Journal of Computational and Applied Mathematics 481 (2026) 117308

Contents lists available at ScienceDirect

Journal of Computational and Applied Mathematics

journal homepage: www.elsevier.com/locate/cam

Memory - and compute-optimized geometric multigrid *GMGPolar* for curvilinear coordinate representations – Applications to fusion plasma

Julian Litz[a,1], Philippe Leleux[b], Carola Kruse[c], Joscha Gedicke[d], Martin J. Kühn[a,e,*]

[a] German Aerospace Center (DLR), Institute of Software Technology, Department for High-Performance Computing, Linder Höhe, Cologne, 51147, Germany
[b] Laboratoire d'Analyse et d'architecture des Systèmes (LAAS), équipe TSF, 7 avenue du Colonel Roche Toulouse cedex 4, 31031, BP 54200, France
[c] Parallel Algorithms Team, CERFACS (Centre Européen de Recherche et de Formation Avancée en Calcul Scientifique), 42 Avenue Gaspard Coriolis, Toulouse Cedex 01, 31057, France
[d] Institut für Numerische Simulation, Universität Bonn Friedrich-Hirzebruch-Allee, 7, Bonn, 53115, Germany
[e] Life & Medical Sciences Institute (LIMES) and Bonn Center for Mathematical Life Sciences, University of Bonn, Bonn, 53115, Germany

ARTICLE INFO

2010 MSC:
68Q25
65Y20
65Y05
65N55
65N06
65B99

Keywords:
Multigrid
Scientific computing
Parallel computing
Performance optimization
Fusion plasma

ABSTRACT

Tokamak fusion reactors are actively studied as a means of realizing energy production from plasma fusion. However, due to the substantial cost and time required to construct fusion reactors and run physical experiments, numerical experiments are indispensable for understanding plasma physics inside tokamaks, supporting the design and engineering phase, and optimizing future reactor designs. Geometric multigrid methods are optimal solvers for many problems that arise from the discretization of partial differential equations. It has been shown that the multigrid solver GMGPolar solves the 2D gyrokinetic Poisson equation in linear complexity and with only small memory requirements compared to other state-of-the-art solvers. In this paper, we present a completely refactored and object-oriented version of GMGPolar which offers two different matrix-free implementations. Among other things, we leverage the Sherman-Morrison formula to solve cyclic tridiagonal systems from circular line solvers without additional fill-in and we apply re-ordering to optimize cache access of circular and radial smoothing operations. With the *Give* approach, memory requirements are further reduced and speedups of four to seven are obtained for usual test cases. For the *Take* approach, speedups of 16 to 18 can be attained. In an additionally experimental setup of using GMGPolar as a preconditioner for conjugate gradients, this speedup could even be increased to factors between 25 and 37.

1. Introduction

Tokamak fusion reactors are one of the most promising approaches for realizing energy production from plasma fusion. However, due to substantial cost and time to construct fusion reactors and run physical experiments, numerical experiments are indispensable

* Corresponding author.
E-mail address: martin.kuehn@dlr.de (M.J. Kühn).
[1] Current address: Institute of Climate and Energy Systems - Energy Systems Engineering (ICE-1), Forschungszentrum Jülich GmbH, 52425 Jülich, Germany

https://doi.org/10.1016/j.cam.2025.117308
Received 15 July 2025; Received in revised form 8 October 2025
Available online 26 December 2025

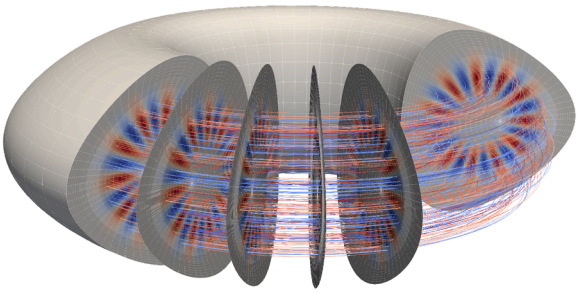

Fig. 1. Particular solution of (1) on a 3D tokamak with Culham geometry. The figure shows a 3D tokamak geometry spanning over 220 degrees in toroidal direction and clipped over the remaining 140 degrees. It furthermore show several 2D cross sections in the clipped part of the geometry.

to understand plasma physics inside tokamaks, to support and speed up the engineering phase, and to optimize future reactor designs. To model and simulate the particular physics, the gyrokinetic framework is used by many authors; see, e.g., [1–6]. The corresponding five-dimensional problem to be solved contains three dimensions for the torus geometry and two dimensions for the velocity [5]. At each time step, the simulation requires solving a 5D Vlasov equation for each species, along with a 3D Poisson-like equation that enforces quasi-neutrality. While of smaller dimension, computing the solution of this three-dimensional system can deteriorate the overall performance and scaling of simulation codes [5]. While some solvers, such as GENE-3D [1], COGENT [2], or EUTERPE [3], solve this system directly, other codes such as GYSELA [5], and ORB5 [4] solve a large number of two-dimensional equations on cross sections of the tokamak; cf. Fig. 1 for a visualization of a tokamak geometry with several cross sections. The type and form of the 2D cross sections differ for various publications (see also the next section). As the formulation by curvilinear coordinates poses an additional difficulty at the section of the separatrix, recently, in [7], a multi-patch geometry for a decomposed domain with an X-point was presented.

In [8], the taylored geometric multigrid method *GMGPolar* was proposed to efficiently solve the resulting subproblems on hundreds or thousands of cross sections, repeated over thousands or millions of time steps. From the beginning, GMGPolar has been designed to be scalable, allow higher order approximations, and reduce the memory footprint to a minimum. In [9], a matrix-free C++ implementation using shared memory parallelism through OpenMP was presented. It was furthermore shown that the GMGPolar algorithm is optimal in the sense that it has linear asymptotic complexity, i.e., the number of floating point operations (FLOP) for the solution process is a linear function of the number of degrees of freedom. In [10], GMGPolar was compared to other state-of-the-art solvers for the gyrokinetic Poisson-like equation on tokamak cross sections. GMGPolar was found to have the smallest memory requirements and to offer a compromise between relatively fast execution and high order of approximation.

In this article, we present a completely refactored version 2 of GMGPolar [11]. The novel version aligns the multigrid components with cache lines, optimizes the compromise between the storage and recomputation of (costly) function evaluations, underwent low level performance engineering through, e.g., function inlining, and boosts the parallel scalability through a substantial reduction of synchronization and waiting times. In addition, GMGPolar now implements F-cycles and full multigrid to speed up the convergence.

The article is structured as follows. In Section 2, we present the considered model problem together with the mathematical background and multigrid components of GMGPolar. In Section 3, we present the object-oriented design of the geometric multigrid algorithm. Here, we focus particularly on optimizing memory usage and cache accesses, improving parallel scalability, and on new multigrid features such as full multigrid. We provide extensive numerical results in Section 4 before concluding with Section 5.

2. Model problem and principles of GMGPolar

2.1. Model problem and geometry

With some simplifications, as explained in [10], we consider the following model problem for a 2D Poisson-like equation

$$\begin{aligned} -\nabla\cdot(\alpha\nabla u)+\beta u &= f \quad \text{in} \quad \Omega,\\ u &= u_D \quad \text{on} \quad \partial\Omega, \end{aligned} \tag{1}$$

which arises in the description of quasi-neutrality. Here, $\Omega\subset\mathbb{R}^2$ is a disk-like domain, and $f:\ \Omega\to\mathbb{R}$, with $f\in C^0(\overline{\Omega})$, is the right-hand side. The functions $\alpha,\beta:\ \Omega\to\mathbb{R}$ are coefficients corresponding to *density profiles*, with $\alpha\in C^1(\Omega)\cap C^0(\overline{\Omega})$ and $\beta\in C^0(\overline{\Omega})$. Furthermore, we prescribe Dirichlet boundary conditions with $u_D\in C^0(\partial\Omega)$.

We next introduce the energy functional

$$J(u) := \int_\Omega \frac{1}{2}\alpha|\nabla u|^2+\frac{1}{2}\beta u^2 - fu\,(x,y), \tag{2}$$

from which we obtain a symmetric linear system after a finite difference discretization as described further below. We note that the weak formulation of problem (1) is equivalent to the minimization of the energy functional $J(u)$ in (2) over a suitable Sobolev space, prescribing the boundary conditions u_D.

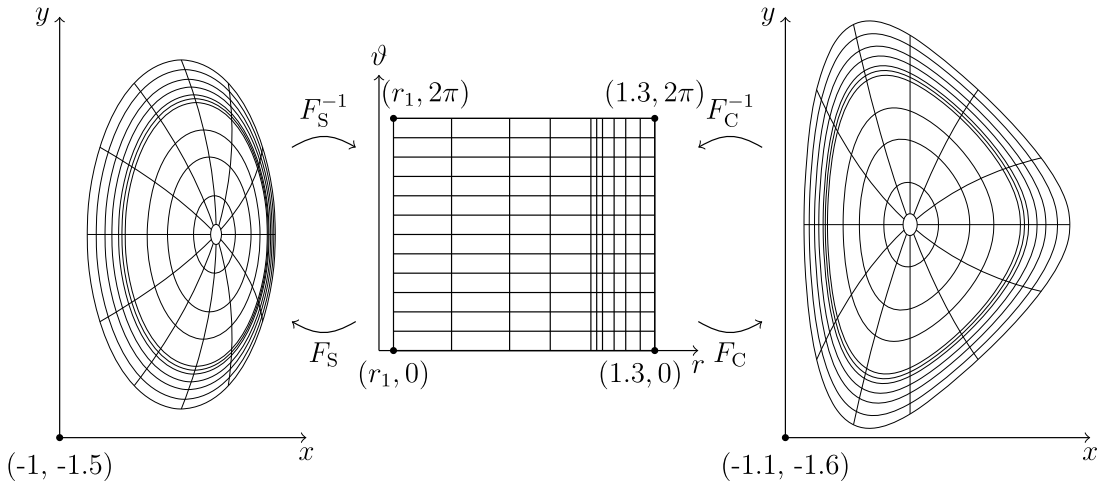

Fig. 2. Visualization of the Shafranov and Czarny cross section geometries of a tokamak. As both geometries can be described in curvilinear coordinates, mappings from a rectangular grid $[r_1, 1.3] \times [0, 2\pi]$ (center) to the considered geometries are indicated. The *Shafranov* geometry (left) is given by the mapping $F_S : [r_1, 1.3] \times [0, 2\pi] \to \mathbb{R}^2$ and the *Czarny* geometry (right) is given by the mapping $F_C : [r_1, 1.3] \times [0, 2\pi] \to \mathbb{R}^2$; see (3) and (4). For the existence of the invertible mappings F_S^{-1} and F_C^{-1}, we need $r_1 > 0$. The depicted grids are symbolically refined at 2/3 of the generalized radius.

In this paper, we consider three different geometries that represent cross sections of a tokamak; cf. [6,12,13]. Two of the three geometries can be described by relatively short representations of curvilinear coordinates, i.e., based on a mapping F_g, $g \in \{S, C\}$, from the curvilinear coordinates $(r, \vartheta) \in [r_1, 1.3] \times [0, 2\pi)$, where r is the (generalized) radius and ϑ the (generalized) angle to Cartesian coordinates (x, y), defined as follows.

The Shafranov geometry is a deformed ellipse which is defined by the mapping

$$F_S(r, \vartheta) := \begin{pmatrix} x(r, \vartheta) \\ y(r, \vartheta) \end{pmatrix} = \begin{pmatrix} x_0 + (1 - \kappa) r \cos \vartheta - \delta r^2 \\ y_0 + (1 + \kappa) r \sin \vartheta \end{pmatrix}; \tag{3}$$

see Fig. 2 (left). Here, κ is the elongation and δ is the Shafranov shift; see [6,12]. For the parameters, we use $x_0 = y_0 = 0$, $\kappa = 0.3$, and $\delta = 0.2$.

The Czarny geometry is a D-shaped geometry and adds triangularity to the shape; see Fig. 2 (right). It is defined by the mapping

$$F_C(r, \vartheta) := \begin{pmatrix} x(r, \vartheta) \\ y(r, \vartheta) \end{pmatrix} = \begin{pmatrix} \frac{1}{\varepsilon}\left(1 - \sqrt{1 + \varepsilon(\varepsilon + 2r \cos \vartheta)}\right) \\ y_0 + \frac{\mathrm{e}\, \xi\, r \sin \vartheta}{2 - \sqrt{1 + \varepsilon(\varepsilon + 2r \cos \vartheta)}} \end{pmatrix}, \tag{4}$$

where ε is the inverse aspect ratio, e the ellipticity, and $\xi = 1/\sqrt{1 - \varepsilon^2/4}$, see [12,14]. For the parameters, we use $y_0 = 0$, $\varepsilon = 0.3$, and $\mathrm{e} = 1.4$.

Note that for $r_1 = 0$, the inverse mappings F_g^{-1}, $g \in \{S, C\}$, do not exist as the functions F_g, $g \in \{S, C\}$, map the whole line $(0, \vartheta)$ on the origin. This, however, is only of theoretical concerns as our method will use an interior radius r_1 such that $0 < r_1 \ll 1$.

The third geometry, the nonanalytical *Culham* geometry [15] has been chosen to take into account more realistic geometries in GYSELA. The rather lengthy development can be found in [10, Section 6.2]. For a visualization of the Culham cross sections in a 3D tokamak; see Fig. 1.

2.2. Discretization

While a matrix-free implementation of iterative solvers based on finite element (FE) discretizations might be cumbersome, finite differences (FD) offer a straightforward approach for matrix-free implementations. However, standard finite difference schemes applied to nonuniform meshes generally lead to nonsymmetric discretizations, even if the considered model problem is symmetric or, more precisely, the considered operator self-adjoint. In [16], we presented a novel approach to derive symmetric FD discretizations for nonuniform meshes. In this FE-inspired FD discretization, the energy functional is localized and discretized on the local elements.

Let us first introduce a nonuniform mesh in product format by $r_1, \ldots, r_{n_r}$ with $0 < r_1 \ll 1$ and $r_{n_r} = 1.3$ as well as $\vartheta_1, \ldots, \vartheta_{n_\vartheta + 1} \in [0, 2\pi]$ with $\vartheta_1 = 0$ and $\vartheta_{n_\vartheta + 1} = 2\pi$. We define

$$h_i := r_{i+1} - r_i, \quad i \in \{1, \ldots, n_r - 1\}, \quad k_j := \vartheta_{j+1} - \vartheta_j, \quad j \in \{1, \ldots, n_\vartheta\}. \tag{5}$$

Furthermore, we add an additional restriction for the discretization, i.e.,

$$\begin{aligned} &n_r \text{ odd}, \quad h_{2i} = h_{2i-1}, \qquad i \in \{1, \dots, (n_r - 1)/2\}, \\ &n_\vartheta \text{ even}, \quad k_{2j} = k_{2j-1}, \qquad j \in \{1, \dots, n_\vartheta/2\}. \end{aligned} \tag{6}$$

Note that this restriction is neither needed for the FD discretization nor for the geometric multigrid in general. It will only be used to raise the convergence order of the geometric multigrid via implicit extrapolation. For more details, see [8,16].

For theoretical purposes, we assume h_i and k_j to be uniformly bounded by

$$0 < h_{\min} \le h_i \le h \quad \text{and} \quad 0 < k_{\min} \le k_j \le k, \tag{7}$$

as well as the existence of $0 < \tau < \infty$ such that $h = \tau k$.

We now localize $J(u)$ of (2), by considering the rectangular elements $R_{ij} := [r_i, r_i + h_i] \times [\vartheta_j, \vartheta_j + k_j]$. By transformation, the local energy functional writes

$$J_{R_{i,j}}(u) := \int_{R_{i,j}} \left(\frac{1}{2}\alpha |DF_g^{-T} \nabla_{(r,\vartheta)} u|^2 + \frac{1}{2}\beta u^2 - fu \right) |\det DF_g|(r, \vartheta), \tag{8}$$

where DF_g is the Jacobian matrix of F_g and $DF_g^{-T} := (DF_g^T)^{-1}$, $g \in \{S, C\}$.

For the sake of simplicity, we do not distinguish notations between functions defined on the logical and physical domain. For a full derivation of the discretization, we refer to [8,16] with $\beta = 0$ and [9] for $\beta \neq 0$. In order to simplify the notation, we will furthermore use

$$\frac{1}{2}\alpha DF_g^{-1} DF_g^{-T} |\det DF_g| =: \begin{pmatrix} a^{rr} & \frac{1}{2}a^{r\vartheta} \\ \frac{1}{2}a^{r\vartheta} & a^{\vartheta\vartheta} \end{pmatrix}. \tag{9}$$

Discretizing the local energy components $J_{R_{i,j}}(u)$ to $\widetilde{J}_{R_{i,j}}(u)$, computing the derivative with respect to $u_{s,t}$, and searching for the critical point, i.e.,

$$\sum_{i=1}^{n_r - 1} \sum_{j=1}^{n_\vartheta} \frac{\partial}{\partial u_{s,t}} \widetilde{J}_{R_{i,j}}(u) \overset{!}{=} 0. \tag{10}$$

yields the nine-point finite difference stencil and right hand side as provided by [9]. Note that in the implementation, the geometric understanding has changed. While this does not change the stencil itself, the particular interpretation of, e.g., left and top differ between the current codebase and [9].

In [8], we suggested a particular discretization *across-the-origin* to handle the artificial singularity at $r_1 = 0$, avoiding to have the origin as a grid point. It could be shown that this approach performed almost identical to incorporating (artificial) Dirichlet boundary conditions on an inner circle with generalized radius $0 < r_1 \ll 1$.

2.3. Multigrid and GMGPolar basics

GMGPolar is a geometric multigrid method that has been optimized to satisfy three desired requirements for the integration in a gyrokinetic framework such as GYSELA [5]: i) it achieves fast convergence for geometries represented in curvilinear or (generalized) polar coordinates, ii) it provides a matrix-free approach with low memory requirements, and iii) it realizes higher order convergence through implicit extrapolation. As a multigrid method, it additionally allows for good parallel scalability by design. In [9], it was shown that the number of floating point operations and memory cost of GMGPolar depend, asymptotically, only linearly on the number of degrees of freedom. In the following, we summarize very briefly GMGPolar's mathematical core properties. For more details and a pseudo-code description of GMGPolar, see [8,9].

2.3.1. Smoothing for curvilinear coordinate representations

Through the transformation of the model problem (1) to a curvilinear coordinate representation, the "strong connections" between grid points change across the grid as we go from $r_1 \approx 0$ to $r_{n_r} = 1.3$. This property strongly affects the choice for suitable smoothing operations in the multigrid algorithm. In general, point-wise smoothers are not sufficient.

In [17], smoothing properties of particular circular and radial line smoothers were considered analytically for polar coordinate representations. Strong connections lie on circular lines near the origin, circle smoothers were found to perform better in this part of the disk-shaped domain. Closer to the boundary, strong connections lie on radial grid lines, and radial smoothers are more efficient. Based on these findings, GMGPolar switches from circle to radial smoothing where $k/h_i r_i > 1$ is satisfied; a schematic picture is given in Fig. 4 (left). Note that in most applications, we assume uniform discretization in the angular direction, i.e., $k = k_j$, $j = 1, \dots, n_\vartheta$. Otherwise, a more general switching condition or overlapping smoothers are needed. The implemented switching yields good smoothing behavior on the whole domain by only treating every grid point once per smoothing iteration.

2.3.2. Coarsening and intergrid transfer operators

Coarsening in GMGPolar is done by standard coarsening, selecting every second node in each dimension. Except for the particular handling of implicit extrapolation, as briefly mentioned in the next section, GMGPolar uses standard bilinear interpolation. With the symmetry-preserving FD scheme, we can use the adjoint operator as restriction operator. Note that we do not need any additional scaling constant between prolongation and restriction as our tailored FD scheme scales the right hand side locally with the surface of the considered rectangle; see also [8, Section 4.2].

2.3.3. Higher order convergence through implicit extrapolation

When using the discretization from Section 2.2, implicit extrapolation allows to raise the convergence order towards the true solution when refining the grid. For the considered model problems and geometries, increases from order 2 to approximately 3.6 - 4.0 were observed [16]. For implicit extrapolation to take effect, GMGPolar makes some adjustments to standard multigrid algorithms. However, these changes only affect the finest two grid levels. All operations between grid levels below the second finest grid comply with standard multigrid practices. For a complete description of the adjusted intergrid transfer operators and smoothing operations, see [8, Section 4.3].

2.3.4. Take and give approaches

A straightforward way to implement the FD stencil is the node-wise computation of all available row entries of the stiffness matrix. This approach has been denoted the *A-take*, or simply *Take*, approach in [9], as it takes the necessary values from the memory locations of the neighboring nodes. From the structure of the stencil, it can be seen that many entries are recomputed with the take approach. Alternatively, we can optimize the computation of the stencil values per node and distribute computed values to the memory of neighboring nodes that need the same function evaluations. This approach was denoted *A-give* or *Give*. Both approaches will be discussed in the sections on the refactored version of GMGPolar.

3. Object-oriented redesign and algorithmic optimization

The refactoring of GMGPolar represents a substantial transition from a functional programming style, which had been inherited from the initial implementation of [8], to a structured and object-oriented design. With this shift, specific functionalities are now better encapsulated in dedicated classes, clarifying the responsibilities of different components within the codebase – also ensuring better maintenance and extension capabilities. The refactoring was essentially done during the master thesis of Julian Litz [18]. Here, we outline the key aspects of the novel implementation.

Compared to the previous version 1 of GMGPolar, the data structure has been separated into more specialized classes, instead of being inside a unique large multigrid *Level* class. A dedicated `PolarGrid` class will manage grid-related data, ensuring efficient access and organization, while a separate `LevelCache` class will handle the storage and retrieval of precomputed data. The `Interpolation` class manages intergrid transfer operations, separating the responsibilities of data movement between grid levels from other functionalities of the solver. In addition, a custom `LinearAlgebra` class manages the fundamental operations on vectors and matrices. In this class, we implemented tailored tridiagonal solver algorithms, which are crucial for the performance of the smoother.

To further modularize the design, distinct operator classes were created to handle specific computational tasks, such as computing the `Residual`, solving the system coarse matrix via `DirectSolver`, and performing smoothing operations via `Smoother`. The refactored layout is presented in Fig. 3. For an overview of particular settings with respect to geometry, multigrid, and particular problem settings, see Table A.1. By adopting this more object-oriented approach, which also optimizes memory usage and enhances the multigrid cycle methodology, we obtain a more flexible and efficient tool for large-scale gyrokinetic simulations.

The original GMGPolar code employed exclusively the Give approach for matrix-free computations and dynamically computed transformation coefficients during matrix-vector operations, whereas the Take approach was only available with the assembled matrix version. While the Give method minimizes memory usage, it introduces a computational overhead, particularly for complex geometries. To provide greater flexibility, the refactored implementation supports, both, the Give and Take stencil implementations, allowing to choose between storing or recomputing transformation coefficients a^{rr}, $a^{r\vartheta}$, $a^{\vartheta\vartheta}$ and $\det DF_g$ from (9) based on specific requirements. With this functionality, the solver adapts to a wider range of problem sizes and computational constraints. In the following, we will present in depth the improvements that were undertaken in GMGPolar version 2.

3.1. Memory usage and cache efficiency

Memory and cache optimization was a key component to be considered in the refactoring phase. The matrix-free GMGPolar implementation had already been designed with small memory requirements [9] compared to other state-of-the-art solvers [10]. Nevertheless, the previous implementation did not yet exploit the symmetry of the smoother matrices, relying instead on a full LU decomposition that introduced additional fill-in. While the total memory consumption for the prior version was computed asymptotically linear as $12n$ [9] (accounting for all multigrid levels), the new Give variant reduces finest-level storage to just $5n$ (asymptotically with all levels: $6.7n$) by employing in-place symmetric Cholesky factorizations for the smoothers and eliminating redundant temporary vectors. Here, $n = n_r \cdot n_\vartheta$ denotes the total number of grid nodes. The Take variant introduces four extra vectors (for arrays `arr`, `att`, `art`, and `detDF`) to streamline data access, bringing its peak requirement to $9n$ (asymptotically with all levels: $12n$).

The smoother constitutes a major computational component of the solver (see [9]); therefore, both the setup and solution phases of the smoother matrices were improved. In the previous version, the assembled matrices were stored in Coordinate List (COO) format and decomposed using a general LU decomposition. The smoother's solver matrices naturally take the form of tridiagonal matrices or cyclic tridiagonal matrices when using circle line smoothing with peridodic boundary conditions; cf. [8]. In the new implementation, we eliminate explicit row-column indices and exploit the symmetry, only storing the upper or lower half of the matrices and additionally use specialized tridiagonal solvers suited to this structure. The tridiagonal matrices of the radial smoother are factorized using Cholesky decomposition in $4n + \mathcal{O}(1)$ operations and solved via forward and backward substitution in $5n + \mathcal{O}(1)$ operations. To factorize the cyclic tridiagonal matrices of the circle smoother, we apply the Sherman-Morrison formula, which expresses the cyclic tridiagonal matrix as a rank-one modification of a standard tridiagonal matrix, enabling efficient factorization without fill-in. The

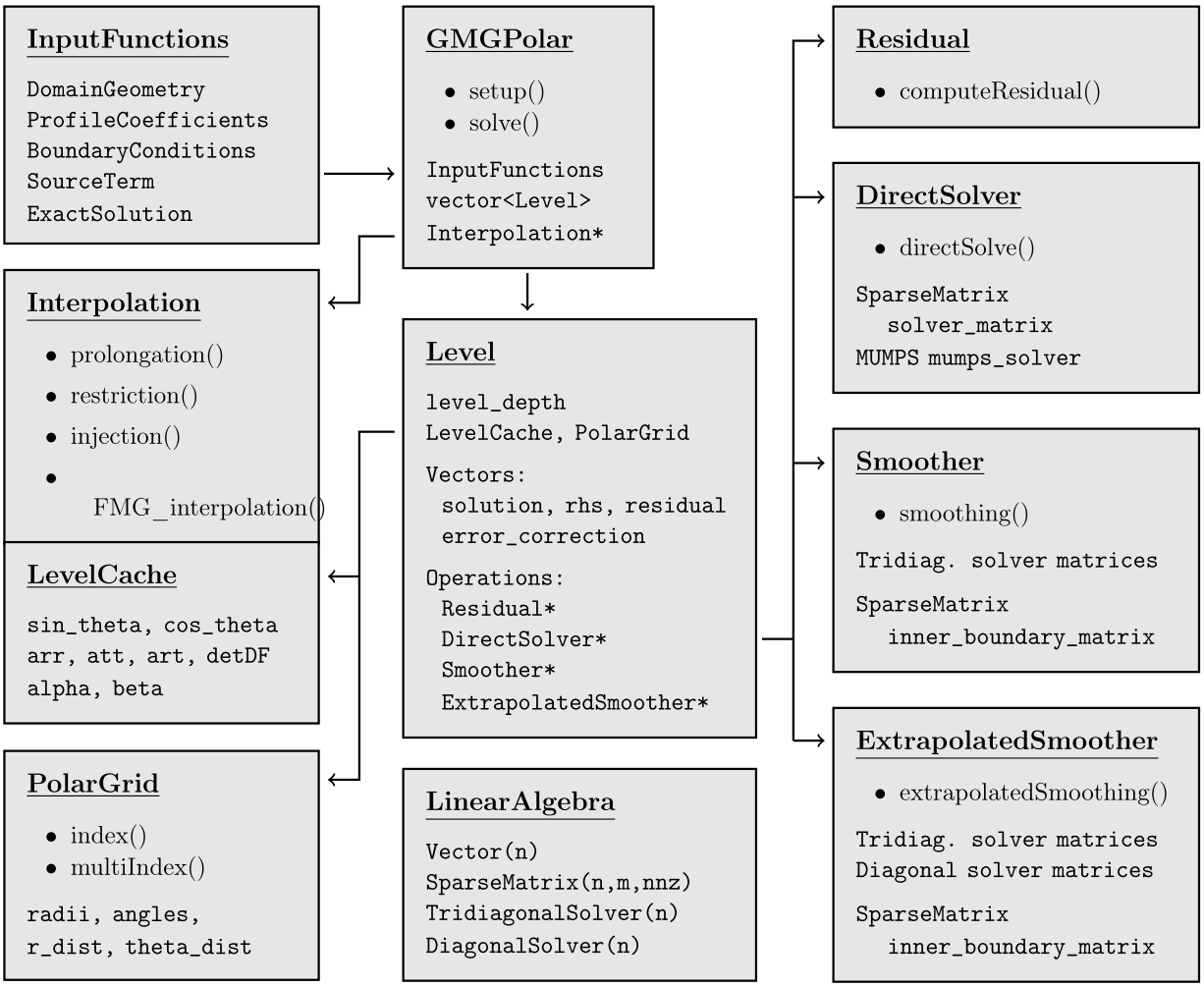

Fig. 3. Refactored class layout highlighting the modular structure and object-oriented design approach.

solution phase involves solving two independent tridiagonal systems and combining their results, requiring $12n + \mathcal{O}(1)$ operations. Although this new approach incurs a slightly higher cost in the solution phase compared to the adapted LU decomposition used previously (cf. [9, Sec. 8.4]), it avoids the fill-in introduced in the last row and column, reducing memory footprint.

Moreover, we optimized the solution step of the smoother matrices on cache level by reordering grid indexing to align with the smoother's line patterns. For a visualization of a smoother-aligned indexing, see Fig. 4. This reorganization increases data locality and improves cache line usage, consequently, leading to faster execution times.

Remark 1. For solving the circle smoothing system on the innermost circle with the default, across-the-origin, discretization, attention has to be paid. Due to the across-the-origin approach, the corresponding submatrix is not tridiagonal. Therefore, we use the COO format and MUMPS for the smoother on the innermost circle.

As sparse linear algebra applications are often memory-bound, improving memory access patterns is also promising for speeding up the particular application. However, as first observed in [10], the matrix-free version of GMGPolar, replacing most memory accesses by (re)computations, did not speed up our method as expected. This observation is a direct consequence of the complex domain geometries of the tokamak cross sections. On these geometries, the evaluation of the sine and cosine functions as well as the factors $a^{rr}, a^{r\vartheta}, a^{\vartheta\vartheta}$ were found to be relatively costly. In [9], we already stored sine and cosine evaluations for the different generalized angles. In the novel version, these values are also stored. By default, the novel version also caches the values of α and β evaluations from (1) and (13) for the Give approach. For the Take approach, we additionally cache the transformation coefficients from (9) – which can also be stored for the Give approach upon selection by the user.

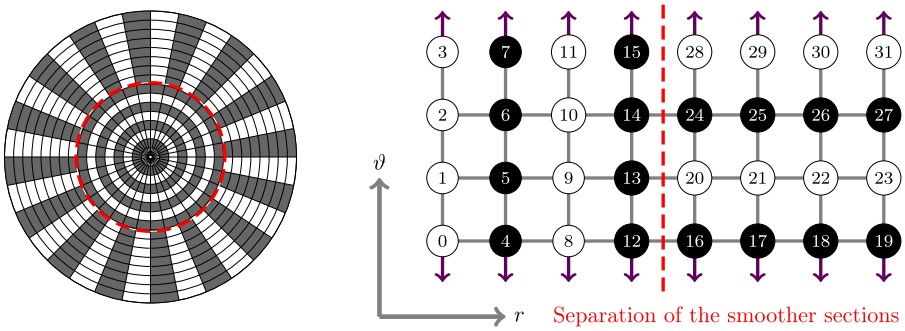

Fig. 4. Combined circle-radial smoother and indexing for the smoothing operations. Circle and radial lines colored black and white on a grid of dimension 16×32 with eight circle lines of 32 nodes and 16 radial lines of eight nodes; for visualization simplification, the curvilinear lines are shown without the nodes (left). Optimized grid indexing for a periodic grid of dimension 8×4 with four circle lines of four nodes and four radial lines of four nodes. Vertical lines of the same color represent circular smoothers, while horizontal lines of the same color correspond to radial smoothers (right).

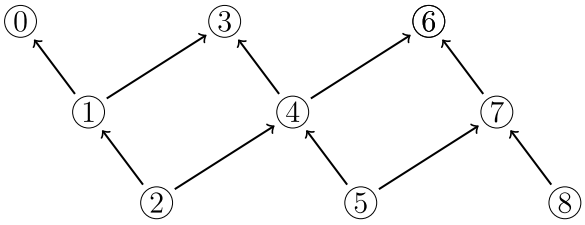

Fig. 5. Dependency graph for the application of the system matrix $A^{(i)}$, of multigrid level $i \in \{0, \dots, L-1\}$, using the Give implementation. Each vertex represents a task corresponding to a line of nodes. The edges visualize the dependencies between these tasks.

3.2. Parallelization

In the original implementation of [9], a task-based parallelism with dependencies was used. At runtime, the OpenMP threads could pick up and perform tasks as they became available. While this approach might be advantageous for unstructured task sets and largely differently sized task, we found that a loop-based parallelism yielded better results for the structured and similarly-sized problems considered in GMGPolar. With the loop-based parallelism, we avoid dynamic scheduling and dependency management and reduce synchronization overhead.

In order to avoid concurrent updates of memory locations when applying the system matrix A with the Give approach, GMGPolar treats every third line in parallel; see [9] and Fig. 5.

For the smoothing operation on multigrid level $i \in \{0, \dots, L-1\}$, we solve subsystems of the original system with matrix $A^{(i)}$, right hand side $f^{(i)}$, and solution $u^{(i)}$. These systems write

$$A^{(i)}_{s_c s_c} u^{(i)}_{s_c} = f^{(i)}_{s_c} - A^{(i)}_{s_c s_c^{\perp}} u^{(i)}_{s_c^{\perp}}, \tag{11}$$

where s refers to the smoothing operation, either circle or radial, and c to the color, either black or white. Furthermore, $A^{(i)}_{s_c s_c}$, $u^{(i)}_{s_c}$, and $f^{(i)}_{s_c}$ correspond to the nodes to be smoothed and the complementary part $A^{(i)}_{s_c s_c^{\perp}}$ and $u^{(i)}_{s_c^{\perp}}$ to the nodes connected, which contribute to the right hand side of the system. In the Take approach, the complementary updates to the right hand side can also be done in parallel. While lines of the same smoother and color can be solved completely in parallel, several dependencies to update the right hand sides have to be considered with the Give approach. Using the Give approach, we obtain a more complex parallelization pattern as nodes change the values of their neighbors. In the prior (Give) implementation, the additional dependencies to be added to Fig. 5 can be found in [9, Fig. 8]. The novel implementation adopts a slightly less complex pattern as visualized in Fig. 6. In this approach, the rows for the application of the complementary part are executed in parallel pattern of 2-4-4 with barriers in-between, meaning that, first, every second line is treated, then, two sweeps of lines with a distance of four are executed before, eventually, the system is solved for every second line in parallel.

3.3. Multigrid features

GMGPolar now also supports W- and F-cycles. The W-cycle performs additional coarse grid corrections by revisiting intermediate levels during the upward traversal, yielding a more thorough error reduction on coarser grids. With the F-cycle, GMGPolar offers a hybrid between the V-cycle and W-cycle. It combines the efficiency of the V-cycle with the robustness of the W-cycle by selectively applying additional coarse-grid corrections; see [19,20].

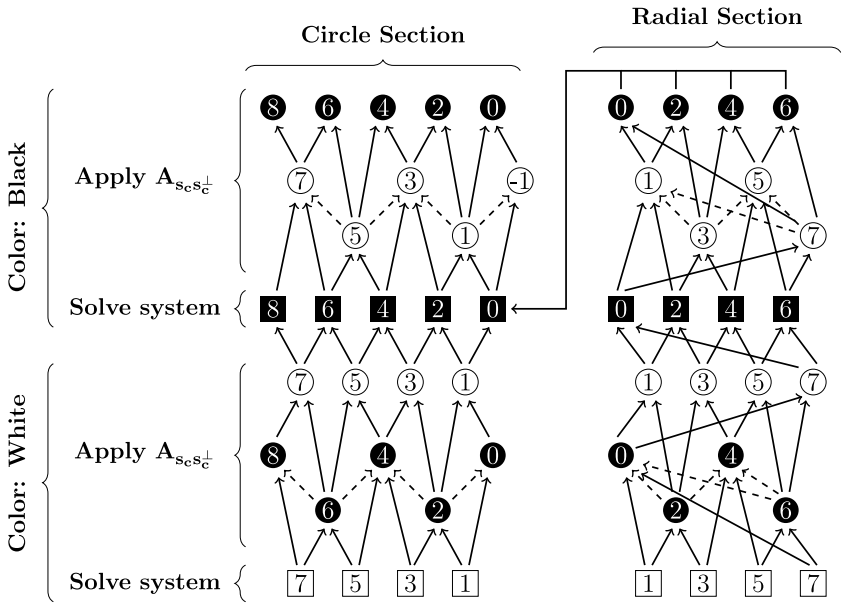

Fig. 6. Parallelization of the smoother using the Give approach. The figure represents a grid which consists of nine circular and eight radial smoother. In this example, inner- and outermost circle smoothed with a circle pattern are colored black. Vertices represent computational tasks corresponding to a line of nodes. Solid edges visualize dependencies between tasks as given by Eq. (11). Dashed edges represent synthetic dependencies introduced to avoid concurrent updates of the same memory location and ensure conflict-free execution during parallel processing. .

In addition, GMGPolar now provides support for a full multigrid cycle (FMG). FMG uses nested iterations to compute a refined initial approximation before standard multigrid cycles are applied. With this approach, we significantly accelerate the convergence process, as shown in Section 4.

4. Numerical results

In this section, we will present numerical results for the model problem (1). The coefficients α and β as well as the manufactured solution are inspired by the simulation of plasma in tokamak fusion reactors and taken from prior benchmarks in [9,10]. We consider a *Polar solution* with oscillations aligned with the polar grid

$$u(x,y) = 0.4096\left(\frac{r(x,y)}{R_{max}}\right)^6\left(1-\frac{r(x,y)}{R_{max}}\right)^6\cos(11\vartheta(x,y)). \tag{12}$$

Fig. 7 illustrates the solution for the Shafranov (left), Czarny (center), and Culham (right) geometries. Note that for the nonanalytical Culham geometry, no exact, manufactured solution is supplied. For the coefficients α and β, we set

$$\alpha(r) = \exp\left[-\tanh\left(\frac{\frac{r(x,y)}{R_{max}}-r_p}{\delta_r}\right)\right], \quad \beta(r) = \frac{1}{\alpha(r)}, \tag{13}$$

where $\delta_r = 0.05$ and $r_p = 0.7$, as in [9,10], and $R_{\max} = 1.3$ as in [8,9].

Our experiments were run on an AMD EPYC 7601, 2.2 GHz, node with two 32-Core sockets and 128 GB DDR4 RAM of the supercomputer CARA at the German Aerospace Center as well as on Intel Xeon "Skylake" Gold 6132, 2.60 GHz, node with four 14-Core sockets with 384GB DDR4 RAM of a small internal cluster. As for coarse level solver, we use MUMPS v5.5.1 [21]. We use LIKWID [22,23] for measuring performance in MFLOPs/s and data transfer in MBytes/s.

We allow a grid-adapted maximum number of levels yielding six multigrid levels for a grid of size 193×256 and 11 levels for a grid of size $6\,145 \times 8\,192$. As in prior publications [8,9], we use an anisotropic grid refinement, approximately where the gradient of coefficient α attains its minimum; [8, Fig. 1] or, symbolically, in Fig. 2. This setup demonstrates that GMGPolar is also capable of handling anisotropic grids efficiently. The absolute or relative convergence criteria were selected depending on the purposes of the individual, following subsections. Convergence is measured in the weighted $\|\cdot\|_{\ell_2}$-norm

$$\|v\|_{\ell_2} = \sqrt{\frac{1}{n}\sum_{i=1}^{n} v_i^2}.$$

In Section 4.1, we provide roofline model results. In Section 4.2, we consider the memory requirements. In Section 4.3, we provide weak scaling results. In Section 4.4, we show strong scaling results. In Section 4.5, we provide results on FMG and different cycle

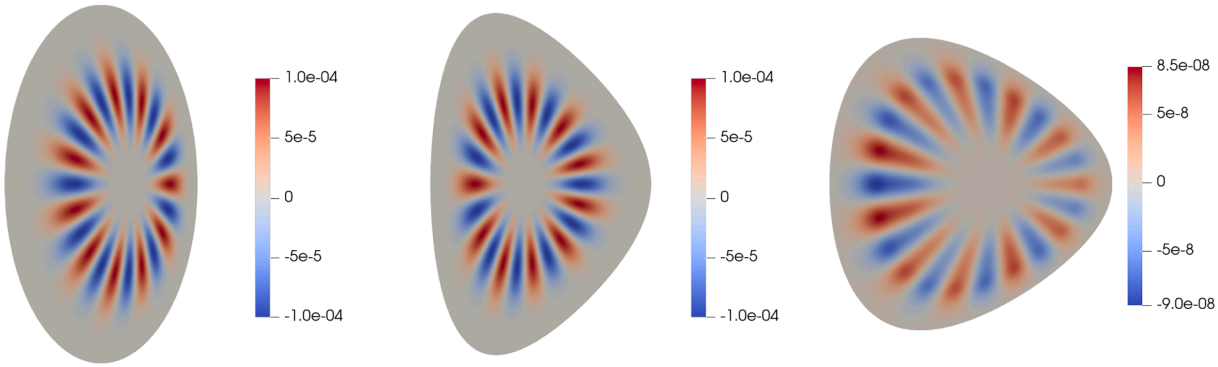

Fig. 7. Illustration of the manufactured solution (12). Visualized solutions for the Shafranov (left), Czarny (center), and Culham (right) geometries.

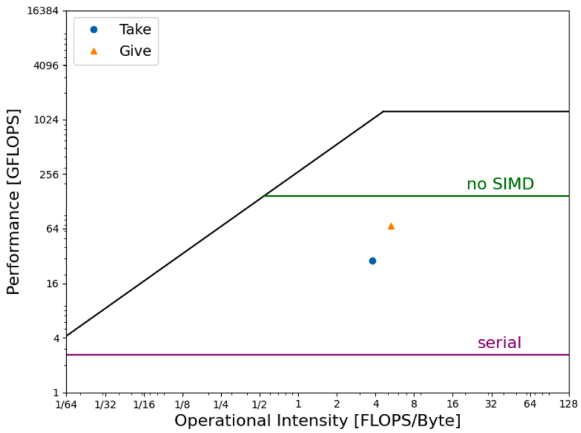

Fig. 8. Roofline model of GMGPolar. Czarny geometry with **Take** and **Give** stencil implementation and problem size $\mathbf{6145 \times 8192}$.

types to obtain algorithmic speedups. Eventually, in Section 4.6, we provide some experimental results when using GMGPolar as a preconditioner for conjugated gradients.

4.1. Roofline model

In this first subsection, we provide a roofline model for GMGPolar. Our roofline model uses the benchmarked peak performance against the operational intensity. This model shows the hardware limitations and the potential benefit in optimizing computational or memory aspects of the considered implementations. We use LIKWID [22,23] for measuring double precision computations in MFLOPs/s, the memory bandwidth in MBytes/s as well as the operational intensity in FLOPs/Byte. We use the roofline model computed on a full node (56 cores) of an Intel Xeon "Skylake" Gold 6132, 2.60 GHz, with four 14-Core sockets. The peak flops of 1256.6 GFlops/s were quantified with the LIKWID benchmark *peakflops_avx* with settings *N:1792kB:56*. The value of 1792 kB was obtained from 32kB for each of the 56 hardware threads so that each vector chunk fits into the L1 cache of one core. The maximum memory bandwith of 272 GBytes/s was obtained with the LIKWID benchmark *stream_mem_avx_fma* with settings *N:2GB:56*. As a test case, we consider the Czarny geometry with a resolution of 6145×8192 nodes. As a convergence criterion, we choose a relative reduction of the initial residual by $1e8$.

As indicated by our first findings in [9,10] and although being a sparse matrix-free implementation, we see that GMGPolar has a rather elevated computational intensity and is not memory bound; cf. Fig. 8. Aside from the potential through AVX SIMD operations which is not yet exploited, we see that in particular the Give approach comes close to the compute limit. We thus see that the matrix-free implementation benefits from storing several expensive function evaluations on the transformed geometry.

4.2. Memory requirements

In this section, we consider the memory requirement of GMGPolar, comparing the novel implementation against the previous one. We compare the new Give implementation where either nothing (*Min. cache*), profile coefficients (*Coeff.*), geometry transformations (*Geom.*), or profile coefficients and geometry transformations (*Coeff. & Geom.*) are cached and with the Take approach where coeffi-

Table 1
Memory requirements of different novel GMGPolar implementations with comparison to old implementation. **Czarny** geometry from grid size $\mathbf{193 \times 256}$ to grid size $\mathbf{6\,145 \times 8\,192}$. n_r provides the resolution in the first dimension, n_ϑ provides the resolution in the second dimension. *Min. chache* stands for the evaluations of sine and cosine functions in one dimension which are always cached. *Coeff.* stands for caching of α and β evaluations from (1) and (13) and *Geom.* stands for caching transformation coefficients from (9).

$n_r \times n_\vartheta$	193×256	385×512	$769 \times 1\,024$	$1\,537 \times 2\,048$	$3\,073 \times 4\,096$	$6\,145 \times 8\,192$
GMGPolar v1						
Estimate (Min. cache)	4.74 MB	18.92 MB	75.60 MB	302.19 MB	1 208.35 MB	4 832.62 MB
Give (Min. cache)	4.96 MB	18.86 MB	73.96 MB	294.97 MB	N/A*	N/A*
GMGPolar v2						
Estimate (Min. cache)	2.64 MB	10.51 MB	42.00 MB	167.88 MB	671.31 MM	2 684.79 MB
Give (Min. cache)	3.22 MB	12.24 MB	47.56 MB	189.04 MB	754.03 MB	3 015.47 MB
Give (Coeff.)	3.22 MB	12.25 MB	47.58 MB	189.09 MB	754.13 MB	3 015.65 MB
Give (Geom.)	5.19 MB	20.08 MB	78.89 MB	314.19 MB	1 254.33 MB	5 016.03 MB
Give (Coeff. & Geom.)	5.20 MB	20.09 MB	78.90 MB	314.21 MB	1 254.38 MB	5 016.12 MB
Take (Coeff. & Geom.)	5.20 MB	20.09 MB	78.90 MB	314.21 MB	1 254.38 MB	5 016.12 MB

* This run was canceled / not executed as it took too long with the massif memory tool.

cients and geometries transformations are both cached; see the end of Section 3.1. We, furthermore, provide estimates on the expected memory requirements as laid out in Section 3.1. For these experiments, the direct solver MUMPS has been replaced by a custom-made solver as it could otherwise not be measured with valgrind's tool massif [24,25]. Furthermore, we replaced the right hand side by a constant vector equal to one and only three iterations were conducted. It has to be noted that the corresponding custom-made solver was not designed to replace MUMPS for a performance-oriented execution but only provides a fallback implementation. With this fallback implementation, the measured memory of the Give approach increases from approximately five vectors (on the finest level) to be stored to 5.6 vectors and for the Take approach from nine vectors to be stored to 9.4 vectors. An optimized version of the custom-build solver is already available with a new pull request of GMGPolar.

From Table 1, we see that the memory requirements of the Give approach were reduced by approximately 36 %, when compared to the prior implementation. On the other hand, the implementation of the Take approach in the novel version 2 comes close to the requirements of the prior Give implementation. We also see that the caching of the density profile coefficients from (1) and (13) is almost negligible with respect to memory requirements while the caching of the domain geometry transformations a^{rr}, $a^{r\vartheta}$, $a^{\vartheta\vartheta}$, and $\det DF_g$ from (9) leads to a relevant memory increase.

4.3. Weak scaling

In this section, we provide weak scaling experiments going from a single core to 64 cores on CARA. We start with a geometry of 769×1024 nodes and end with a grid of size 6145×8192, effectively scaling from approximately 800 000 to 50 million nodes. For the Give stencil implementation, we use the default setting of caching the profile coefficient values. For the Take implementation, profile coefficient and geometry values are cached. As a convergence criterion, we choose a relative reduction of the initial residual by $1e8$. From Table 2, we obtain weak scaling efficiencies of 25.64 % and 41.06 % for the Shafranov and Czarny geometry and the Give implementation. The corresponding values were 29.71 % and 46.14 % for a very similar test case in the previous implementation; cf. [9]. However, with the algorithm itself substantially sped up by a factor of two to four (cf. Fig. 10), the results appear acceptable. For the Culham geometry, we obtain a weak scaling efficiency of 72.95 % from one to 64 cores. From one to 16 cores, we obtain weak scaling efficiencies of 69.91 to 91.50 % for the three different geometries. For the Take implementation, we obtain worse weak scaling results but can substantially speed up the computation by factors of 2.7 to 14.6. As indicated by the roofline model, we see that the Take approach, where more information is stored in memory, is the more advantageous the more complex the geometry is.

4.4. Strong scaling

In this section, we first present strong scaling results for the novel GMGPolar implementation on different geometries. Eventually, we also provide a comparison with respect to scaling behavior of our prior implementation of [9,10]. As a convergence criterion, we choose a relative reduction of the initial residual by $1e8$. Our experiments were again run on CARA. We double the number of cores from one to 64 and consider the strong scaling behavior solver timings of GMGPolar v2 and, in comparison, setup and solver timings of the novel implementation against GMGPolar v1.

Table 2
Weak scaling of GMGPolar for different geometries and the two different stencil implementations. n_r provides the resolution in the first dimension, n_ϑ provides the resolution in the second dimension, *Cores* provides the numbers of cores used, *Time* provides the total solver time, *Eff.* provides the weak scaling efficiency computed with respect to the single core run on grid size **769 × 1 024** and 64 cores on **6 145 × 8 192**, and *G/T* provides the speedup obtained by using the Take approach instead of the Give approach.

Shafranov geometry								
$n_r \times n_\vartheta$	Cores	Stencil	Time	Eff.	Stencil	Time	Eff.	G/T
769 × 1 024	1	Give	32.42 s	100 %	Take	11.83 s	100 %	2.74
1 537 × 2 048	4		34.67 s	93.51 %		13.39 s	88.35 %	2.58
3 073 × 4 096	16		46.37 s	69.91 %		25.98 s	45.54 %	1.78
6 145 × 8 192	64		126.41 s	25.64 %		112.32 s	10.53 %	1.12
Czarny geometry								
$n_r \times n_\vartheta$	Cores	Stencil	Time	Eff.	Stencil	Time	Eff.	G/T
769 × 1 024	1	Give	41.77 s	100 %	Take	9.18 s	100 %	4.55
1 537 × 2 048	4		43.60 s	95.80 %		10.16 s	90.35 %	4.29
3 073 × 4 096	16		53.00 s	78.81 %		19.48 s	47.12 %	2.72
6 145 × 8 192	64		101.72 s	41.06 %		77.89 s	11.78 %	1.30
Culham geometry								
$n_r \times n_\vartheta$	Cores	Stencil	Time	Eff.	Stencil	Time	Eff.	G/T
769 × 1 024	1	Give	115.52 s	100 %	Take	7.90 s	100 %	14.62
1 537 × 2 048	4		118.38 s	97.58 %		8.64 s	91.43 %	13.70
3 073 × 4 096	16		126.26 s	91.50 %		15.40 s	51.29 %	8.19
6 145 × 8 192	64		158.36 s	72.95 %		63.29 s	12.48 %	2.50

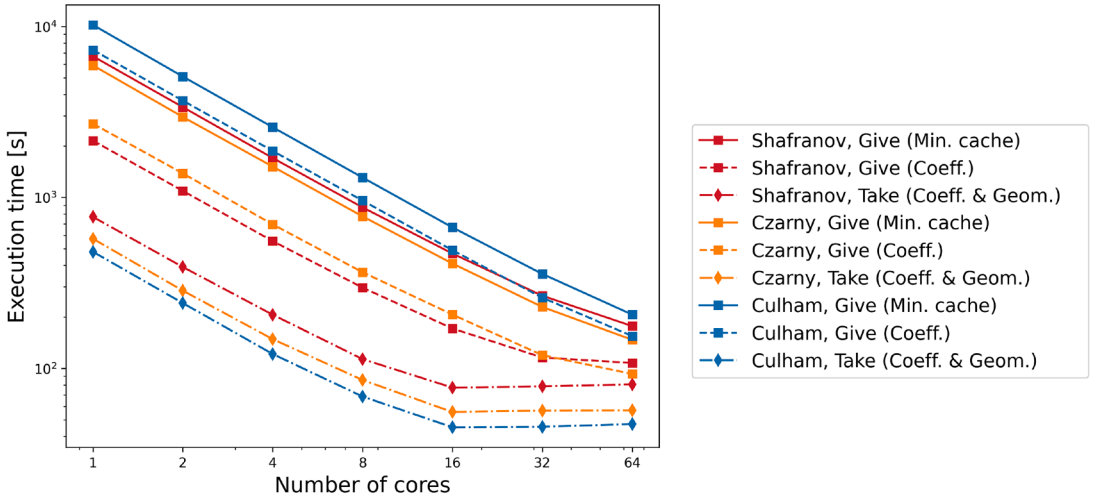

Fig. 9. Strong scaling and runtime comparison of GMGPolar for different geometries and stencil and caching implementations. Visualization of strong scaling for **Shafranov**, **Czarny**, and **Culham** geometry with **Give** and **Take** stencil implementations on problem size **6145 × 8192** with convergence criterion of a relative reduction of the initial residual by $1e8$.

In Fig. 9, we provide computing times and strong scaling of the novel GMGPolar on Shafranov, Czarny, and Culham geometry, respectively, with a grid size of 6145 × 8192, i.e., approximately 50 million degrees of freedom. We first see that additional caching of the profile coefficients in the Give approach, which only minimally increases the requirement memory (see Table 1), substantially reduces the runtime of the Give approach. Additionally, the Take approach, where coefficients and geometry information is stored, drastically reduces the runtime – at the cost of approximately 66 % of additional memory. However, we also see that the different geometries benefit differently from storing coefficient and geometry information. Intuitively, the more complex the geometry, the less beneficial is the sole caching of the coefficients, recomputing geometry-dependent values. For the faster Take implementation, we see that parallelization stagnates after 16 cores. This means that not enough data is available with 6145×8192 nodes for the compute parallelism offered through 32 to 64 cores and that, at best, four of these cross sections could be computed on a single node; cf. Fig. 1.

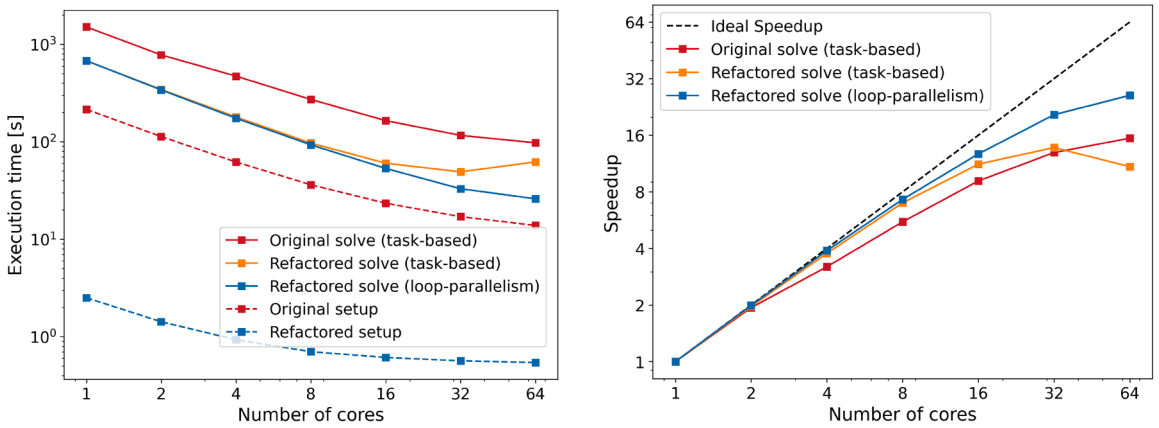

Fig. 10. Strong scaling and runtime comparison of old and new implementation of GMGPolar. Solver execution time in seconds (left) and speedup of solver times (right). **Czarny** geometry with **Give** stencil implementation and problem size $\mathbf{3073 \times 4096}$ with convergence criterion of a relative reduction of the initial residual by $1e8$.

Additionally, we compare old and new implementations on the Czarny geometry with a resolution of 3073×4096 nodes. We consider the Give stencil implementation which was chosen for Version 1 in [9]. In an intermediate step, we also compare the novel GMGPolar implementation with a task-based parallelism for the multigrid smoothers, which was implemented in Version 1. From Fig. 10, we see first that both, setup and solve, timings could be substantially reduced. For the setup phase we obtain a speedup of 86 and 26 for one and 64 cores, respectively. For the solve phase we obtain speedups of 2.2 and 3.8 for one and 64 cores respectively. We can furthermore state that the task-based parallelism reaches its limit of optimal performance with 16 to 32 cores and that the loop-based parallelism is better suited for the structured parallelism that we exploit in GMGPolar. With 66 % efficiency from one to 32 cores, the novel version scales very well for this test case.

4.5. Multigrid cycles, smoothing steps, and full multigrid initialization

After having considered scaling properties and computational performance in the previous sections, we now consider the novel algorithmic features available with GMGPolar v2. Therefore, we compare multigrid V−cycles with newly implemented W- and F-cycles. Furthermore, we compare methods initialized by zero with an FMG initialization with either 1, 2, or 3 cycles of type V, W, or F. As the Take implementation of the stencil performed best with respect to compute time, with reasonable increase in memory needs, we consider it here. As a test case, we consider the Czarny geometry with a resolution of 6145×8192 nodes. For a fair comparison with and without FMG, we require the iteration to reach an absolute residual of $1e-16$.

While we see from Table 3 that without FMG, the V-cycle with just one pre- and postsmoothing step performs best with respect to the solve time, we can speed up the algorithm by a factor of three when using FMG with two initial F-cycles.

Eventually, in Table 4, we provide the speedup for the Czarny geometry with a smaller grid size of approximately 780k nodes, as also considered in [10]. With the newly refactored GMGPolar, also using new multigrid features such as optimized initialization through FMG, we obtain substantial speedups compared to the old version. With the Give approach, the speedup ranges between four and seven (for one to 16 cores) and with the Take approach between 16 and 18 (for one to 16 cores). While the Take approach uses approximately the same amount of memory as the Give implementation in Version 1, the Give implementation of version 2 reduces the memory by approximately one third (36 %).

4.6. GMGPolar in preconditioned conjugate gradients

In our prior publications as well as in the previous sections, GMGPolar was used as a standalone multigrid solver. However, in order to speed up convergence through an optimized construction of iterates, we can also use Krylov subspace methods such as the conjugate gradient (CG) method. In this section, we present preliminary and experimental results using GMGPolar as a preconditioner for CG (PCG-GMGPolar). In this setting, we consider the system

$$M^{-1} A_{ex} u = M^{-1} f_{ex}, \tag{14}$$

where A_{ex} and f_{ex} are the extrapolated system and right hand side, as described briefly above and in more detail around [16, Eq. (4.10)] and [9, Eq. (32)].

In PCG, we first compute $r_0 = f_{ex} - A_{ex} u_0$ and then either apply M^{-1} or solve $M z_0 = r_0$ to obtain the preconditioned residual, which is used to initialize the search direction for solving (14). In our experiments, we use the nonextrapolated system matrix $M = A$ as the preconditioner, as it resulted in a shorter time-to-solution than $M = A_{ex}$. Within each PCG iteration, an approximate solution to $M z_k = r_k$ is obtained by performing a single FMG 1xF-cycle iteration. This provides a computationally efficient yet sufficiently accurate solution to the preconditioning step.

Table 3
Performance of different multigrid settings with and without full multigrid (FMG). **Czarny** geometry with **Take** stencil implementation and problem size $\mathbf{6145 \times 8192}$. The *Initial cycle* column provides the cycle used with FMG initialization with the corresponding number of iterations. The *Cycle* column provides the cycle used in the multigrid iteration with the number of pre- and post-smoothing iterations. The *its* column provides the number of iterations until convergence of the multigrid scheme. The *Time (Init)* column provides the time for the FMG initialization and the *Time (MG)* column provides the time of the multigrid scheme with convergence checks for an absolute residual smaller than $1e-16$.

Initialization	Initial cycle	Cycle	its	Time (Init)	Time (MG)
No FMG	–	**V(1,1)**	**48**	–	**77.64 s**
	–	W(1,1)	38	–	117.44 s
	–	F(1,1)	38	–	79.78 s
	–	V(2,2)	39	–	89.97 s
	–	W(2,2)	28	–	138.30 s
	–	F(2,2)	28	–	83.63 s
FMG	1x V	V(1,1)	31	1.50 s	36.21 s
	1x W	V(1,1)	21	4.24 s	33.81 s
	1x F	V(1,1)	21	2.59 s	33.99 s
	2x V	V(1,1)	18	3.50 s	28.12 s
	2x W	V(1,1)	13	8.39 s	21.02 s
	2x F	**V(1,1)**	**13**	**5.01 s**	**21.00 s**
	3x V	V(1,1)	15	5.29 s	24.44 s
	3x W	V(1,1)	12	12.62 s	19.31 s
	3x F	V(1,1)	12	7.48 s	19.53 s

Table 4
Speedup of the novel GMGPolar. **Czarny** geometry with **Take** and **Give** stencil implementation and problem size $\mathbf{769 \times 1\,024}$. Solver timings shown for 1, 4, and 16 cores for the novel GMGPolar (v2) compared to the prior version (v1) in seconds and speedup shown between parentheses in bold face.

Cores Version	1	4	16
v1, Give	83.96 s	27.81 s	13.79 s
v2, Give, FMG (2x F)	20.21 s (**4.15**)	5.61 s (**4.96**)	1.95 s (**7.07**)
v2, Take, FMG (2x F)	4.83 s (**17.38**)	1.64 s (**16.96**)	0.88 s (**15.67**)

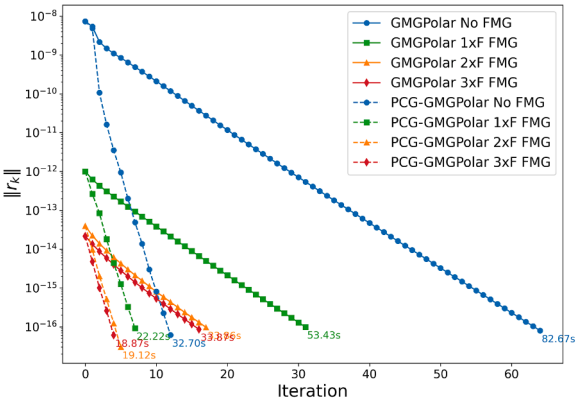

Fig. 11. Comparison of the experimental branch of standalone GMGPolar and GMGPolar in PCG. The plot shows the convergence behavior in terms of the norm of the residual and the number of PCG iterations. Solver execution time in seconds shown after the last iteration. **Shafranov** geometry with **Take** stencil implementation and problem size $\mathbf{6145 \times 8192}$ with convergence criterion of an absolute residual smaller than $1e-16$.

In the following, we present results for standalone GMGPolar and PCG-GMGPolar using an initial approximation that is either set to zero or obtained via 1, 2, or 3 FMG iterations prior to the start of the multigrid or CG iterations. In Fig. 11, we observe that the algorithm can achieve computational speedups of approximately 1.8 to 2.5, particularly when the initial FMG iterations alone are insufficient to reach the required tolerance.

Table 5
Speedup of the experimental branch of standalone GMGPolar and GMGPolar in PCG. **Czarny** geometry with **Take** stencil implementation and problem size $\mathbf{769 \times 1\,024}$. Solver timings shown for 1, 4, and 16 cores compared to the prior version (v1) in seconds and speedup shown between parentheses in bold face; cf. Table 4.

Cores Version	1	4	16
v2.2-experimental, Take, FMG (2x F)	4.08 s (**20.57**)	1.43 s (**19.45**)	0.78 s (**17.68**)
v2.2-experimental-PCG, Take, FMG (2x F)	2.26 s (**37.15**)	0.88 s (**31.60**)	0.55 s (**25.07**)

In contrast to the prior results which have been fully merged to the productive main branch, using GMGPolar with PCG is still experimental and available on development branch `v2_paper_conjugate_gradient`[1]. In between production version 2.0 and this branch some other minor adaptations and optimizations have taken place. A direct comparison with the prior release version has thus to be conducted with care. In Table 5, we have thus recomputed the row of the Take approach from Table 4 and additionally evaluated the PCG version of GMGPolar. First of all, we observe some minor speedups from approximately 16–17 before to 17–20 with the minor improvements. However, we also see that the additional speedup through PCG yields an overall speedup of more than 37 in serial execution and more than 25 in 16-core execution.

5. Conclusion

In this paper, we presented a completely refactored version of GMGPolar, an optimized state-of-the-art matrix-free multigrid solver that has been designed for complex 2D cross sections of tokamaks. GMGPolar has been developed to minimize the memory footprint, to allow fast and scalable execution for curvilinear coordinate representations and to achieve higher order approximations on tensor product structured grids.

With the improved stencil Give implementation, GMGPolar minimizes the already small memory requirements by reducing it by approximately 36 %. This yields an 8- to 14-fold reduction of memory compared to the spline solver, as demonstrated in [10], efficiently allowing to compute many different cross sections on a single compute node. With improved weak and strong scaling, both stencil implementations, Take and Give, realize substantial speedups compared to the prior implementation. For a use case as considered in [10], the Give implementation attains speedups between four and seven while the Take implementation attains speedups of approximately 16 to 18. In an experimental setting, we additionally considered GMGPolar as a preconditioner in the conjugate gradient method, which yielded an additional speedup factor of 1.8 to 2.5. In this experimental setup, we obtained speedups of more than 37 in serial execution and of more than 25 when executing on 16 cores.

While in [10], GMGPolar was found to represent "a compromise between relatively fast execution and high order of approximation", when compared to other state-of-the-art solvers, the novel version of GMGPolar combines an even reduced memory footprint with faster execution and better scalability. It can directly be used for domains without X-points, such as described in Fig. 7, or in a multipatch decomposition for a fast and precise computation on the core part of the domain as presented in [7]. In addition, GMGPolar's object-oriented redesign offers a more intuitive use of the taylored geometric multigrid for physicists and plasma fusion engineers. Future research will include porting the application to GPU accelerators and considering domains with X-points.

Data availability

Data generation scripts and program code is fully available on zenodo and github.

Acknowledgements & Funding

The authors gratefully acknowledge the scientific support and HPC resources provided by the German Aerospace Center (DLR). The HPC system CARA is partially funded by "Saxon State Ministry for Economic Affairs, Labour and Transport" and "Federal Ministry for Economic Affairs and Climate Action".

This project has received funding from the European High Performance Computing Joint Undertaking under grant agreement n°101144014.

Funded by the European Union. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the EuroHPC JU. Neither the European Union nor the granting authority can be held responsible for them.

[1] https://github.com/SciCompMod/GMGPolar/tree/paper_v2_conjugate_gradient

Appendix A

For potential users, we provide the most important simulation parameters in Table A.1.

Table A.1
Summary of simulation parameters with descriptions.

Type	Parameter	Description
General	`verbose`	Controls output verbosity.
	`paraview`	Enables Paraview output files.
	`maxOpenMPThreads`	Maximum OpenMP threads.
	`stencilDistributionMethod`	Stencil distribution: 'Take' or 'Give'.
	`cacheProfileCoefficients`	Caches profile coefficients α and β.
	`cacheDomainGeometry`	Caches transformation coefficients a^{rr}, $a^{r\theta}$ and $a^{\theta\theta}$.
	`DirBC_Interior`	Interior boundary condition: Across-the-origin or Dirichlet.
Polar Grid	`R0`	Generalized radius of the innermost circle.
	`Rmax`	Generalized radius of the outermost circle.
	`nr_exp`	Number of discretization points in radial dimension.
	`ntheta_exp`	Number of discretization points in angular dimension.
	`anisotropic_factor`	Anisotropic refinement radius.
	`divideBy2`	Refines grid globally `divideBy2` times to obtain identical grids for scaling experiments.
Multigrid Settings	`FMG`	Enables full multigrid / nested iteration for initial approximation.
	`FMG_iterations`	Number of FMG iterations.
	`FMG_cycle`	FMG Cycle type: V-, W-, or F-cycle.
	`extrapolation`	Extrapolation: None, implicit or full grid smoothing.
	`maxLevels`	Maximum multigrid levels.
	`preSmoothingSteps`	Pre-smoothing steps.
	`postSmoothingSteps`	Post-smoothing steps.
	`multigridCycle`	Multigrid Cycle type:V-, W-, or F-cycle.
	`residualNormType`	Residual norm type: $\|\cdot\|_2$, weighted $\|\cdot\|_{l_2}$, or $\|\cdot\|_\infty$.
	`maxIterations`	Maximum multigrid iterations.
	`absoluteTolerance`	Absolute tolerance for convergence.
	`relativeTolerance`	Relative tolerance for convergence.
Test Problem	`geometry`	Cross section shape: Shafranov, Czarny, Culham, etc.
	`alpha_jump`	Radius of rapid decay for density profile.
	`kappa_eps`	Geometry elongation.
	`delta_e`	Outward radial displacement of flux center.
	`problem`	Defines the solution: Cartesian, Polar, Multi-scale.
	`alpha_coeff`	Alpha coefficient: Poisson, Sonnendrucker, Zoni.
	`beta_coeff`	Beta coefficient: Zero or inverse of `alpha_coeff`.

References

[1] M. Maurer, A. Bañón Navarro, T. Dannert, M. Restelli, F. Hindenlang, T. Görler, D. Told, D. Jarema, G. Merlo, F. Jenko, GENE-3D: a global gyrokinetic turbulence code for stellarators, J. Comput. Phys. 420 (2020) 109694. https://doi.org/10.1016/j.jcp.2020.109694

[2] M. Dorf, M. Dorr, Progress with the 5D full-F continuum gyrokinetic code COGENT, Contrib. Plasma Phys. 60 (5-6) (2020) e201900113. https://doi.org/10.1002/ctpp.201900113

[3] R. Hatzky, T.M. Tran, A. Könies, R. Kleiber, S.J. Allfrey, Energy conservation in a nonlinear gyrokinetic particle-in-cell code for ion-temperature-gradient-driven modes in θ-pinch geometry, Phys. Plasmas 9 (3) (2002) 898–912. https://doi.org/10.1063/1.1449889

[4] S. Jolliet, A. Bottino, P. Angelino, R. Hatzky, T.M. Tran, B.F. Mcmillan, O. Sauter, K. Appert, Y. Idomura, L. Villard, A global collisionless PIC code in magnetic coordinates, Comput. Phys. Commun. 177 (5) (2007) 409–425. https://doi.org/10.1016/j.cpc.2007.04.006

[5] V. Grandgirard, J. Abiteboul, J. Bigot, T. Cartier-Michaud, N. Crouseilles, G. Dif-Pradalier, C. Ehrlacher, D. Esteve, X. Garbet, P. Ghendrih, et al., A 5D gyrokinetic full-f global semi-Lagrangian code for flux-driven ion turbulence simulations, Comput. Phys. Commun. 207 (2016) 35–68. https://doi.org/10.1016/j.cpc.2016.05.007

[6] N. Bouzat, C. Bressan, V. Grandgirard, G. Latu, M. Mehrenberger, Targeting realistic geometry in Tokamak code Gysela, ESAIM Proc. Surv. 63 (2018) 179–207. https://doi.org/10.1051/proc/201863179

[7] P. Vidal, E. Bourne, V. Grandgirard, M. Mehrenberger, E. Sonnendrücker, Local cubic spline interpolation for Vlasov-type equations on a multi-patch geometry, (2025). https://doi.org/10.48550/arXiv.2505.22078

[8] M.J. Kühn, C. Kruse, U. Rüde, Implicitly extrapolated geometric multigrid on disk-like domains for the gyrokinetic Poisson equation from fusion plasma applications, J. Sci. Comput. 91 (1) (2022) 1–27. https://doi.org/10.1007/s10915-022-01802-1

[9] P. Leleux, C. Schwarz, M.J. Kühn, C. Kruse, U. Rüde, Complexity analysis and scalability of a matrix-free extrapolated geometric multigrid solver for curvilinear coordinates representations from fusion plasma applications, J. Parallel Distrib. Comput. (2025) 105143. https://doi.org/10.1016/j.jpdc.2025.105143

[10] E. Bourne, P. Leleux, K. Kormann, C. Kruse, V. Grandgirard, Y. Güclü, M.J. Kühn, U. Rüde, E. Sonnendrücker, E. Zoni, Solver comparison for Poisson-like equations on tokamak geometries, J. Comput. Phys. 488 (2023) 112249. https://doi.org/10.1016/j.jcp.2023.112249

[11] J. Litz, P. Leleux, C. Kruse, U. Rüde, M.J. Kühn, GMGPolar v2.0.1, Zenodo, 2025. https://doi.org/10.5281/zenodo.15732483

[12] E. Zoni, Y. Güçlü, Solving hyperbolic-elliptic problems on singular mapped disk-like domains with the method of characteristics and spline finite elements, J. Comput. Phys. 398 (2019) 108889.

[13] E. Zoni, Theoretical and Numerical Studies of Gyrokinetic Models for Shaped Tokamak Plasmas, Ph.D. Thesis, Technische Universität München, 2019.

[14] O. Czarny, G. Huysmans, Bézier surfaces and finite elements for MHD simulations, J. Comput. Phys. 227 (16) (2008) 7423–7445.

[15] J.W. Connor, S.C. Cowley, R.J. Hastie, T.C. Hender, A. Hood, T.J. Martin, Tearing modes in toroidal geometry, Phys. Fluids 31 (3) (1988) 577–590.

[16] M.J. Kühn, C. Kruse, U. Rüde, Energy-minimizing, symmetric discretizations for anisotropic meshes and energy functional extrapolation, SIAM J. Sci. Comput. 43 (4) (2021) A2448–A2473. https://doi.org/10.1137/21M1397520
[17] S.R.M. Barros, The Poisson equation on the unit disk: a multigrid solver using polar coordinates, Appl. Math. Comput. 25 (2) (1988) 123–135.
[18] J. Litz, Parallel Matrix-Free Computation of the Gyrokinetic Poisson Equation from Fusion Plasma Applications using Extrapolated Geometric Multigrid, 2025. University of Bonn. URL: https://elib.dlr.de/214029/.
[19] U. Trottenberg, C.W. Oosterlee, A. Schüller, Multigrid, Academic Press, London San Diego, London San Diego, 2001.
[20] W. Hackbusch, Multi-Grid Methods and Applications, 4 of *Springer Series in Computational Mathematics*, Springer Berlin / Heidelberg, Berlin, Heidelberg, 1 edition, Berlin, Heidelberg, 1985.
[21] P. Amestoy, C. Ashcraft, O. Boiteau, A. Buttari, J.-Y. L'Excellent, C. Weisbecker, Improving multifrontal methods by means of block low-rank representations, SIAM J. Sci. Comput. 37 (3) (2015) A1451–A1474. https://doi.org/10.1137/120903476
[22] G. Hager, G. Wellein, J. Treibig, LIKWID: a lightweight performance-oriented tool suite for x86 multicore environments, in: 2012 41st International Conference on Parallel Processing Workshops, IEEE Computer Society, Los Alamitos, CA, USA, 2010, pp. 207–216. https://doi.org/10.1109/ICPPW.2010.38
[23] T. Gruber, J. Eitzinger, G. Hager, G. Wellein, LIKWID v5.2.2, Zenodo, 2022. https://doi.org/10.5281/zenodo.4275676
[24] N. Nethercote, J. Seward, Valgrind: a framework for heavyweight dynamic binary instrumentation, ACM Sigplan Not. 42 (6) (2007) 89–100.
[25] N. Nethercote, R. Walsh, J. Fitzhardinge, Building workload characterization tools with valgrind, in: 2006 IEEE International Symposium on Workload Characterization, IEEE, 2006, pp. 2.