E-Infrastructures

H2020-INFRAEDI-2018-1

INFRAEDI-2-2018: Centres of Excellence on HPC

EoCoE-II

Energy oriented Center of Excellence: toward exascale for energy

Grant Agreement Number: INFRAEDI-824158

D3.3

Updated results and new releases of LA solvers

Project and Deliverable Information Sheet

EoCoE	Project Ref:	INFRAEDI-824158
	Project Title:	Energy oriented Centre of Excellence
	Project Web Site:	http://www.eocoe.eu
	Deliverable ID:	D3.3
	Lead Beneficiary:	ULB
	Contact:	Yvan Notay
	Contact's e-mail:	ynotay@ulb.ac.be
	Deliverable Nature:	Report
	Dissemination Level:	PU*
	Contractual Date of Delivery:	M24 30/06/2021
	Actual Date of Delivery:	M24 30/06/2021
	EC Project Officer:	Evangelia Markidou

* - The dissemination level are indicated as follows: PU – Public, CO – Confidential, only for members of the consortium (including the Commission Services) CL – Classified, as referred to in Commission Decision 2991/844/EC.

Document Control Sheet

Document	Title :	Updated results and new releases of LA solvers
	ID :	D3.3
	Available at:	http://www.eocoe.eu
	Software tool:	LaTeX
Authorship	Written by:	Yvan Notay
	Contributors:	Abdeselam El-Haman Abdeselam, Daniele Bertaccini, Siham Boukhris, Pasqua D'Ambra, Vincent Darrigrand, Fabio Durastante, Salvatore Filippone, Philippe LELEUX, Herbert Owen
	Reviewed by:	Mathieu Lobet, Andrea Quintiliani

Contents

List of Figures

List of Tables

1. Introduction

Solving Linear Algebra (LA) problems is a main computational kernel in three out of five EoCoE II Scientific Challenges (SC) and thus the availability of exascale-enabled LA solvers is fundamental in preparing the SC applications for the new exascale ecosystem. More specifically, "LA problem" refers here to the solution of systems of algebraic linear equations, with numbers of unknowns and equations that are increasingly larger going towards exascale.

The goal of WP3 is to design and implement exascale-enabled LA solvers for the selected applications and to integrate them into the flagship codes.

This deliverable is not only a report, but also refers *Code Demonstrators* for LA solvers. For general purpose solvers, this is in fact quite straightforward: they all have a dedicated Web page from where the code can be obtained and tested. In Section 3 below, we provide the list of these Web pages.

Besides, within the framework of this project, some solvers are specifically developed for a partner application code, which we sometimes refer to (perhaps somehow extending the usual concept) as co-design (in the sense that the design result from close collaboration between LA experts and applications developers). For these solvers, having a general distribution makes little sense since they cannot be used outside the application they have been developed for. Moreover, as seen below (Sections 5.1 and 5.2), the required efforts in terms of design and testing makes logically that software developments are less advanced, and therefore not mature enough to be publicly released.

Update results are reported in Sections 4–7. More precisely, Sections 4–6 report progress made in the three SC which involve LA solvers: Water (Section 4), Fusion (Section 5) and Wind (Section 6), while Section 7 gathers progress in transversal activities. Note that most reported results are complementary to those presented in previous deliverables [55, 28], and this deliverable should therefore not be seen as a comprehensive summary of all activities so far.

The grant agreement also refers some WP3 task for the Material SC ("Task 3.1: Linear Algebra solvers for Materials"), which does not appear in the above list. The reason is the following. The task planned in the proposal has eventually been canceled because the concerned flagship code (PVnegf) was changed during the first half of the project while the FZJ partner lost a resource. The new flagship code libNEGF, which replaces PVnegf, has no focus on the use of LA solver and no WP3 activities are planned for it. The very limited resources allocated to this task within WP3 (about 0.5 PM) has eventually been spent on working on other SC, in particular for Alya code (Wind SC).

Considering the deliverable contents more in detail, one noticeable fact is that the CNR Partner has merged, improved and extended its solvers which are now distributed through the package PSCToolkit. This is reported in transversal activities (Section 7.1) but has also an impact in SC that uses these solvers, namely Water (see Section 4.1) and Wind (see Section 6.2). On the other hand, progress made with the integration of AGMG in SHEMAT are presented in Section 4.2, while Section 6.1 reports on the Integration of MUMPS in Alya. Sections 5.1 and 5.2 report interesting, yet preliminary, results obtained

with solvers specifically designed for the considered application: namely, the solver Gmgpolar developed for the code GyselaX (Section 5.1) and the solver HyTeG developed for the code SOLEDGE3X (Section 5.2). Regarding transversal activities, besides the one already mentioned (Section 7.1), Section 7.2 reports on the integration of MUMPS within the HPDDM package, while Sections 7.3 and 7.4 present how AGMG has been ported to GPUs, considering two different but complementary approaches. Eventually, a short summary is given in Section 8.

2. Acronyms

Acronym	Partner and institute
BSC:	Barcelona Supercomputing Center
CEA:	Commissariat à l'énergie atomique et aux énergies alternatives
CNR:	Consiglio Nazionale delle Ricerche
CNRS:	Centre Nationale de la Recherche Scientifique
CERFACS:	Centre Européen de Recherche et de Formation Avancée en Calcul Scientifique
FZJ:	Forschungszentrum Jülich GmbH
INRIA:	Institut National de Recherche en Informatique et en Automatique
IRIT:	Institut de Recherche en Informatique de Toulouse
IRFM:	Institute for Magnetic Fusion Research
MPG:	Max-Planck- Gesellschaft zur Förderung der Wissenschaften e.V
RWTH:	Rheinisch-Westfälische Technische Hochschule Aachen, Aachen University
ULB:	Université Libre de Bruxelles
UNITOV:	University of Rome Tor-Vergata

Acronym	Software and codes
	Application codes
Alya:	High Performance Computational Mechanics
libNEGF:	General library for Non Equilibrium Green's Functions
GyselaX:	GYrokinetic SEmi-LAgrangian
ParFlow:	Parallel Flow
SHEMAT:	Simulator of HEat and MAss Transport
SOLEDGE3X:	Transport and turbulence in the edge plasma of tokamaks
TOKAM3X:	Transport and turbulence in the edge plasma of tokamaks
	LA software libraries
AGMG:	Iterative solution with AGgregation-based algebraic MultiGrid [53]
AMG4PSBLAS:	Algebraic MultiGrid for PSBLAS [22]
Fabulous:	Fast Accurate Block Linear Krylov Solver [30]
MaPHyS:	Massively Parallel Hybrid Solver [50]
MUMPS:	MUltifrontal Massively Parallel sparse direct Solver [51]
PaStiX:	Parallel Sparse matriX package [59]
PSBLAS:	Parallel Sparse Basic Linear Algebra Subroutines [33]
PSCToolkit:	Parallel Sparse Computation Toolkit [22]
HPDDM:	High-Performance unified framework for Domain Decomposition Methods [39]

3. Code demonstrators

	Location of code demonstrators for LA packages
AGMG:	http://agmg.eu
AMG4PSBLAS:	https://psctoolkit.github.io/
Fabulous:	https://gitlab.inria.fr/solverstack/fabulous
MaPHyS:	https://gitlab.inria.fr/solverstack/maphys
MUMPS:	http://mumps-solver.org/
PaStiX:	https://gitlab.inria.fr/solverstack/pastix
PSBLAS:	https://psctoolkit.github.io/
PSCToolkit:	https://psctoolkit.github.io/
HPDDM:	https://github.com/hpddm/hpddm

4. Task 3.2: Linear Algebra solvers for Water

4.1 PSCToolkit in ParFlow

Partners: CNR, UNITOV, FZJ
Software packages: PSCToolkit, AMG4PSBLAS, PSBLAS, ParFlow,

During the first phase of the project, as reported in the Deliverable 3.1 [55], we developed the software interface between, PSBLAS (rel. 3.6), PSBLAS-EXT (rel. 1.2), and its sibling preconditioner package MLD2P4 (rel. 2.2) with the SUNDIALS/KINSOL package that is used in turn inside Parflow to approach the solution of the nonlinear systems arising from the discretization of hydro-geological models. Due to the release of the new version of our software packages, please refer to the details about the new versions discussed in Section 7, including the new release of PSBLAS and the new preconditioner package AMG4PSBLAS, we updated the set of interfaces to encompass these new developments. Furthermore, we have updated the version of the SUNDIALS/KINSOL library we interface to the Version 5.4.0 (GitHub repository github.com/psctoolkit/psctoolkit).

In the following, we discuss the first preliminary results obtained within the KINSOL interface for the solution of the discretized Richards equation for the simulation of groundwater flow in the unsaturated zone. We stress that this is indeed one of the main models used in ParFlow.

Test Case Description

For complying with the procedure adopted in ParFlow to discretize *the mixed form* of Richards equation [32], we consider here a cell-centered finite difference approximation on a regular tensor mesh in a three-dimensional parallelepipedal domain Ω of size $[0, L_x] \times [0, L_y] \times [0, L]$, averaging for interface values the hydraulic conductivity by upstream and harmonic means. To completely specify the model, we adopt the Van Genuchten choice for both water content and hydraulic conductivity functions from [65]. The test case is then a wetting problem in which we apply water at height $z = L$ such that the pressure-head becomes zero in a square region at the center of the top-layer and is fixed to the same constant negative value on all the remaining boundaries.

An important feature of the sequence of Jacobians $\{J_{\mathbf{N}}\}_{\mathbf{N}}$ produced by this discretization strategy is that we can predict its asymptotic spectral properties, formally this means that we can compute a measurable function $f : D \subset \mathbb{R}^3 \to \mathbb{C}$ such that

$$\lim_{\mathbf{N}\to\infty} \frac{1}{N} \sum_{i=1}^{N} F(\lambda_i(J_{\mathbf{N}})) = \frac{1}{\mu_3(D)} \int_D F(f(\mathbf{x}))\mathrm{d}\mathbf{x}, \qquad \forall\, F \in C_c(\mathbb{C}),$$

for $\mu_3(\cdot)$ represent the Lebesgue measure on $\mathbb{R}^3$, and $C_c(\mathbb{C})$ is the space of continuous functions with compact support. Informally, this means that if we assume that N is large enough, then the eigenvalues $\lambda_i(J_{\mathbf{N}})$ of the matrix $J_{\mathbf{N}}$, except possibly for $o(N)$ outliers, are approximately equal to the samples of f over a uniform grid in D. What we can prove is that such function f is determined only by the terms in the Richards equation that are relative to the Darcy flux of the Jacobians, i.e., it is the same as an opportune symmetric positive definite matrix [12, 13, 14]. This means that in principle one can simplify the build phase of the preconditioner by working only on this auxiliary sequence. Indeed such

observation had already been empirically made [40], what we have achieved here is giving a theoretical foundation to it.

Algorithmic scalability

In this phase, we have focused on testing the algorithmic scalability of some of the preconditioners available in the PSCToolkit. For this task, we adopted a weak scalability analysis in the same fashion of [19]. This means that we fix the number of degree of freedoms for a processor to be $\mathbf{N}(k) = (2^k N_x, 2^k N_y, N_z)$ for $np = 4^k$ processors, $k = 0, \ldots, 5$, for $N_x = N_y = 50$, and $N_z = 40$, for a domain $\Omega(k) = [0, 2^k \times 4.0]^2 \times [0, 2.5]$ and $N_t = 10$ dealing with a global mesh size ranging between 10^5 to 0.1×10^9. This means that the number of time steps N_t is fixed independently from the number of processes np, thus the analysis is done on the quantities averaged on the number of time steps relative to the given np. The experiments are run on the SuperMUC-NG machine, ranked 15th in the November 2020 TOP500 list[1], having 311,040 compute cores (Intel Xeon Skylak' processors) with the main memory of 719 TB, a peak performance of 26.9 PetaFlop/s, and an OmniPath network with 100 Gbit/s.

We solve the associated linear systems employing the GMRES(10) algorithm to a tolerance that is automatically tuned by the Newton implementation in SUNDIALS to avoid over–solving, heuristically, this means that the tolerance becomes stricter as we move toward the solution of the non-linear system at the given time-step. The Newton method automatically tries to reduce the request of new Jacobians by adopting a reuse strategy. This means that we compute a new preconditioner only when a new Jacobian is requested.

We tested two Algebraic Multigrid preconditioners that we identify with the acronyms relative to the underlying aggregation scheme, that are

VSBM standing for the decoupled smoothed aggregation by Vaněk, Mandel, Brezina, and that is built on each new Jacobian produced by the Newton method;

VSMATCH3 the smoothed aggregation preconditioner based on three sweeps of parallel-coupled weighted matching, that is built on the part relative to the Darcy flux of the Jacobians, i.e., on a symmetric positive definite matrix, as suggested by the spectral properties of the Jacobian matrices we developed in [12, 13, 14] and that we have briefly reminded in the test case description.

Both V-cycle preconditioners are then completed by a single sweep of a Hybrid-FBGS as pre/post-smoother. As discussed in the scalability analysis in [22], we reduce the size of the matrix on the coarse grid in such a way as to have 200 equations per core. Then, for the VSBM preconditioner, we employ 30 iterations of a Block-Jacobi method with the ILU(0) factorization as a coarse solver, while we adopt the previous as a preconditioner for a PCG algorithm on the coarse grid for the VSMATCH3 approach. From the results collected in Table 1, we observe that both the strategies do manage in keeping the number of linear iteration fixed. In the VSMATCH3 this is interpreted in two ways, firstly we have an empirical confirmation of the fact that the spectral information of the matrix sequence $\{J_{\mathbf{N}}\}_{\mathbf{N}}$ are captured by the symmetric approximation, secondly, we observe that

[1] `https://www.top500.org`

np	Average Number of Newton Iterations	Number of Jacobians	Average Number of Linear Iterations per Newton Iteration	
			VSMATCH3	VSBM
1	3	11	63	35
4	3	12	65	44
16	3	12	62	43
64	3	12	59	34
256	3	12	58	34
1024	3	12	56	33

Table 1: Algorithmic scalability for the Richards test problem.

the multigrid hierarchy, in turn, gives a reliable approximation of the effect of inverting the latter sequence. Furthermore, we observe that the spectral approximation underlying the VSMATCH3 strategy does get better in terms of approximation properties as we increase the problem size highlighting the effectiveness of the asymptotic analysis. The second observation, is then that also the VSBM strategy when directly applied to the sequence of the Jacobian performs satisfactorily, the smaller number of iterations for the VSMATCH3 case is explained by the fact that we are applying one less level of approximation for the sequence of matrices. Nevertheless, this latter strategy is applicable only in the case in which we discretize also the transport term in the Richards equations through *centered* differences or, equivalently, when we have a pattern symmetric sequence of Jacobians that is not always the case.

To push further this analysis we still need to investigate *update* and *reuse* strategy for the two preconditioners. This would reduce the number of full preconditioners builds and thus enhance the overall parallel efficiency of the code. After experimenting a successfull preconditioner *update* and *reuse* strategy, our code will be tested on larger mesh size and number of cores, as well as on hybrid architectures embedding GPUs.

4.2 AGMG for the SHEMAT-Suite

Partners: RWTH, ULB
Software packages: AGMG, SHEMAT

The initial plan was to make an interface between AGMG and PETSc[2], an US package that contains various solvers. This package is used by PETSHEM, a code for multi-phase and multi-components flow simulations originating from SHEMAT. This way AGMG would be available to PETSHEM as well. However, after discussion between the partners, it has been found both more productive and more coherent with respect to the objectives of the scientific tasks to directly interface AGMG within the SHEMAT-Suite for single-phase heat flow simulations in porous media.

In this way, a FORTRAN interface for AGMG has been integrated in the SHEMAT-Suite code. The goal is to test AGMG as an alternative solver for solving conductive heat flow problems in geothermal applications; the current solver is BiCGStab with ILU

[2] https://www.mcs.anl.gov/petsc/

preconditioning.

A first application to a 2D conductive heat flow model shows that AGMG can successfully solve the problem. Moreover, despite the modest size of the problem, AGMG already offers significant saving: the nonlinear model was solved using only globally 64 linear iterations with AGMG, whereas 1480 linear iterations are needed when using the BiCGStab solver with ILU preconditioner. Regarding the computing time, it is decreased by a factor of about 7.

Moreover, in this very preliminary test, AGMG still uses a vanilla stopping criterion whereas the native solver benefits from a tuned criterion that takes into account the progress of the Newton outer iteration. More results will be reported in the next deliverable after the needed adaptations are made to get the full potentialities of AGMG.

The next step is a detailed comparison with the BiCGStab solver and performance evaluation. For the latter we will use a large-scale three-dimensional subsurface heat flow model. Depending on its performance, we will consider including AGMG for production runs.

5. Task 3.3: Linear Algebra solvers for Fusion

5.1 Geometric Multigrid Solver for Plasma Fusion Simulations in GyselaX

Partners: CERFACS, MPG-IPP, IRFM-CEA
Software packages: Gmgpolar, GyselaX

Several meetings have been held between MPG-IPP, IRMF-CEA and CERFACS in order to design a scalable geometric multigrid solver for the solution of the 2D quasi-neutrality equation defined on a deformed circular geometry arising in the GyselaX code [18, 36]. Our developed code Gmgpolar is based on a co-design effort aimed at providing optimal cost efficiency for the respective class of problems. The principles behind this multigrid solver were detailed in the previous deliverable [28] and the references [43, 44].

Further regular meetings have taken place between these teams since the beginning of 2021. The goal of these meetings was to establish an extensive comparison between three different solvers: Gmgpolar [44], the parallel framework AMRex [71], and a spline based approach [72]. The main interest of this comparison is to establish the advantages and disadvantages of each method, in terms of accuracy, convergence, and theoretical computational and memory complexity, in order to integrate them in the plasma simulation code depending on its actual requirements.

As first essential step towards this comparison, we carried out a detailed analysis of the computational and memory complexity for the Gmgpolar solver. In particular, we have demonstrated that the asymptotic complexity of the solver is optimal in the sense that it is only growing linearly with the problem size.

The problem setting

We are interested in the solution of the gyrokinetic Poisson equation which is simplified here by

$$\begin{aligned} -\nabla \cdot (\alpha \nabla u) &= f \text{ in } \Omega, \\ u &= 0 \text{ on } \partial\Omega, \end{aligned} \tag{1}$$

where $f : \Omega \to \mathbb{R}$, and $\alpha : \Omega \to \mathbb{R}$ is a coefficient corresponding to a *density profile*. The domain Ω is a disk-like domain described with curvilinear coordinates, with *right hand side* (RHS) f. These coordinates are based on an invertible mapping from the Cartesian coordinates x, y to the generalized polar coordinates $r, \theta \in [R_0, R] \times [0, 2\Pi)$ where r is the (generalized) radius and θ the angle. We focus here on the so called *Target* geometry as defined in [18], i.e. a stretched ellipse. The polar plane is divided in n_r and n_θ nodes in the respective directions with a possible non-uniform refinement in the r direction to accurately capture the strong variations in the coefficient of the gyrokinetic equation.

The multigrid solver

In these coordinates, the *partial differential equation* (PDE) is discretized using a 9-point finite difference stencil as described in [43]. The discretization is designed to handle the specific grid, and the strong anisotropy introduced by the curvilinear coordinates. We obtain the operator $A \in \mathbb{R}^{m \times m}$, where $m = n_r \cdot n_\theta$. In order to overcome memory issues when considering large scale problems, Gmgpolar follows a matrix-free implementation [9], i.e., the discretized operator is not stored in memory but rather applied on-the-fly using the stencil representation given in Figure 1. Based on this stencil, the finite-differences scheme induces the computation of the Jacobian of the mapping, i.e., of three functions a^{rr}, $a^{r\theta}$, and $a^{\theta\theta}$, which correspond respectively to the radial, polar, and diagonal updates in the stencil. In the representation of the stencils in Figure 1, we display for each update the computationally significative functions applied to the neighboring nodes, i.e. either a^{rr}, $a^{r\theta}$, or $a^{\theta\theta}$. The central update a^c is a sum containing all 3 functions. The RHS $f \in \mathbb{R}^m$ is constructed and kept in memory.

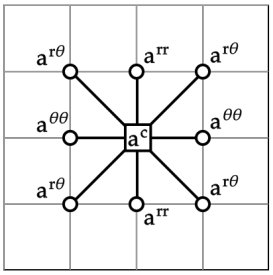


Figure 1: 9-point stencil for the operator A. We neglect the boundaries $r = R_0$ and $r = R$. For each update, we give which function a^{rr}, $a^{r\theta}$, or $a^{\theta\theta}$ needs to be computed.

In order to use a geometric multigrid scheme, we then construct L grid levels using standard coarsening of the initial grid. Level $l = 0$ is the finest, then $m_0 = m$ and $m_l \approx m/4^l$ in our 2D problem. Between two consecutive grid levels, the prolongation operator $P_{l+1}^l \in \mathbb{R}^{l \times l+1}$ is defined as the bilinear interpolation taking into account the unstructured

grid. Again, the prolongation is not stored but applied in the matrix-free implementation, following the stencil shown in Figure 2. We still need to store the operator on the coarsest grid A_L for the coarse grid correction obtained with a sparse direct solver.

In this study, we do not incorporate yet the implicit extrapolation [41] and use a classical multigrid V-cycle in Gmgpolar, with ν_1 and ν_2 iterations or pre- and post-smoothing. The implicit extrapolation should not result in any significant computational overhead, but should improve significantly the order of approximation. Note that this feature of the approach chosen, i.e. to deliver higher order at only minimally raised cost, will show its advantages in a later stage of the project.

The algorithm for the V-cycle is given in Algorithm 1. In this algorithm, $\mathcal{S}_\nu(u, A, f)$ is the application of ν relaxations. In the following, we drop the notation l and only specify the level when really needed, e.g. P_{l+1}^l is simply noted P and A_L is the coarse grid operator.

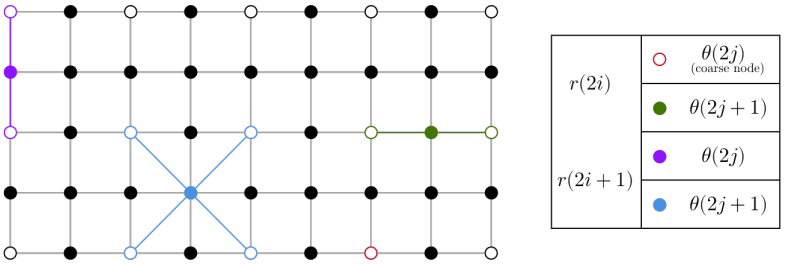


Figure 2: Standard coarsening of the (nested) grids: empty circles are the coarse nodes, plain circles are the fine nodes. We give the stencil for the prolongation operator depending on its neighboring nodes. In order, there are 4 types of nodes with the corresponding stencils: injected coarse nodes, fine nodes with neighbor coarse nodes in the radial or in the polar direction, and fine nodes with neighbor coarse nodes only in the diagonals. We neglect the boundaries $r = R_0$ and $r = R$.

Algorithm 1 MG(u, l): Multigrid V-cycle with ν_1 pre-smoothing and ν_2 post-smoothing steps on level l (L levels with $l = 0 \equiv$ finest)

1: $\mathcal{S}_{\nu_1}(u, A, f)$
2: residual: $r = P^T(f - Au)$
3: **if** l = L **then**
4: return $A^{-1}r$
5: **else**
6: $e_c = MG(r, l + 1)$
7: **end if**
8: $\mathcal{S}_{\nu_2}(Pe_c, A, f)$

The main contribution from [44] is the smoother that is specifically constructed to provide optimal convergence rates at minimal cost for the disk-like domain. The literature on multigrid methods designed for such a type of domain is sparse. In [4], one of the few references, smoothing factors are estimated for different zebra line smoothers. In accordance with multigrid theory it was observed that in the interior, i.e. towards the center point, circle smoothers are best while radial smoothers are best in the exterior. Based on these results, the chosen smoother in Gmgpolar is a combination of zebra circle line smoother in the interior, and zebra radial line smoother in the exterior. We employ a

switch between the two of them when the radius r_{switch} is such that

$$\frac{k}{h_{switch}} r_{switch} > 1, \tag{1}$$

where k is the interval size in the polar direction (uniform), and h_{switch} in the radial direction. Figure 3 shows the domain split in two and colored in black and white for each zebra relaxation. Let us define $\mu = r_{switch}/R$, then there are respectively $m_{Circle,c} = \mu m/2$ and $m_{Radial,c} = (1-\mu)m/2$ nodes for the radial and circle smoothers with color $c \in \{\text{Black, White}\}$. Since we use a 9-point stencil of length one, black lines are pair-wise independent and so are white lines for both smoothers. The algorithm for the smoother is given in Algorithm 2. In this algorithm, $A_{S,c} \in \mathbb{R}^{m_{S,c} \times m_{S,c}}$, and $u_{S_c}, f_{S_c} \in \mathbb{R}^{m_{S,c}}$ represent the restriction of the operator A and the vector u on the degrees of freedom corresponding to smoother S and color c. $A^{\perp}_{S,c} \in \mathbb{R}^{m_{S,c} \times (m - m_{S,c})}$ and $u^{\perp}_{S,c} \in \mathbb{R}^{m - m_{S,c}}$ are the complement of the matrix, i.e. the remaining part of the rows of A for this smoother, and the remaining part of u. The matrices $A^{\perp}_{S,c}$ are applied in a matrix-free implementation. The complementary stencils corresponding to these matrices are given in Figure 3. Again, we display in the stencils only the computationally significant functions applied to the neighboring nodes for each update. In the code, a system based on the matrices $A_{S,c}$ need to be solved for each relaxation sweep, thus they are stored in a sparse format. These matrices are block diagonal where each block corresponds to a line and has the structure given in Figure 4. Given that in Algorithm 2, 1) the applications of all smoothers S, c are independent, and 2) the lines with same color are independent, the solver has a very high potential for parallelism.

Again we wish to point out the advantages of this problem-specific co-design effort. They will become essential in later stages of the project. The particular algorithms have been developed since they are not only well suited for vectorization and for exploiting instruction-level parallelism, but they will also be well suited for future accelerator-based architectures. Thus the co-design here is a key step towards performance portability for future architectures.

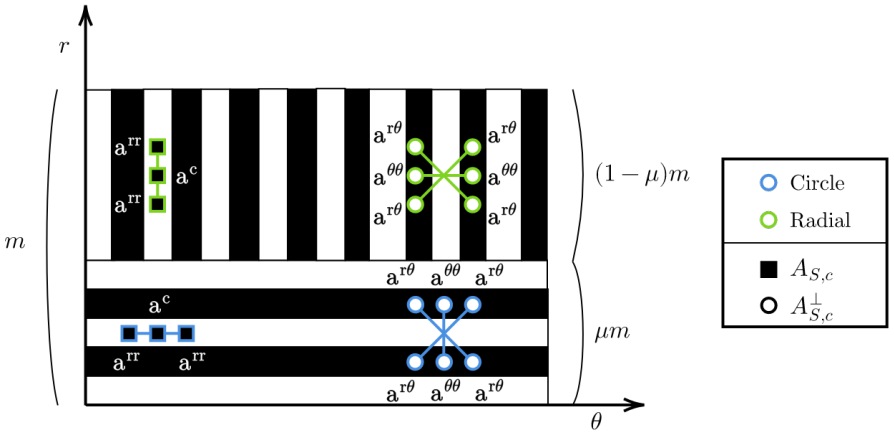


Figure 3: Decomposition of the polar plane into circle and radial smoothers with black and white coloring. The stencils for the smoother matrix $A_{S,c}$ and its complement $A^{\perp}_{S,c}$ are also displayed. For each update, we give which function a^{rr}, $a^{r\theta}$, or $a^{\theta\theta}$ needs to be computed.

Now, using all the elements for the multigrid scheme introduced above, we compute the overall complexity of the Gmgpolar solver.

Algorithm 2 Alternating zebra smoother $\mathcal{S}_\nu(u, A, f)$

1: **for** ν iterations **do**
2: **for** $(S, c) \in \{Circle, Radial\} \times \{White, Black\}$ **do**
3: Solve $A_{S,c}u_{S,c} = f_{S,c} - A^{\perp}_{S,c}u^{\perp}_{S,c}$
4: **end for**
5: **end for**

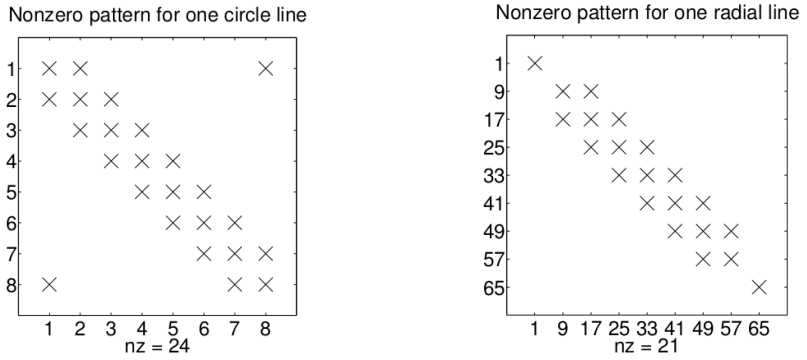


Figure 4: Matrix corresponding to 1 line for *(Left)* the Circle relaxation and *(Right)* the radial relaxation.

Computational and Memory complexity

On each grid level in Gmgpolar, only the coarse grid operator, the RHS and the smoother matrices $A_{S,c}$ are stored. We thus have a total memory consumption of the order

$$\text{Mem(Gmgpolar)} = 9m_L + \sum_{l=0}^{L}\left(m_l + \sum_{S,c} 3m_{lS,c}\right) = \left(\frac{9}{4^L} + \frac{4^L - 1}{3.4^{L-2}}\right)m. \tag{2}$$

Though we do not detail the calculus here, the complexity for each element required in the multigrid cycle is easily obtained:

- The construction of A_L, $A_{S,c}$, as well as the application of A, P, $A^{\perp}_{S,c}$ are naturally obtained from the stencils presented in Figures 1, 2, and 3,
- The solution of the problem on the coarse grid is obtained with a sparse direct solver. Since the operator is SPD, we use a Cholesky decomposition $chol(A_L)$ during setup, and only apply the forward and backward substitution at each iteration of the V-cycle. These have well known computational complexities,
- Due to the specific structures of the matrices $A_{S,c}$ from Figure 4, the system of the corresponding system performed during each relaxation sweep has a linear complexity.

The actual complexity for each of these elements is given in Table 2, together with their number of applications and the relevant grid levels. We do not give the complexity of the RHS construction since it depends on the solved problem.

Table 2: Cost and number of application for the functions used in the multigrid cycle. μ characterizes the switch between circle and radial smoothers. ν_1 and ν_2 are the number of pre- and post-smoothing sweeps. it is the number of iterations for the convergence of the multigrid cycle. Note that $m_{Circle,s} = \mu m/2$ and $m_{Radial,s} = (1-\mu)m/2$.

Function	Cost	Applications	When	Levels
$chol(A_L)$	$\mathcal{O}(m_L^3/3)$	Once	*(Setup)*	$l = L$
$A_{S,c}$	$\mathcal{O}(42m)$			$l = \{0, \dots, L-1\}$
$A_{Circle,c}^{-1}$	$\mathcal{O}(12m_{Circle,s})$	$(\nu_1+\nu_2)it$	*(Relaxation)*	$l = \{0, \dots, L-1\}$
$A_{Radial,c}^{-1}$	$\mathcal{O}(8m_{Radial,s})$			
$A_{S,c}^{\perp}$	$\mathcal{O}(51m)$			
Au	$\mathcal{O}(70m)$	it	*(Residual)*	
Pu	$\mathcal{O}(17m/2)$	$2it$	*(Restrict res., prolongate error)*	$l = \{0, \dots, L-1\}$
${A_L}^{-1}$	$\mathcal{O}(2m_L^2 - m_L)$	it	*(Coarse solve)*	$l = L$

The total complexity of the solver is

$$\text{Cost}(\text{Gmgpolar}) = \left(\frac{1}{3.4^{3L}}m^3 + \frac{2it}{4^{2L}}m^2 - \frac{it}{4^L}m\right) + (42 + (\nu_1+\nu_2)(4\mu+59)it + 87it)\,\frac{4^L-1}{3.4^{L-1}}m. \quad (3)$$

The memory cost is linear with respect to the size m of the problem. Also, the computational cost of the relaxation is linear in $\mathcal{O}(m)$. However, the solution of the problem on the coarse grid is of the order $\mathcal{O}(m^3)$! Gmgpolar is a multigrid solver, and the goal is to have a sufficiently high number of grids such that we have enough layers to obtain a coarsest grid with few unknowns. This is the case here since the cost to solve the coarse problem is $\mathcal{O}\left(\frac{1}{3.4^{3L}}m^3\right)$ and this very quickly tends to 0. Asymptotically, we can neglect the cost of the coarse problem solve and the cost of the whole solver becomes linear. It is typical when analyzing the complexity of a multigrid method to express it in terms of *Work Unit* (WU), where 1 WU is the cost of 1 relaxation sweep. Here, if we consider that the switch between smoothers is for $\mu = 1/3$, then we have $1\text{WU} = (4\mu+59) \approx 60m$ flops. If we apply only 1 relaxation sweep in pre- and post-smoothing ($\nu_1 = \nu_2 = 1$), we get the limit costs:

$$\begin{aligned} &\lim_{L\to\infty} \text{Cost}(\text{Gmgpolar}) = \mathcal{O}\left((56+277it)m\right)\text{flops} = \mathcal{O}(0.93+4.60it)\text{WU}, \\ &\lim_{L\to\infty} \text{Mem}(\text{Gmgpolar}) = \mathcal{O}(5.33m). \end{aligned} \quad (4)$$

Summarizing, Gmgpolar is an efficient solver for the solution of the gyrokinetic Poisson equation on disk-like domains using curvilinear coordinates to represent the problem geometry in the best possible way. We have shown that despite its nontrivial design, it can be implemented with low memory cost and with low computational cost. The details of the computation for the complexity of the Gmgpolar solver are soon to be published in a technical report [45]. In the remaining time of the project, it is planned to perform an extensive comparison of this solver with the AMRex framework and the spline based solvers that are proposed as alternatives.

Currently, a C++ implementation of the Gmgpolar code is being realized including the matrix-free techniques and exploiting node-level OpenMP parallelism. This implementation is expected to be completed before the end of the EoCoE-2 project. A possible integration inside the GyselaX code is still considered. As pointed out above, another

important perspective is the possibility to port the C++ code to GPUs or other accelerators. An application for the participation to the GPU Hackaton EuroHack21 will be submitted soon. We point out, however, that the full realization of GPU parallelism before the end of the project may not be possible. Future work will be required to exploit the Gmgpolar solver to the full extent of its capabilities and its advantages that it derives from the application-specific co-design. In particular, it may not be possible to fully realizing the implicit extrapolation given the funding constraints, although this can provide much better approximation quality and would thus further improve the efficiency considerably.

5.2 HyTeG for large scale problems from Tokamak simulations

Partners: CERFACS, IRIT-CNRS, FAU Erlangen-Nürnberg, INRIA, ULB
Software packages: HyTeG, SOLEDGE3X

The simulation of magnetically confined plasma in Tokamak reactors is pursued in EoCoE with the two flagship codes developed mainly at IRFM-CEA

- GYSELA is a gyrokinetic code solving the 5D Vlasov equation coupled with a 3D quasi-neutrality equation [36, 18]. In this code, 2D Poisson-like equations on cross-sections of the torus must be solved at each time step.
- SOLEDGE3X is a code, an evolution of TOKAM3X [62], for the simulation of transport and turbulence of the plasma. In this code, the 3D vorticity equation must be solved at each time step.

Both of these codes require at each time step the solution of a large linear system. Based on several meetings between MPG-IPP and CERFACS about GYSELA on the one hand, and between CNRS-IRIT and IRFM-CEA about SOLEDGE3X on the other hand, it was identified that there is a need for scalable solvers. In particular, in the case of the SOLEDGE3X code, AGMG and several solvers from PETSc have been tested in the early stages of the project on problems of size up to $\mathcal{O}(10^6)$ degrees of freedom (DOFs) [28]. However, weak scaling efficiency is still too low for realistic problems such as those arising in the ITER project[3] with $\mathcal{O}(10^9)$ DOFs.
In collaboration between CERFACS, CNRS-IRIT, and FAU Erlangen-Nürnberg, we have investigated scalable multigrid solvers employed on the full 3D torus. This work is based on a new open source code, the *Hybrid Tetrahedral Grids* (HyTeG) finite element multigrid framework.

Hybrid Tetrahedral Grids (HyTeG)

The HyTeG framework[4] [42] is a new, more flexible and sustainable open source implementation of the *Hierarchical Hybrid Grids* (HHG) framework [9, 28]. The framework is based on several ingredients enabling parallel performance on supercomputers. First, HyTeG overcomes the memory limitation arising from extreme scale computations with a matrix-free implementation, i.e. operators are not stored but applied. This does not only reduce overall memory consumption, but also avoids memory traffic. This is potentially a critical advantage, since many sparse matrix codes are limited by memory bandwidth. Computational experience with HHG shows that this design enables the solution of larger

[3] `https://www.iter.org/fr/accueil`
[4] `https://i10git.cs.fau.de/hyteg/hyteg`

problems due to its excellent scaling behaviour. Secondly, HyTeG also supports unstructured coarse meshes. This together with blending techniques can be used to represent complex geometries. The principle is to uniformly refine each element of a coarse grid, thus building hierarchical hybrid grids, see Figure 5. The local uniformity is then exploited to realize the algorithms in the form of efficient stencil-based operations. Note that the data structures are specifically designed to avoid indirect addressing for memory access, as would be typical in more traditional sparse matrix codes. This results in a significant performance advantage on many current supercomputer architectures.

In terms of data structure, HyTeG follows the same principle as HHG and decomposes

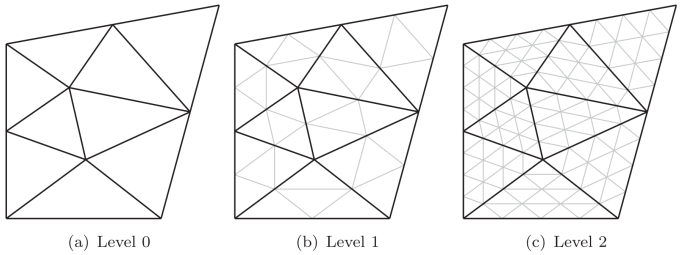


Figure 5: Local uniform refinement on an unstructured mesh. *Source:* [42].

the grids in so-called macro-primitives separated in nodes, edges, faces, or volumes of a finite-element mesh to handle all DOFs. Efficient graph-based methods to partition the system are embedded in HyTeG in order to decompose the whole mesh in subdomains with balanced workloads. Figure 6 shows the macro-primitives and partitioning for a simple mesh. This decomposition is used to distribute the workload over processes in the MPI parallel code.

Finally, we recall that multigrid methods have been proven to offer asymptotic optimal

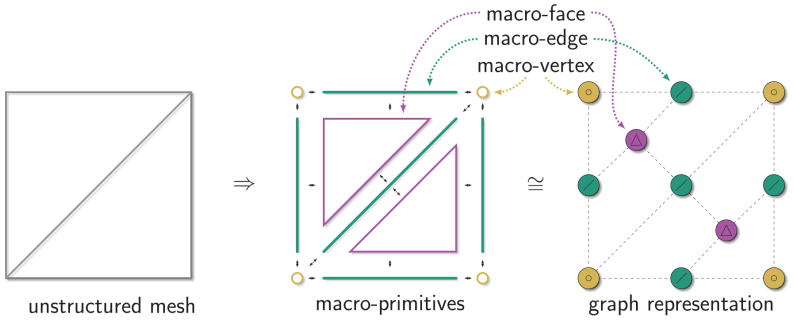


Figure 6: Macro-primitives and graph-based domain decomposition for a simple mesh in HyTeG. *Source:* [42].

complexity for systems arising from the discretization of elliptic *partial differential equations* (PDEs), in the sense that their convergence rate is bounded independently of the mesh size. Thus the computational cost grows linearly with the problem size. This is mathematically necessary to realize scalable behaviour for extreme scale computing.

The problem and multigrid solver

For the current scalability study, we have focused on the Poisson equation with homogeneous Dirichlet boundary conditions

$$\begin{aligned} -\nabla \cdot (k \nabla u) &= f \quad \text{in } \Omega, \\ u &= 0 \quad \text{on } \partial\Omega, \end{aligned} \tag{5}$$

where Ω is the toroidal geometry, and f is the source term. k is a coefficient depending on the radial direction with strong variations in the domain. Unlike other approaches proposed in the project [72, 44], Cartesian coordinates are used here. The torus is defined following the ITER geometry, as described in [58].
In the HyTeG framework, we start from a coarse unstructured tetrahedral mesh, here a toroidal polyhedron, then uniform refinement is applied to each coarse grid element. We thus construct a hierarchy of nested grid levels. In order to get an accurate curved domain even after refinement, we introduce a differentiable blending function mapping the computational domain Ω_{comp}, corresponding to a grid, and the physical domain Ω. This mapping is used in the weak formulation of (5). Mathematically, the blending function can also be interpreted as a variable coefficient in the equation. As shown in Figure 7, the mapping is applied in 3 steps:

1. mapping of the computational domain $\Omega_{comp} = \Omega_0$ to a torus in toroidal direction Ω_1,
2. mapping of Ω_1 on an actual torus Ω_2,
3. mapping of the torus Ω_2 on the Tokamak geometry Ω.

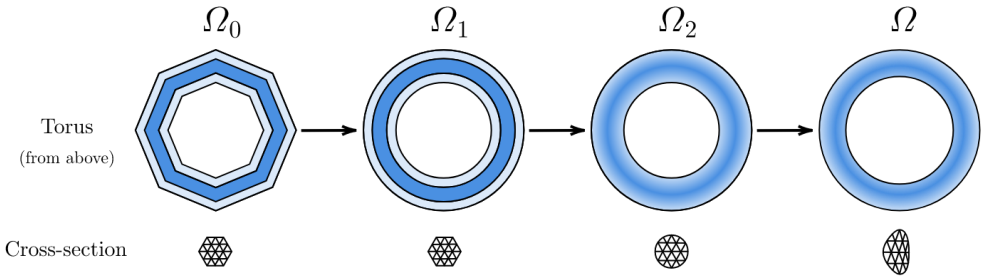
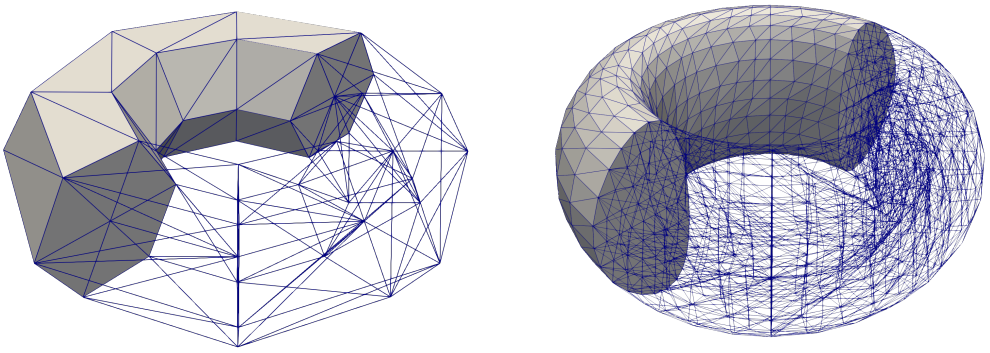


Figure 7: Blending of the computational domain, a toroidal polyhedron, to the geometry of the torus. For each step, the torus is seen from above, and the corresponding cross-section is shown below.

All-in-all, starting from a very coarse grid with the geometry of a toroidal polyhedron, HyTeG constructs refined levels of grids to get the ITER geometry, as presented in Figure 8. We discretize equation (5) using P1 finite-elements on all grid levels, and define bilinear interpolation as prolongation operator between 2 consecutive levels. Finally, we use a classical multigrid V-cycle using weighted Jacobi relaxation. To solve the problem on the coarsest grid level, we use a *conjugate gradient* (CG) algorithm.

Large scale experiments

We have performed weak scaling experiments in order to assess the potential parallel efficiency of using HyTeG in the context of plasma simulation. For our tests, we use the



(a) Unblended coarse mesh (level 0)

(b) Blended mesh after two refinement steps (level 2)

Figure 8: Creation of the mesh for the ITER geometry used in the scaling experiments . In those experiments a refinement of up to level seven is conducted.

exact solution defined by

$$u(x,y,z) = -\overline{z}^2[\overline{r} + \cos(\arcsin(\overline{z}) - \overline{\delta})][\overline{r} - \cos(\arcsin(\overline{z}) + \overline{\delta})]\sin(\pi\overline{z})\sin(\delta - \overline{r}), \tag{6}$$

and the coefficient

$$k(x,y,z) = k_{min} + \frac{k_{max} - k_{min}}{2}\left(\tanh\left(3.5\frac{r - r_{jump}}{d_{jump}}\right) + 1\right), \tag{7}$$

with

$$\overline{r} = \frac{\sqrt{(x^2+y^2)} - R_0}{R_1},\ r = \sqrt{\overline{z}^2 + \overline{r}^2},\ \overline{z} = \frac{z}{R_2},\ \overline{\delta} = \overline{z}\arcsin(\delta), \tag{8}$$

where R_0, R_1, R_2 are parameters of the torus, and k_{min},k_{max},r_{jump},d_{jump} are characteristics of the variations of the coefficient k in (5).

Our experiments are performed on the two petascale supercomputers ranked respectively in positions 15 and 16 of the TOP500 list[5] (November 2020):

- SuperMUC-NG, at the LRZ in Leibniz[6], consists of 6 336 Intel Xeon Skylak' processors with 48 cores and 96 GB memory each. SuperMUC-NG uses the Intel OmniPath interconnect. The supercomputer has 304 128 cores in total for a theoretical peak performance of 26.9 PFLOPS/s.
- Hawk, at HLRS in Stuttgart[7], consists of 5 632 AMD EPYC 7742 processors with 2 sockets of 64 cores and 128 GB memory each. Hawk uses InfiniBand HDR200 interconnect. The supercomputer has 720 896 cores in total for a theoretical peak performance of 26 PFLOPS/s.

Here, we show the results from a weak scaling study performed using HyTeG to compute the solution of (5), with the parameters introduced above. To obtain comparable results on different node counts, we choose to refine the coarse grid once while increasing the number of nodes by a factor of eight in each step. Since the refinement of a tetrahedron results in eight smaller tetrahedrons, the number of tetrahedrons per core is constant.

[5] https://www.top500.org
[6] https://doku.lrz.de/display/PUBLIC/SuperMUC-NG
[7] https://www.hlrs.de/systems/hpe-apollo-hawk/

Starting from this coarse mesh we build up a multigrid hierarchy with six (SuperMuc-NG) or seven (HAWK) levels of refinement, respectively.
In the multigrid solver, V-cycles are used with 3 steps of pre- and post-smoothing. The convergence of the multigrid method is reached when the residual has been reduced by a factor of 10^6, compared to the initial residual. The CG solver from the PETSc library is used as a coarse grid solver. Its convergence is obtained for a relative residual reduction of 10^{-6}, or for an absolute residual of 10^{-12}.
It is important to note that in this particular experiment we don't follow a strictly matrix free scenario but store the local element stiffness matrices for each element. Even though our current implementation would be capable of not storing the matrices and perform the computation on the fly this would result in a performance drawback. This will be improved in the future see 5.2.

Table 3 shows the results of the current weak scaling analysis performed on SuperMUC-NG for the solution of the problem (5) with $5.1 \cdot 10^7$ DOFs up to $3.2 \cdot 10^9$ DOFs respectively solved on 288 to 18 432 MPI processes. The total execution time for the multigrid solver is given as well as the timing for the coarse grid processing. We also present the parallel efficiency for the average total runtime over the iterations, in relation to the smallest run with 288 processors. The ideal result would be that the execution time, and thus the parallel efficiency, stays unchanged when we increase the problem size and number of cores.

We observe that the current implementation does not yield as good a parallel efficiency as could be expected [35, 7]. Efficiency decreases to 77% when increasing the problem and the number of processors by a factor of 64. In the current study, this is mainly due to the runtime for the coarse grid solver increasing more than twenty-fold. In particular, the average number of coarse grid iterations for the convergence of the coarse grid solver increases by a factor of 105 between the smallest and largest problems. As for the convergence of the multigrid scheme itself, the number of iterations also increases by around 32%. This shows that additional improvements need to be applied to the multigrid scheme in order to improve its convergence. We introduce some possibilities below.

As for the average runtime, we have encountered heavy issues with getting consistent results when running on the supercomputer SuperMUC-NG as well as on the supercomputer Hawk. In fact, at this extreme scale some instabilities can appear which are partly due to differences from the placement of the processes at runtime as well as asynchronous MPI communication, and differences in the use of the cache between two runs [60]. Several identical runs would be needed to get better results. Currently such extended studies are not yet available since the waiting time in queues for these machines is also high and the compute time budget is limited.

Finally, Table 4 shows the results of the weak scaling, with same settings, performed on the supercomputer Hawk for the problems with $5.1 \cdot 10^7$ DOFs and $2.7 \cdot 10^8$ DOFs respectively solved on 288 to 4 096 MPI processes. The results in terms of parallel efficiency are very similar with what is obtained on SuperMUC-NG. This presents our first attempt at showing the potential of HyTeG on an AMD architecture which are more and more used in the HPC community.

Table 3: Weak scaling of the V-cycle multigrid cycle in HyTeG on SuperMUC-NG. We display the number of iterations for the convergence of the V-cycle and also for the convergence of the CG solver applied to the coarse grid. We distinguish the time to process the coarse grid and the time for the fine grids. The parallel efficiency compares the average total run-time of the smallest case to the average total run-time of each run. Furthermore, we observe the expected quadratic L^2 convergence of the discretization error $\|u - u_h\|_{0,h}$ with FE solution u_h and analytic solution u from (6). The discrete L^2 norm is defined by $\|v\|^2_{0,h} := \mathbf{v}^T \mathbf{M}_h \mathbf{v}$ with FE mass matrix $\mathbf{M}_h$ and vector of nodal values $\mathbf{v}$ corresponding to v.

proc.	DOFs		iterations		time (s)		eff.	disc. L2 error
	fine	*coarse*	MG	avg coarse	MG	coarse grid		
288	$5.1 \cdot 10^7$	$3.8 \cdot 10^2$	26	20	28.85	0.39	1.00	$1.15 \cdot 10^{-04}$
2 304	$4.1 \cdot 10^8$	$2.2 \cdot 10^3$	35	131	39.17	0.69	0.99	$2.80 \cdot 10^{-05}$
18 432	$3.2 \cdot 10^9$	$1.5 \cdot 10^4$	38	2107	54.96	9.34	0.77	$7.07 \cdot 10^{-06}$

Table 4: Weak scaling of the V-cycle multigrid cycle in HyTeG on HAWK. The settings are exactly the same as in Table 3 except that one additional level of refinement is used.

proc.	DOFs		iterations		time (s)		eff.	disc. L2 error
	fine	*coarse*	MG	avg coarse	MG	coarse grid		
512	$3.4 \cdot 10^8$	$3.2 \cdot 10^2$	26	19	79.2	0.12	1	$3.13 \cdot 10^{-05}$
4 096	$2.7 \cdot 10^9$	$1.9 \cdot 10^3$	35	92	113.6	0.82	0.94	$7.77 \cdot 10^{-06}$

Discussion and perspectives

With this study, we explore the potential of using a modern matrix-free code like HyTeG for the solution of very large scale problems posed on a torus. Though we believe that solving for $2.7 \cdot 10^9$ unknowns in less than 2 minutes compute time already shows a good potential, we point out that this is merely a feasibility demonstrator. For the solution of such a problem, the scaling results are still very sub-optimal and far from what we could expect when using HyTeG to its full capabilities. In fact, the current world record in terms of the largest linear system ever solved, of size $\mathcal{O}(10^{13})$, was established using this framework [7]. Also we recall that the equation (5) that we solve is simplified compared to the actual application problems, and our manufactured solution is quite smooth. In order to target realistic applications, many further aspects will have to be worked on:

- First, it is necessary to improve on the convergence of the multigrid scheme, e.g. with different smoothers, prolongation operators, and a better coarse grid solver. With correctly chosen, application-specific components, the multigrid scheme exhibits a mesh independent convergence, the algorithmic basis to achieve full scalability.
- Hybrid methods are of particular interest in high performance computing. It is possible to employ an interface for coupling HyTeG with sparse direct solvers, such as MUMPS. In the current weak scaling study, it was observed (as theoretically expected) that the convergence, and runtime of the coarse grid solver was degrading with larger problem sizes. In fact, these problems were already observed in [35]. In [20], we propose using an agglomeration technique combined with the use of an approximate direct solver on the coarse grid in order to overcome this issue. This approach appeared in the previous deliverable for the solution of a Stokes

problem defined on a spherical shell [28]. Again it is possible to adapt it for the plasma simulation.

- Techniques to improve the approximation order of our multigrid solver could be implemented, e.g. implicit extrapolation [41]. The advantage of such a method is that they do not require the explicit use of higher order elements, thus allowing better approximation order for a similar computational cost. For 2D cross sections of the Tokamak this is represented and illustrated in gmpolar as part of the current deliverables to EoCoE2.
- As mentioned above, the current approach stores the local element stiffness matrices for improved performance but at the cost of higher memory consumption. In order to maximize the possible number of degrees of freedom, a matrix-free method will have to be realized. If naïvely implemented, this results in a substantial computational overhead due to redundant recomputation of the matrix entries. However, we are able to avoid the redundant evaluations of costly numerical quadrature rules by replacing the entire stiffness matrix with a small set of surrogate polynomials as developed in [6, 5, 7].

These techniques together would further improve the efficiency of HyTeG, so that problems of equal size could be solved much faster (strong scaling) or larger problems could be solved at similar times (weak scaling). Additional extensions however, will be needed for a full integration in a simulation code such as Gysela. By the end of the project, we are planning to present a systematic comparison of the existing HyTeG geometric multigrid solver with other solvers. In particular, an algebraic multigrid solver, as implemented in PETSc, as well as classical Krylov solvers will be used. Thus, within the EoCoE-2 project, we can explore the potential of HyTeG as a building block in plasma simulation, but it is foreseeable that full scale production runs will remain out of reach in the given time and funding limits. A complete co-design, as required to exploit the potential of advanced multigrid methods is shown to be very promising, but can currently not be fully developed.

6. Task 3.4: Linear Algebra solvers for Wind

6.1 Solid physics solver for Alya and MUMPS coupling

Partners: BSC, IRIT-CNRS, INRIA
Software packages: MUMPS, Alya

Solid mechanics problems are known to be very stiff. In the case of composite materials, the situation is even more complex as material anisotropy makes the difficulty directional. In Alya, the most efficient algebraic solver currently available to solve such problems is the GMRES method coupled with a restricted additive Schwarz (RAS) preconditioning. Additionally, since the problem considers large deformations, the resulting modelisation is a non linear elastic problem. The GMRES-RAS is encapsulated into a Newton-Raphson (NR) method. Furthermore, the problem is time-dependant therefore, at each time step, NR performs several iterations, each of which involves the convergence of a GMRES-RAS.

GMRES and RAS preconditioner. Although its overall convergence and, thus, its performance, are strongly related to the mesh partitions size, orientation and shape, the GMRES-RAS method has proved to be the most robust approach to solve solid mechanics problems in the Alya application. A description of the algorithm as implemented in Alya can be found in [49]. In the RAS preconditioner, the original problem is partitioned into smaller sub-problems which are, each, assigned to one of the processes participating in the computation which is, therefore, in charge of its resolution. Overlapping between the local sub-problems ensures a global coherency and a faster transmission of information; this can be further using a deflation technique (as explained below). Due to the relatively small size of the sub-problems and the need to solve them reliably and accurately, a direct method is often preferred despite its potentially large cost. As a result, the main bottlenecks of the RAS preconditioner are:

- The numerical factorization of the matrix associated with each sub-problem; this is executed once, in the preconditioner setup phase. The factorization must be preceded by a so-called symbolic analysis which preprocesses the sub-problem matrix based solely on its structure in order to improve the efficiency of the numerical factorization.
- The backward elimination and forward substitution to compute the local sub-problem solution; these are executed every time the preconditioner is applied, i.e., at each GMRES iteration.

Currently, for the production runs of the Alya application, the factorizations and substitutions are performed with a sequential in-house direct solver based on the LU factorization.

Coarse problem. To provide a global communication mechanism across the partitions, a coarse problem correction is also available. This coarse problem is constructed using the agglomeration technique used in the deflation strategy, for the deflated conjugate gradient described in [48]. Because the targeted problems are indefinite, this agglomeration is surely not optimal in terms of convergence properties but is very cheap to construct and, thus, worth being used. The coarse matrix is independent of the partitioning, which enables a control of the coarse problem size, when the number of cores increases. The coarse operator is handled with the same direct method used for solving the local sub-problems.

MUMPS integration

The MUMPS sparse direct solver has been integrated in the Alya software in order to replace the in-house LU solver and to evaluate its potential to improve the efficiency of the solution of the above-mentioned problems.

We consider the structural mechanic problem of a wind power generator blade. The simulation only embraces the elastic regime of the material – the structure does not undergo damage. Figure 9 displays an example of such structure. The discretization of the airfoil generates a mesh with around $15 \cdot 10^6$ degrees of freedom (dofs).

In the context of this study, we first consider only one time step that involves several iterations of a NR to solve the non-linear problem. We consider 15 computing nodes of the MareNostrum supercomputer using only 32 cores per node out of the 48 available.

We first investigate the use of MUMPS to replace the GMRES-RAS approach. In

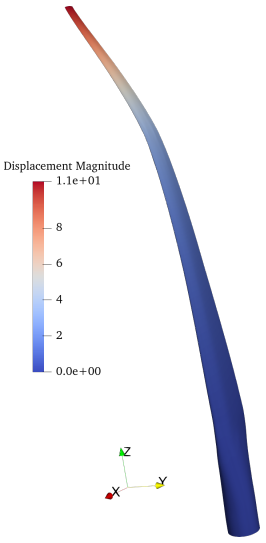


Figure 9: Blade of a wind power generator submitted to displacement of its tip of 10m after ten time steps

order to improve its efficiency, the numerical pivoting is disabled, because the factorization was found to be sufficiently accurate and stable. Additionally, a recent MUMPS feature was activated: the block analysis. This feature allows for compressing the matrix graph in the case where multiple unknowns are associated with each node of the discretization mesh; this allows for considerably reducing the time spent in the symbolic analysis. Note that we can reuse the analysis throughout the iterations of the non-linear solver and of the time dependency as long as the material does not break because, in this case, the structure of the sub-problem matrices remains the same. We also enable the Block Low-Rank (BLR) technique recently implemented in MUMPS [1]. The idea of BLR is to compress some off-diagonal blocks by their low-rank approximation to reduce the workload and memory whenever possible. The low-rank approximation is the representation of a block by a product of matrices of smaller rank. Finally, we took advantage of the recent advanced multi-threading technique, the so-called L0-thread [47] to improve the parallel efficiency.

For the elastic deformation, the use of MUMPS as direct solver shows an improvement with respect to the original GMRES-RAS approach (run GMRES-RAS-ALYA-t1 versus run MUMPS-t1 of Tab 5).

Now we consider the strategy where the current (in-house) LU solver is replaced with MUMPS to tackle the aforementioned bottleneck in the RAS preconditioner. The size of the coarse operator is set to 6150 dofs which is the value that experimentally provides the best convergence for the iterative solver for that case.

In table 5, we show three representative runs: GMRES-RAS-ALYA-t1, GMRES-RAS-MUMPS-t1, and GMRES-RAS-MUMPS-OMP-t1. We detail the timing of the LU solvers in Figure 10 for those runs. Used as a reference, the first run, GMRES-RAS-ALYA-t1, is the fastest configuration we could obtain with the production code version of Alya: each core runs one MPI process which is in charge of one sub-domain; as a result, each sub-domain has around 33k dofs. The second run, GMRES-RAS-MUMPS-t1, is obtain with MUMPS as LU solver using the same configuration. We observe that the time spent in the LU solver is divided by two thanks to the use of MUMPS with respect to the in-house solution. From Figure 10, we observe that the cumulative time spent in the factorization

ID	MPI	Threads	Time step	Solver Time (s)	LU Time (s)	LU package
GMRES-RAS-ALYA-t1	480	1	1	377	223	ALYA
GMRES-RAS-MUMPS-t1	480	1	1	212	102	MUMPS
GMRES-RAS-MUMPS-OMP-t1	240	2	1	276	114	MUMPS
MUMPS-t1	30	16	1	231	231	MUMPS
MUMPS-t10	30	16	10	1156	1156	MUMPS
GMRES-RAS-ALYA-t10	480	1	10	1271	917	ALYA
GMRES-RAS-MUMPS-t10	480	1	10	519	328	MUMPS

Table 5: Examples of runs on MareNostrum with 480 cores spread on 15 nodes. The coarse deflation operator has 6150 dofs. We show significant improvement by replacing ALYA LU solver by MUMPS.

is largely reduced by almost a factor 10. As the last version of MUMPS implements an advanced multi-threading technique, we explored the possibility of using this feature. We show the results with half the number of MPI and with two threads each in GMRES-RAS-MUMPS-OMP-t1. Because the sub-domains are larger (around 68k dofs) the factorization and solve become more expensive; unfortunately the multi-threading does not compensate for the operational overhead. Additionally, due to the relatively small size of the local problems, the Block Low Rank technique cannot achieve sufficient compression.

Finally, we extend the tests to up to 10 time steps. The use of MUMPS as a direct solver, MUMPS-t10, shows slightly better performance than GMRES-RAS-ALYA-t10 with Alya LU solver. However, with MUMPS in the GMRES-RAS solver, case GMRES-RAS-MUMPS-t10, we observe an even more significant improvement. Indeed we divided the total time spent in the solver by a factor of almost two. This is mostly due to the gains on the factorization phase.

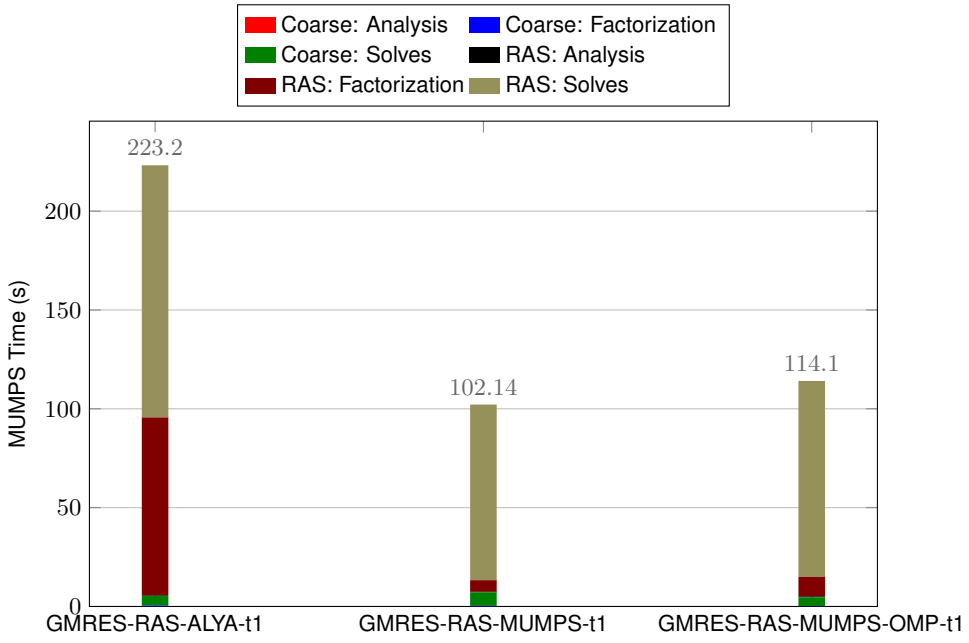


Figure 10: Distribution of the time spent in the LU solvers for three configurations reported in table 5

6.2 CFD solver for Alya: PSCToolkit

Partners: BSC, CNR, UNITOV
Software packages: PSCToolkit, PSBLAS, AMG4PSBLAS, Alya

During the first phase of the project, as reported in the Deliverable 3.1 [55], we developed a software module included in the Alya's kernel, to interface PSBLAS (rel. 3.6) and its sibling preconditioner package to the code. This allows to Alya the exploitation of linear solvers and preconditioners from those external packages for solving linear systems arising from the different physics modules. Preliminary results obtained by testing the solvers on systems stemming from fluid dynamics simulation exploiting the NASTIN module, which deals with the incompressible Navier-Stokes equations for turbulent flows, were included in the previous deliverable [28]. In the following we present results on the test case already employed in [28] by using the new versions of the software packages, as extended and improved within this project (see Section 7 for details). Many new features, both in terms of available methods and in terms of software organization and implementation, have been added to the preconditioner package, so that its scope was largely extended; then we decided to change his former name (MLD2P4) in AMG4PSBLAS, whose first version (1.0.0) is currently available as component of a more comprehensive software framework we named PSCToolkit *Parallel Sparse Computation Toolkit*. The new AMG4PBSLAS package, including Algebraic MultiGrid (AMG) preconditioners for PSBLAS, required improvements and extensions also to the basic PSBLAS kernels, whose last release (3.7.0.1) is also available in the PSCToolkit framework. In details, in this section we first remind the test case designed by BSC, then we discuss both strong scalability and weak scalability results of the employed solvers, focusing on the best algorithmic choices and on the comparison among the most promising available AMG preconditioners included in AMG4PSBLAS.

Test Case Description

The mathematical model is the set of 3D incompressible unsteady Navier-Stokes equations for the Large Eddy Simulations (LES) of turbulent flows in a bounded domain with mixed boundary conditions. The LES formulation is closed by an appropriate expression for the subgrid-scale viscosity; in this analysis, the eddy-viscosity model proposed in [68] is used. Discretization is based on a low-dissipation mixed finite-element scheme, using linear finite elements both for velocity and pressure unknowns. A non-incremental fractional-step method is used to stabilise the pressure, whereas for the explicit time integration of the set of discrete equations a fourth order Runge-Kutta explicit method is applied [46]. Note that time steps used during simulations, for increasing problem size, are generally different due to the CFL stability constraint for velocity which is dealt in explicit way. The pressure field is obtained at each step by solving a discretization of a Poisson-type equation. The test case is based on the Bolund experiment, a classical benchmark for microscale atmospheric flow models over complex terrain [8]. The Reynolds number based on the friction velocity is approximately $RE_\tau = 10^7$. We run both strong scalability analysis for unstructured meshes of tethrahedra of fixed sizes as well as weak scalability analysis, fixing different mesh sizes per cores, for increasing number of cores up to 12288 and a mesh size up to $345276325 \approx 0.35 \times 10^9$ nodes (dofs). At each time step, we solve the symmetric positive definite (s.p.d) linear systems arising from the pressure equation employing a flexible version of the Conjugate Gradient (FCG) method of PSBLAS, coupled with different

AMG preconditioners available from AMG4PSBLAS. In detail, we used right preconditioned FCG starting from an initial guess for pressure from the previous step, and stop iterations when the Euclidean norm of the relative residual is not larger than $TOL = 10^{-3}$. A general row-block data distribution based on Metis 4.0 is applied for the parallel runs. The simulations have been performed with the Alya code interfaced to PSBLAS (3.7.0.1) and AMG4PSBLAS (1.0), built with GNU compilers 7.2, on the Marenostrum 4 Supercomputer composed of 3456 nodes with 2 Intel Xeon Platinum 8160 chips with 24 cores per chip (ranked 42^o in the TOP500 list[8], with more than 10 petaflops of peak performance), operated by BSC. The facility was made available by a grant dedicated to the EoCoE II project from PRACE.

Strong Scalability

In this section we focus on strong scalability results obtained on the Bolund experiment for three fixed size problems including $n = 5570786 \approx 6 \times 10^6$, $n = 43619693 \approx 4.4 \times 10^7$ and $n = 345276325 \approx 0.35 \times 10^9$ dofs, respectively. Three different configurations of number of cores are employed for the three different mesh sizes: from 48 to 192 cores in the case of the smallest mesh, from 384 to 1536 cores for the medium size mesh, and finally from 3072 to 12288 cores for the largest mesh. We analyze parallel efficiency and convergence behaviour of the linear solvers for 20 time steps after a pre-processing phase so that we focus on the solvers behaviour in the simulation of a fully developed flow. Note that in the Alya code a master-slave approach is employed, where the master process is not involved in the parallel computations. Taking into account the comparison analysis discussed in the previous deliverable (see [28]), in the following we only consider AMG methods implemented in AMG4PSBLAS, which already shown better results with respect to the one-level available methods; in more details we selected a symmetric V-cycle employing 4 iterations of the hybrid forward/backward Gauss-Seidel smoother at the intermediate levels and a coarsest solver based on the CG method preconditioned by ILU(1), with a stopping criterion based on the reduction of the relative residual of 3 orders of magnitude or a maximum number of iterations equal to 30. The multilevel hierarchies are built by applying the decoupled smoothed aggregation coarsening already available in the previous version of the preconditioner package and inherited in AMG4PSBLAS, in this case we refer to the AMG preconditioner as *MLVSBM*, then we apply the new parallel coupled aggregation scheme implemented in AMG4PSBLAS, relying on the *coarsening based on compatible weighted matching* [22], to build two hierarchies characterized by different sizes of the aggregates; in the first case we built aggregates of size at most 8 and we refer to the corresponding preconditioner as *MLVSMATCH3*, while in a second case a more aggressive coarsening characterized by aggregates of size at most 16 is employed; in this last case we refer to the corresponding preconditioner as *MLVSMATCH4*. In all cases the coarsening procedure is stopped when the size of the global coarsest matrix is no more than a fixed default value, and in our test cases this always corresponds to hierarchies having a total of 4 levels. In Figs. 11-12 we report a comparison of the different methods in terms of the total number of iterations of the linear solvers and of the solve time per iteration (in seconds), respectively. Note that in the figures we also have results obtained with a version of Deflated CG (*AlyaDefCG*), available from the original Alya code. We can observe that the total number of linear iterations is smaller than the original *AlyaDefCG*, per each one of the three mesh sizes, when AMG4PSBLAS multilevel preconditioners are applied. For the smallest size problem, min-

[8] `https://www.top500.org`

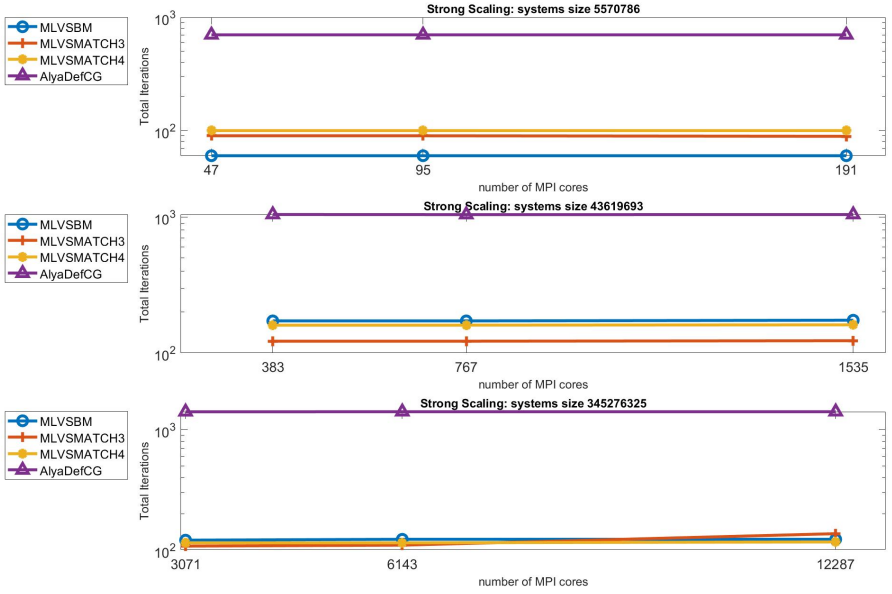


Figure 11: Strong scalability: total iteration number of the linear solvers

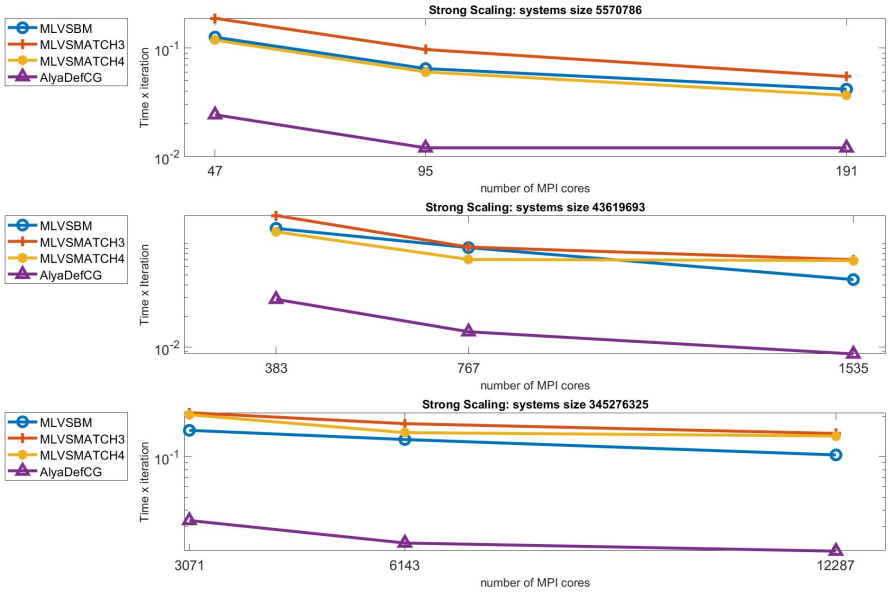


Figure 12: Strong scalability: time per iteration of the linear solvers

imum number of linear iterations is obtained by *MLVSBM* which shows a fixed number of 60 iterations for all number of cores, while *MLVSMATCH3* requires 90 iterations for all number of cores and *MLVSMATCH4* requires 100 iterations. In this case, the original *AlyaDefCG* requires 700 iterations for all number of cores. In the case of the medium size problem, we observe a larger number of iterations of the solvers employing AMG4PSBLAS precondi-

tioners with respect to the largest size problem. We have a minimum number of iterations with *MLVSMATCH3* equal to 123 for all number of cores, while *MLVSMATCH4* requires 160 iterations and *MLVSBM* requires 172 iterations. On the contrary, in the case of the largest size problem, the number of iterations required by *MLVSMATCH3* ranges between 108 on 3072 cores and 137 on 12288 cores, while *MLVSMATCH4* requires a more stable number of iterations ranging from 115 to 117; similar stability is observed for *MLVSBM* which requires a number of iterations ranging from 121 to 123. The oscillations in the number of iterations seem to be largely dependent on the data partitioning obtained by Metis which seems to have larger impact on the AMG4PSBLAS preconditioners in the case of the medium size problem. Some deeper analysis on the impact of the data partitioner on the solver behaviour, although interesting, is out of the scope of our current work and requires a larger availability in terms of computer resources. Indeed, we point out that the limited access to the computer resources is a main issue that limits our current performance analysis also in terms of increasing number of cores. A stable number of iterations is observed in all cases for *AlyaDefCG*, where the total number of linear iterations is always 1042 for the medium size problem and 1406 for the largest size problem.

In all cases, the time needed per each iteration decreases for increasing number of cores and, as expected, it is larger for the AMG preconditioners, where the cost for the preconditioner application at each FCG iteration is larger than that of *AlyaDefCG*. Depending on the problem size and number of cores, the AMG preconditioners show a very similar behaviour, although *MLVSBM* often requires a smaller time per iteration, especially for the largest size problem.

In Figs. 13-14 we can see the total solve time spent in the linear solvers and the resulting speedup for the preconditioners. Here we define speedup as the ratio $Sp = T_{min_p}/T_p$, where T_{min_p} is the total time for solving linear systems when the minimum number of total cores, per each problem size, is involved in the simulation, and T_p is the total time spent in linear solvers for all the increasing number of cores used for the specified problem size. We observe that all the AMG preconditioners by AMG4PSBLAS generally show smallest execution times with respect to the original *AlyaDefCG*, indeed the larger time per iteration, as expected, is largely compensated by the large reduction in the number of iterations especially for the smallest and largest problem size. In good agreement with the behaviour in terms of iterations and time per iteration, we observe that *MLVSBM* generally shows the smallest execution time, but in the medium size problem, where on 768 has a worse behaviour. The best speedup are generally obtained, except for the smallest size problem, by the original *AlyaDefCG*, while in the case of AMG preconditioners, the very good convergence behaviour and solve time on the smallest number of cores limits the speedup for increasing number of cores. In all cases *MLVSBM* shows the best speedup among the AMG4PSBLAS preconditioners, due to its general smallest cost per iteration.

In Figs. 15-16 we report the time for setup of the AMG4PSBLAS preconditioners and the resulting speedup (scaled to 48 cores, 384 and 3072, respectively, as for solve). We observe that, due to some instabilities of the Marenostrum 4 in all the simulations with the largest problem size, when the number of cores increases the setup time of the AMG preconditioners increases. This aspect is under investigation with the system administrators. On the other hand, in the other two cases, we see that the setup cost decreases with increasing number of cores, and the best behaviour is observed with *MLVSBM*, which implements a decoupled aggregation scheme, therefore, the setup of the aggregation phase is embarrassingly parallel. In conclusion, the selected solvers from the PSCToolkit gener-

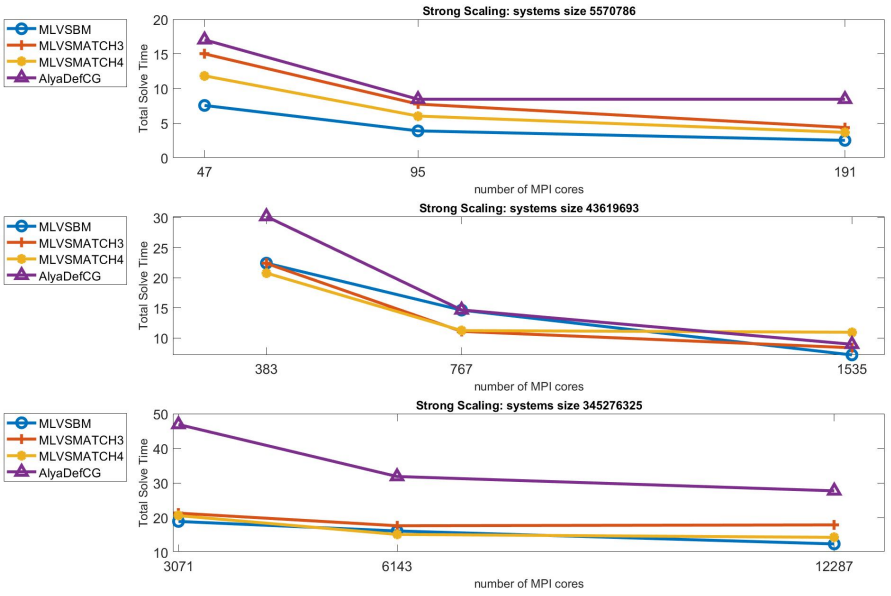


Figure 13: Strong scalability: total solve time of the linear solvers

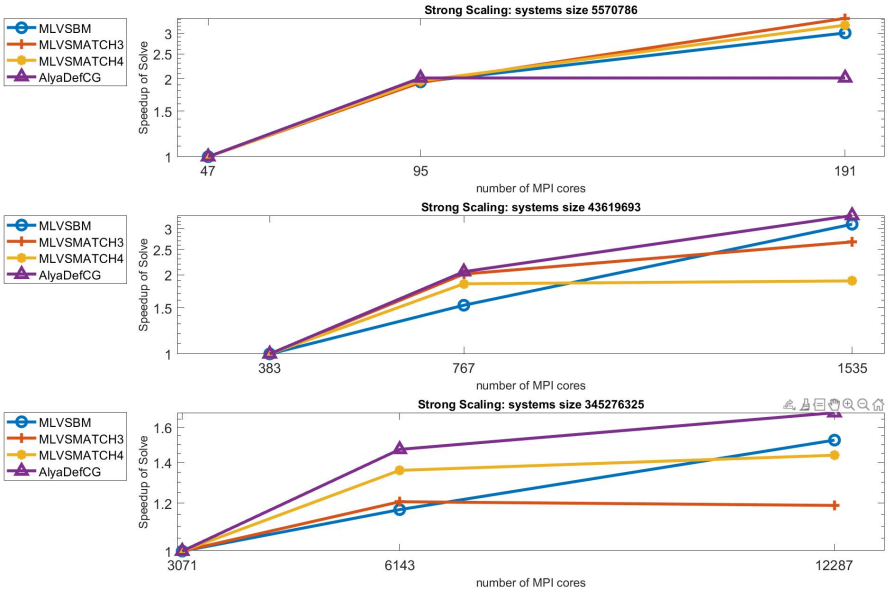


Figure 14: Strong scalability: speedup of the linear solvers

ally outperform the original Alya solver for the employed test case, and in the better case of FCG coupled with the AMG preconditioner *MLVSBM* we generally obtained the best strong scalability.

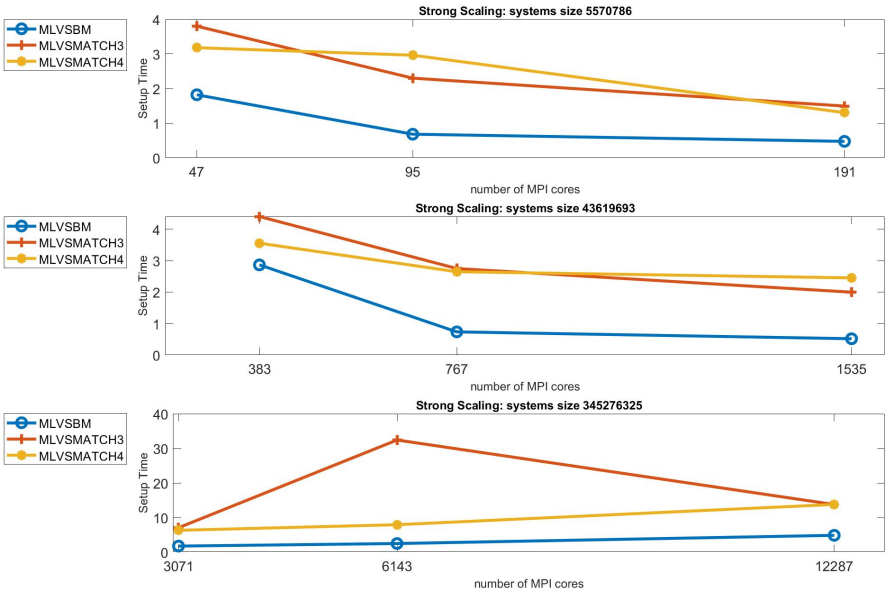


Figure 15: Strong scalability: AMG4PSBLAS preconditioners setup time

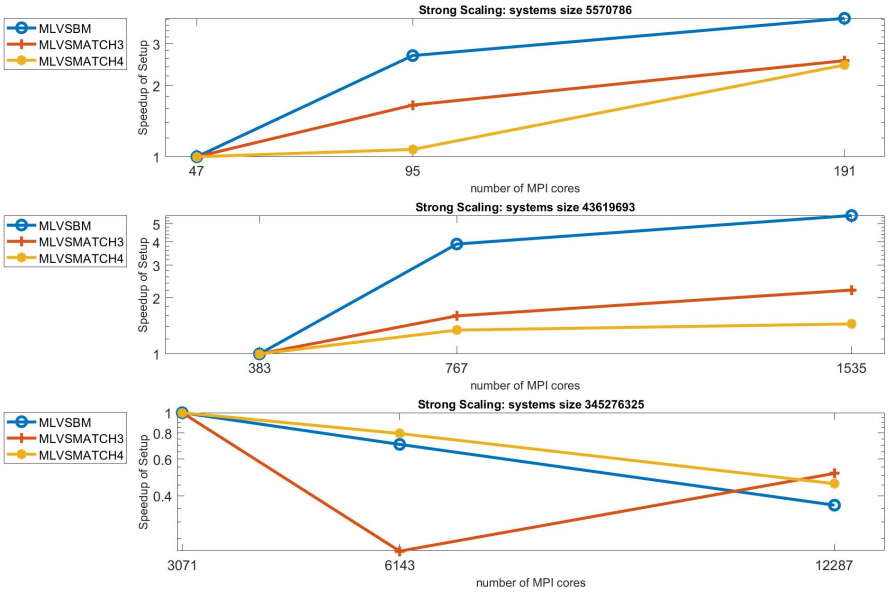


Figure 16: Strong scalability: AMG4PSBLAS preconditioners speedup

Weak Scalability

In this section we analyze weak scalability of the AMG4PSBLAS preconditioners, i.e., we observe the solvers looking at their behaviour when we fix the mesh size per core and increase number of cores. Indeed, main aim in parallel computation is both to use

the available resources at the best and to be able to efficiently solve larger problems when larger resources are employed. We considered the same test case and mesh sizes of the previous section, in the three possible configurations of computational cores, from 48 up to 3072, from 96 up to 6144 and from 192 to 12288, corresponding to three different mesh sizes per core equal to $1.1e5$, $5.9e4$, and $2.9e4$, respectively. Note that the medium and the largest mesh total sizes correspond to scaling factors of 8 and 64, respectively, with respect to the smallest mesh size, then in the same way we scale the number of cores for our weak scalability analysis. We can limit our analysis to observe the average number of linear iterations of the different employed preconditioners per each time step in the various simulations and to analyze exectuion times and scaled speedup for the solve and AMG preconditioner setup phases. In Fig. 17, we report the average number of iterations per each time step. We can observe a general increase, ranging from 35 to 70 for increasing number of cores, when the original *AlyaDefCG* is employed, while a very good algorithmic scalability, with an average number of linear iterations per each time step ranging from 4 to 8, when AMG preconditioners from AMG4PSBLAS coupled with FCG by PSBLAS are applied. In Figs. 18-19 we can see the total solve time and the corresponding scaled

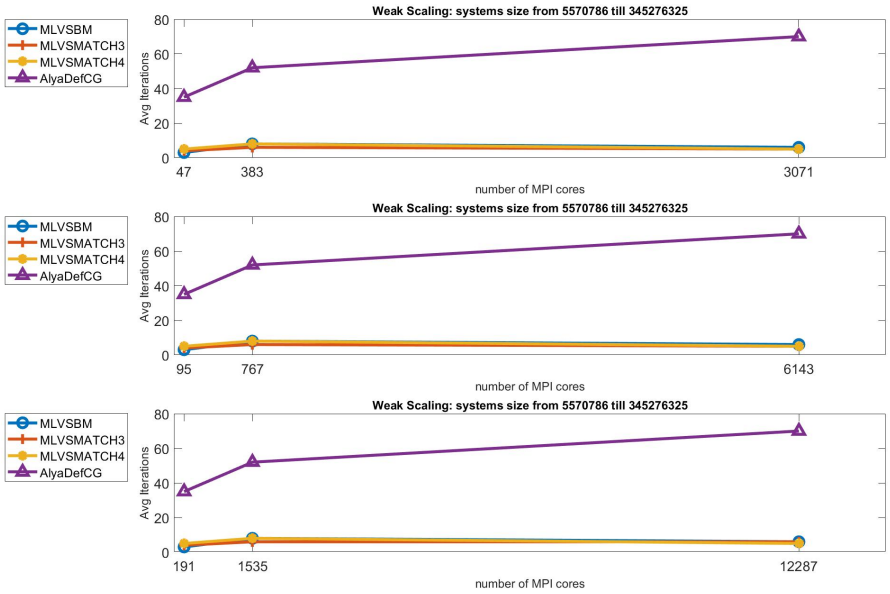


Figure 17: Weak scalability: average number of linear iterations per time step. $1.1e5$ dofs per core (top), $5.9e4$ dofs per core (middle), $2.9e4$ dofs per core (bottom)

speedup. We can observe that the good algorithmic scalability of the AMG4PSBLAS leads to a very small increase in the execution time for solve, especially in the case of the largest mesh size per core (top figure). In this setting, we observe that, for the medium size mesh per core, the smallest increase in the execution time is generally obtained with *MLVSMATCH3*. On the contrary, the original *AlyaDefCG* shows a very large increase for increasing number of cores and problem size, in all the mesh-size-per-core configurations. Then we look at the scaled speedup, defined as $scalfactor * T_{min_p}/T_p$, where $scalfactor = 1, 8, 64$, for the three increasing number of cores, T_{min_p} is the total time for solving linear systems when the minimum number of total cores is involved in the simulation, per each problem size per core, and T_p is the total time spent in linear solvers for all the increasing number

of cores used for the specified problem size per core. We observe that the best values are obtained with the *MLVSMATCH3* and *MLVSMATCH4* preconditioners when the largest and medium mesh size per core are used. In details, for the largest mesh size per core, *MLVSMATCH3* reaches about 71% of scaled efficiency on 3072 cores and about 44% of scaled efficiency on 6144 core when the medium size per core is employed, that are very good values for this memory bound problems. This shows that the scalability of *MLVSMATCH3* is very promising in facing the exascale challenge, especially when the resources are used at their best in terms of node memory capacity and bandwidth. On the other hand, in the case of the smallest mesh size per core (bottom figure), the scaled speedup of *AlyaDefCG* is better; this is essentially due to the very large solve time spent by this solver on 192 cores. In Figs. 20-21 we can see the AMG4PSBLAS preconditioners setup time and the

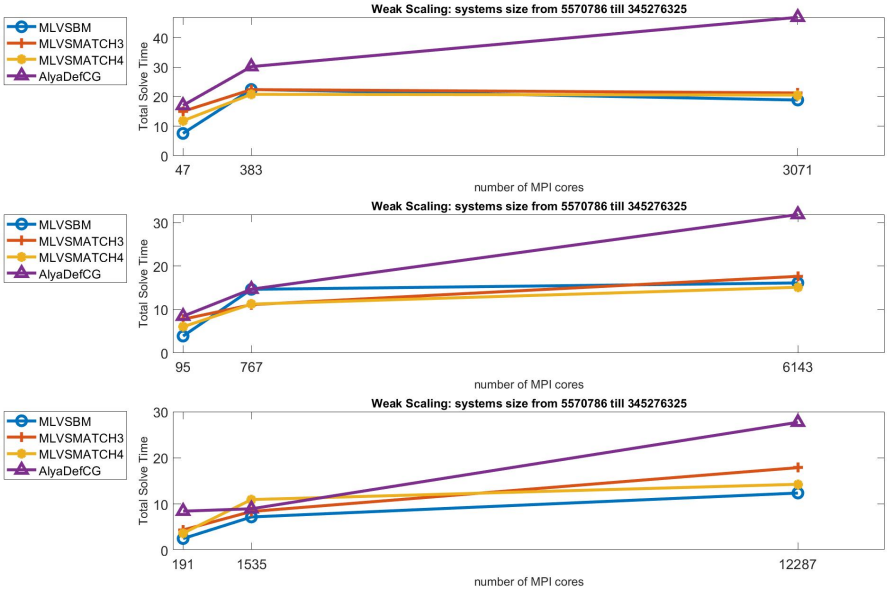


Figure 18: Weak scalability: total solve time of the linear solvers. $1.1e5$ dofs per core (top), $5.9e4$ dofs per core (middle), $2.9e4$ dofs per core (bottom)

corresponding scaled speedup. We observe that, as expected, also in this case the best scalability in the setup phase is obtained by *MLVSBM* in all the cases. Also for setup as for solve, the best scaled speedup values for all the preconditioners are obtained when the largest mesh size per core is used. On the other hand, the case of the smallest size per core requires a more deep investigation, due to the large oscillations in the execution times we observed on the Marenostrum 4 supercomputers, when many nodes are used at their full capability, i.e. using 48 cores per node.

7. Task 3.5: Transversal activities

7.1 PSCToolkit: PSBLAS and AMG4PSBLAS

Partners: CNR, UNITOV
Software packages: PSCToolkit, PSBLAS, AMG4PSBLAS

During the duration of EoCoE-II, we have revised and improved the PSBLAS linear

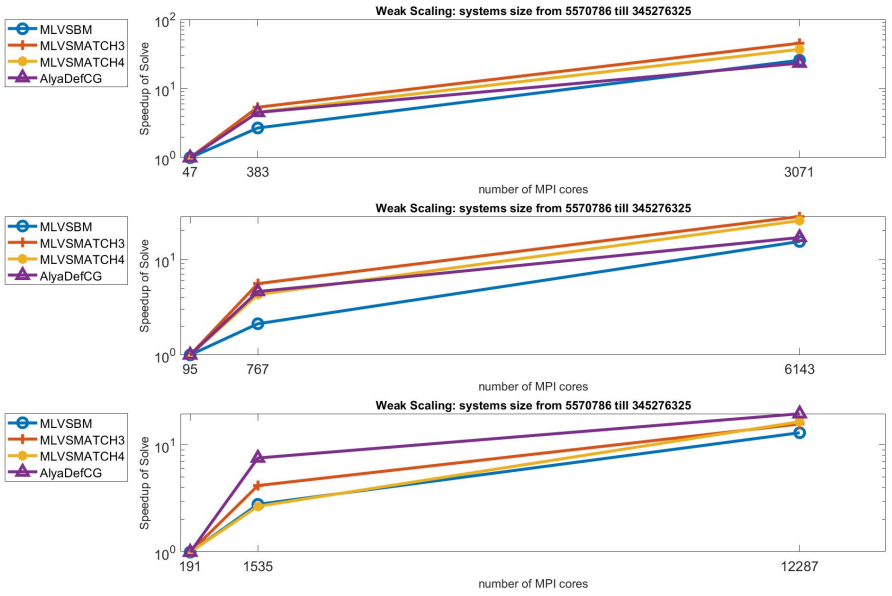


Figure 19: Weak scalability: scaled speedup of the linear solvers. $1.1e5$ dofs per core (top), $5.9e4$ dofs per core (middle), $2.9e4$ dofs per core (bottom)

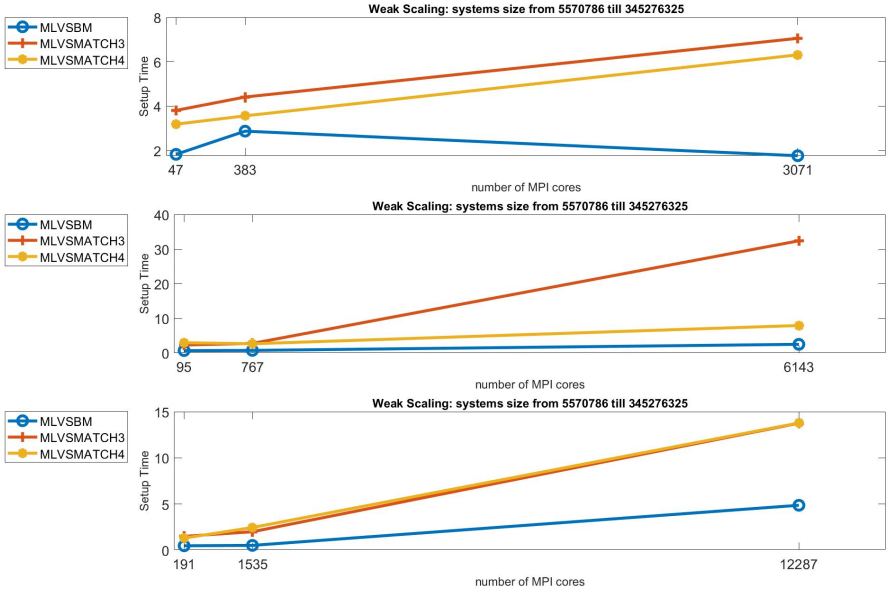


Figure 20: Weak scalability: AMG4PSBLAS preconditioners setup time. $1.1e5$ dofs per core (top), $5.9e4$ dofs per core (middle), $2.9e4$ dofs per core (bottom)

algebra package, we have started a novel package AMG4PSBLAS, which is a substantial evolution of the previous MLD2P4 package, and we have defined a combined toolkit PSCToolkit containing both packages, together with some other support tools.

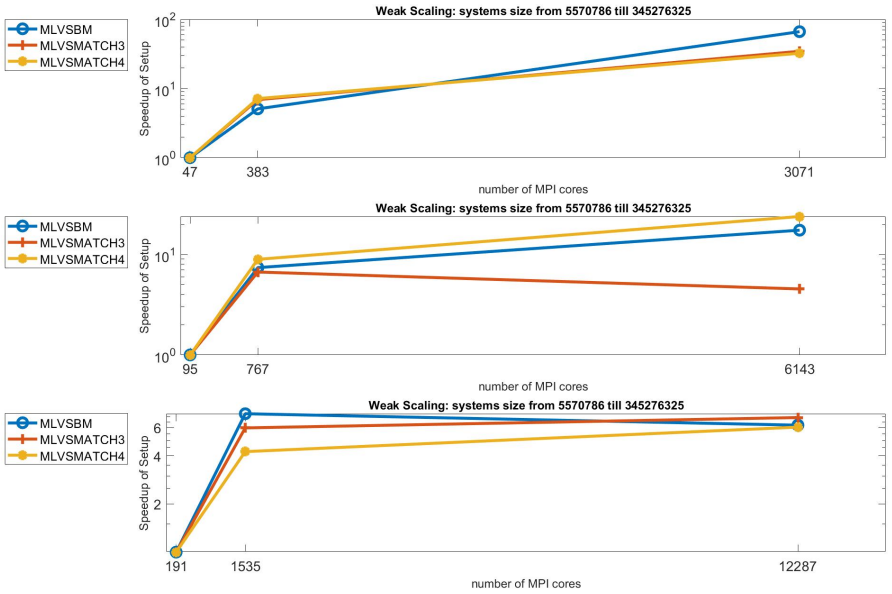


Figure 21: Weak scalability: scaled speedup of AMG4PSBLAS preconditioners setup. $1.1e5$ dofs per core (top), $5.9e4$ dofs per core (middle), $2.9e4$ dofs per core (bottom)

PSBLAS. Our development is based on the PSBLAS framework [33, 34]. Originally introduced for clusters that at the time were large-scale, it has gone through a number of revisions to keep up with the technology development of the past two decades, and the movement towards exascale is no exception. The software framework contains the computational building blocks for Krylov-type linear solvers on parallel computers, as well as support infrastructure to ease the writing of a parallel application using them. In particular, we introduced:

1. a framework for handling the mapping between the global index space of the problem and the local portions of the data structures;
2. the handling and optimization of the halo data exchange, also known as nearest-neighbour data exchange, the essential communication kernel;
3. an object-oriented architecture that enables choosing storage formats for sparse matrices and switching them at runtime to adapt to the application needs [33];
4. a plugin for seamless integration of GPUs [34].

During the development of the EoCoE project we have improved the handling of large index spaces requiring 8-byte integers, streamlined the process of setting up the data structures for halo data exchange, and also implemented some new computational kernels prompted by the extension of the preconditioners package.

In applications dealing with a large and sparse linear system, the system matrix is typically associated with a graph, examples being the discretization mesh of a PDE and the graph representing a complex network. All such applications handle the global numbering of the graph, which induces the global numbering of unknowns and matrix indices. In

normal practice the global graph/matrix is partitioned and split among processes, and each portion local to a process is handled through a local numbering scheme. The solution in PSBLAS is to have an *index map* object contained in the *communication descriptor* to keep track of the correspondence between local and global indices. With the target of handling more than 10^{10} degrees of freedom (dofs), it is clear that global indices require 8-byte integers, but that does not necessarily mean that any individual portion will require the same; indeed, having the local portions of the matrices run over 4-byte integers enables memory savings that can be quite significant, especially when we consider accelerators such as NVIDIA GPUs which do not support virtual memory and for which memory management is a major concern for the developer. In the current development version of PSBLAS we can choose at configuration time the number of bytes for local and global numbering separately, with the default of using 4 bytes for local and 8 bytes for global indices.

One of the main design points of PSBLAS was to make it as easy as possible for the application developer to specify the distribution of the index space, with the only constraint that each global index/dof point is owned by one process; this is done at the time the descriptor for the index space is created. After this step, all processes need to figure out with whom they need to exchange data. In general there will be some mesh points whose value is needed to carry out the local part of the computation but are not locally owned, and are known as the *halo*; for each halo index, we need to know the owner process. This question would be easy to answer if we had available a vector mapping each index to a process; indeed, that is one of the possible ways to partition an index space, but for very large index spaces this would imply an excessive memory footprint. Instead, we normally keep an amount of auxiliary memory that is proportional to the number of local and halo indices on the current process, a solution scalable for increasing number of computational cores; this can be done in two main variants, with a set of hash tables, or by imposing the constraint that the global indices owned by a process must be contiguous. To help with the construction of the data exchange lists:

1. we have devised a new iterative algorithm to identify the owner process for a given non-local index;
2. we defined a new interface for the user to provide additional information about the process topology, if available;
3. we create, when necessary, a copy of an existing index map employing a renumbering into a block-contiguous format, so as to speed up subsequent halo ownership identification.

Finding the process owning halo indices is equivalent to establishing a process topology mesh; the algorithm to identify index owners is based on the concept of *neighbouring processes*, i.e. processes that own indices needed by each other. The new index-owner identification is implemented with an iterative procedure which improves memory handling; the procedure can also employ user provided process topology information, if available. As an example consider the simple 2D finite-difference mesh in fig. 22a; with a standard 5-point stencil, we get a process communication topology as in fig. 22b. The procedure would be able to reconstruct the topology automatically for an *arbitrary* discretization; the user can improve the setup time if topology information is available in advance.

All these aspects are handled internally by the software with minimal input by the

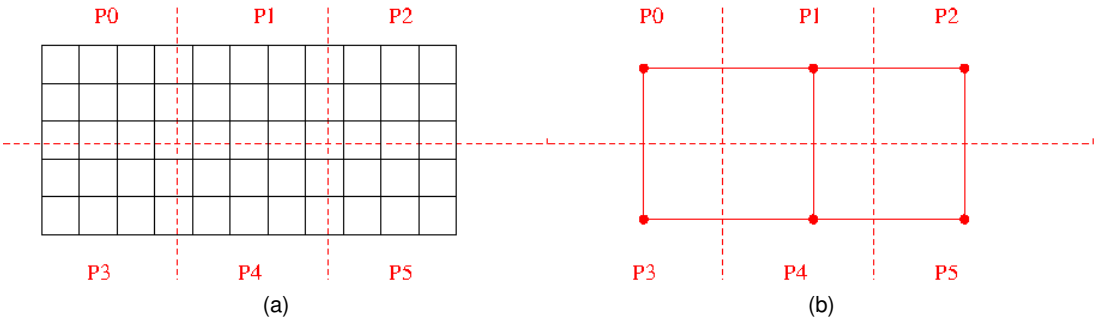


Figure 22: An example 2D mesh and its process topology.

user; they influence the setup time of the linear system and of the preconditioner, but have essentially no impact on the runtime of the solver methods. For further details see [22].

AMG4PSBLAS. In [23] we proposed a package of AMG preconditioners built on top of the PSBLAS framework; the first version of the package implemented a multilevel version of some domain decomposition preconditioners of additive-Schwarz type and was based on a parallel decoupled version of the smoothed aggregation method described in [66] to generate the multilevel hierarchy of coarser matrices. In the course of EoCoE 2 we have designed and introduced a new and improved package, described in [22], which inherits all the new features of the PSBLAS infrastructure and provides significant extensions over the previous version in terms of algorithms and software modules.

We are therefore improving flexibility, robusteness and computational complexity, but we preserve the numerical scalability and concurrency of the preconditioners when tens of thousands cores are used, whilst at the same time including support for GPU accelerators.

We have implemented a parallel aggregation scheme for coarsening on large distributed-memory architectures. The method, named *coarsening based on compatible weighted matching* was first introduced in [26] and is already available in the sequential package described in [25]. A first parallel version of the method, exploiting fine-grained parallelism and specifically tailored for single GPU device is described in [10, 11]. The method is independent of any heuristics or a priori information on the *near kernel* of A, i.e., the lower part of the range of eigenvalues of the system matrix A which is generally used to obtain good-quality aggregates, and it is a completely automatic procedure applicable to general s.p.d. systems. Furthermore, the coupled coarsening based on compatible weighted matching has the advantage of building coarse matrices which are well-balanced among parallel processes; there is no need for special treatment of process-boundary dofs accounting for inter-processes coupling, as often happens in the coarsening procedures available in widely used software libraries. Finally, there is a significant flexibility in the choice of the size of aggregates: it is possible to have an almost arbitrarily aggressive coarsening.

The *coarsening based on compatible weighted matching* is a recursive procedure which starts from the adjacency graph $G = (\mathcal{V}, \mathcal{E})$ associated with the sparse matrix A, where the vertex set $\mathcal{V}$ consists of the row/column indices of A and the edge set $\mathcal{E}$ corresponds to the index pairs (i, j) of the nonzero entries in A. A *matching* $\mathcal{M}$ in the graph G is a subset of edges such that no two edges are incident on the same vertex. In our method we associate to the graph G a suitable edge weight matrix C, computed from the matrix A and an arbitrary

vector w, and exploit the maximum product matching as a tool to obtain good quality aggregates for fast convergent AMG preconditioners.

Let C be the following weight matrix:

$$(C)_{i,j} = c_{i,j} = 1 - \frac{2a_{i,j}w_i w_j}{a_{i,i}w_i^2 + a_{j,j}w_j^2}, \tag{9}$$

where $a_{i,j}$ are the entries of A and $w = (w_i)_{i=1}^n$ is a given vector, and let $\mathcal{M}$ be a *maximum product matching* in the graph G with edge weight matrix C, i.e. $\mathcal{M} = \arg\max_{\mathcal{M}'} \prod_{(i,j)\in\mathcal{M}'} c_{ij}$. By applying a maximum product matching we can define the aggregates $\{\mathcal{G}_p\}_{p=1}^{n_p}$ for the row/column indices $\mathcal{I}$ of matrix A, consisting of pairs of indices, where $n_p = |\mathcal{M}|$ is the cardinality of the graph matching $\mathcal{M}$. Equivalently, we are decomposing the index set as

$$\mathcal{I} = \bigcup_{p=1}^{n_p} \mathcal{G}_p, \quad \mathcal{G}_p \cap \mathcal{G}_r = \emptyset \text{ if } p \neq r.$$

An unmatched vertex corresponds to a singleton G_s, and n_s is the total number of singletons. In our parallel coarsening we use the `MatchBox-P` software library, which implements the parallel algorithm for the computation of half-approximate maximum weight matching described in [21]. This algorithm has a complexity $\mathcal{O}(|E|\Delta)$, where $|E|$ is the cardinality of the graph edge set and Δ is the maximum vertex degree[9], and guarantees a solution that is at least half of the optimal weight. The `MatchBox-P` algorithm is based on the idea of identifying *locally dominant* edges, i.e., edges with largest weight for both end-vertices.

Based on the matching, we build a prolongator matrix P that is a piecewise constant interpolation operator; the recursive application of the above procedure defines an unsmoothed-type aggregation coarsening whose quality and convergence analysis have been discussed in [24] If the prolongation operator P is a piecewise constant interpolation operator, the V-cycle proves inadequate to obtain an optimal AMG; it is then necessary to employ more robust cycles such as general Algebraic Multilevel Iteration (AMLI) [67]. IN our library we also employ a Krylov-based MG cycle called the K-cycle, where at each level except the fine and the coarsest ones we apply two iterations of a Flexible Conjugate Gradient (FCG) method with the already defined AMG method starting on the current level as preconditioner [57]. An alternative to improve convergence while still employing a single V-cycle is to consider the use of a more regular interpolation operator obtained by applying one step of a weighted-Jacobi smoother to the basic piecewise constant interpolation. Scalable AMG relies on smoothers that are both highly parallel and robust. For s.p.d. matrices, a common choice with a good smoothing factor is the *Hybrid Gauss Seidel (HGS)* method, which has been demonstrated to be convergent with better smoothing properties than the block-Jacobi method when the local diagonal block of the matrix is sufficiently large with respect to the off-diagonal portion [2]; a weighted version of the method, named ℓ_1–HGS, can also be useful.

An alternative smoother can be based on sparse approximate inverses, which are quite efficient and well suited to GPU implementations since their main kernel is a sparse matrix-vector product. There exist several different algorithms for computing a sparse approximate inverse; we focus here on the inversion and sparsification of an incomplete factorization introduced in [64]. This strategy is based on the application of a sparse inversion technique for the triangular factors of an existing incomplete factorization in the

[9] The degree of a vertex is the number of edges that are incident on it.

form $M = LDL^T$, where, as usual, D is a diagonal matrix and L is lower triangular with an all ones main diagonal. In this way an expression for

$$M^{-1} = L^{-T}D^{-1}L^{-1} = ZD^{-1}Z^T$$

is obtained, and the application of the smoother is reduced to the computation of a matrix-vector product. To have sparse expressions for the incomplete factorization of A^{-1} it is necessary to employ a sparsification process during the computation of the matrix M^{-1}, i.e., a sparsification process for the matrix Z. As analyzed in detail in [16], the sparsification can be based on either a thresholding procedure or a positional dropping; we refer to [15, Chapter 3.5] for a complete discussion.

Performance Check

To measure the overall performance of the Krylov methods with parallel AMG preconditioners, we focus on both *algorithmic* and *implementation* scalability. Perfect algorithmic scalability is achieved when $\rho(B^{-1}A) \approx 1$ independently of the global size n of the linear system; implementation scalability on the other hand corresponds to having an optimal application cost for B^{-1} that is $\mathcal{O}(n)$ flops per iteration while achieving parallel speedup proportional to the number of processes employed. In all cases we use the AMG methods as preconditioners for a Flexible Conjugate Gradient (FCG) algorithm. Algorithmic scalability can be analyzed by looking at the number of iterations needed by FCG to achieve a relative residual norm of 10^{-6} as the size n grows; implementation scalability is analyzed by considering both the total solve time for the procedure as well as the average time per iteration. We also analyze the timings for the preconditioner setup with increasing number of processes; all the timings reported in the figures are in seconds.

Regarding the memory footprint of the proposed multigrid hierarchies as well as the cost of the application of a V-cycle, a quantitative measure is given by the operator complexity

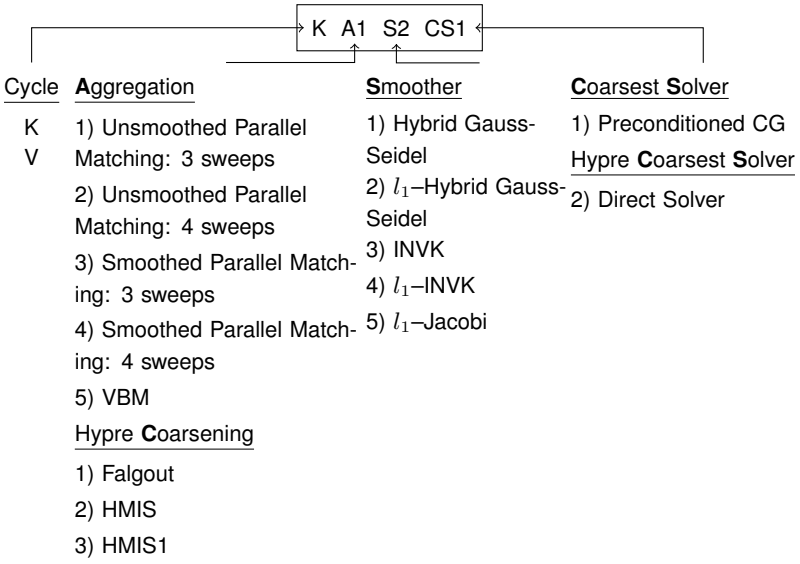
$$\text{opc} = \frac{\sum_{l=0}^{\text{nl}-1} \text{nnz}(A_l)}{\text{nnz}(A_0)} > 1.$$

For both AMG cycles we tested (K- and V-cycle), we always apply 1 pre-smoothing and 1 post-smoothing step. For the coarsest solver we use a version of the CG method coupled with a block-Jacobi preconditioner, with incomplete LU factorization (ILU(0)) on the diagonal blocks; the iterative solution of the coarsest system is stopped when the relative residual is less than 10^{-4} or the number of iterations is larger than 30. In the experiments discussed below, we applied the parallel coarsening based on compatible weighted matching starting from a vector w of all ones, since this vector is in the near kernel of our model test case. Furthermore, we used the library default approach which sets a target size of the coarsest-level matrix and stops the coarsening procedure as soon as the coarse matrix size is less than or equal to the target; in the experiments with the pure MPI version of the library we set $maxsize = 200 \times np$, where np is the number of cores.

To identify the different algorithmic variants implemented in `AMG4PSBLAS` and analyzed in our experiments, we use the labeling convention described in Table 6: the overall name reported in the figures and in the analysis is obtained by combining together the labels of the various components, e.g., KA1S1CS1 is the preconditioner employing a K-cycle, a hierarchy built with three sweeps of parallel coarsening based on matching and

unsmoothed prolongator (unsmoothed parallel matching, for brevity), one sweep of the forward/backward Hybrid Gauss-Seidel smoother and the preconditioned CG method as coarsest solver.

Table 6: Strings identifying each preconditioner are built by combination of the strings identifying the various algorithmic variants.



HGS shows a consistently good behavior with increasing number of cores which is reflected in the total solve time, as shown in Fig. 23a. In Fig. 23b we show the setup time for the different preconditioners, including the setup of the AMG hierarchy, the setup of the smoother operators at each level of the hierarchy, and the setup of the coarsest-level solver. The setup of the preconditoners with the HINVK smoothers requires a somewhat larger time; this is due to the additional (local) computations needed to perform the approximate inversion, whereas the HGS smoothers only require memory copies to generate the matrix splitting. The smoother setup time is completely flat, since it only depends on the size of the local matrix, whereas the hierarchy setup time tends to grow with the number of cores/size of the systems, hence the gap between the curves for the total setup time in Fig. 23b tends to close.

In all cases the setup times show good scalability with a sub-linear increase for increasing number of cores and dofs. We did the same analysis with a more aggressive coarsening using 4 sweeps of basic pairwise aggregation, i.e., when the size of aggregates is at most 16, which we denote as KA2-type preconditioners.

In Fig. 24a-24b we report number of iterations and solve time per iteration, respectively, when the different preconditioners are applied. As expected, in this case the number of iterations is generally larger than the case of KA1-type preconditioners, however we still get a fairly good numerical scalability, with a moderate increase in the number of iterations for increasing number of cores. The best convergence behavior is obtained when HINVK smoothers are employed, while the minimum number of iterations is generally obtained by $\ell 1$-HINVK. This good convergence behavior balances the small increase in solve time per iteration required by the HINVK smoothers, resulting in a total solve time which is generally better when $\ell 1$-HINVK is employed.For the sake of completeness we report in Fig. 25b the setup time for the different preconditioners based on the more aggressive

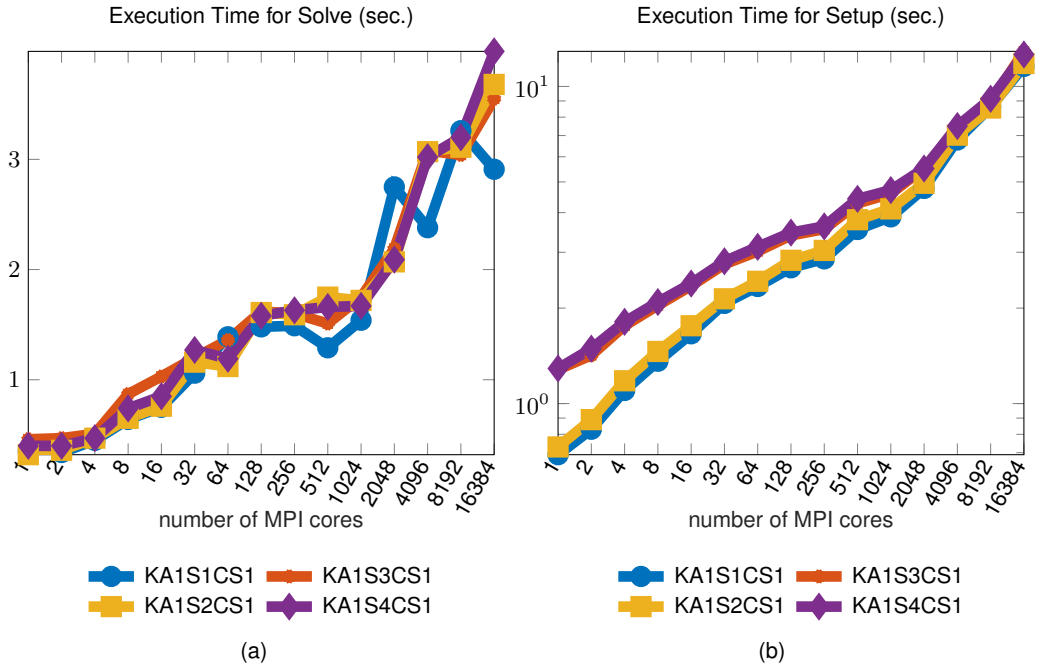


Figure 23: Weak scaling results 256k dofs per core. Execution times for the solve and setup for different smoothers when KA1-type preconditioners are used.

coarsening, showing a behavior similar to that of the KA1-type preconditioners.

Comparison with Hypre. We now compare our preconditioners with some of those available in the `Hypre` library [31], using default algorithmic parameters for best practice. Specifically, we compare with three different coarsening approaches, i.e., Falgout [37], [69, section 3.2], HMIS [27], and HMIS1 [70, section 3]. These coarsening schemes are used to generate an AMG hierarchy which is applied as a V-cycle preconditioner for a CG method, with one sweep of forward/backward HGS as pre/post-smoother. Default choices are used for the coarsest system, where a direct method is employed; the three `Hypre` preconditioners are denoted, respectively, VC1S1CS2, VC2S1CS2, and VC3S1CS2.

We compare these preconditioners from `Hypre` with the KA1S1CS1 and VA3S1CS1 methods in `AMG4PSBLAS` and with the VA5S1CS1 preconditioner, based on the parallel decoupled version of the smoothed aggregation strategy from [66, 63], already implemented in the previous version of the library. For these experiments we used the Piz Daint machine up to 8192 CPU cores. To complement this information we look also at the setup time for all the preconditioners. From the results in Fig. 27 we observe that using KA1S1CS1 has a similar cost to that of VC1S1CS2, while VA3S1CS1 shows a small increase in the setup time due to the application of 1 step of the weighted Jacobi smoother to the hierarchy prolongators. VA5S1C1, based on a decoupled smoothed aggregation, has a clear advantage in the setup cost due to the absence of communication in the aggregation algorithm, and obtains the best speedup for the setup phase. Very good speedups are also shown by KA1S1CS1 and VA3S1CS1, which confirms the effectiveness of the parallel implementations of all the computational kernels described. If we look at the solve phase in Fig.28, we can see that VA3S1CS1 has generally the best solve time with respect to the other preconditioners, with a very small increase with increasing number of cores. This efficiency compensates the small increase in the setup time with respect to the `Hypre` preconditioners

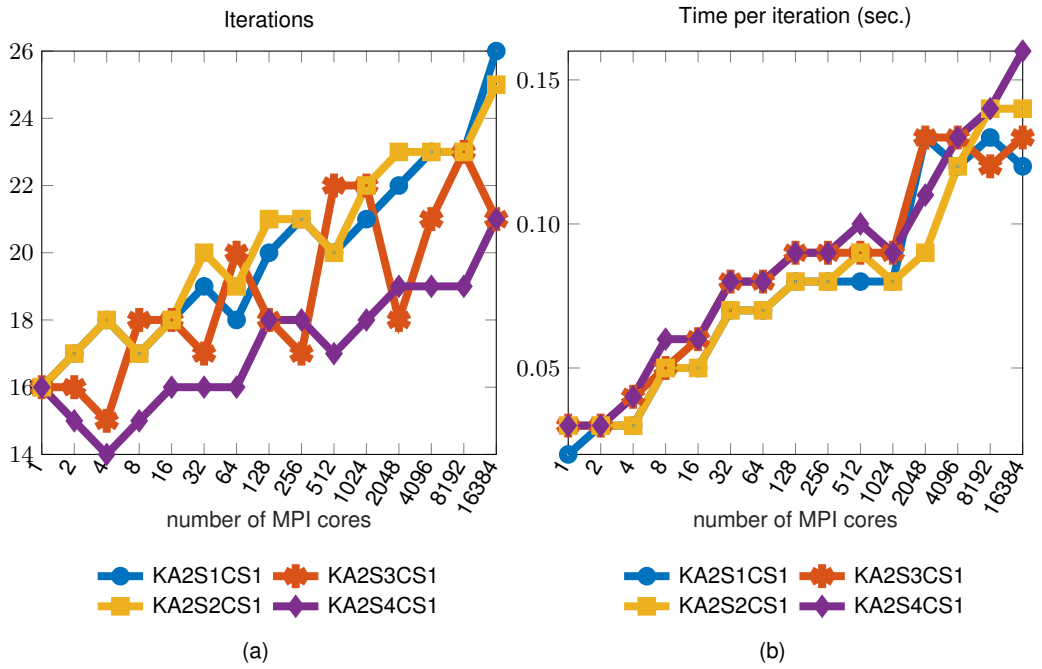


Figure 24: Weak scaling results 256k dofs per core. Number of iterations and time per iteration for different smoothers when KA2-type preconditioners are used.

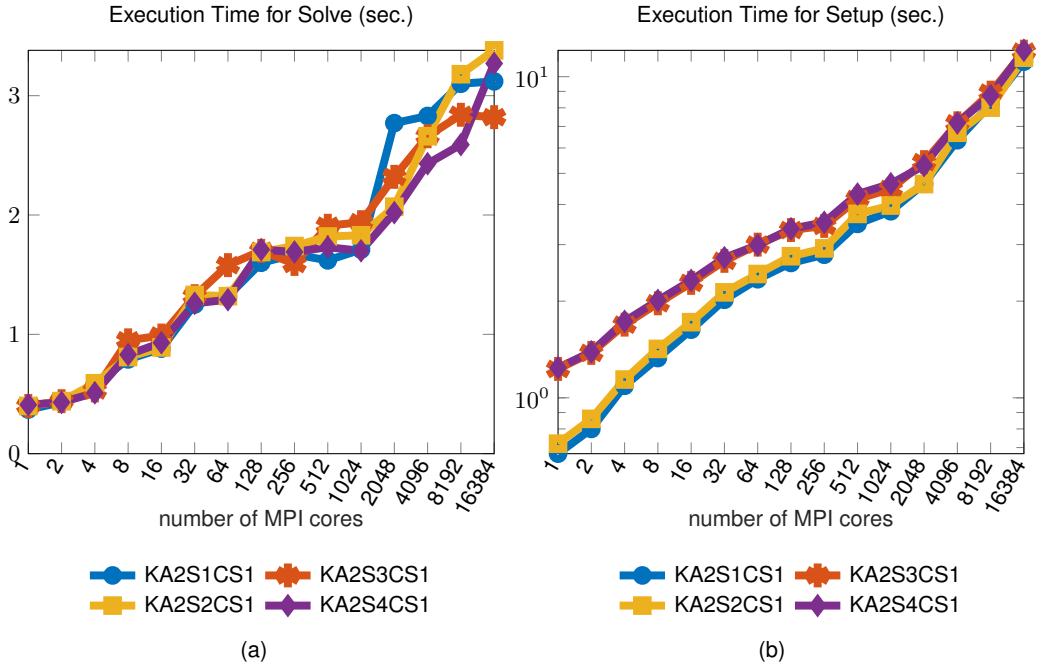


Figure 25: Weak scaling results 256k dofs per core. Execution times for the solve and setup for different smoothers when KA2-type preconditioners are used.

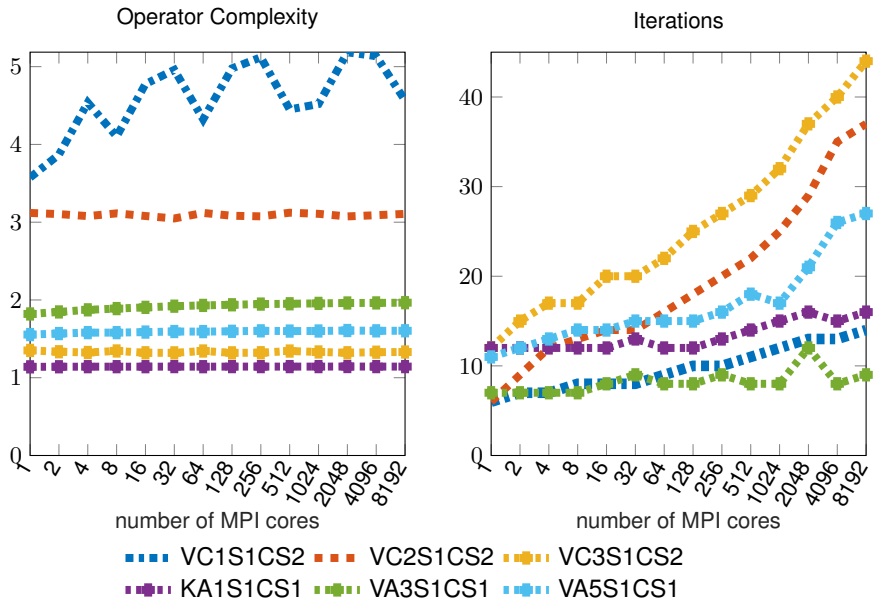


Figure 26: Comparison with `Hypre` 256k dofs per core. Operator complexity and number of iterations for different preconditioners.

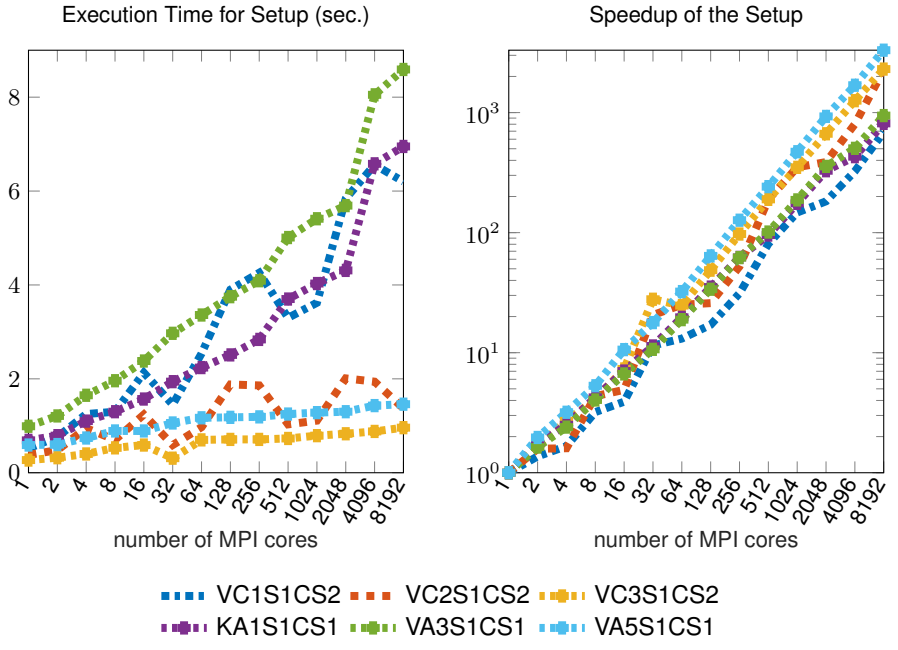


Figure 27: Comparison with `Hypre` 256k dofs per core. Preconditioners setup: execution time (left), speedup (right).

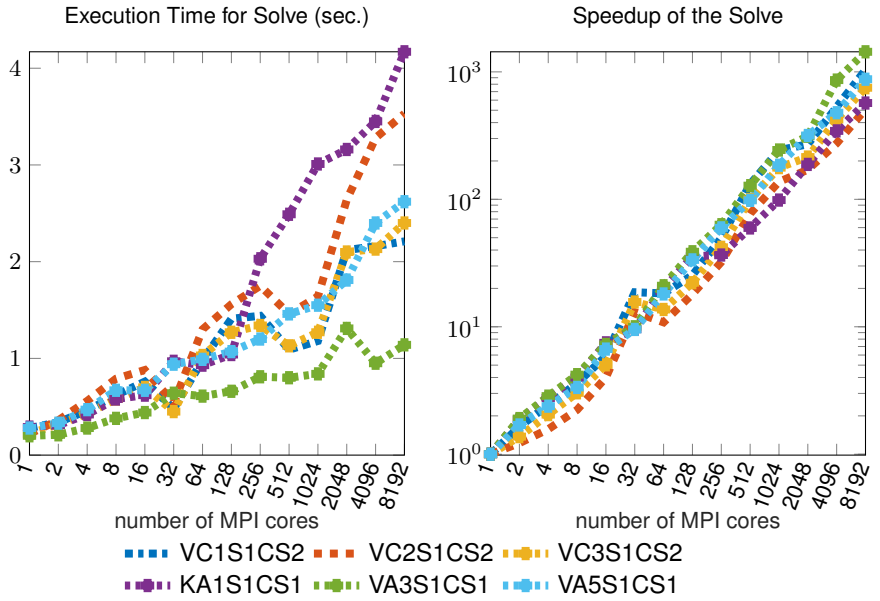


Figure 28: Comparison with `Hypre` 256k dofs per core. Solve: execution time (left), speedup (right).

when applied to time-dependent or non-linear problems, where the same preconditioner can be reused for multiple external iterations. We observe that our KA1S1CS1, although showing a very good algorithmic scalability, has a rapid increase in the solve time for increasing number of cores, due to the K-cycle application. Indeed this cycle requires $2^{nl-2}+1$ coarsest level visit and solutions per iteration, leading to a worse ratio between computation and communication with respect to the simpler V-cycle; hence, when the number of levels grows with the problem size so as to keep the coarsest matrix reasonably small, it shows poor scalability. The speedups of the solve phase for all preconditioners are broadly comparable.

Performance results towards extreme scale. In this section we discuss scalability results obtained on Piz Daint, running tests with 512×10^3 dofs per core up to 27000 cores, i.e., we reach an overall number of $\sim 1.4 \times 10^{10}$ dofs; in the same vein, we also analyze results obtained with GPU accelerators (in the solve phase). In the latter case we run tests with $12 \times 512 \times 10^3 \sim 6.2 \times 10^6$ dofs per GPU, and up to 2048 GPUs, i.e, we reach an overall number dofs of more than 1.2×10^{10}; in these experiments on GPUs we keep the same amount of memory per node, hence each GPU will handle the same number of dofs as 12 CPU cores. In the same vein, we stop the coarsening process for the setup of the multilevel hierarchy when the maximum size of the coarsest matrix is $12 \times 200 \times np$, where np is the number of GPUs, i.e. with the same size of coarsest matrix per node.

We begin with results on pure MPI; for the sake of space, we limit our discussion to preconditioners using the HGS smoother. We compare the KA1S1CS1 and KA2S1CS1 preconditioners (KA1/A2-types), using the K-cycle in the application of an AMG hierarchy where unsmoothed prolongators are employed, with the VA3S1CS1 and VA4S1CS1 preconditioners (VA3/A4-types), which use the V-cycle coupled with the smoothed version of the prolongators. In Fig. 29a we show the operator complexity of the multilevel hierarchies corresponding to all preconditioners. As expected, the operator complexity of the VA3/A4-type preconditioners, with smoothed prolongators, is larger than that of the corresponding KA1/A2-type preconditioners, since the coarse matrices are denser than

those built with the unsmoothed prolongators. Nevertheless, when aggregates of size 8 are built, the operator complexity is about 1.9, while when aggregates of size 16 are employed, the operator complexity is about 1.3. This indicates that even for VA3/A4-type preconditioners, the memory requirements for the AMG hierarchies are less than double the memory needed for the system matrix. Moreover, despite the small operator complexity, the numerical scalability of all the preconditioners is very satisfactory, even optimal for some of them; in Fig. 29b we see that KA1S1CS1 requires a number of iterations ranging from 12 to 17, while KA2S1CS1 requires a number of iterations ranging from 17 to 24, showing a small increase in iterations despite a reduction in operator complexity, and preserving numerical scalability for increasing number of cores. The VA3/A4-type

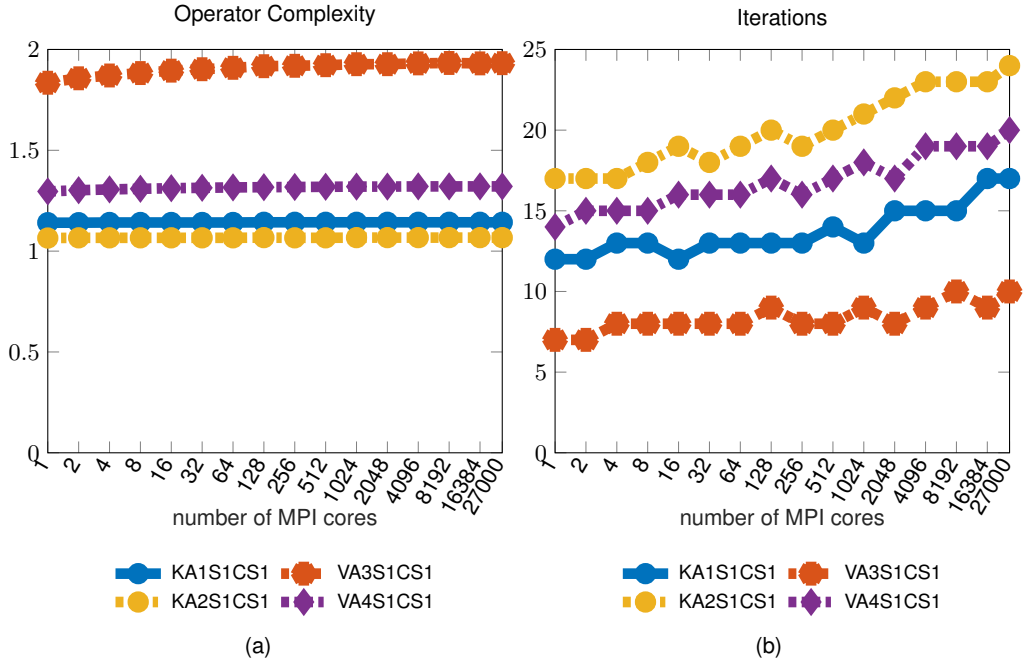


Figure 29: Weak Scaling results for 512k dofs per core. Operator complexity and number of iterations.

preconditioners, employing the smoothed version of the prolongators and a less expensive cycle, require a smaller number of iterations than the corresponding KA1/A2-type preconditioners. In more details, VA3S1CS1 continues to show an almost perfect algorithmic scalability with a number of iterations ranging from 7 to 10 going from 1 core to 27700. A very good algorithmic scalability is also observed for VA4S1CS1, where the number of iterations ranges from 14 to 20. This behavior confirms that the new parallel coarsening based on weighted matching is able to detect, in a completely automatic way, good quality aggregates of variable size and, exploiting smoothed operators allows to obtain optimality also using V-cycle at low operator complexity. VA3S1CS1 always obtains the best solve time per each number of cores, with a very slow increase for increasing number of cores; it solves the biggest size problem on 27000 cores in less than 2 seconds. In Fig. 30a the application of the more expensive K-cycle shows its effect on the solve time, especially when increasing the number of cores. Better solve times than the KA1/A2-type preconditioners are also obtained by VA4S1CS1 for increasing number of cores, showing that the use of V-cycle coupled with A3/A4 aggregation types is the best choice for extreme scalability, albeit at a small increase in the setup cost.For the sake of completeness we report in Fig. 30b the speedups obtained; we observe a very similar behavior for all meth-

ods, displaying a smooth increase with increasing number of cores, and demonstrating the good implementation scalability of all computational kernels.

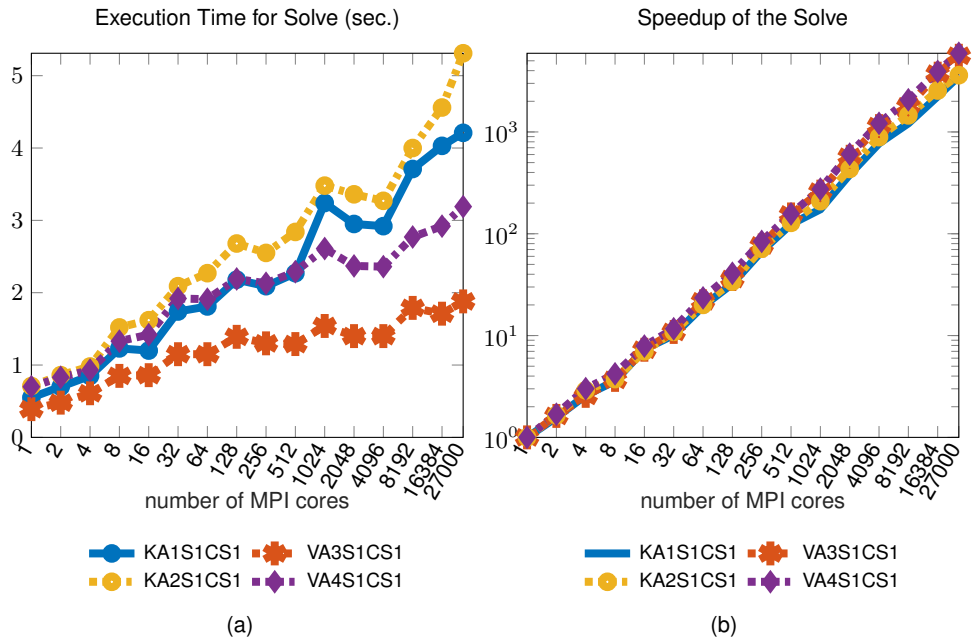


Figure 30: Weak scaling results 512k dofs per core. Setup and solve time with the relative speed-up.

The GPU plugin of PSBLAS allows us to run the solve phase on a cluster of GPUs, implementing the preconditioned FCG method coupled with some of the preconditioners included in AMG4PSBLAS. The plugin implements efficient GPU versions of the sparse matrix-vector products and vector-vector operations, such as vector updates and scalar products, including the necessary MPI communications [34]. In the following we discuss results obtained by employing VA3-type preconditioners coupled with both the HINVK and ℓ_1–Jacobi smoothers, whose implementation on GPU uses the sparse matrix-vector kernel. We compare VA3S3CS1 and VA3S5CS1, when they are coupled with the FCG iterative solver; in the case of VA3S3CS1, we apply HINVK also as preconditioner of the CG method at the coarsest level, while for VA3S5CS1 we use ℓ_1–Jacobi both as smoother and as preconditioner for CG at the coarsest level. For the ℓ_1–Jacobi smoother we apply 4 pre/post-smoothing sweeps of the method at each V-cycle application, whereas for HINVK we only apply 1 sweep. From Fig. 31a we see that both methods have a very similar behavior, showing a number of iterations ranging from 7 to 23 for VA3S3CS1 and from 7 to 26 for VA3S5CS1. The cost per iteration of VA3S3CS1 is generally better, as shown in Fig. 31b: we can observe a cost per iteration ranging from 0.04 to 0.14 seconds, corresponding to a solve time per dof in the range $[10^{-12} : 10^{-9}]$.

For the total solve time reported in Fig.32a, we observe that the two preconditioners have a similar behavior but VA3S3CS1 is generally better. In Fig. 32b we also show the setup time for both preconditioners. In the current version of the library the setup of the preconditioners is not yet implemented on the GPU: indeed, it is carried out on a single core of the CPU host device. We see that, as expected, VA3S3CS1 shows larger setup costs with respect to VA3S5CS1 due to the larger cost for the setup of the HINVK smoother. The smoother setup cost, as expected, is about constant for all numbers of CPU cores,

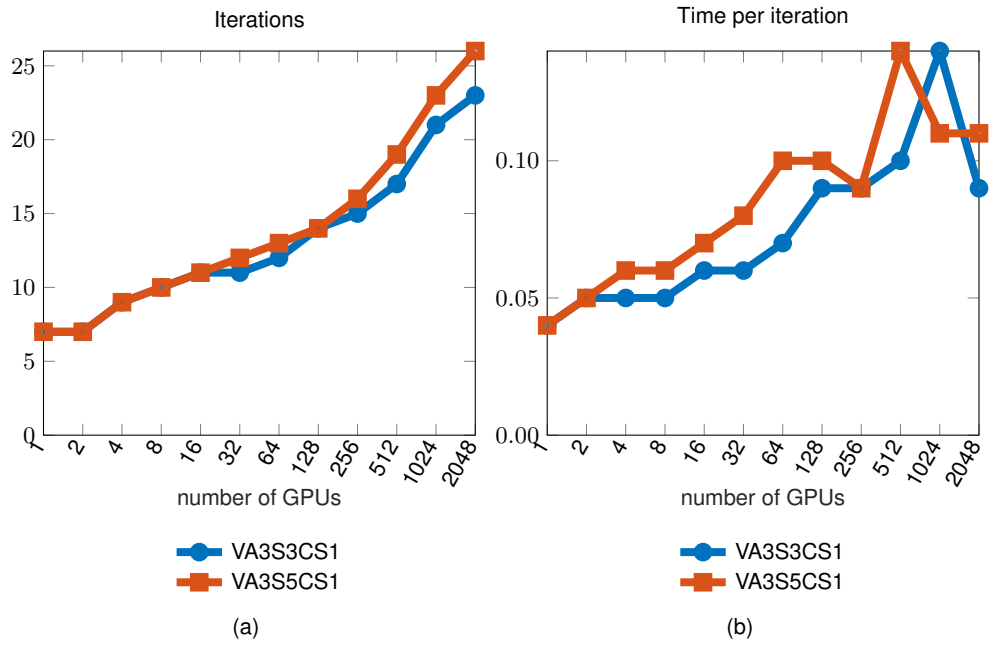


Figure 31: Weak scaling results 6M dofs per GPU. Number of iterations and time per iteration on GPUs.

with the ℓ_1-Jacobi smoother being less expensive by about 2 orders of magnitude. Further development activities planned for future releases of the library include the implementation of hybrid OpenMP-MPI and possibly CUDA versions of the HINVK setup phase to make better use of hybrid computing nodes. Let us observe that in time-dependent problems, such as in most of the test cases from the EoCoE-II project, large setup times are generally well-tolerated by the computational procedure if the solve phase is very efficient, since the same preconditioner is applied in a very large number of time steps.

7.2 HPDDM and MUMPS coupling

Partners: IRIT-CNRS
Software packages: HPDDM, MUMPS

The goal of this work is to investigate the efficient use of MUMPS solver within HPDDM, two linear algebra packages of the EoCoE II project. In particular, we study the potential computational gains that MUMPS and its BLR feature enhanced by the recent advances in multithreading optimization (see section 6) may provide.

We use PETSc's interface of HPDDM described in [38, 39] as preconditioner for a GMRES solver. HPDDM is a high-performance unified framework for domain decomposition methods. One of its main features is the construction of an abstract coarse grid by solving Generalized Eigenproblems in the Overlap (GenEO)[61] which allows the domain decomposition method to be robust. Another important improvement is the aggregation of the coarse operator on a subset of processes which prevents the coarse solver from being limited by an excessive amount of communications therefore ensuring good overall scalability when the number of sub-domains increase.

Let us recall the solve stages of the method where MUMPS can be used:

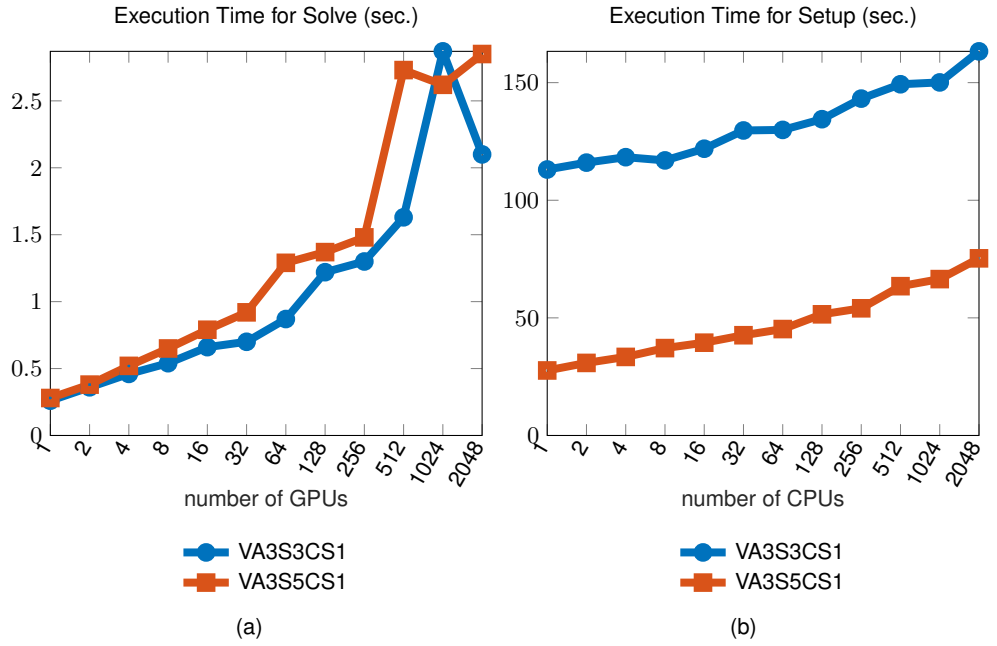


Figure 32: Weak scaling results 6M dofs per GPU. Solve time on GPUs and Setup time on CPU.

1. local solves of the restricted problem;
2. localized eigenvalue problems (GenEO) solved using SLEPc to compute the local deflation matrix;
3. one aggregated solve of the coarse problem.

Each local problem runs on a single MPI process. The coarse problem is solved on an aggregation of p MPI processes.

Experimental analyses reported in the literature [39], show that, in the method under evaluation, it is beneficial to use a high number of relatively small local subdomains, due to the superlinear complexity of the direct solver used within each subdomain; in this case, the direct solver can be run sequentially due to the small local problem size. On the other hand, the BLR and advanced multithreading features of MUMPS express their full potential on problems of relatively large size. The following study aims at determining the best configuration possible given fixed computational resources.

We consider 10 computing nodes of the Olympe supercomputer installed at the CALMIP supercomputing center of Toulouse (France). These include a total of 360 processors (bi-socket with 18 cores each socket). The considered problem is the linear elastic deformation of a heterogeneous rigid body clamped horizontally and subject to gravity as illustrated in Figure 33.

Since the matrix for the GenEO solver and the one for the restricted problem have the same connectivity pattern, we improved the HPDDM solver by reusing the MUMPS symbolic analysis for both problems.

We set the following options for HPDDM:

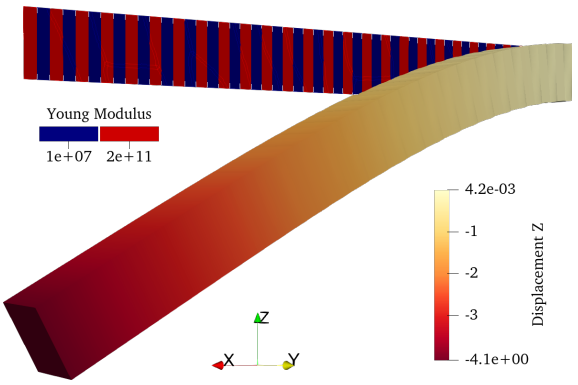


Figure 33: Horizontal heterogeneous beam subject to gravity

1. We reuse the analysis for both the GenEO and the restricted problem,
2. We ask the GenEO solver to provide 40 eigenvectors on each subdomain,
3. We agglomerate the coarse problem on 2 MPI processes.

As for MUMPS, we enabled the advanced multithreading optimization. Additionally, we disabled the pivoting. For each solver, we use an instance of the MUMPS Cholesky decomposition.

We performed several runs, with the different possible configurations from pure MPI to 1 MPI per socket for problem with up to 61 millions dofs.

In table 7, we report the timings of the three most representative runs. For this case, we saturate as much as possible the memory of all the 10 computing nodes by constructing a problem with 61 016 007 dofs. Note that the pressure on the memory decreases with the size of the matrix in each local problem. Indeed, the memory consumption for the factorization is of the order of $\mathcal{O}(n^{1.3})$.

The first run uses 360 MPI processes (one for each available core) so the local overlapping problems size is approximately 250 000 dofs. The coarse operator is of size 14 400 hence of negligible cost compared with the total time. For the second run we enable the BLR with a tolerance of 10^{-8}, for the local problems (both in the GenEO solver and the fine grid solver). MUMPS reports that, thanks to the BLR, the factorization performed 46% of the operations of the full rank case. For the third case, we enable multithreading on each MUMPS instance by dividing by two the number of subdomains and set the number of threads to two since the submatrices are of size 433 000 and the BLR compression allow the factorization to perform 30% of the full rank operations. It illustrates that reducing the number of sub-domains and therefore increasing the size of each sub-matrix is not compensated by the multithreading capability of MUMPS. This is partly due to the fact that we cannot (yet) use more than one MPI per sub-domain with HPDDM. An other factor that justifies this observation is that the coarse operator allows the outer GMRES to remain stable in number of iterations.

We detail the time spent in the direct solvers in figure 7. As mentioned previously, the cost of the resolution of the coarse operator is negligible.

Our observations show that the best configuration is full MPI where so that each local problem is as small as possible. In the case where the size of the local problem

JobID	BLR	MPI	Threads	Solver Time (s)	MUMPS Time (s)
1	0	360	1	359.11	199.98
2	$1 \cdot 10^{-8}$	360	1	326.18	175.02
3	$1 \cdot 10^{-8}$	180	2	415.43	202.29

Table 7: snapshot of the results of the coupling HPDDM-MUMPS, for a linear elastic problem with $61\,016\,007$ dofs.

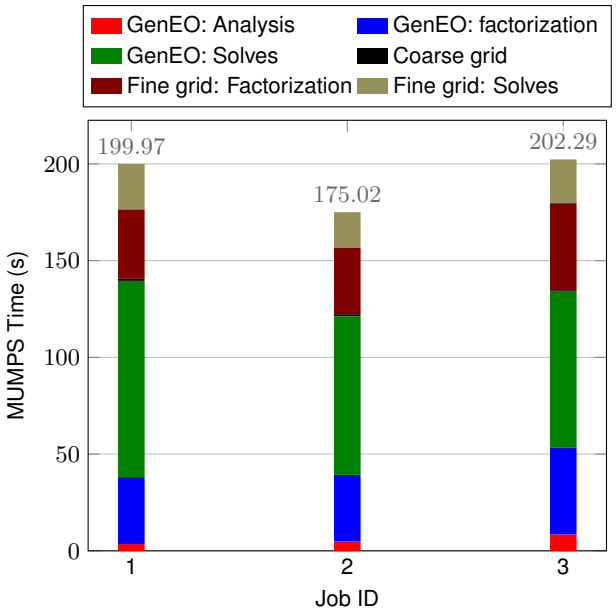


Figure 34: Repartition of the time spent in MUMPS solver for the three configurations reported in table 7

remains large, we can improve the local solvers with the BLR feature.

7.3 Porting AGMG to GPUs: Solve phase of the generic solver

Partners: ULB
Software package: AGMG

In this section we present the portage of the AGMG code [53] to GPUs. At this stage, we consider more particularly the portage of the sequential version of AGMG to 1 computing node having 1 GPU besides the CPU. Multi GPU variants will be developed thereafter based on this work and the multi-node (MPI) variant of AGMG.

AGMG implements an aggregation-based AMG method. Such type of solvers has two phases. In the *setup phase* a hierarchy of coarse systems is constructed, key to the efficiency of the iterative solution method that is used in the subsequent *solve phase*. The distinction is important because when several linear systems have to be solved with the same system matrix, the setup has to be done only once, hence accelerating the setup phase is relatively less important, especially when hundreds or thousands of solves are needed. This situation occurs, e.g., when solving fluid problems with a time stepping scheme or a pressure correction technique, that requires to solve at each step a discrete

Poisson equation for the pressure unknowns. Alya is an example of code that uses such technique.

This motivates the focus on the solve phase in the present work, the setup being performed with the standard CPU implementation of AGMG. A further motivation to go in this direction is that it seems hard to port the aggregation algorithm which is at the heart of the setup phase without making compromises regarding the quality of the aggregates and, therefore, the convergence rate. Hence, in case of many solves for a single setup, the variant presented here, which exploits the setup of the sequential version of AGMG, may well remain the best option.

More details on the results presented in this section can be found in [29].

Adaptation of the AMG solver

As already written, the setup is retrieved from standard (sequential) AGMG and hence nothing has to be adapted here.

The solve phase is based on an outer Krylov accelerator with multigrid preconditioning. The Krylov subspace method is the flexible conjugate gradient method if the system matrix is symmetric and the GCR method otherwise. The portage of these on GPUs is straightforward.

The multigrid preconditioner alternates smoothing iterations and coarse grid corrections. Regarding smoothing iterations, standard AGMG uses the Gauss–Seidel method, which is intrinsically sequential and, therefore, not competitive for a GPU implementation. Instead, we selected a combination of ℓ_1 Jacobi smoothing as presented in [3] with polynomial smoothing as presented in the same reference; see [29] for details. The used smoothing schemes amount to 2 pre- and 2 post-smoothing iterations (i.e., degree 2 polynomials) at fine grid level while only 1 pre- and 1 post-smoothing iteration (i.e., degree 1 polynomial) are applied at coarse levels.

The coarse grid correction amounts to solve approximately the problem transferred to a coarser grid, based on the aggregation computed during the setup phase. In standard AGMG, this approximate solution is obtained by performing 2 iteration of the same multigrid method at the considered coarse level. The method is thus used recursively, and since Krylov acceleration is used at all levels, this approach is called K-cycle. We implemented this approach in the GPU version.

However, since the computation of the inner products needed by the Krylov acceleration may be time consuming in parallel, we also tried to get rid of them using the W-cycle (which uses the same 2 iterations but without Krylov acceleration). Results were not satisfactorily with the standard W-cycle, because of a significant impact on the convergence rate. However, with a variant which we call *relaxed W-cycle* [29], it turns out that this impact is quite limited and that the method remains robust, as shown by the results below.

Finally, at some point, the coarsest level is reached, meaning that the recursion is stopped and the problem is transferred to a *bottom level solver*. Standard AGMG uses a sparse direct solver for this step. With the GPU version, we found that the best results are obtained when the bottom level solver is not applied on the GPU, but consists in a

transfer to the CPU on which a standard CPU solver can be applied. Indeed, the coarsest level has relatively few unknowns, hence GPU acceleration brings no more speedup.

For this CPU bottom level solver we could use a sparse direct solver as does standard AGMG, but, inspired by [56] we obtained even better results using this standard (CPU version) AGMG; that is, the bottom level solver consists in one application of the sequential AGMG preconditioner. This is because the grids on which a GPU brings no more speedup are significantly larger than those for which a sparse direct solver is faster than AGMG.

Test problems and specification

We tested the performance of our implementation on a collection of linear systems that correspond to finite difference (FD) and finite element (FE) discretizations of second order elliptic PDEs.

FD examples have been obtained considering structured uniform grids for PDEs defined on the unit square (2D) or cube (3D), either with constant coefficients (Mod2D, Mod3D) or with jumps and/or anisotropy in the PDE coefficients (Jump2D, Ani2D, Jump3D, Ani3D_a, Ani3D_b). Structured problems also include BFE, a uniform bilinear finite element discretization of a strongly anisotropic Laplacian on the unit square. More details on these problems can be found in [54].

Other problems are FE discretizations with unstructured meshes. In the 2D case (LRfUst), the problem is the Laplace equation on an L-shaped domain and the mesh has been strongly locally refined in the neighborhood of the reentering corner. In the 3D case (SphRf), the domain is the unit cube but contains a sphere in which the diffusion coefficient is 10^3 times bigger. For both these problems, classical Lagrangian Pi finite elements have been used, $i = 1, \ldots, 4$, implying quite diverse matrix properties and connectivity patterns. In the figures below, writing LRfUst_i or SphRf_i we refer to Pi discretization scheme. More details on these problems can be found in [52].

Each problem comes with three sizes, except 2D P4 and 3D P3 FE discretizations (2 sizes), as well as the 3D P4 FE discretization (1 size), because of memory limits (with these problems, the number of nonzero entries per row is significantly larger).

In each reported test, the flexible conjugate gradient method is used with the zero vector as initial approximation and the iterations are stopped when the relative residual error is below 10^{-6}. Results were obtained on a single-processor unit running with a i7-7800X processor with 6 cores and 12 threads at 4 GHz, a RTX 2080 Ti GPU (11GB GPU RAM) and 16GB of RAM.

Because, as discussed above, this study focus on the implementation of the solve phase, all timings reported are elapsed times (in seconds) for this phase only (set up time is neither included nor reported).

Speed-up with respect to sequential AGMG

We first report the time comparison with the original CPU implementation of AGMG. A sample of the results is shown in Figure 35. Results obtained for other test problems are similar.

One sees that the speed-ups are impressive, except sometimes for smaller problems for which the CPU version is actually already quite fast.

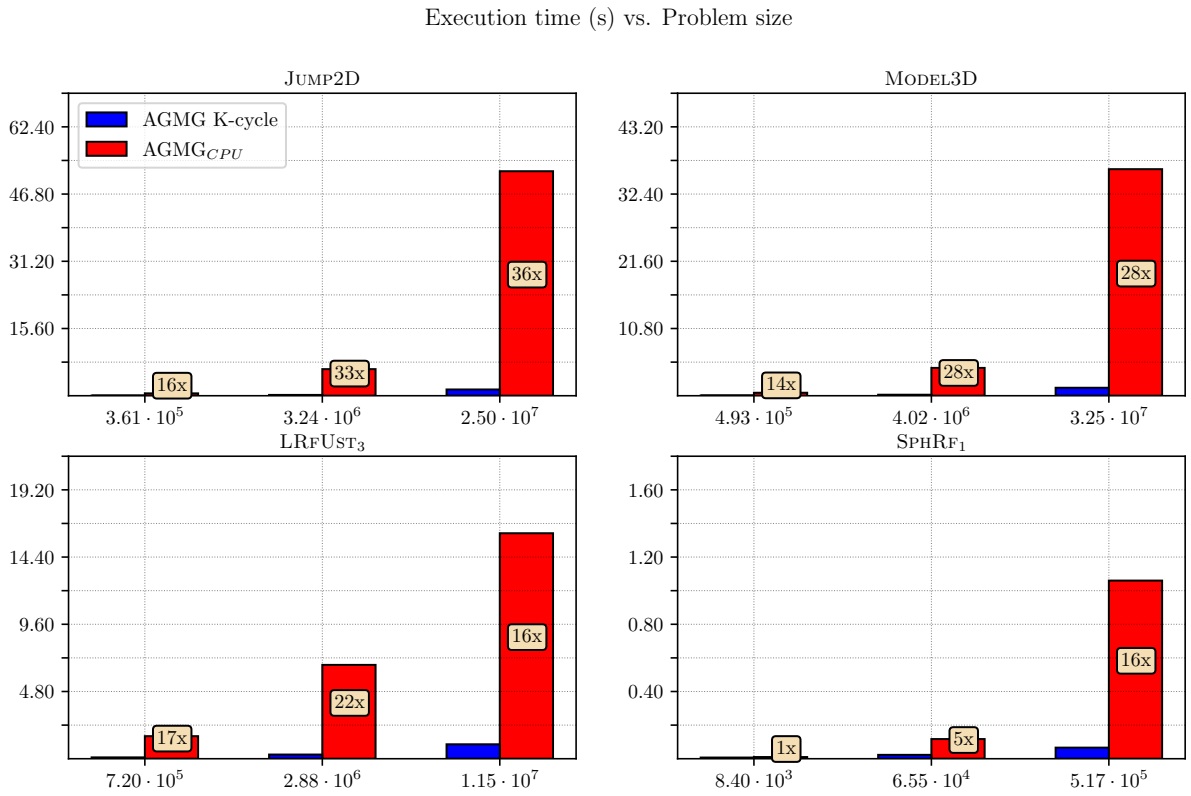


Figure 35: Comparison of AGMG ported to GPU (K-cycle variant) with the standard sequential (CPU) AGMG; the speedup is annotated over the bars.

Comparison with a reference GPU solver

Here we compare our solver with AmgX[10], a GPU AMG solver provided by NVIDIA. In our experiments, AmgX is configured to use an AMG V-cycle as preconditioner for the conjugate gradient method. AmgX is tested with two coarsening schemes: classical AMG with D2 interpolation (*AmgX classical*) and aggregation AMG with target aggregate size set to 4 (*AmgX aggregation*). These are similar to "standard" configuration files provided with AmgX. Options (D2 interpolation and aggregates of size 4) have been chosen as they give overall better results (lower running times) than alternatives. For the classical AMG case, 2 levels of aggressive coarsening (`aggressive_levels` option in AMGX configuration files) are used in order to prevent as far as possible AmgX setup from running out of memory. The maximal number of iterations is 300 for both configurations.

The results for structured meshes are reported in Figure 36 and those for unstructured ones are reported in Figure 37.

We first compare the performance of the K-cycle with the relaxed W-cycle. The K-cycle performs slightly better than the relaxed W-cycle because the K-cycle AGMG often requires slightly less iterations. However, in some examples the relaxed W-cycle wins over

[10] https://developer.nvidia.com/amgx

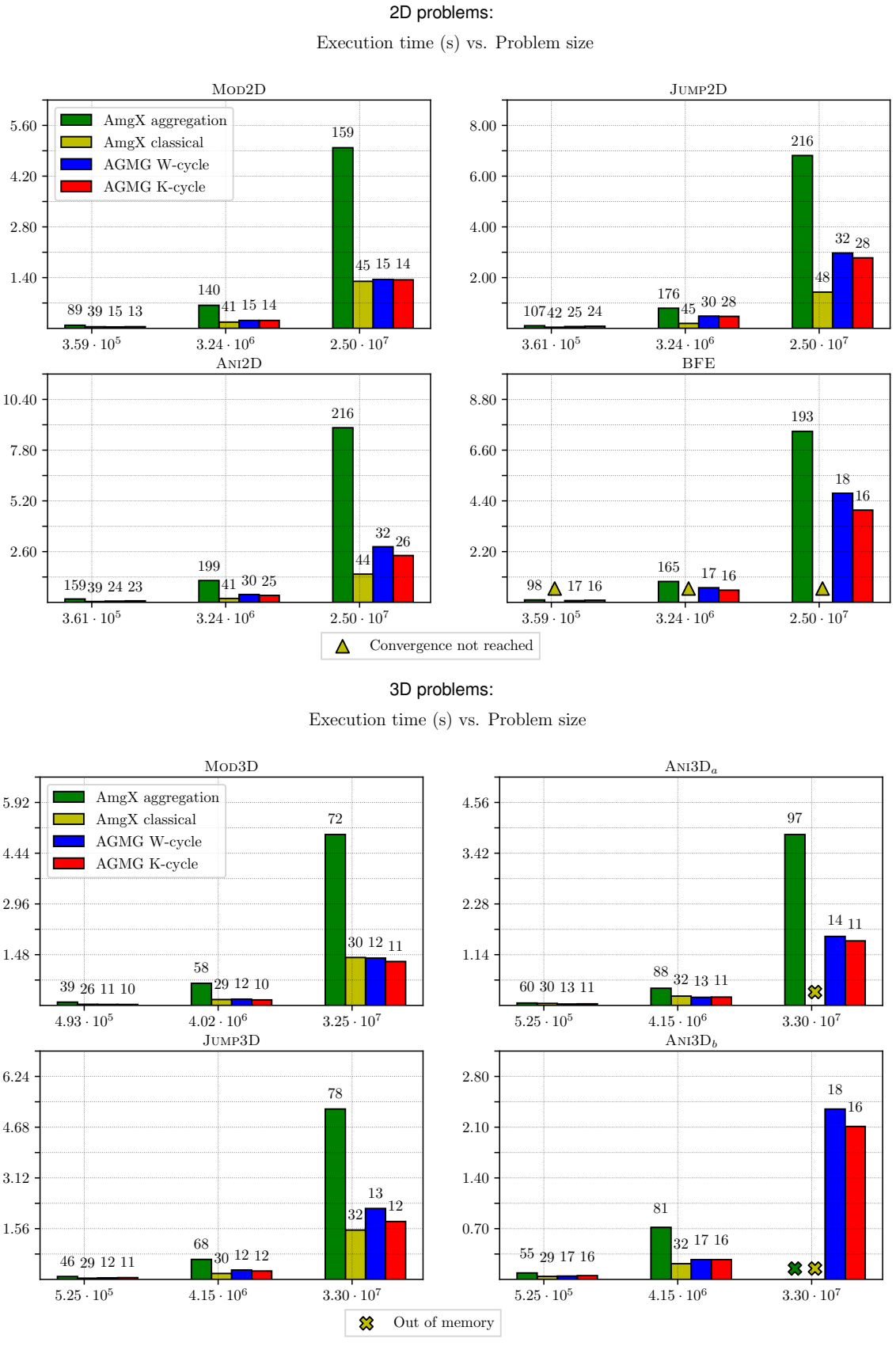


Figure 36: Comparison of AGMG with AmgX (both on GPU): Results for structured meshes; the number on top of each bar is the number of iterations. Solution time (y axis) is in seconds.

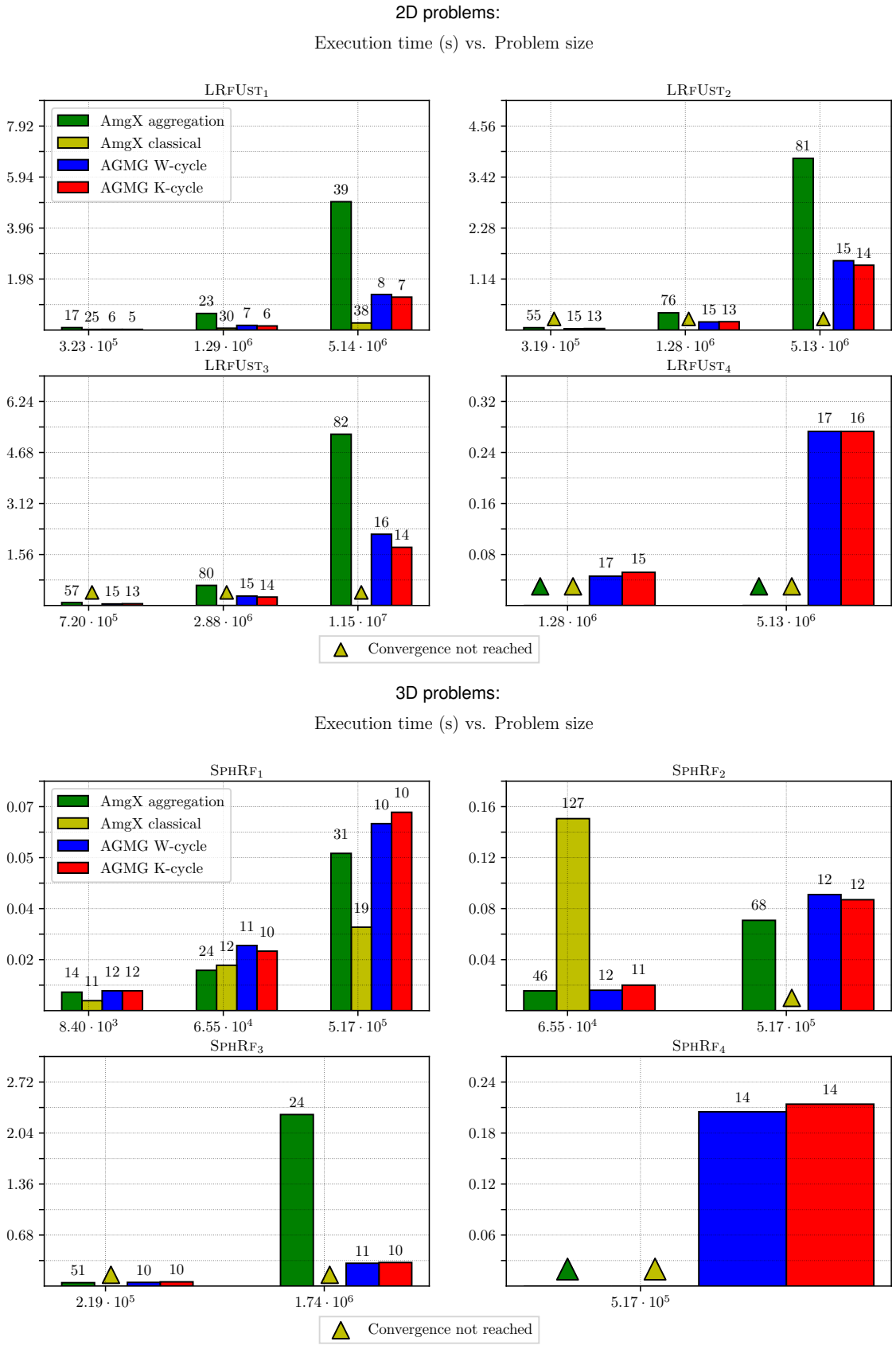


Figure 37: Comparison of AGMG with AmgX (both on GPU): Results for unstructured meshes; the number on top of each bar is the number of iterations. Solution time (y axis) is in seconds.

the K-cycle because the number of iterations are equal or close while the time per iteration is smaller for the relaxed W-cycle. Note the stability in the performance of both cycles for every problem.

This stability is even more noticeable when comparing to both AmgX configurations; AmgX aggregation requires always significantly more iterations and is nearly always significantly slower. In a few case it is faster that AGMG, but only marginally. On the other hand, AmgX classical may be significantly faster than AGMG, but lacks of robustness: for some problems it never converges, for some others it runs out of memory for the biggest sizes.

7.4 Porting AGMG to GPUs: Specific variant based on Stencil-CSR format

Partners: ULB
Software package: AGMG

Most PDE solver codes use general sparse matrix formats to encode the system matrix A, such as the common "Compressed Sparse Row" (CSR) or "coordinates" (COO) formats. However, A may have an underlying structure which can be exploited to reach the solution faster. Alternative approaches attempt to use this structure to accelerate linear algebra operations. In some approaches, called "matrix-free", the PDE solver code is designed differently: the matrix A is never explicitly formed and the linear system solver algorithm is tightly coupled to the rest of the code. In other approaches, the matrix A is formed but it is encoded in a restrictive, specialized sparse matrix format, such as the "constant stencil" format, in which the coefficients of every row (interpreted using the structure of the tensor product grid) are determined by the PDE discretization stencil.

These restrictive sparse matrix formats are more efficient in terms of memory and computation complexity than general formats such as CSR or COO. This is notably true on SIMD-like platforms such as vector computers or graphical processing units (GPUs). However, solvers based on these formats often lack of generality since they are typically embedded in the application they have been developed for, and can hardly be used outside it.

In this work, we consider a specialized general sparse matrix format suited for PDE problems with *piecewise*-constant coefficients and we apply it to the simpler case of constant coefficients PDEs, with a GPU implementation of operations. This format is based on separating the matrix into rows encoded in CSR format and blocks of rows in a flexible variant of a constant stencil format, hence we call it "hybrid stencil-CSR format", or "hybrid format" for short. This gives enough flexibility to encompass relatively general problems (with different geometries, boundary conditions, discretization schemes, etc) while obtaining, as will be seen, significant speed-up.

More details on the results presented in this section can be found in [17].

Hybrid format

The hybrid format introduced in this section is aimed at piecewise-constant coefficients PDE problems. It combines stencil representation for certain blocks of rows of the system matrix A, typically corresponding to regions of the domain with constant coeffi-

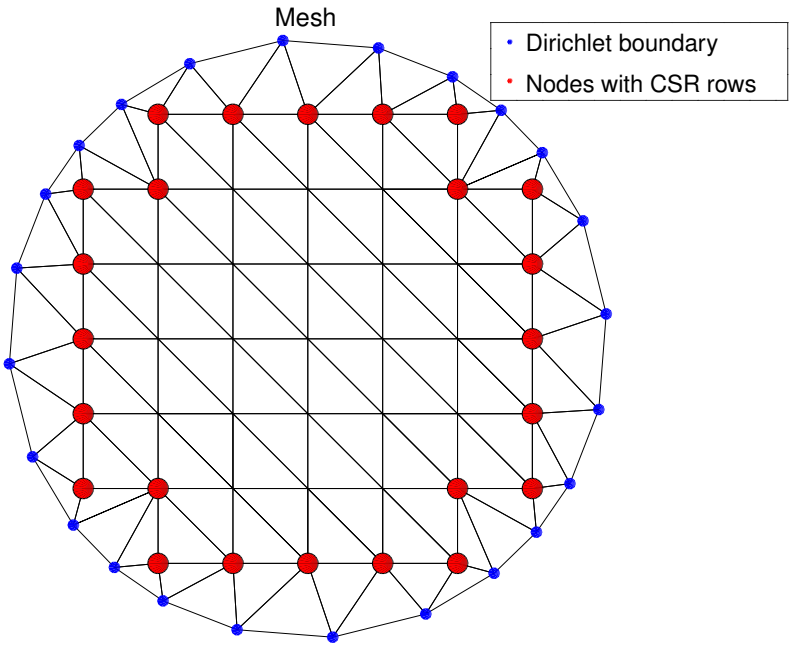


Figure 38: Example application of the hybrid format: P_1 finite elements discretization of the Laplacian $-\Delta$ on a disk. There are no *outer CSR* rows in this example.

cients, with CSR representation for rows of A which cannot be encoded in this way. The latter would usually correspond to internal (between constant-coefficients regions) and domain boundaries. In this description we implicitly consider that the discretization mesh is highly structured within the constant coefficients regions of the domain.

The stencil representation is similar to the "constant stencil" format mentioned above, but more flexible. A stencil-structured block of rows in the matrix is defined by specifying the stencil coefficients for the block, the geometry associated with the unknowns corresponding to the rows in the block, and finally CSR rows corresponding to the boundary of the stencil-structured region associated with the block.

Additional CSR rows are used to encode the other rows of A, those which do not belong to any stencil-structured block. These "outer CSR" rows are not given a geometric meaning in the format. Inversely, we will say that an unknown is *geometric* if the corresponding row of the system matrix A (CSR or stencil) is not an outer CSR row.

An example is given in Figure 38 for Poisson's equation $-\Delta u = f(x, y)$ on a disk domain with Dirichlet boundary conditions imposed by elimination (corresponding nodes are not associated with unknowns). In this case, there is only one stencil-structured block of rows and no outer CSR rows. The data for this stencil-structured block consists in: the logical coordinates (in $\mathbb{Z}^2$) of each geometric unknown on the cartesian grid visible in the mesh; the encoding for the CSR rows, shown as filled disks in Figure 38; and the stencil itself, defined consistently with the choice of grid:

$$\text{example stencil (5-point Laplacian)} = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}. \tag{10}$$

name	Short description	Dim.	Asymptotic nnz per row
2D FD	FD on L-shaped domain	2D	5
2D P_1	P_1 FE on acute triangle ($30^\circ - 75^\circ - 75^\circ$)	2D	7
2D Q_1	Q_1 FE on square (isotropic)	2D	9
3D FD	FD on stretched cube (dimensions: 1 – 1 – 100)	3D	7
3D P_2	P_2 FE on cube	3D	23.25
3D Q_1	Q_1 FE on cube (isotropic)	3D	21

Table 8: Test problems used in numerical experiments. All the problems arise from the discretization of Poisson's equation. FD = finite differences and FE = finite elements.

AMG solver

We aim at implementing an aggregation-based AMG method using this hybrid Stencil-CSR format.

We refer to the previous subsection (Section 7.3) for a rough description of the involved components. Regarding the solve phase, our approach uses essentially the same ingredients, except that here we only consider the relaxed W-cycle; see [17, 29] for more details.

Now, this has to be combined with an aggregation scheme adapted to the format of the data. For the stencil region, we designed an algorithm that retrieves the nearly optimal aggregation pattern at fairly negligible cost. For the CSR region, it turns out that a mere pairwise aggregation for this region is sufficient to obtain a relatively fast overall aggregation while ensuring robustness. We refer to [17] for additional details about the aggregation.

Test problems and specification

To assess the performance of our solver, we run it on 2D and 3D test problems listed in Table 8. These are typical of linear problems arising from the discretization of Poisson's equation on highly structured grids using the finite difference or finite element methods. Each of these problems can be instanced with a chosen resolution (determining the number of unknowns). All have square and symmetric matrices. When encoded into the hybrid (stencil-CSR) format, the domain in each problem becomes a single stencil-structured region bordered by nodes with associated CSR rows.

Each problem is discretized in 3 or 4 different sizes (denoted S1, S2, S2.5, S3) as shown in Table 9. For 2D problems, a jump in size from S1 to S2 or S2 to S3 corresponds to about 10 times as many unknowns for the discretized problem. For 3D problems, the corresponding factor is close to 8 and an additional intermediate size S2.5 is introduced between S2 and S3.

In each reported test, the flexible conjugate gradient method is used with the zero vector as initial approximation and the iterations are stopped when the relative residual error is below 10^{-6}. Results were obtained on a single-processor unit running with a i7-7800X processor with 6 cores and 12 threads at 4 GHz, a RTX 2080 Ti GPU (11GB GPU RAM) and 16GB of RAM.

Reported timings always refer to the total elapsed time (in seconds or milliseconds)

Problem	n	Stencil-CSR T	Stencil-CSR Iters	AGMG seq. T	AGMG seq. Iters	$\frac{T_{\text{AGMG seq.}}}{T_{\text{Stencil-CSR}}}$
2D FD	$6.6 \cdot 10^4$	27.	16	62.	17	2.3
	$6.6 \cdot 10^5$	53.	21	908.	19	17.3
	$6.6 \cdot 10^6$	257.	22	10560.	20	41.1
2D P_1	$4.9 \cdot 10^4$	29.	12	60.	21	2.1
	$4.9 \cdot 10^5$	74.	14	853.	22	11.6
	$4.9 \cdot 10^6$	285.	14	10420.	24	36.5
2D Q_1	$9.8 \cdot 10^4$	29.	18	140.	18	4.8
	$9.9 \cdot 10^5$	61.	20	1776.	20	29.1
	$9.9 \cdot 10^6$	337.	22	19740.	21	58.5
3D FD	$4.7 \cdot 10^5$	72.	11	628.	15	8.8
	$3.9 \cdot 10^6$	281.	11	6760.	16	24.0
	$1.3 \cdot 10^7$	677.	11	28200.	17	41.6
	$3.2 \cdot 10^7$	1401.	12	72800.	17	52.0
3D P_2	$4.5 \cdot 10^5$	132.	14	2252.	14	17.1
	$3.8 \cdot 10^6$	557.	14	21250.	16	38.2
	$1.3 \cdot 10^7$	1320.	14	74100.	16	56.1
3D Q_1	$4.7 \cdot 10^5$	95.	11	1811.	13	19.0
	$3.9 \cdot 10^6$	317.	11	17050.	14	53.7
	$1.3 \cdot 10^7$	742.	11	64000.	14	86.3

Table 9: Total times (T) in milliseconds, numbers of iterations and speed-up for the GPU Stencil-CSR solver in comparison with the standard sequential (CPU) AGMG.

including both the setup and solve phases. (Figure 39 allows one to see how this time is spread between setup and solve.)

Speed-up with respect to sequential AGMG

The results are displayed in Table 9. The speed-up is modest for the smallest and simplest problems (2D configurations), and ranges from 36 to 86 on the biggest sizes. Interestingly, the number of iterations is often smaller for the Stencil-CSR variant, showing the efficiency of the selected aggregation scheme.

Comparison with reference GPU solver

Here we compare our solver with AmgX[11], a GPU AMG solver provided by NVIDIA. We tested the same two configurations of AmgX described in Section 7.3, *AmgX classical* being here denoted *AMGX-Cl2* and *AmgX aggregation* being denoted *AMGX-Ag4*.

Running times are presented graphically in Figure 39. For all but one problem instance, the hybrid (stencil-CSR) solver has the fastest solve time. Considering total time, the hybrid solver is consistently the fastest at larger problem sizes ($> 4.8 \cdot 10^5$ unknowns). Considering set-up time, it is often beaten at smaller problem sizes by AMGX-Ag4 and just once by AMGX-Cl2. However, AMGX-Ag4 is never the fastest solver when adding solve time in the comparison.

[11] https://developer.nvidia.com/amgx

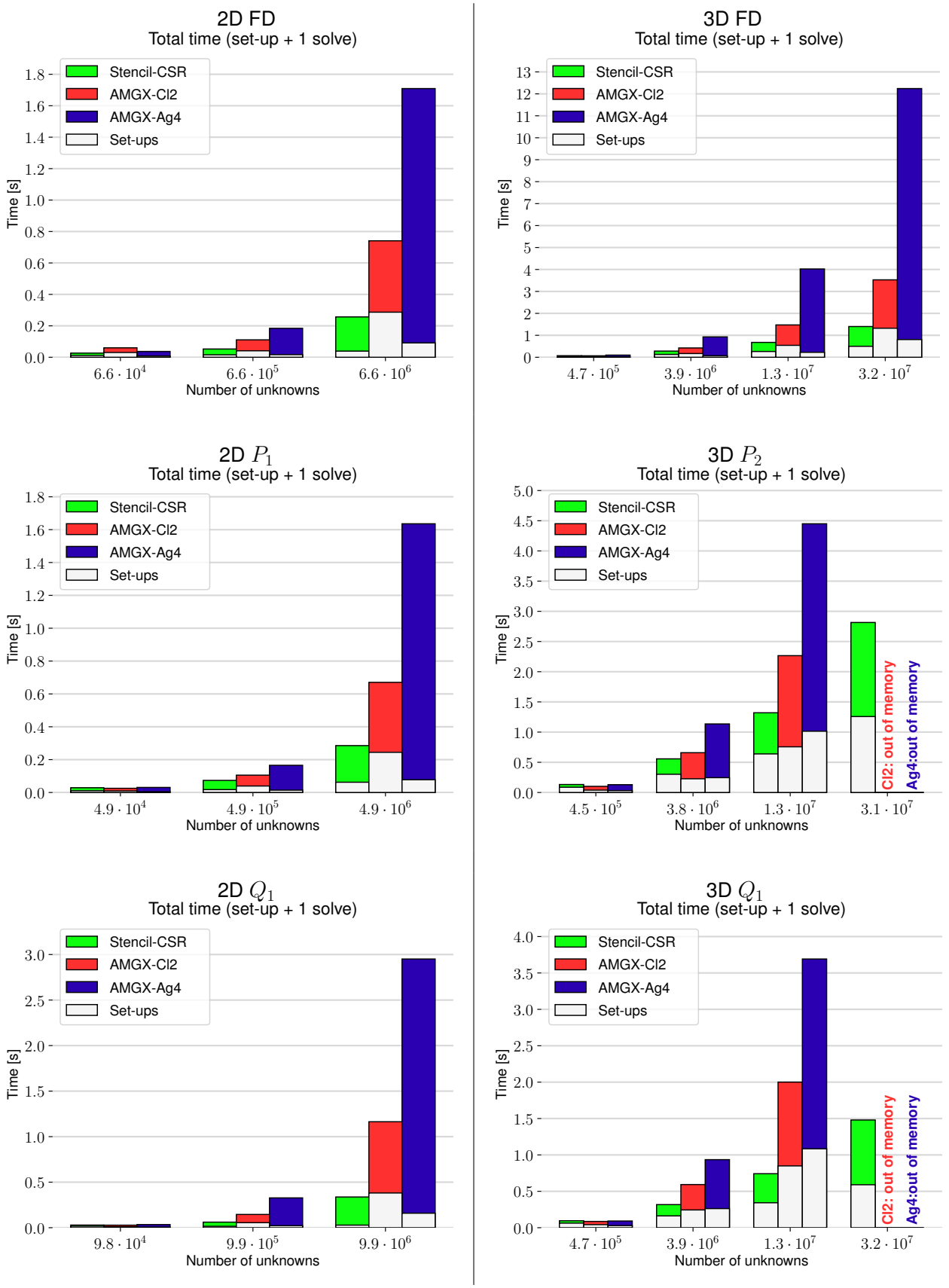


Figure 39: Comparison of Stencil-CSR AGMG with AmgX (both on GPU).

8. Summary

The results presented in previous sections reflect the diversity of the research actions. The most mature are oriented towards exascale. This includes the development of the PSCToolkit package considered in Section 7.1, where extensive scalability tests are reported. This LA package has further been fully integrated in the Alya flagship code and has been interfaced to the KINSOL package as a main step towards the integration in the Parflow code. Related scalability results are reported in Section 4.1 and Section 6.2 (respectively).

Regarding the number of cores involved in these experiments, one may object that one is still far from exascale. Here, the involved researchers unanimously complain that PRACE resources allocated to this EoCoE-II project have been constantly decreasing, making increasingly difficult to run very large scale experiments. Their feeling is that they have to face with an inconsistent policy from EU, which on the one hand insists on the orientation of the work towards the development of exascale-enabled solutions, and, on the other hand, does not provide a fair access to the largest facilities available so far.

The results in Sections 7.3 and 7.4 are also oriented towards exascale, as they fill a gap regarding the AGMG software by porting it to GPUs. Although currently only one GPU is used at a time, the combination of these results with the MPI implementation of AGMG [56] will enable in a near future to run on clusters of GPUs, which is one the prominent pre-exascale architecture.

Other sections are not yet concerned with pre-exascale, but report real success resulting from the integration of LA codes in EoCoE flagship codes. This includes Section 4.2 where very promising, although preliminary results, are obtained thanks to the integration of AGMG in SHEMAT-Suite. On the other hand, impressive savings are also reported in Section 6.1 thanks to the integration of MUMPS as subdomain solver in the domain decomposition method used by Alya to solve solid mechanics problems. The same type of integration is also considered in Section 7.2, which reports on the integration of MUMPS within the HPDDM package, this latter being a high-performance unified framework for domain decomposition methods. (This latter activity is reported as transversal activity because both MUMPS and HPDDM are LA packages.)

Finally, the results with solvers developed specifically for EoCoE flagship code are logically less mature, since these were developed from scratch at project start. Nevertheless, as can been seen in Sections 5.1 and 5.2, they are on the good way. Note that the development of HyTeG for GyselaX (Section 5.2) was not foreseen in the proposal, but has been decided as an alternative to the algebraic multigrid codes that have been tested previously [28], which lead to only mixed results due to the very particular nature of the underlying PDE problem.

Acknowledgments

We acknowledge PRACE for awarding us access to: SuperMUC-NG at GCS@LRZ, Germany; MareNostrum at Barcelona Supercomputing Center (BSC), Spain; HAWK at GCS@HLRS, Germany; Piz Daint at CSCS, Switzerland.

References

[1] Patrick Amestoy, Alfredo Buttari, Jean-Yves L'Excellent, and Theo Mary. On the complexity of the block low-rank multifrontal factorization. *SIAM Journal on Scientific Computing*, 39(4):A1710–A1740, 2017.

[2] Allison H. Baker, Robert D. Falgout, Tzanio V. Kolev, and Ulrike Meier Yang. Multigrid smoothers for ultraparallel computing. *SIAM J. Sci. Comput.*, 33(5):2864–2887, 2011.

[3] Allison H. Baker, Robert D. Falgout, Tzanio V. Kolev, and Ulrike Meier Yang. Multigrid smoothers for ultraparallel computing. *SIAM J. Sci. Comput.*, 33:2864–2887, 2011.

[4] Saulo RM Barros. The poisson equation on the unit disk: a multigrid solver using polar coordinates. *Applied Mathematics and Computation*, 25(2):123–135, 1988.

[5] S. Bauer, M. Huber, S. Ghelichkhan, M. Mohr, U. Rüde, and B. Wohlmuth. Large-scale simulation of mantle convection based on a new matrix-free approach. *J. Comput. Sci.*, 31:60–76, 2019.

[6] S. Bauer, M. Mohr, U. Rüde, J. Weismüller, M. Wittmann, and B. Wohlmuth. A two-scale Approach for Efficient on-the-fly Operator Assembly in Massively Parallel High Performance Multigrid Codes. *Applied Numerical Mathematics*, 122:14–38, 2017.

[7] Simon Bauer, Hans-Peter Bunge, Daniel Drzisga, Siavash Ghelichkhan, Markus Huber, Nils Kohl, Marcus Mohr, Ulrich Rüde, Dominik Thönnes, and Barbara Wohlmuth. Terraneo—mantle convection beyond a trillion degrees of freedom. *Software for Exascale Computing SPPEXA*, page 569, 2020.

[8] A. Bechmann, N. Sorensen, J. Berg, J. Mann, and P.E.. Rethore. The bolund experiment, part ii: flow over a steep, three-dimensional hill. *Bound Layer Meteorol.*, 141(2):245—271, 2011.

[9] B. Bergen and F. Hülsemann. Hierarchical hybrid grids: Data structures and core algorithms for multigrid. *Numer. Linear Algebra Appl.*, 11:279–291, 2004.

[10] Massimo Bernaschi, Pasqua D'Ambra, and Dario Pasquini. AMG based on compatible weighted matching for GPUs. *Parallel Comput.*, 92:102599, 13, 2020.

[11] Massimo Bernaschi, Pasqua D'Ambra, and Dario Pasquini. BootCMatchG: An adaptive Algebraic MultiGrid linear solver for GPUs. *Software Impacts*, 6, 2020.

[12] D. Bertaccini, P. D'Ambra, F. Durastante, and S. Filippone. Hybrid preconditioning richards equation for variably saturated flow. SIAM Conference on Applied Linear Algebra (LA21), 2021.

[13] D. Bertaccini, P. D'Ambra, F. Durastante, and S. Filippone. Solving richards equations for extreme scale applications in hydrology. SIAM Conference on Mathematical & Computational Issues in the Geosciences (GS21), 2021.

[14] D. Bertaccini, P. D'Ambra, F. Durastante, and S. Filippone. Hybrid Preconditioning Richards Equation for Variably Saturated Flow. **In Preparation**. 2021.

[15] D. Bertaccini and F. Durastante. *Iterative methods and preconditioning for large and sparse linear systems with applications*. Monographs and Research Notes in Mathematics. CRC Press, Boca Raton, FL, 2018.

[16] Daniele Bertaccini and Salvatore Filippone. Sparse approximate inverse preconditioners on high performance GPU platforms. *Comput. Math. Appl.*, 71(3):693–711, 2016.

[17] S. Boukhris, A. Napov, and Y. Notay. Algebraic multigrid using a stencil-CSR hybrid format on GPUs. Technical Report GANMN 21–04, Université Libre de Bruxelles, Brussels, Belgium, 2021. `http://homepages.ulb.ac.be/~ynotay/`.

[18] Nicolas Bouzat, Camilla Bressan, Virginie Grandgirard, Guillaume Latu, and Michel Mehrenberger. Targeting realistic geometry in tokamak code gysela. *ESAIM: Proceedings and Surveys*, 63:179–207, 2018.

[19] Carsten Burstedde, Jose A Fonseca, and Stefan Kollet. Enhancing speed and scalability of the parflow simulation code. *Computational Geosciences*, 22(1):347–361, 2018.

[20] A. Buttari, M. Huber, P. Leleux, T. Mary, U. Ruede, and B. Wohlmuth. Block low rank single precision coarse grid solvers for extreme scale multigrid methods. working paper or preprint, April 2020.

[21] Ü. V. Catalyürek, F. Dobrian, A. Gebremedhin, M. Halappanavar, and A. Pothen. Distributed-memory parallel algorithms for matching and coloring. In *PCO'11 New Trends in Parallel Computing and Optimization, IEEE International Symposium on Parallel and Distributed Processing Workshops*. IEEE CS, 2011.

[22] P. D'Ambra, F. Durastante, and S. Filippone. AMG preconditioners for linear solvers towards extreme scale. *arXive 2006.16147*, 2021.

[23] Pasqua D'Ambra, Daniela di Serafino, and Salvatore Filippone. MLD2P4: a package of parallel algebraic multilevel domain decomposition preconditioners in Fortran 95. *ACM Trans. Math. Software*, 37(3):Art. 30, 23, 2010.

[24] Pasqua D'Ambra, Fabio Durastante, and Salvatore Filippone. On the quality of matching-based aggregates for algebraic coarsening of SPD matrices in AMG, 2020.

[25] Pasqua D'Ambra, Salvatore Filippone, and Panayot S. Vassilevski. BootCMatch: a software package for bootstrap AMG based on graph weighted matching. *ACM Trans. Math. Software*, 44(4):Art. 39, 25, 2018.

[26] Pasqua D'Ambra and Panayot S. Vassilevski. Adaptive AMG with coarsening based on compatible weighted matching. *Comput. Vis. Sci.*, 16(2):59–76, 2013.

[27] H. De Sterck, U. Yang, and J. J. Heys. Reducing complexity in parallel algebraic multigrid preconditioners. *SIAM J. Matrix Anal. Appl.*, 27(4):1019–1039, 2006.

[28] P. D'Ambra and F. Durastante. Preliminary results and performance evaluation of linear algebra solvers, 2020. Deliverable 3.2 of EoCoE II.

[29] A. El Haman Abdeselam, A. Napov, and Y. Notay. Porting an aggregation-based algebraic multigrid method to GPUs. Technical Report GANMN 21–03, Université Libre de Bruxelles, Brussels, Belgium, 2021. `http://homepages.ulb.ac.be/~ynotay/`.

[30] Fabulous team. Fabulous software and documentation. `https://gitlab.inria.fr/solverstack/fabulous`.

[31] Robert D. Falgout, Jim E. Jones, and Ulrike Meier Yang. The design and implementation of hypre, a library of parallel high performance preconditioners. In *Numerical solution of partial differential equations on parallel computers*, volume 51 of *Lect. Notes Comput. Sci. Eng.*, pages 267–294. Springer, Berlin, 2006.

[32] Matthew W. Farthing and Fred L. Ogden. Numerical solution of richards' equation: A review of advances and challenges. *Soil Sci. Soc. Am. J.*, 81(6):1257–1269, 2017.

[33] S. Filippone and A. Buttari. Object-oriented techniques for sparse matrix computations in Fortran 2003. *ACM TOMS*, 38(4):23:1–23:20, 2012.

[34] Salvatore Filippone, Valeria Cardellini, Davide Barbieri, and Alessandro Fanfarillo. Sparse matrix-vector multiplication on GPGPUs. *ACM Trans. Math. Software*, 43(4):Art. 30, 49, 2017.

[35] Björn Gmeiner, Markus Huber, Lorenz John, Ulrich Rüde, and Barbara Wohlmuth. A quantitative performance study for Stokes solvers at the extreme scale. *J. Comput. Sci.*, 17:509 – 521, 2016.

[36] V Grandgirard, Y Sarazin, X Garbet, G Dif-Pradalier, Ph Ghendrih, N Crouseilles, G Latu, E Sonnendrücker, N Besse, and P Bertrand. Gysela, a full-f global gyrokinetic semi-lagrangian code for itg turbulence simulations. In *Aip conference proceedings*, volume 871, pages 100–111. American Institute of Physics, 2006.

[37] Van E. Henson and U. Yang. BoomerAMG: a parallel algebraic multigrid solver and preconditioner. *Appl. Numer. Math.*, 41(1):155–177, 2002. Developments and trends in iterative methods for large systems of equations—in memoriam Rüdiger Weiss (Lausanne, 2000).

[38] Pierre Jolivet, Frédéric Hecht, Frédéric Nataf, and Christophe Prud'homme. Scalable domain decomposition preconditioners for heterogeneous elliptic problems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis on - SC '13*. ACM Press, 2013.

[39] Pierre Jolivet, Jose E. Roman, and Stefano Zampini. Ksphpddm and pchpddm: Extending petsc with advanced krylov methods and robust multilevel overlapping schwarz preconditioners. *Computers & Mathematics with Applications*, 84:277–295, 2021.

[40] Jim E Jones and Carol S Woodward. Newton–krylov-multigrid solvers for large-scale, highly heterogeneous, variably saturated flow problems. *Advances in Water Resources*, 24(7):763–774, 2001.

[41] Michael Jung and Ulrich Rüde. Implicit extrapolation methods for variable coefficient problems. *SIAM Journal on Scientific Computing*, 19(4):1109–1124, 1998.

[42] Nils Kohl, Dominik Thönnes, Daniel Drzisga, Dominik Bartuschat, and Ulrich Rüde. The HyTeG finite-element software framework for scalable multigrid solvers. *International Journal of Parallel, Emergent and Distributed Systems*, 34(5):477–496, 2019.

[43] Martin Joachim Kühn, Carola Kruse, and Ulrich Rüde. Energy-minimizing, symmetric finite differences for anisotropic meshes and energy functional extrapolation. 2020.

[44] Martin Joachim Kühn, Carola Kruse, and Ulrich Rüde. Implicitly extrapolated geometric multigrid on disk-like domains for the gyrokinetic poisson equation from fusion plasma applications. 2020.

[45] Kühn, Martin Joachim and Leleux, Philippe and Kruse, Carola and Rüde, Ulrich. Gmgpolar/iexmg: complexity analysis. to be published, 2021.

[46] V. Lehmkuhl, G. Houzeaux, H. Owen, G. Chrysokentis, and I. Rodriguez. A low-dissipation finite element scheme for scale resolving simulations of turbulent flows. *Journal of Computational Physics*, 390:51 – 65, 2019.

[47] Jean-Yves L'Excellent and Wissam M. Sid-Lakhdar. A study of shared-memory parallelism in a multifrontal solver. *Parallel Computing*, 40(3-4):34–46, March 2014.

[48] Rainald Löhner, Fernando Mut, Juan Raul Cebral, Romain Aubry, and Guillaume Houzeaux. Deflated preconditioned conjugate gradient solvers for the pressure-poisson equation: Extensions and improvements. *International Journal for Numerical Methods in Engineering*, 87(1-5):2–14, June 2010.

[49] Frédéric Magoulès, François-Xavier Roux, and Guillaume Houzeaux. *Parallel Scientific Computing*. John Wiley & Sons, Inc., December 2015.

[50] MaPHyS team. MaPHyS software and documentation. `https://gitlab.inria.fr/solverstack/maphys`.

[51] MUMPS team. MUMPS: Multifrontal massively parallel solver. `http://mumps-solver.org/`.

[52] A. Napov and Y. Notay. Algebraic multigrid for moderate order finite elements. *SIAM J. Sci. Comput.*, 36:A1678–A1707, 2014.

[53] Y. Notay. AGMG software and documentation. `http://agmg.eu`.

[54] Y. Notay. An aggregation-based algebraic multigrid method. *Electron. Trans. Numer. Anal.*, 37:123–146, 2010.

[55] Y. Notay, P. D'Ambra, M.J. Kühn, and P. Tamain. Co-design of la solvers, specification of characteristics and interfaces of the la solvers for all target applications, 2019. Deliverable 3.1 of EoCoE II.

[56] Y. Notay and A. Napov. A massively parallel solver for discrete poisson-like problems. *J. Comput. Physics*, 281:237–250, 2015.

[57] Yvan Notay and Panayot S. Vassilevski. Recursive Krylov-based multigrid cycles. *Numer. Linear Algebra Appl.*, 15(5):473–487, 2008.

[58] H Oueslati, T Bonnet, N Minesi, M-C Firpo, and A Salhi. Numerical derivation of steady flows in visco-resistive magnetohydrodynamics for jet and iter-like geometries with no symmetry breaking. In *AIP Conference Proceedings*, volume 2179, page 020009. AIP Publishing LLC, 2019.

[59] PaStiX team. PaStiX software and documentation. `https://gitlab.inria.fr/solverstack/pastix`.

[60] David Skinner and William Kramer. Understanding the causes of performance variability in hpc workloads. In *IEEE International. 2005 Proceedings of the IEEE Workload Characterization Symposium, 2005.*, pages 137–149. IEEE, 2005.

[61] N. Spillane, V. Dolean, P. Hauret, F. Nataf, C. Pechstein, and R. Scheichl. Abstract robust coarse spaces for systems of PDEs via generalized eigenproblems in the overlaps. *Numerische Mathematik*, 126(4):741–770, August 2013.

[62] Patrick Tamain, Hugo Bufferand, Guido Ciraolo, Clothilde Colin, Davide Galassi, Ph Ghendrih, Frédéric Schwander, and Eric Serre. The tokam3x code for edge turbulence fluid simulations of tokamak plasmas in versatile magnetic geometries. *Journal of Computational Physics*, 321:606–623, 2016.

[63] R.S. Tuminaro and C. Tong. Parallel smoothed aggregation multigrid: aggregation strategies on massively parallel machines. In *SC '00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*. IEEE CS, 2000.

[64] Arno C. N. van Duin. Scalable parallel preconditioning with the sparse approximate inverse of triangular matrices. *SIAM J. Matrix Anal. Appl.*, 20(4):987–1006, 1999. Sparse and structured matrices and their applications (Coeur d'Alene, ID, 1996).

[65] M. Th. van Genuchten. A closed-form equation for predicting the hydraulic conductivity of unsaturated soils. *Soil Sci. Soc. Am. J.*, 44(5):892–898, 1980.

[66] P. Vaněk, J. Mandel, and M. Brezina. Algebraic multigrid by smoothed aggregation for second and fourth order elliptic problems. *Computing*, 56(3):179–196, 1996. International GAMM-Workshop on Multi-level Methods (Meisdorf, 1994).

[67] P.S. Vassilevski. *Multilevel block factorization preconditioners: matrix-based analysis and algorithms for solving finite element equations*. Springer, New York, USA, 2008.

[68] A. W. Vreman. An eddy-viscosity subgrid-scale model for turbulent shear flow: Algebraic theory and applications. *Physics of Fluids*, 16(10):3670—36681, 2004.

[69] U. Yang. Parallel algebraic multigrid methods—high performance preconditioners. In *Numerical solution of partial differential equations on parallel computers*, volume 51 of *Lect. Notes Comput. Sci. Eng.*, pages 209–236. Springer, Berlin, 2006.

[70] U. Yang. On long-range interpolation operators for aggressive coarsening. *Numer. Linear Algebra Appl.*, 17(2-3):453–472, 2010.

[71] Weiqun Zhang, Ann Almgren, Vince Beckner, John Bell, Johannes Blaschke, Cy Chan, Marcus Day, Brian Friesen, Kevin Gott, Daniel Graves, et al. Amrex: a framework for block-structured adaptive mesh refinement. *Journal of Open Source Software*, 4(37):1370–1370, 2019.

[72] Edoardo Zoni and Yaman Güçlü. Solving hyperbolic-elliptic problems on singular mapped disk-like domains with the method of characteristics and spline finite elements. *Journal of Computational Physics*, 398:108889, 2019.