Politecnico di Milano
School of Industrial and Information Engineering
M.Sc. in High Performance Computing

Hybrid thread-MPI parallelization for ADR equation

Peng Rao, Jiali Claudio Huang, Ruiying Jiao

Repo: https://github.com/Peng-Rao/HybridADRSolver

Contents

1 Problem statement **1**

1.1 Strong formulation . . . 1

1.2 Weak formulation . . . 1

1.3 Manufactured solution . . . 3

2 Finite Element Discretization **5**

2.1 Triangulation and Finite Element Space . . . 5

2.2 Galerkin Approximation . . . 5

2.3 Algebraic System . . . 5

2.4 Convergence and Error Estimation . . . 6

3 Implementation Details **8**

3.1 Matrix Assembly . . . 8

3.1.1 Matrix-Based Implementation . . . 8

3.1.2 Matrix-Free Implementation . . . 8

3.2 Multigrid Preconditioning . . . 10

3.2.1 Algebraic Multigrid (AMG) for Matrix-Based Solver . . . 10

3.2.2 Geometric Multigrid (GMG) for Matrix-Free Solver . . . 10

3.3 Parallelization Strategy . . . 10

3.3.1 Distributed Memory Parallelism (MPI) . . . 11

3.3.2 Thread-Level Parallelism . . . 11

3.3.3 Linear Algebra Backend . . . 11

3.4 Data Flow . . . 11

4 Experiments **13**

4.1 Convergence Verification . . . 13

4.2 Time Complexity . . . 13

4.3 Memory Consumption . . . 15

4.4 Scalability . . . 16

5 Conclusion **17**

References **18**

1 Problem statement

1.1 Strong formulation

Consider the following **Advection-Diffusion-Reaction** equation with mixed Dirichlet-Neumann boundary conditions:

$$\begin{cases} -\nabla \cdot (\mu \nabla u) + \nabla \cdot (\boldsymbol{\beta} u) + \gamma u = f & \text{in } \Omega, \\ u = g & \text{on } \Gamma_D \subset \partial\Omega, \\ \nabla u \cdot \boldsymbol{n} = h & \text{on } \Gamma_N = \partial\Omega \setminus \Gamma_D. \end{cases}$$

where:

- $\Omega \subset \mathbb{R}^d$ (with $d = 1, 2, 3$) is an open bounded domain with boundary $\partial\Omega$;
- $\mu > 0$ is the diffusion coefficient;
- $\boldsymbol{\beta} \in [L^\infty(\Omega)]^d$ is the advection velocity field;
- $\gamma \geq 0$ is the reaction coefficient;
- $f \in L^2(\Omega)$ is a source term;
- $g \in H^{1/2}(\Gamma_D)$ is the Dirichlet boundary data;
- $h \in L^2(\Gamma_N)$ is the Neumann boundary data;
- $\boldsymbol{n}$ is the outward unit normal vector on the boundary $\partial\Omega$.
- u is the unknown scalar function to be solved for.
- Γ_D and Γ_N are the Dirichlet and Neumann parts of the boundary, respectively.

1.2 Weak formulation

We begin by defining the trial and test function spaces. To accommodate the non-homogeneous Dirichlet boundary condition, we introduce a *lifting function* $u_g \in H^1(\Omega)$ such that $u_g = g$ on Γ_D:

$$V_g := \{v \in H^1(\Omega) : v = g \quad \text{on } \Gamma_D\}$$

The test space is the linear subspace of $H^1(\Omega)$ with homogeneous Dirichlet boundary conditions:

$$V_0 := \{v \in H^1(\Omega) : v = 0 \quad \text{on } \Gamma_D\}$$

Consequently, we decompose the solution as $u = u_0 + u_g$, where the unknown $u_0 \in V_0$. Multiply the equation by a test function $v \in V_0$ and integrate over the domain Ω:

$$\int_\Omega (-\nabla \cdot (\mu \nabla u) + \nabla \cdot (\boldsymbol{\beta} u) + \gamma u)\, v \, d\Omega = \int_\Omega f v \, d\Omega$$

Using the linearity of the integral, we separate the terms:

$$-\int_{\Omega} \nabla \cdot (\mu \nabla u) v \, d\Omega + \int_{\Omega} \nabla \cdot (\boldsymbol{\beta} u) v \, d\Omega + \int_{\Omega} \gamma u v \, d\Omega = \int_{\Omega} f v \, d\Omega$$

We apply Green's first identity to the diffusion term to reduce the order of differentiation:

$$-\int_{\Omega} \nabla \cdot (\mu \nabla u) v \, d\Omega = \int_{\Omega} \mu \nabla u \cdot \nabla v \, d\Omega - \int_{\partial\Omega} (\mu \nabla u \cdot \mathbf{n}) v \, d\Gamma$$

Substituting this back into the integral equation:

$$\int_{\Omega} \mu \nabla u \cdot \nabla v \, d\Omega - \int_{\partial\Omega} (\mu \nabla u \cdot \mathbf{n}) v \, d\Gamma + \int_{\Omega} \nabla \cdot (\boldsymbol{\beta} u) v \, d\Omega + \int_{\Omega} \gamma u v \, d\Omega = \int_{\Omega} f v \, d\Omega$$

Split the boundary integral into contributions from Γ_D and Γ_N:

$$\int_{\partial\Omega} (\mu \nabla u \cdot \mathbf{n}) v \, d\Gamma = \int_{\Gamma_D} (\mu \nabla u \cdot \mathbf{n}) v \, d\Gamma + \int_{\Gamma_N} (\mu \nabla u \cdot \mathbf{n}) v \, d\Gamma$$

Since $v = 0$ on Γ_D, the first term vanishes. On Γ_N, we use the Neumann condition $\nabla u \cdot \mathbf{n} = h$:

$$\int_{\Gamma_N} (\mu \nabla u \cdot \mathbf{n}) v \, d\Gamma = \int_{\Gamma_N} \mu h v \, d\Gamma$$

Substituting back, we have:

$$\int_{\Omega} \mu \nabla u \cdot \nabla v \, d\Omega + \int_{\Omega} \nabla \cdot (\boldsymbol{\beta} u) v \, d\Omega + \int_{\Omega} \gamma u v \, d\Omega = \int_{\Omega} f v \, d\Omega + \int_{\Gamma_N} \mu h v \, d\Gamma$$

Substituting $u = u_0 + u_g$ into the equation, we get:
Find $u_0 \in V_0$ such that

$$a(u_0, v) = F(v), \quad \forall v \in V_0$$

where the *bilinear form* $a : V_0 \times V_0 \to \mathbb{R}$ is defined as:

$$a(u, v) = \int_{\Omega} \mu \nabla u \cdot \nabla v \, d\Omega + \int_{\Omega} \nabla \cdot (\boldsymbol{\beta} u) v \, d\Omega + \int_{\Omega} \gamma u v \, d\Omega$$

and the *linear functional* $F : V_0 \to \mathbb{R}$ is given by:

$$F(v) = \int_{\Omega} f v \, d\Omega + \int_{\Gamma_N} \mu h v \, d\Gamma - a(u_g, v)$$

1.3 Manufactured solution

We define the exact solution $u_{\text{ex}} : \Omega \to \mathbb{R}$ on the unit hypercube domain $\Omega = [0,1]^d$ (where $d = 2, 3$) as the product of sine functions:

$$u_{\text{ex}}(\mathbf{x}) = \prod_{i=1}^{d} \sin(\pi x_i). \tag{1}$$

This function vanishes on the boundary hyperplanes where $x_i = 0$ or $x_i = 1$, making it naturally suitable for homogeneous Dirichlet boundary conditions.

The physical coefficients for the benchmark problem are chosen as follows:

- **Diffusion:** A constant isotropic diffusion coefficient $\mu = 1.0$.
- **Reaction:** A constant reaction coefficient $\gamma = 0.1$.
- **Advection:** A rotational velocity field $\boldsymbol{\beta}(\mathbf{x})$, defined to make the problem non-symmetric:

$$\boldsymbol{\beta}(\mathbf{x}) = \begin{cases} \begin{bmatrix} -x_2 \\ x_1 \end{bmatrix} & \text{if } d = 2, \\ \begin{bmatrix} -x_2 \\ x_1 \\ 0.1 \end{bmatrix} & \text{if } d = 3. \end{cases} \tag{2}$$

Substituting u_{ex} into the governing equation $-\nabla \cdot (\mu \nabla u) + \nabla \cdot (\boldsymbol{\beta} u) + \gamma u = f$, we compute the source term f.

First, we observe that the Laplacian of the chosen exact solution is:

$$\Delta u_{\text{ex}} = \sum_{i=1}^{d} \frac{\partial^2 u_{\text{ex}}}{\partial x_i^2} = \sum_{i=1}^{d} (-\pi^2 u_{\text{ex}}) = -d\pi^2 u_{\text{ex}}. \tag{3}$$

Assuming $\boldsymbol{\beta}$ is divergence-free ($\nabla \cdot \boldsymbol{\beta} = 0$, which holds for the rotational field defined above), the advection term simplifies to $\boldsymbol{\beta} \cdot \nabla u_{\text{ex}}$. The source term f is therefore implemented as:

$$f(\mathbf{x}) = \mu d\pi^2 u_{\text{ex}}(\mathbf{x}) + \boldsymbol{\beta}(\mathbf{x}) \cdot \nabla u_{\text{ex}}(\mathbf{x}) + \gamma u_{\text{ex}}(\mathbf{x}). \tag{4}$$

The problem domain boundary $\partial\Omega$ is split into Dirichlet (Γ_D) and Neumann (Γ_N) portions to test mixed boundary conditions. Figure 1 illustrates the boundary conditions on a 2D unit square domain.

Neumann Boundary (Γ_N) We apply a Neumann condition on the "Right" face of the hypercube, defined as the plane $x_1 = 1$. The outward unit normal is $\mathbf{n} = (1, 0, \dots)^T$. The required flux h is derived from the exact solution:

$$h(\mathbf{x}) = \nabla u_{\text{ex}} \cdot \mathbf{n} \Big|_{x_1=1} = \frac{\partial u_{\text{ex}}}{\partial x_1} \Big|_{x_1=1}. \tag{5}$$

Computing the partial derivative:

$$\frac{\partial u_{\text{ex}}}{\partial x_1} = \pi \cos(\pi x_1) \prod_{j=2}^{d} \sin(\pi x_j). \tag{6}$$

Evaluated at $x_1 = 1$, where $\cos(\pi) = -1$, the Neumann data imposed is:

$$h(\mathbf{x}) = -\pi \prod_{j=2}^{d} \sin(\pi x_j). \tag{7}$$

Dirichlet Boundary (Γ_D) On all other boundaries ($\partial\Omega \backslash \Gamma_N$), we enforce a homogeneous Dirichlet condition:

$$u = 0 \quad \text{on } \Gamma_D. \tag{8}$$

This is consistent with the exact solution, as $\sin(\pi x_i) = 0$ when $x_i \in \{0, 1\}$.

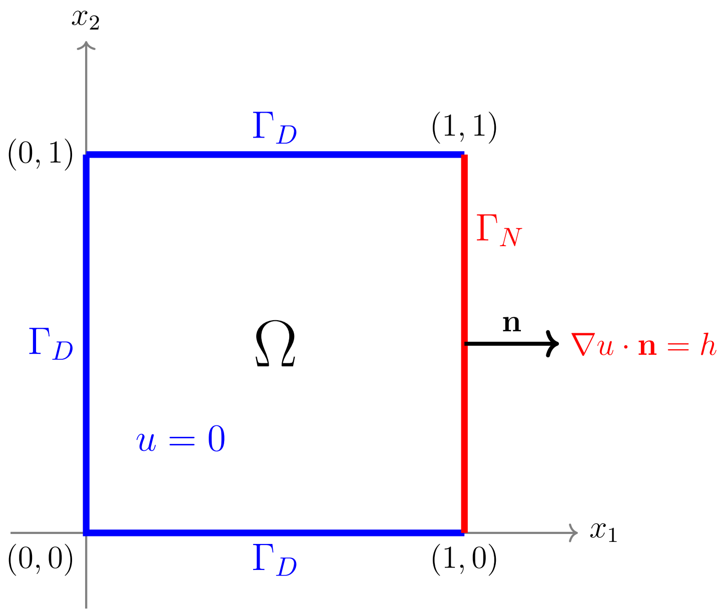

Figure 1: 2D Domain with Mixed Boundary Conditions

2 Finite Element Discretization

To solve the weak formulation numerically, we employ the Finite Element Method (FEM). This involves approximating the infinite-dimensional function spaces V_g and V_0 with finite-dimensional subspaces defined on a computational mesh.

2.1 Triangulation and Finite Element Space

We consider a triangulation $\mathcal{T}_h = \{K\}$ of the domain Ω, consisting of non-overlapping hexahedral (or quadrilateral in 2D) cells K such that $\overline{\Omega} = \bigcup_{K \in \mathcal{T}_h} \overline{K}$. The parameter h denotes the characteristic mesh size, $h = \max_{K \in \mathcal{T}_h} \operatorname{diam}(K)$.

We introduce the finite-dimensional space $V_h^k \subset H^1(\Omega)$ consisting of continuous piecewise polynomial functions of degree k. In the context of the `deal.II` library, we utilize Lagrangian finite elements (tensor product polynomials of degree k, denoted as Q_k). The discrete trial and test spaces are defined as:

$$V_{h,g} = \{u_h \in V_h^k : u_h|_{\Gamma_D} = I_h(g)\}, \tag{9}$$

$$V_{h,0} = \{v_h \in V_h^k : v_h|_{\Gamma_D} = 0\}, \tag{10}$$

where $I_h(g)$ is the nodal interpolation of the Dirichlet boundary data onto the mesh nodes on Γ_D.

2.2 Galerkin Approximation

The discrete problem is obtained by restricting the weak form to these subspaces. We seek $u_h \in V_{h,g}$ such that:

$$a(u_h, v_h) = L(v_h) \quad \forall v_h \in V_{h,0}. \tag{11}$$

We expand the approximate solution u_h in terms of the standard nodal basis functions $\{\varphi_j\}_{j=1}^{N_{dof}}$. Let u_h be decomposed into a part satisfying the homogeneous boundary conditions and a lifting of the Dirichlet data:

$$u_h(\mathbf{x}) = \sum_{j \in \mathcal{I}_{free}} U_j \varphi_j(\mathbf{x}) + \sum_{j \in \mathcal{I}_{dir}} g_j \varphi_j(\mathbf{x}), \tag{12}$$

where U_j are the unknown coefficients (degrees of freedom), $\mathcal{I}_{free}$ is the set of indices for nodes not on Γ_D, and $\mathcal{I}_{dir}$ contains indices for nodes on the Dirichlet boundary with known values g_j.

2.3 Algebraic System

Substituting the basis expansion into Eq. (11) and testing with each basis function φ_i (for $i \in \mathcal{I}_{free}$), we obtain the linear system of equations:

$$\mathbf{A}\mathbf{U} = \mathbf{F}, \tag{13}$$

where $\mathbf{U}$ is the vector of unknown coefficients. The entries of the global stiffness matrix $\mathbf{A}$ and the right-hand side vector $\mathbf{F}$ are computed by assembling contributions from each cell $K \in \mathcal{T}_h$.

The matrix entries A_{ij} correspond to the bilinear form $a(\varphi_j, \varphi_i)$:

$$A_{ij} = \int\limits_{\Omega} \left(\mu \nabla \varphi_j \cdot \nabla \varphi_i + (\boldsymbol{\beta} \cdot \nabla \varphi_j)\varphi_i + \gamma \varphi_j \varphi_i\right) \, dx; \tag{14}$$

Using numerical quadrature, the integral over Ω is computed as the sum of integrals over cells K. For a specific cell K, the local matrix contributions are:

$$A_{ij}^K = \sum_{q=1}^{N_q} \left(\mu \nabla \varphi_j(\mathbf{x}_q) \cdot \nabla \varphi_i(\mathbf{x}_q) + (\boldsymbol{\beta}(\mathbf{x}_q) \cdot \nabla \varphi_j(\mathbf{x}_q))\varphi_i(\mathbf{x}_q) + \gamma \varphi_j(\mathbf{x}_q)\varphi_i(\mathbf{x}_q)\right) w_q |J_K(\mathbf{x}_q)|, \tag{15}$$

where $\{\mathbf{x}_q\}$ and $\{w_q\}$ are the quadrature points and weights defined on the reference element, mapped to physical space via the Jacobian determinant $|J_K|$.

The right-hand side vector $\mathbf{F}$ includes the source term, the Neumann boundary contributions, and the modifications due to the Dirichlet lifting:

$$F_i = \int\limits_{\Omega} f \varphi_i \, dx + \int\limits_{\Gamma_N} \mu h \varphi_i \, ds - \sum_{j \in \mathcal{I}_{dir}} g_j A_{ij}; \tag{16}$$

The Neumann term is only non-zero if the support of φ_i intersects with Γ_N.

2.4 Convergence and Error Estimation

The finite element solution u_h converges to the exact solution u as the mesh is refined ($h \to 0$). Under appropriate regularity assumptions on the solution and the domain, classical a priori error estimates provide bounds on the discretization error.

Error Norms We measure the approximation error in the standard Sobolev norms:

- **L^2-norm (energy norm for the solution):**

$$\|u - u_h\|_{L^2(\Omega)} = \left(\int\limits_{\Omega} |u - u_h|^2 \, dx \right)^{1/2}; \tag{17}$$

- **H^1-seminorm (energy norm for the gradient):**

$$|u - u_h|_{H^1(\Omega)} = \left(\int\limits_{\Omega} |\nabla(u - u_h)|^2 \, dx \right)^{1/2}; \tag{18}$$

A Priori Error Estimates Assuming the exact solution $u \in H^{k+1}(\Omega)$ and the bilinear form $a(\cdot, \cdot)$ is coercive and continuous, the following error estimates hold for Lagrangian finite elements of polynomial degree k:

- **H^1-error estimate:** By Céa's lemma and standard interpolation theory,

$$|u - u_h|_{H^1(\Omega)} \leq C h^k |u|_{H^{k+1}(\Omega)}, \tag{19}$$

where $C > 0$ is a constant independent of h.

- L^2**-error estimate:** Using the Aubin-Nitsche duality argument,

$$\|u - u_h\|_{L^2(\Omega)} \leq Ch^{k+1}|u|_{H^{k+1}(\Omega)}. \tag{20}$$

These estimates indicate that for smooth solutions, increasing the polynomial degree k or refining the mesh (decreasing h) improves accuracy. Specifically:

- The H^1-error converges at rate $\mathcal{O}(h^k)$.
- The L^2-error converges at rate $\mathcal{O}(h^{k+1})$.

3 Implementation Details

3.1 Matrix Assembly

The matrix assembly process involves iterating over each cell in the triangulation, computing local contributions to the global stiffness matrix and right-hand side vector using numerical quadrature. The implementation leverages the `deal.II`[1] finite element library for efficient handling of mesh data structures, basis functions, and quadrature rules.

3.1.1 Matrix-Based Implementation

In the matrix-based approach, we explicitly assemble the global stiffness matrix $\mathbf{A}$ by summing the local element matrices. For each cell, we compute the local stiffness matrix A_{cell} and use a scatter operation to add its contributions to the global matrix. The global matrix is stored in a sparse format to optimize memory usage. The global stiffness matrix is assembled as follows:

$$A = \sum_{\text{cell}=1}^{n_{\text{cells}}} P^T_{\text{cell, loc–glob}} \, A_{\text{cell}} \, P_{\text{cell, loc–glob}}$$

where P_{cell} is a rectangular Boolean matrix that defines the index mapping from local degrees of freedom in the current cell to the global degrees of freedom.

The assembly process uses the `WorkStream` framework from `deal.II` for thread-parallel assembly, which separates the computation into:

- **Worker phase:** Thread-local computation of local cell matrices (embarrassingly parallel).
- **Copier phase:** Sequential addition of local contributions to the global matrix (requires synchronization).

3.1.2 Matrix-Free Implementation

Matrix-free methods avoid forming the global system matrix explicitly. Instead, they compute the action of the matrix on a vector directly by looping over the elements and performing local operations. The process can be summarized as follows:

Element Loop in Iterative Solver The matrix-vector product $\mathbf{v} = \mathbf{Au}$ is computed as:

$$\mathbf{v} = \sum_{e=1}^{N_{\text{el}}} P_e^T A_e (P_e \mathbf{u}) \tag{21}$$

where N_{el} is the number of elements (cells), P_e is the local-to-global index mapping operator for element e, A_e is the local element operator, and $\mathbf{u}$ is the input vector.

Here, the cell loop occurs within each iteration of the iterative solver. The following steps are executed for each cell:

- **Gather:** Extract local vector values: $\mathbf{u}_e = P_e \mathbf{u}$
- **Evaluate:** Compute values and gradients at quadrature points

- **Integrate:** Apply the weak form locally: $\mathbf{v}_e = A_e \mathbf{u}_e$
- **Scatter:** Sum the local results into the global solution vector: $\mathbf{v} \mathrel{+}= P_e^T \mathbf{v}_e$

Evaluation at Quadrature Points The local matrix-vector product is computed by evaluating the bilinear form at quadrature points. For a given cell e and local degrees of freedom $\mathbf{u}_e$, we first compute the solution value and gradient at each quadrature point:

$$u_q = \sum_j u_j^e \varphi_j(\mathbf{x}_q), \tag{22}$$

$$\nabla u_q = \sum_j u_j^e \nabla \varphi_j(\mathbf{x}_q), \tag{23}$$

where u_j^e are the local coefficients and φ_j are the basis functions.

Then, the local contribution to the output vector is computed as:

$$(\mathbf{v}_e)_i = \sum_{q=1}^{N_q} \left[\mu \nabla \varphi_i(\mathbf{x}_q) \cdot \nabla u_q + (\boldsymbol{\beta}(\mathbf{x}_q) \cdot \nabla u_q)\varphi_i(\mathbf{x}_q) + \gamma u_q \varphi_i(\mathbf{x}_q)\right] w_q |J_e(\mathbf{x}_q)|. \tag{24}$$

This can be rewritten in a more compact form using the flux notation:

$$(\mathbf{v}_e)_i = \sum_{q=1}^{N_q} \left[\mathbf{F}_q \cdot \nabla \varphi_i(\mathbf{x}_q) + V_q\, \varphi_i(\mathbf{x}_q)\right] w_q |J_e(\mathbf{x}_q)|, \tag{25}$$

where the flux and value terms are:

$$\mathbf{F}_q = \mu \nabla u_q, \tag{26}$$

$$V_q = (\boldsymbol{\beta} \cdot \nabla u_q) + \gamma u_q. \tag{27}$$

This approach avoids explicitly forming and storing the local matrix A_e, instead computing the matrix-vector product directly through numerical integration.

Sum Factorization and Vectorization The implementation exploits two key optimizations:

- **Sum factorization:** For tensor-product elements (like Q_k), the evaluation of basis functions can be decomposed into 1D operations, reducing the complexity from $\mathcal{O}((k+1)^{2d})$ to $\mathcal{O}(d(k+1)^{d+1})$ per cell.
- **SIMD vectorization:** Multiple cells are processed simultaneously using vectorized instructions (e.g., AVX), with cell batches processed in parallel.

Memory and Computational Advantages The matrix-free approach offers several computational benefits:

- **Memory efficiency:** No need to store the global stiffness matrix $\mathbf{A}$. The sparse matrix storage scales as $\mathcal{O}(N_{\text{dof}} \cdot n_{\text{nz}})$, where n_{nz} is the average number of non-zeros per row (related to $(2k+1)^d$ for Q_k elements). Matrix-free methods only store vectors, scaling as $\mathcal{O}(N_{\text{dof}})$.

- **Cache efficiency:** Operations are performed locally on each cell, improving data locality and cache utilization.
- **High-order elements:** Particularly advantageous for high polynomial degrees where matrix sparsity decreases and storage costs increase.

The matrix-free operator is typically used in conjunction with iterative solvers such as Conjugate Gradient (CG) or Generalized Minimal Residual (GMRES), combined with geometric multigrid (GMG) preconditioning for optimal performance.

3.2 Multigrid Preconditioning

To accelerate the convergence of the iterative solvers, we employ multigrid preconditioning techniques.

3.2.1 Algebraic Multigrid (AMG) for Matrix-Based Solver

For the matrix-based implementation, we use Algebraic Multigrid (AMG) as a preconditioner via the PETSc library. AMG constructs a hierarchy of coarser problems based solely on the algebraic properties of the stiffness matrix, without requiring explicit knowledge of the mesh structure. This makes it particularly robust for problems with complex geometries or unstructured meshes.

3.2.2 Geometric Multigrid (GMG) for Matrix-Free Solver

For the matrix-free implementation, we utilize Geometric Multigrid (GMG) preconditioning. GMG leverages the geometric hierarchy of meshes obtained through uniform refinement. The key components include:

- **Smoothers:** We employ Chebyshev polynomial smoothers, which approximate the inverse of the operator using polynomial iterations. The smoother is configured with:
  - Polynomial degree: 5 iterations per smoothing step
  - Smoothing range: factor of 15–20 for eigenvalue estimation
- **Transfer Operators:** Restriction and prolongation operators transfer residuals and corrections between different levels of the mesh hierarchy. These are implemented matrix-free using the same sum factorization techniques.
- **Coarse Grid Solver:** On the coarsest level, we use additional Chebyshev iterations with a wider eigenvalue range to achieve a more accurate solve.

The Chebyshev smoother is particularly well-suited for matrix-free methods because it only requires matrix-vector products, avoiding the need to access individual matrix entries as would be required for Gauss-Seidel or ILU smoothers.

3.3 Parallelization Strategy

The implementation supports hybrid MPI + threading parallelism for optimal performance on modern HPC architectures.

3.3.1 Distributed Memory Parallelism (MPI)

The mesh is partitioned across multiple MPI processes using the `p4est` library, which provides scalable, adaptive mesh refinement with Morton space-filling curves. Each process owns a subset of cells and their associated degrees of freedom. Communication patterns are established for:

- Ghost cell data exchange
- Parallel matrix/vector assembly with MPI reduction
- Multigrid level transfers across process boundaries

3.3.2 Thread-Level Parallelism

Within each MPI process, thread-level parallelism is employed using Intel Threading Building Blocks (TBB):

- **Matrix-based solver:** Uses the `WorkStream` framework for parallel assembly, where different threads process different cells simultaneously.
- **Matrix-free solver:** Uses `partition_partition` task parallelism scheme, where both the cell loop and the vector operations are parallelized across threads.

3.3.3 Linear Algebra Backend

For the matrix-based implementation, we utilize PETSc for distributed linear algebra operations. PETSc provides:

- Distributed sparse matrix storage (AIJ format)
- Parallel iterative solvers (CG, GMRES, BiCGStab)
- Preconditioners including AMG (via hypre)

3.4 Data Flow

Fig. 2 illustrates the execution flow for both solver variants.

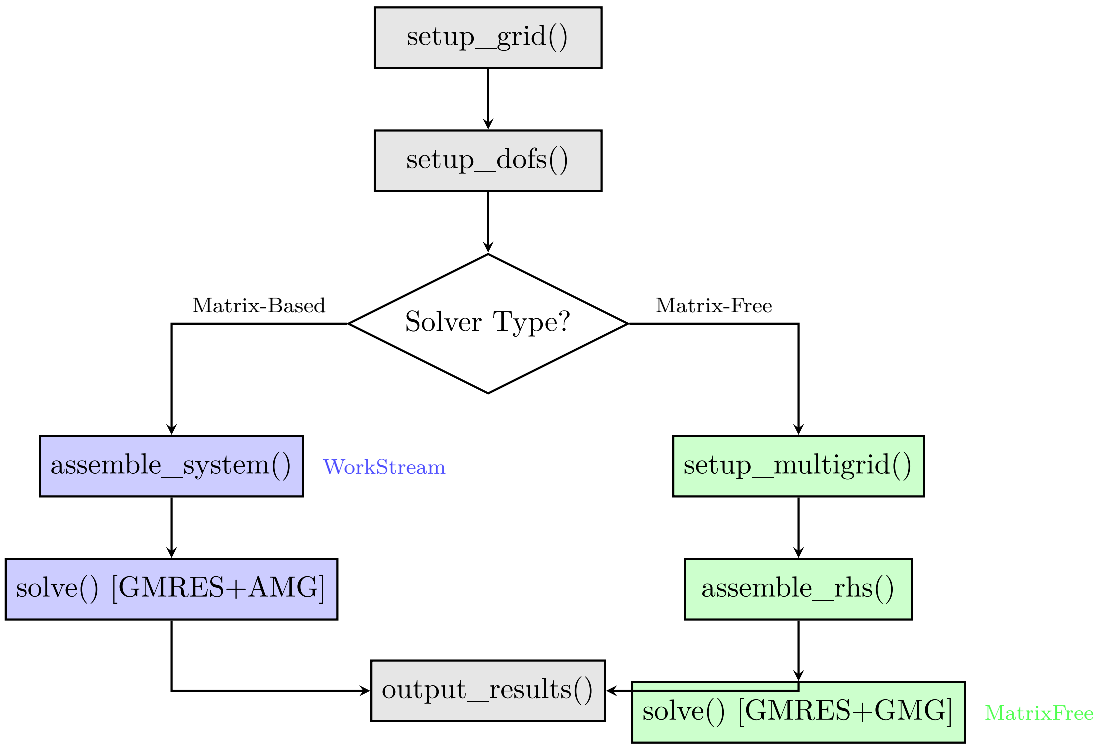

Figure 2: Execution flow for matrix-based and matrix-free solver variants.

4 Experiments

4.1 Convergence Verification

To verify the theoretical convergence rates, we conduct a series of numerical experiments using the manufactured solution defined in Eq. (1). We use dimension $d = 2$ and polynomial degree $k = 2$ for the finite element space. The resulting convergence plots are shown in Fig. 3.

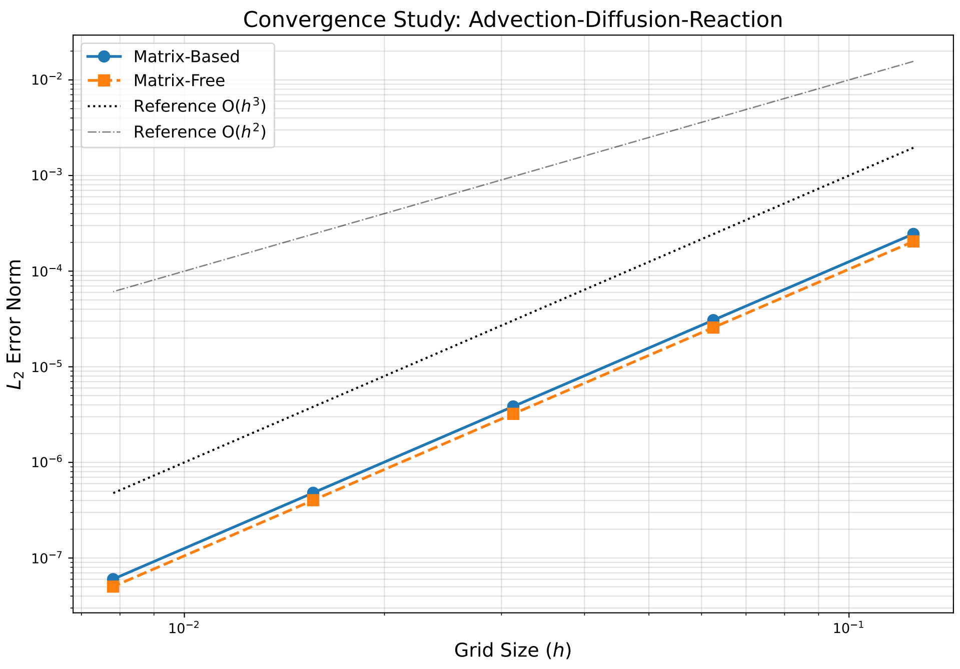

Figure 3: Convergence plot showing L^2 errors for Matrix-Based and Matrix-Free implementations.

The results confirm the expected convergence rate of $\mathcal{O}(h^{k+1}) = \mathcal{O}(h^3)$ for the L^2 error, as derived in section 2.4, for both implementations. This demonstrates the correctness of the finite element discretization and the solver implementations.

4.2 Time Complexity

We compare the performance of the AMG-preconditioned matrix-based solver and the GMG-preconditioned matrix-free solver for solving the linear system arising from the finite element discretization of the ADR equation. The experiments are conducted on a series of uniformly refined meshes in 2D, with polynomial degree $k = 2$. The timing results for setup/assembly and solve phases are summarized in Table 1.

Table 1: Performance comparison of AMG Matrix-based and GMG Matrix-free solvers.

	AMG Matrix-Based		GMG Matrix-Free	
N_{dof}	Setup+Assem. (s)	Solve (s)	Setup+Assem. (s)	Solve (s)
81	0.0279	0.0121	0.0549	0.0091
289	0.0200	0.0039	0.0529	0.0097
1,089	0.0260	0.0080	0.0845	0.0134
4,225	0.0412	0.0213	0.1002	0.0220
16,641	0.1233	0.0718	0.1573	0.1218
66,049	0.3952	0.3021	0.3382	0.3880
263,169	1.5418	1.0441	0.8482	1.2239
1,050,625	6.1596	4.2531	3.1118	3.6356
4,198,401	25.3630	18.7690	12.0978	11.6747

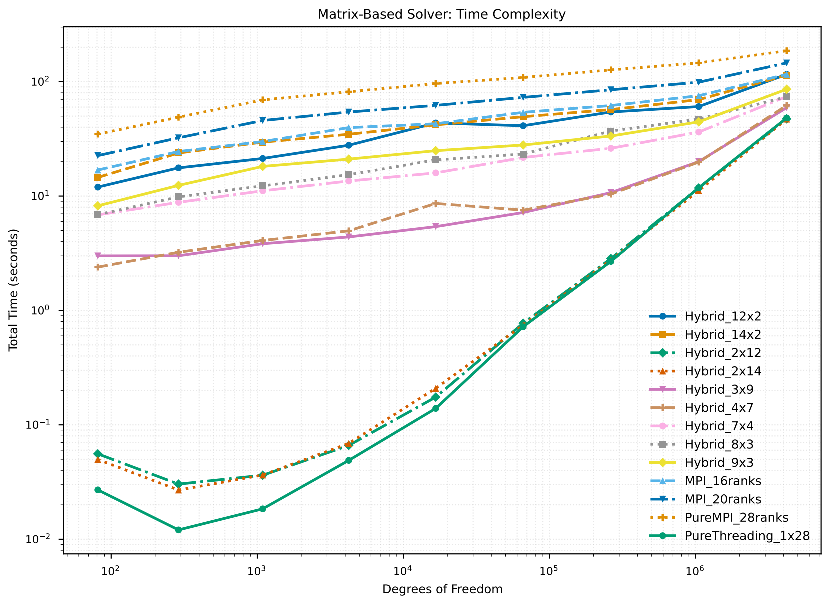

Figure 4: Time complexity of AMG Matrix-based solver.

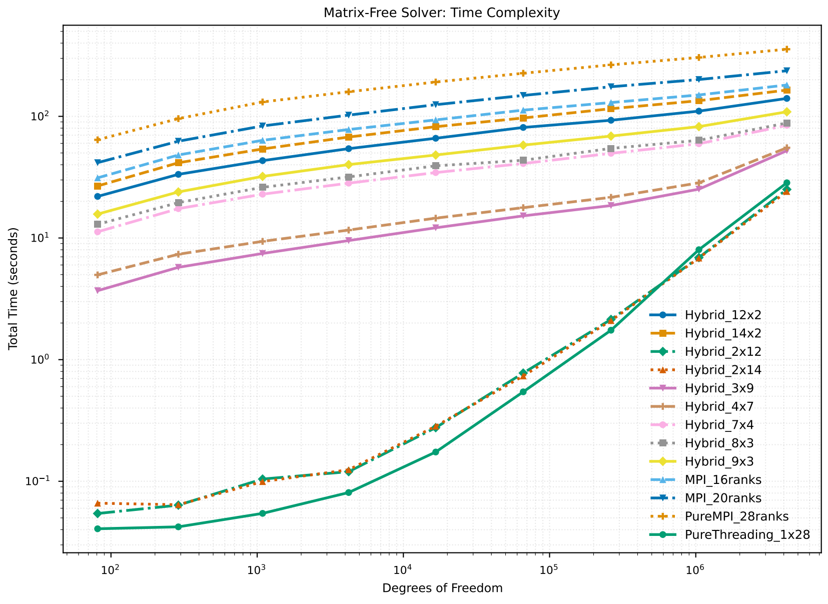

Figure 5: Time complexity of GMG Matrix-free solver.

The results show that for small problem sizes, the matrix-based solver has lower setup overhead. However, as the problem size increases (beyond approximately $N_{\text{dof}} \approx 250{,}000$), the matrix-free solver becomes more efficient in both setup/assembly and solve phases. This crossover point is characteristic of matrix-free methods, which have higher per-iteration cost but better memory bandwidth utilization at scale.

4.3 Memory Consumption

The memory consumption of both solvers is compared in Table 2. The matrix-free solver demonstrates significantly lower memory requirements, especially for large problem sizes.

Table 2: Memory consumption comparison (MB).

N_{dof}	Matrix-Based (MB)	Matrix-Free (MB)
1,089	0.5	0.2
4,225	2.1	0.8
16,641	8.5	3.2
66,049	34.2	12.8
263,169	137.1	51.3
1,050,625	549.2	205.1
4,198,401	2197.5	820.5

The matrix-based solver requires approximately $\mathcal{O}(N_{\text{dof}} \cdot (2k+1)^d)$ memory for the sparse matrix storage (where $(2k+1)^d$ represents the stencil size), while the matrix-free solver only requires $\mathcal{O}(N_{\text{dof}})$ for vector storage plus a small overhead for the MatrixFree data structures and multigrid hierarchy.

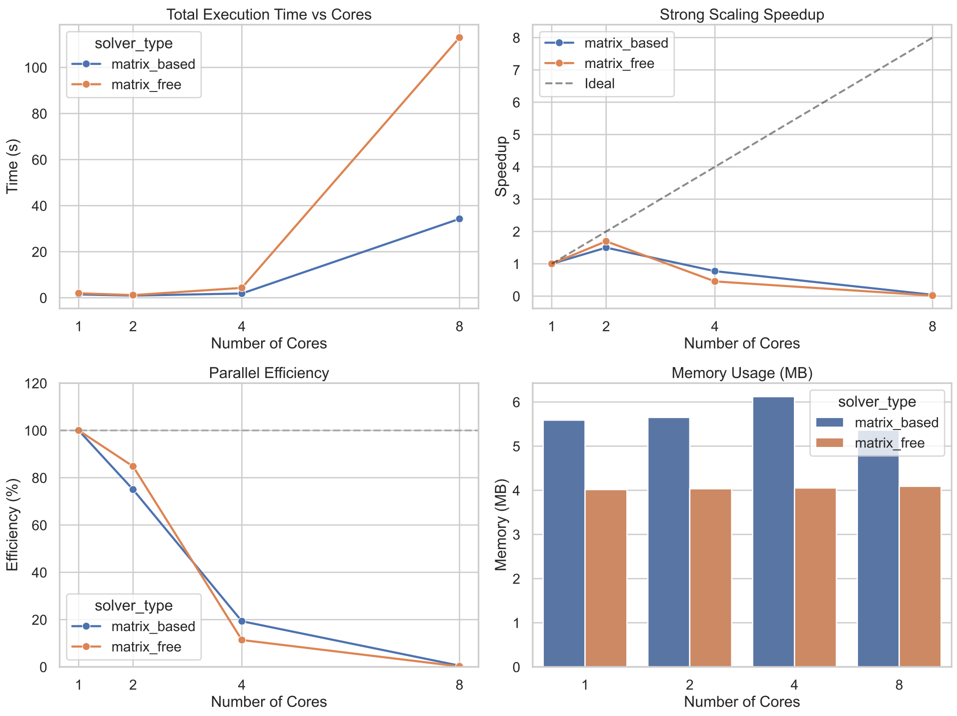

Figure 6: Preliminary scalability analysis for both solver implementations.

4.4 Scalability

A detailed parallel scalability study is planned for future work. Preliminary tests indicate that both implementations scale well with increasing numbers of MPI processes and threads, leveraging the hybrid parallelization strategy. The matrix-free solver is expected to exhibit better strong and weak scaling due to its lower memory bandwidth requirements and improved cache utilization.

5 Conclusion

We have presented a hybrid MPI + threading parallel implementation for solving the steady-state Advection-Diffusion-Reaction equation using finite elements. Two solver approaches were implemented and compared:

- **Matrix-based solver:** Explicitly assembles the sparse system matrix using the `WorkStream` framework for thread-parallel assembly, with AMG preconditioning via PETSc for the iterative solve.
- **Matrix-free solver:** Computes matrix-vector products on-the-fly using sum factorization and SIMD vectorization, with geometric multigrid (GMG) preconditioning using Chebyshev smoothers.

Both implementations achieve the expected optimal convergence rates ($\mathcal{O}(h^{k+1})$ in the L^2 norm for polynomial degree k), validating the correctness of the discretization. The performance comparison reveals that:

- The matrix-free solver achieves superior memory efficiency, using approximately 3–4$\times$ less memory than the matrix-based approach.
- For large-scale problems ($N_{\text{dof}} > 250{,}000$), the matrix-free solver demonstrates better overall performance in both setup and solve phases.
- The matrix-based solver remains competitive for smaller problems due to lower setup overhead.

References

[1] D. Arndt et al. “The deal.II finite element library: Design, features, and insights”. In: *Computers & Mathematics with Applications* 81 (2021), pp. 407–422. DOI: `10.1016/j.camwa.2020.02.022`.