1.

3.2 Error Frequency Change and Grid Size

One can best describe the change of the frequency using the simplified graph 7. The graph depicts a finer grid $\mathcal{T}_h$ with $\frac{1}{h} = 12$ and the coarser grid $\mathcal{T}_{2h}$ with $\frac{1}{h} = 6$ with a simple wave that remains mathematically the same on both grids.

When one doubles the grid spacing, the sample points fall farther apart along the curve. One can only capture fewer peaks and troughs, making the sampled wave appear more oscillatory i.e smooth modes on a fine grid become more oscillatory when viewed on a coarser grid. This is because the reduced spatial resolution of the coarse grid limits its ability to capture smooth variations.

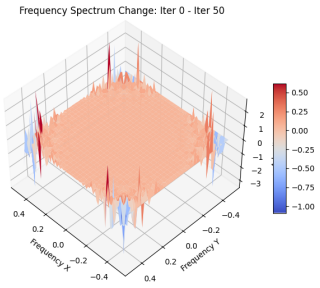

Figure 6: Difference in frequency components of the error after 50 Jacobi iterations: $\Delta e_h = e_h^{(50)} - e_h^{(0)}$.

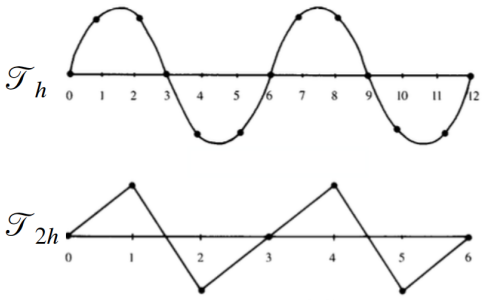

Figure 7: Oscillatory Change

3.3 Time Convergence Rate and Grid Size

In addition to the change of frequency in the error when one changes grid size, there is another benefit to the traversal of the grids.

The convergence rate of the Jacobi method is governed by the spectral radius $\rho(B)$, and the error satisfies the inequality:

$$\|e_h^{(k)}\| \leq \rho(B)^k \|e_h^{(0)}\|.$$

To reduce the error below a tolerance ε, the required number of iterations k must satisfy:

$$k \geq \frac{\log(\varepsilon / \|e_h^{(0)}\|)}{\log(\rho(B))}.$$

For structured problems, such as the 1D or 2D Poisson equation, the spectral radius scales with grid spacing h as

$$\rho(B) \approx 1 - Ch^2,$$

where C is a problem-dependent constant. This implies that the number of iterations required for convergence grows with grid refinement:

$$k \sim \mathcal{O}(h^{-2}) \sim \mathcal{O}(n^{2/d}),$$

where n is the number of unknowns and d is the number of spatial dimensions, hence solving the linear system on a coarser grid can benefit the time convergence rate.
This constitutes the preliminaries necessary to motive the Multigrid method.

4 The Multigrid Method

The following chapter introduces the mathematical implementation of operator with which one traverses grids and the Multigrid algorithm including different, commonly used cycles that exist for the method.

4.1 Traversing Grids

A key component of the multigrid method is the transfer between grids of different resolutions or step sizes. In particular, moving from a fine grid to a coarser one allows the algorithm to address low-frequency error components more effectively. This process is known as *restriction*. This enables the solver to operate on a reduced system where low-frequency components of the error (which converge slowly on fine grids) become high frequency components and hence easier to eliminate.

Restriction and Prolongation Operation Let $\mathcal{G}_h$ and $\mathcal{G}_H$ with $H = 2h$ denote the fine and coarse grids, respectively. The restriction operator $R_h^H : \mathcal{G}_h \to \mathcal{G}_H$ maps a function defined on the fine grid onto the coarse grid.

Each value on the coarse grid is computed as a weighted average of the corresponding 3×3 block on the fine grid. Given a fine-grid residual r_h defined on a grid with spacing h, and a coarse grid with spacing H, the restriction operator computes the coarse-grid residual as:

$$r_H(i,j) = \frac{1}{\sum w_{kl}} \sum_{k=-1}^{1} \sum_{l=-1}^{1} w_{kl} \cdot r_h(2i+k, 2j+l), \quad \text{with } i,j \in \{0,\ldots,(N/2)^2\}$$

where w_{kl} are the stencil weights, chosen as:

$$\begin{bmatrix} 0.25 & 0.5 & 0.25 \\ 0.5 & 1.0 & 0.5 \\ 0.25 & 0.5 & 0.25 \end{bmatrix}$$

The correction computed on the coarse grid is then interpolated back to the fine grid using a complementary operator known as *prolongation*, and added to the fine-grid solution. Each coarse-grid value contributes to a 3×3 block in the fine-grid correction, with weights assigned using the same weight stencil as the restriction operator, resulting in the following attribution:

$$r_h(x,y) = \sum_{k=-1}^{1} \sum_{l=-1}^{1} w_{kl} \cdot r_H(i,j), \quad \text{with } x = 2i+k,\ y = 2j+l, \quad \forall\, i,j \in \{1,\ldots,N^2\}$$

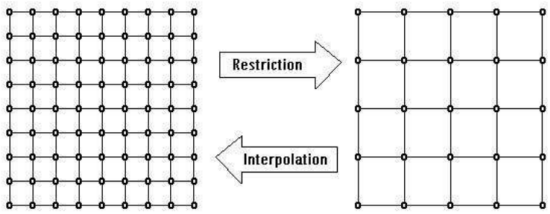

Figure 8: Diagrammatic Explanation of Restriction and Interpolation

4.2 The Multigrid Algorithm

The Multigrid (MG) algorithm is an iterative scheme of the form

$$x^{(k+1)} = MG(x^{(k)}),$$

designed to accelerate the convergence of basic iterative methods. Its core idea is to enhance the approximation of the fine-grid solution by incorporating corrections computed on coarser grids. By transferring the problem to a coarser level, where smooth (low-frequency) error components become easier to eliminate, the multigrid method achieves rapid convergence across all error frequencies, a concept that will be developed in section.

To understand the full system of the Multigrid cycle we look at the pseudo code to understand the structure. We consider a mesh defined by $h = \frac{1}{2^i}$, $i \geq 2$, a matrix $\mathbf{A_h}$, a right-hand-side $\mathbf{f}_h$, and an initial guess $\mathbf{u}_{h,0}$.

The parameters $\nu_1 \geq 0$ and $\nu_2 \geq 0$ give the number of times the smoothing operators are applied, and $\gamma > 0$ the number of recursive calls to perform (for $\gamma = 1$ this is a V-cycle and for $\gamma = 2$ this is a W-cycle which will be discussed in section 4.3).

4.3 Cycle Types in the Multigrid Method

The multigrid method offers flexibility in how it traverses the hierarchy of discretization levels. This traversal is governed by the parameter γ, which controls the number of recursive calls made to coarser levels during each multigrid cycle.

- For $\gamma = 1$, the method performs a **V-cycle**, where each coarser level is visited once per cycle.
- For $\gamma = 2$, the method performs a **W-cycle**, visiting each coarser level twice. This results in increased computational work per cycle but generally leads to improved convergence.

Algorithm 1 Multigrid

procedure MULTIGRID_CYCLE($\mathbf{A}_l, \mathbf{u}_l, \mathbf{f}_l, l, \nu_1, \nu_2, \gamma, h$)
  if $l == N_{\text{coarsest}}$ **then**
    $\mathbf{u}_l \leftarrow$ Direct_Solver($\mathbf{A}_l, \mathbf{f}_l$)
  else
    $\mathbf{u}_l \leftarrow$ Iterative_Smoother($\mathbf{A}_l, \mathbf{u}_l, \mathbf{f}_l, \nu_1$)
    $\mathbf{u}_{l+1} \leftarrow 0$
    $\mathbf{f}_{l+1} \leftarrow$ RESTRICT($\mathbf{f}_l$) {Grid spacing becomes $2h$}
    for $i = 0$ to $\gamma - 1$ **do**
      MULTIGRID_CYCLE($\mathbf{A}_{l+1}, \mathbf{u}_{l+1}, \mathbf{f}_{l+1}, l+1, \nu_1, \nu_2, \gamma, 2h$)
    end for
    $\mathbf{u}_l \leftarrow$ PROLONGATE($\mathbf{u}_{l+1}$) {Grid spacing becomes $h/2$}
    $\mathbf{u}_l \leftarrow$ Iterative_Smoother($\mathbf{A}_l, \mathbf{u}_l, \mathbf{f}_l, \nu_2$)
  end if
end procedure

The parameter γ can also be generalized to other values to balance computational cost and convergence efficiency.

Figure 9 illustrates the recursive structure of the V-cycle ($\gamma = 1$), W-cycle ($\gamma = 2$), and the Full Multigrid (FMG) cycle. The FMG approach combines accurate coarse-grid solutions with multigrid iterations to efficiently compute a high-quality solution on the finest grid.

The process of the FMG cycle is detailed in Algorithm 2.

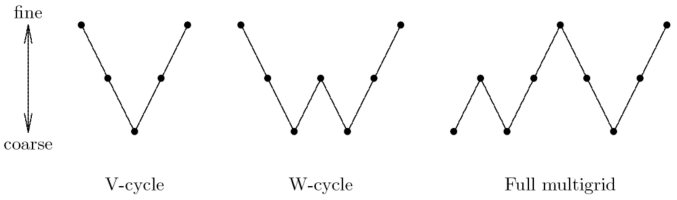

Figure 9: V-cycle ($\gamma = 1$), W-cycle ($\gamma = 2$), and Full Multigrid cycle

4.4 The Full Multigrid method

Algorithm 2 Full Multigrid Cycle

Input: p, ν_1, ν_2, γ, μ
Let $h \leftarrow h_0 = \frac{1}{2}$ and solve exactly: $\mathbf{A}_h \mathbf{u}_h = \mathbf{f}_h$
for $l \leftarrow 1$ to p **do**
  $\mathbf{u}_{h/2} \leftarrow \mathcal{I}_h^{h/2} \mathbf{u}_h$ {Interpolate to finer grid}
  $h \leftarrow \frac{h}{2}$
  $\mathbf{u}_h \leftarrow \text{MG}^{\mu}(\mathbf{A}_h, \mathbf{u}_h, \mathbf{f}_h, \nu_1, \nu_2, \gamma)$
end for
return $\mathbf{u}_h$

The parameter p gives the highest level of refinement on which one wishes to obtain the solution and the parameter μ is the number of Multigrid cycles to perform at each refinement level.

The Full Multigrid method (FMG) functions more like a direct solver than a purely iterative one. Unlike the basic multigrid cycle, which begins with an initial guess on the finest grid and iteratively reduces the error, the FMG starts with an exact solution on the coarsest grid. From there, the solution is interpolated to progressively finer grids, and at each level, it is refined using multigrid corrections. This process continues until the desired grid resolution and solution accuracy are reached. As illustrated in Figure 9, the solution "builds up" level by level, leveraging coarse-grid accuracy to accelerate convergence.

As such FMG takes its benefit versus the basic Multigrid cycle by avoiding the need to iteratively propagate corrections from a poor initial guess on the finest grid, instead building a high-quality solution from the ground up.

5 Multigrid Cycle Results

In order to demonstrate what has so far been mathematically portrayed, we analyse the following graphs that are a result of the implementation of different variations of the Multigrid method.

Error Dampening: Jacobi Method vs. Multigrid Method Figures 10 and 11 show the total error in frequency component for the Poisson equation under both the Jacobi and Multigrid methods. At first glance, the Jacobi method appears to produce little visible error reduction, although dampening is indeed occurring. This becomes clearer in Figure 6, which highlights the change in frequency components of the error between iteration 0 and iteration 50, demonstrating the method's slow but consistent dampening of high-frequency errors. In contrast, the Multigrid method shows significantly faster and more comprehensive error reduction, with the total error shrinking markedly in just a few V-cycles.

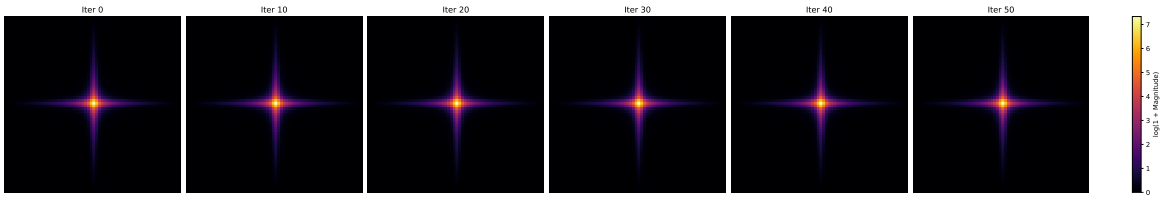

Figure 10: Error convergence using the Jacobi method

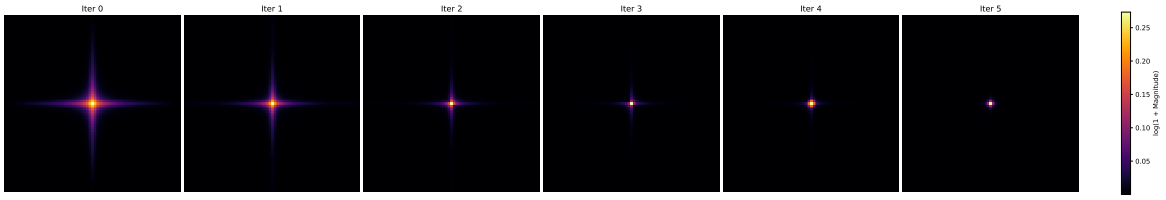

Figure 11: Error convergence using the Multigrid method

Error Convergence: Different Methods The Figure 12 emphasizes how effective each multigrid strategy is after just one cycle, reinforcing the idea that not all multigrid schemes are equally efficient per pass. The Full Multigrid (F-cycle) outperforms both the V- and W-cycles in reducing error even though the W-cycles lays not far behind.

One can related back to the error Figure 3 of the stationary methods, where the smoother (eg Jacobi) took 6000 iterations to achieve a similar error norm as any of the Multigrid methods.

Error Convergence: W-cycle The Figure 13 shows the error convergence behaviour of the W-cycle method. Notably, the W-cycle improves its convergence rate as the number of cycles increases, as expected. Remarkably, after just three cycles, the W-cycle reaches the optimal error decay rate of $\mathcal{O}(N^{-2})$. In comparison, the Conjugate Gradient (CG) method eventually achieves the same error rate but requires significantly more iterations to do so. A comment to add is that the parameters are $\nu_1 = \nu_2 = 3$.

Computation Time of Multigrid Methods Figure 14 presents the computation time of the V-cycle, W-cycle, and F-cycle multigrid methods as a function of problem size N, with N^2 unknowns. All three methods show a time scaling behaviour that follows $\mathcal{O}(N^2)$.

Among the methods, the F-cycle is marginally more expensive than the V- and W-cycles, though all remain close in overall runtime.

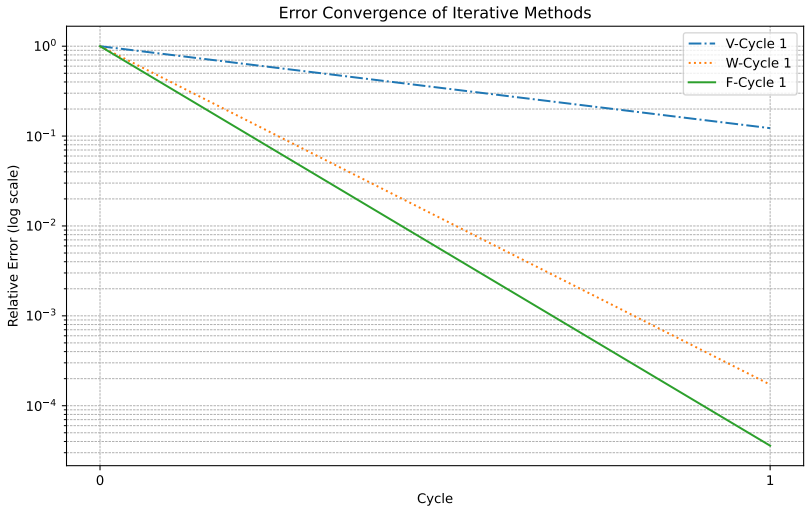

Figure 12: V-, W- and F- cycle MG method after 1 cycle

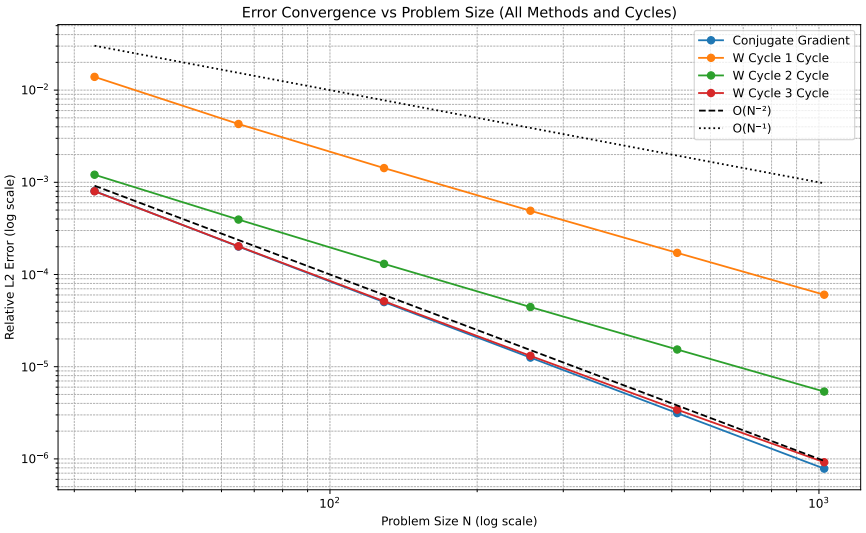

Figure 13: Error Norm Convergence of W-cycle method

6 Parallel Implementation

An advantages of the Multigrid (MG) method is its components are suitability for parallel implementation. In this section, we examine how each of them can be transformed from a sequential to a parallel version, along with the resulting speed-up, which leads to an overall speed-up of the MG algorithm.

A note to be made for the parallel implementation is the computer limitations on which the parallelisation was implemented, see section 8.

6.1 Parallel Components

6.1.1 Parallel Jacobi

The Jacobi method solves $Ax = b$ iteratively with the update:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{\substack{j=1 \\ j \neq i}}^{n} a_{ij} x_j^{(k)} \right), \quad i = 1, \ldots, n.$$

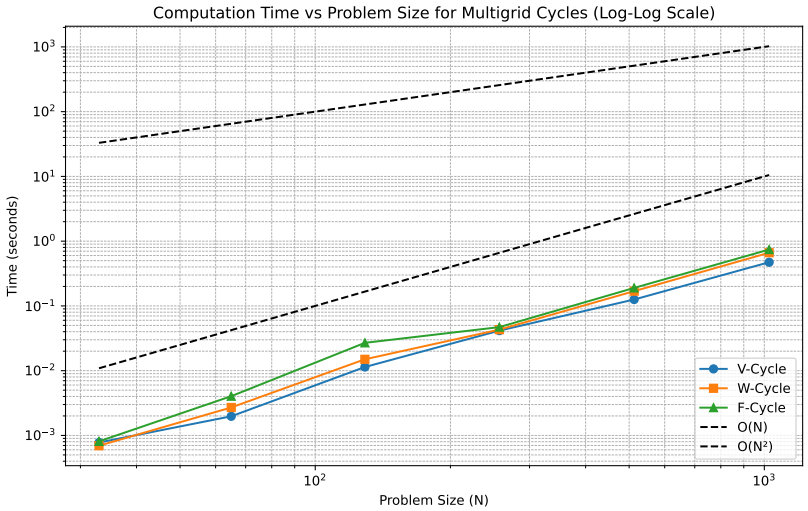

Figure 14: Time Convergence of Different MG methods

Parallelism is achieved by computing each $x_i^{(k+1)}$ independently using a temporary vector x_{tmp}, based only on values from $x^{(k)}$. Each thread updates a distinct portion of x_tmp, avoiding write conflicts and enabling efficient, lock-free execution.

6.1.2 Parallel Residual Computation

The formula for the residual is:

$$r_i = f_i - \frac{1}{h^2}\left(4x_i - x_{i-1} - x_{i+1} - x_{i-w} - x_{i+w}\right),$$

where h is the grid spacing and w is the grid width.

The parallel approach is the same as in the Jacobi method: each r_i is computed independently using values from the shared vector x, allowing efficient parallelization without synchronization.

6.1.3 Parallel Restriction

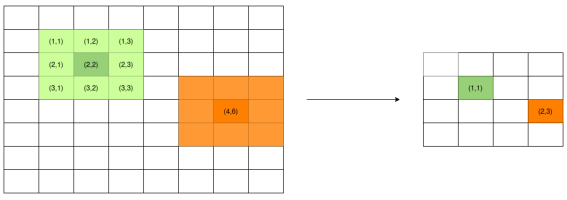

Figure 15: Parallel restriction strategy: each coarse cell is computed from a 3×3 stencil on the fine grid.

In the parallel restriction algorithm, we launch one thread per cell in the coarse grid. Each thread is identified by `thread_X` and `thread_Y`, and is responsible for computing a single output value.

Given a thread located at (`thread_X`, `thread_Y`) on the coarse grid, the corresponding center on the fine grid is:

$$F_x = 2 \cdot \texttt{thread\_X}, \quad F_y = 2 \cdot \texttt{thread\_Y}$$

Each thread then performs a weighted average over a 3×3 neighborhood in the fine grid using full-weighting. The computation is summarized below:

Algorithm 3 Restriction per thread

```
sum ← 0
for dx, dy ∈ {−1, 0, 1} do
  i ← F_x + dx, j ← F_y + dy
  if (i, j) in bounds then
    sum += weight[dy + 1][dx + 1] · input[j][i]
  end if
end for
output[thread_Y][thread_X] ← sum / total weight =0
```

Boundary threads set their output to zero and exit early. This approach ensures full parallelization with no memory conflicts, as each thread reads from the shared fine grid and writes to a unique location in the memory.

6.1.4 Parallel Prolongator

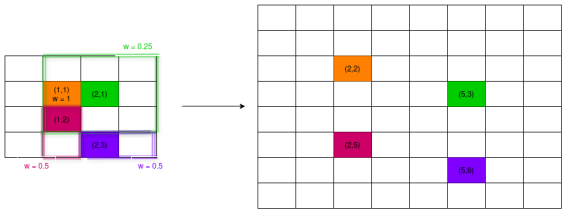

Figure 16: Prolongation: each fine cell is interpolated from the coarse grid based on its coordinates.

In the parallel prolungator algorithm, one thread is launched per cell in the fine grid. Each thread is identified by `thread_X` and `thread_Y`, and is responsible for computing one fine grid value.

Depending on the parity of (`thread_X`, `thread_Y`), the interpolation uses different surrounding coarse grid values:

- If `thread_X` mod 2 = 0 and `thread_Y` mod 2 = 0: direct injection from coarse grid.
- If one coordinate is odd and the other even: linear interpolation from two coarse neighbors.
- If both coordinates are odd: bilinear interpolation from four coarse neighbors.

Algorithm 4 Prolongation per thread

```
if on boundary then
  output ← 0
else if x even and y even then
  output ← coarse[y/2, x/2]
else if x odd and y odd then
  output ← average of 4 coarse neighbors
else if x even and y odd then
  output ← average of top/bottom coarse values
else if x odd and y even then
  output ← average of left/right coarse values
end if=0
```

Each thread reads from the coarse grid and writes to a unique fine grid location, enabling efficient parallel execution without memory conflicts.

6.2 Parallel vs Sequential method timings

The parallel performance improves significantly with larger grids. GPU performance outpaces CPU for all operations at large N. Smaller grids show less benefit in the parallel implementation with respect to larger grids due to the memory overhead.

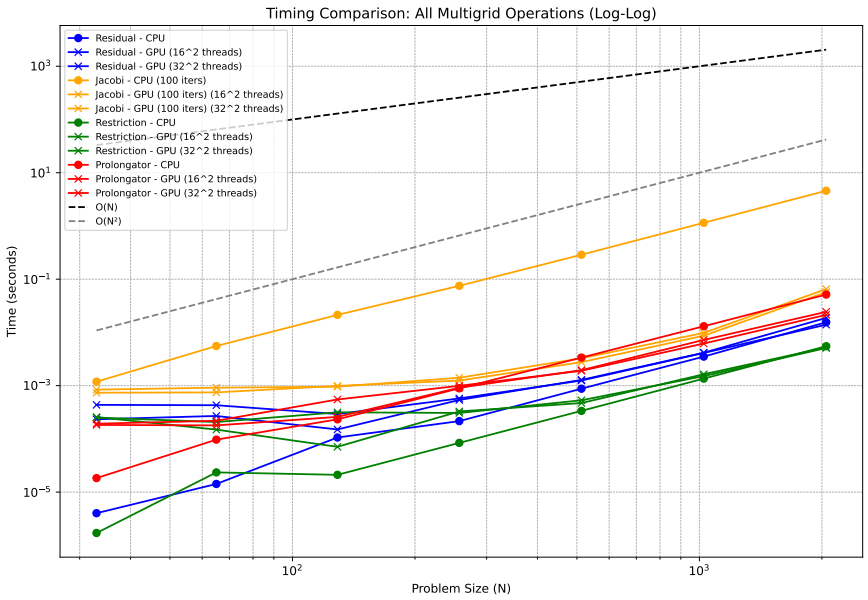

Figure 17: Log-log plot comparing execution times of parallel (GPU) and sequential (CPU) implementations for residual, Jacobi, restriction, and prolongation operations across varying problem sizes.

Figure 17 presents the different components of that MG algorithm that are susceptible to parallelisation. Noticeable, is that due to the smaller grid size the speed-up is not great. However, what stands out is the Jacobi method, being the bottle neck of the MG algorithm due to its time consuming computation, does have significant speed-up, making the computational time comparable to the other components.

6.3 Parallel MG

In the parallel implementation of the Multigrid method, a V-cycle strategy is employed, due to its simple implementation. When the grid size is reduced to $N \leq 16$ through the restriction operation, the computation is transferred from the GPU to the CPU. This switch is motivated by the fact that, at such small problem sizes, the overhead of GPU execution (including memory transfer and kernel launch latency) outweighs its computational benefits.

As shown in Figure 18, this hybrid approach achieves a maximum speed-up of approximately $\times 35$. Moreover, the use of GPU acceleration enables the solution of larger problem sizes with greater precision, overcoming the limitations imposed by CPU-only computations.

As discussed in Figure 13, achieving the desired convergence rate of $\mathcal{O}(h^2)$ typically requires three W-cycles. The corresponding performance results for this case are presented in Figure 19, where a speed-up of up to $\times 40$ is observed.

In Figure 20 one can see the solution attained by the MG algorithm with the given parameters for p and q. Visible are the sin curves.

7 Conclusion

In this work, we evaluated several numerical methods for solving the Poisson equation, including stationary iterative methods, multigrid approaches, and parallel implementations on both the GPU and CPU architectures.

Among the methods studied, stationary iterative solvers such as Jacobi were found to be the slowest, both in terms of convergence and runtime. In contrast, the multigrid method demonstrated substantial improvements, achieving significant speed-ups even in the sequential setting. Notably, we observed that as few as three W-cycles were sufficient to reach optimal convergence, and that a single iteration of the full multigrid (FMG) cycle yielded the fastest convergence of all sequential methods tested.
Parallelisation further amplified these gains. GPU-based implementations in particular achieved speed-ups of up to $40\times$ for the W-cycle and $35\times$ for the V-cycle compared to their sequential counterparts by applying targeted parallel strategies to key multigrid components, such as restriction, prolongation, and smoothing.

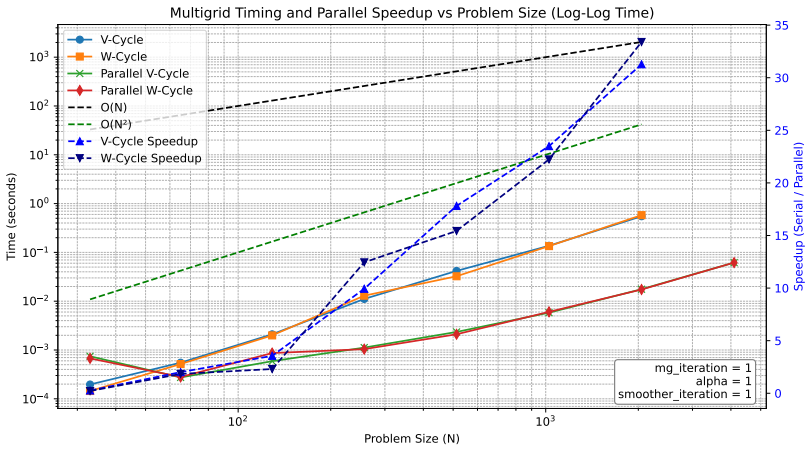

Figure 18: Time and Speed up of the Multigrid Method

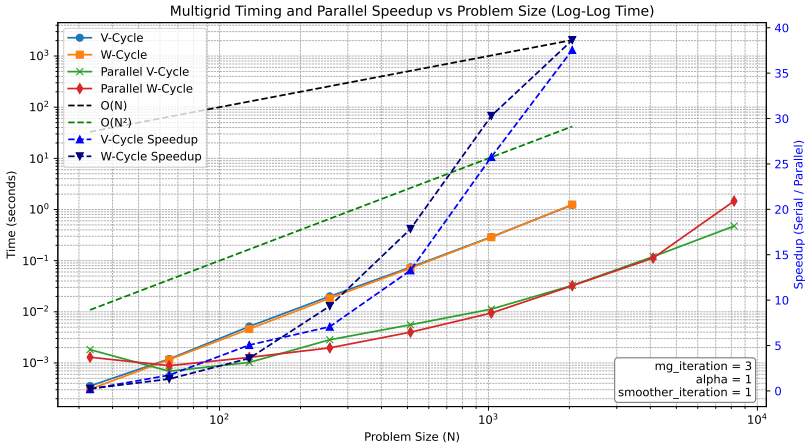

Figure 19: Time and Speed up of the Multigrid Method

Further research could explore optimizing the parallelization strategy itself to push performance.

Figure 20: Approximate solutions for two different parameter configurations.

8 System Configuration

The code was mainly written in C++, with the additional Python code used for plotting purposes.

All experiments were performed on a high-performance laptop system with the following specifications:

- **CPU:** Intel Core i9-14900HX, 24 physical cores, 32 threads, x86_64 architecture (Little Endian)
  - Max frequency: 5.8 GHz
  - Min frequency: 800 MHz
  - Frequency scaling at test time: 19%
- **GPU:** NVIDIA GeForce RTX 4070 Laptop GPU
  - Memory: 8 GB (8188 MiB)
  - CUDA Version: 12.9
  - Driver Version: 575.51.03
  - Power Envelope: 45 W
  - GPU Utilization at time of check: 2%
- **CPU Architecture:**
  - CPU op-mode(s): 32-bit and 64-bit
  - Address sizes: 39-bit physical, 48-bit virtual
  - Vendor ID: GenuineIntel
  - Byte Order: Little Endian

No other GPU-intensive processes were running during the benchmarks to ensure consistent performance measurements.

The full source code and CMake configuration for building and running the project can be found at:

`github.com/FraVirgu/Parallel-Geometric-Multigrid-for-Poisson-problem_final`

References

[1] Paola Antonietti. Multigrid methods. In *Lecture: Numerical Linear Algebra*, 2024. Politecnico di Milano lecture notes.

[2] John S. Butler. Finite difference methods for the poisson equation. `https://john-s-butler-dit.github.io/NumericalAnalysisBook/Chapter%2009%20-%20Elliptic%20Equations/903_Poisson%20Equation-Boundary.html`, 2021. Online article.

[3] James W. Demmel. *Applied Numerical Linear Algebra*. SIAM, Philadelphia, PA, 1997.

[4] Michael T. Heath. Lecture notes on iterative methods. `https://www.cs.princeton.edu/courses/archive/fall12/cos323/notes/cos323_f12_lecture16_multigrid.pdf`, 2012. Princeton University lecture notes.

[5] John Millar and John C. Bancroft. Multigrid principles. *CREWES Research Report*, 15, 2003.

[6] Gilbert Strang. Lecture notes on multigrid methods. `https://math.mit.edu/classes/18.086/2006/am63.pdf`, 2006. Massachusetts Institute of Technology.