The Journal of Supercomputing (2024) 80:13533–13556
https://doi.org/10.1007/s11227-024-05989-y

Alya toward exascale: algorithmic scalability using PSCToolkit

Herbert Owen[1] · Oriol Lehmkuhl[1] · Pasqua D'Ambra[2] · Fabio Durastante[2,3] · Salvatore Filippone[2,4]

Accepted: 9 February 2024 / Published online: 9 March 2024

Abstract
In this paper, we describe an upgrade of the Alya code with up-to-date parallel linear solvers capable of achieving reliability, efficiency and scalability in the computation of the pressure field at each time step of the numerical procedure for solving a Large Eddy Simulation formulation of the incompressible Navier–Stokes equations. We developed a software module in the Alya's kernel to interface the libraries included in the current version of `PSCToolkit`, a framework for the iterative solution of sparse linear systems, on parallel distributed-memory computers, by Krylov methods coupled to Algebraic MultiGrid preconditioners. The Toolkit has undergone various extensions within the EoCoE-II project with the primary goal of facing the exascale challenge. Results on a realistic benchmark for airflow simulations in wind farm applications show that the `PSCToolkit` solvers significantly outperform the original versions of the Conjugate Gradient method available in the Alya's kernel in terms of scalability and parallel efficiency and represent a very promising software layer to move the Alya code toward exascale.

Keywords Navier–Stokes equations · Iterative linear solvers · Algebraic MultiGrid · Parallel scalability

Mathematics Subject Classification 65F08 · 65F10 · 65M55 · 65Y05 · 65Z05

1 Introduction

Alya is a high-performance computational mechanics code for complex coupled multi-physics engineering problems. In this work, we present the interfacing between Alya and the `PSCToolkit` to overcome one of Alya's main obstacles in the path toward exascale, namely the lack of state-of-the-art parallel algebraic linear solvers with adequate algorithmic scalability, as already identified in [29], where

Extended author information available on the last page of the article

Alya's strengths and weaknesses in facing the exascale challenge have been analyzed by scalability studies up to one hundred thousand cores.

Although Alya can be applied to a wide range of problems, in this work, we shall concentrate on solving turbulent incompressible flow problems using a Large Eddy Simulation (LES) approach. Due to the wide range of scales present in turbulent high-Reynolds-number flows, their accurate solution requires computational meshes with a huge number of degrees of freedom (dofs). Alya uses a Finite Element (FE) spatial discretization, while its time discretization is based on finite difference methods; when an implicit time discretization is applied, the two main kernels of a simulation are the assembly of stiffness matrices and the solution of the associated linear system at each time step. In [29], the authors observed that the FE assembly implemented in Alya showed nearly perfect scalability, as one could a priory expect, while the solution of linear systems by available iterative linear solvers was the main weakness in the path toward exascale. The problem is related to Alya's lack of solvers with optimal algorithmic scalability, i.e., solvers able to obtain a given accuracy employing an almost constant number of iterations for an increasing number of dofs.

Alya's sparse linear algebra solvers are specifically developed with tight integration with the overall parallelization scheme; they include Krylov-based solvers, such as Generalized Minimal Residual (GMRES) or Conjugate Gradient (CG), coupled to some deflation approach or a simple diagonal preconditioner. As shown in [29], when incompressible flow problems are considered, the solution of a Poisson-type equation for the pressure field becomes challenging as the size of the problem increases. Indeed, when a uniform mesh multiplication [19] is used to have successively finer mesh, each time the mesh is refined to obtain elements with half the size, the number of iterations for solving the pressure equation is approximately doubled, showing a mesh-size-dependent behavior. To overcome these scalability issues, we interfaced `PSCToolkit` to Alya to take advantage of the Algebraic MultiGrid (AMG) preconditioners available through the `AMG4PSBLAS` library; this effort has been carried out in the context of the European Center of Excellence for Energy (EoCoE) applications.

The rest of the paper is organized as follows. In Sect. 2, we describe the general framework of Alya and the type of fluid dynamics problem we wish to test the new solvers on; in Sect. 3, we give an overall description of `PSCToolkit`, and then we focus on the AMG preconditioners employed in Sect. 3.1. Section 4 discusses the new module written to interface the solver library to the Alya software and the related issues. Section 5 describes the actual test case, while Sect. 6 analyzes the numerical scalability results in detail. Finally, Sect. 7 summarizes the results obtained and illustrates the new lines of development.

2 Alya description

Alya is a high-performance computational mechanics code for complex coupled multi-physics engineering problems. It can solve problems in the simulation of turbulent incompressible/compressible flows, nonlinear solid mechanics, chemistry, particle transport, heat transfer, and electrical propagation. Alya has been designed for massively parallel supercomputers and exploits several parallel programming models/tools. It relies on MPI to support a distributed-memory model; some kernels support vectorization at the CPU level and GPU accelerators are exploited through OpenACC pragmas or CUDA.

Multi-physics coupling is achieved following a multi-code strategy that uses MPI to communicate different instances of Alya. Each instance solves a particular physics, enabling asynchronous execution. Coupled problems can be solved by retaining the scalability properties of the individual instances. Alya is one of the two Computational Fluid Dynamic (CFD) codes of the Unified European Applications Benchmark Suite (UEBAS) [25]. It is also part of the Partnership for Advanced Computing in Europe (PRACE) Accelerator benchmark suite [26].

As mentioned in Sect. 1, large-scale CFD applications are the main problems targeted by Alya; hence, the basic mathematical models include various formulations of the Navier–Stokes equations, whose strong form for incompressible flows in a suitable domain is the following:

$$\partial_t \mathbf{u} - 2\nu\nabla \cdot \varepsilon(\mathbf{u}) + \mathbf{u} \cdot \nabla\mathbf{u} + \nabla p = \mathbf{f}, \tag{1}$$

$$\nabla \cdot \mathbf{u} = 0, \tag{2}$$

where $\mathbf{u}$ and p are the velocity and pressure field respectively, $\varepsilon(\mathbf{u}) = \frac{1}{2}\left(\nabla\mathbf{u} + \nabla^T\mathbf{u}\right)$ is the velocity strain rate tensor, ν is the kinematic viscosity, and $\mathbf{f}$ denotes the vector of external body forces. The problem is supplied with an initial divergence-free velocity field and appropriate boundary conditions.

The flow is turbulent for most real-world flow problems, and some turbulence modeling is needed to make the problem solvable with currently available computational resources. For all the examples presented in this work, we rely on the functionalities of Alya, which apply the spatially filtered Navier–Stokes equations coupled to the Vreman subgrid-scale model [30] for turbulence closure. In practice, a spatially varying turbulent viscosity supplements the laminar viscosity and the velocity and pressure unknowns correspond to spatially filtered values. Finally, since the size of the dynamically important eddies at high Reynolds numbers becomes too small to be grid resolved close to the wall, we employ a wall modeling technique [24] to impose the boundary conditions for the LES equations. For simplicity, the nonlinear term has been written in its convective form, which is most commonly encountered in computational practice.

Space discretization is based on a Galerkin FE approximation, employing hybrid unstructured meshes, which can include tetrahedra, prisms, hexahedra, and pyramids. Temporal discretization is performed through an explicit third-order Runge–Kutta scheme, where the Courant–Friedrichs–Lewy number is set to CFL = 1.0 for the cases presented in this work. A non-incremental fractional step method is used to stabilize the pressure, allowing the use of finite element pairs that do not satisfy the inf-sup condition [12], such as the equal order interpolation for the velocity and pressure applied in this work. A detailed description of the above numerical method, together with examples for turbulent flows, showing its high accuracy and low dissipation, can be found in [22].

The fractional step method allows uncoupling the solution of velocity and pressure [12]. At each Runge–Kutta substep, an explicit approach computes an intermediate velocity, and then a linear system coming from a Poisson-type equation is solved for the pressure; finally, the incompressible velocity is recovered. In the path toward exascale, the solution of the linear system for the pressure is the most demanding step. To reduce the computational burden, for most problems, an approximate projection method for Runge–Kutta time-stepping schemes is applied, which allows solving for the pressure only at the final substep [10].

It is important to note that most flow problems solved with Alya use a fixed mesh. For such problems, the linear system matrix for the pressure equation remains constant during the whole simulation. Therefore, the matrix assembly and the setup of a matrix preconditioner are needed only once at the beginning of the numerical procedure. Given that the number of time steps for LES is usually of the order of 10^5, it is clear that the linear solver computational times and scalability are the most relevant issues to be tackled.

3 PSCToolkit: PSBLAS and AMG4PSBLAS

We have interfaced Alya to exploit the solvers and preconditioners developed in the PSCToolkit[1] software framework for parallel sparse computations, proven on current petascale supercomputers and targeting the next-generation exascale machines. PSCToolkit is composed of two main libraries, named PSBLAS (Parallel Sparse Basic Linear Algebra Subprograms) [17, 18], and AMG4PSBLAS (Algebraic MultiGrid Preconditioners for PSBLAS) [16].

Both libraries are written in modern Fortran; PSBLAS implements algorithms and functionalities of parallel iterative Krylov subspace linear solvers, while AMG4PSBLAS is the package containing sophisticated preconditioners. In particular, AMG4PSBLAS provides one-level Additive Schwarz (AS) and Algebraic Multi-Grid (AMG) preconditioners. In the following, we will describe in some detail the AMG preconditioners we use within the Alya test cases.

3.1 AMG preconditioners

Algebraic MultiGrid methods can be viewed as a particular instance of a general stationary iterative method:

$$\mathbf{x}^{(k)} = \mathbf{x}^{(k-1)} + B\big(\mathbf{b} - A\mathbf{x}^{(k-1)}\big), \quad k = 1, 2, \dots \text{ given } \mathbf{x}^{(0)} \in \mathbb{R}^n,$$

for the solution of a linear system

$$A\mathbf{x} = \mathbf{b}, \qquad A \in \mathbb{R}^{n \times n}, \; \mathbf{b} \in \mathbb{R}^n,$$

[1] See psctoolkit.github.io on how to obtain and run the code.

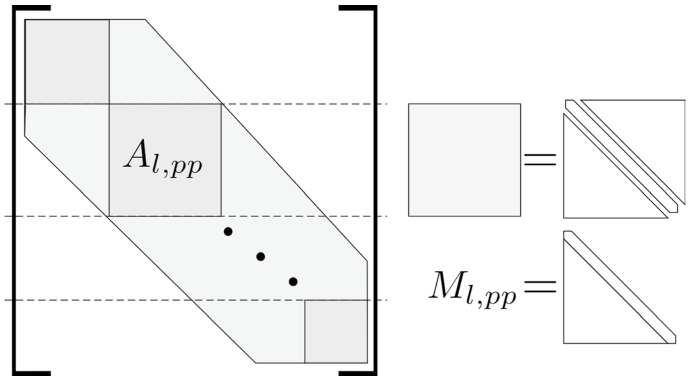


Fig. 1 Depiction of the structure of the hybrid forward/backward Gauss–Seidel method on a general row-block parallel distribution of symmetric positive definite matrix A

where A is symmetric and positive-definite (SPD), and the iteration matrix B is defined recursively; see, e.g., [28] for an exhaustive account. AMG methods are often employed as preconditioners for Krylov subspace solvers; what distinguishes the methods implemented in `AMG4PSBLAS` are the specific details of the construction procedure for the B matrix.

We define $A_0 = A$, and consider the sequence $\{A_l\}_{l=0}^{n_\ell -1}$ of coarse matrices computed by the triple-matrix Galerkin product:

$$A_{l+1} = P_l^T A_l P_l, \quad l = 0, \dots, n_\ell - 1,$$

where $\{P_l\}_{l=0}^{n_\ell -1}$ is a sequence of prolongation matrices of size $n_l \times n_{l+1}$, with $n_{l+1} < n_l$ and $n_0 = n$. To complete the formal construction we need also a sequence $\{M_l\}_{l=0}^{n_\ell -2}$ of A_l-convergent smoothers for the coarse matrices A_l, i.e., matrices M_l for which $\|I_l - M_l^{-1}A_l\|_{A_l} < 1$ holds true, where I_l is the identity matrix of size n_l and $\|\mathbf{v}\|_{A_l} = \sqrt{\mathbf{v}^T A_l \mathbf{v}}$ is the A_l norm. The preconditioner matrix B for the V-cycle with ν pre- and post-smooth iteration is then given by the multiplicative composition of the following error propagation matrices,

$$I_l - B_l A_l = (I_l - M_l^{-T} A_l)^\nu (I_l - P_l B_{l+1} P_l^T A_l)(I_l - M_l^{-1} A_l)^\nu \; \forall l < n_\ell, \tag{3}$$

with $B_{n_\ell} \approx A_{n_\ell}^{-1}$, either as a *direct solution* or as a convergent iterative procedure with a fine enough tolerance.

For the case at hand, we select each iteration matrix of the smoother sequence $\{M_l\}_{l=0}^{n_\ell -1}$ as the one representing four iterations ($\nu = 4$) of the hybrid forward/backward Gauss–Seidel method. We consider having A in a general row-block parallel distribution over n_p processes, i.e., A is divided into n_p blocks of size $n_b \times n$, and we call A_{pp} the corresponding diagonal block of A. We then decompose each block A_{pp} as $A_{pp} = L_{pp} + D_{pp} + L_{pp}^T$, where $D_{pp} = \text{diag}(A_{pp})$, L_{pp} is strictly lower triangular. To enforce symmetry in (3), we select $M_{l,pp}$ as the block diagonal matrices (Fig. 1)

$$M_l = \text{blockdiag}(M_{l,pp})_{pp=1}^{n_p/n_b}, \qquad M_{l,pp} = \omega\big(L_{l,pp} + D_{l,pp}\big), \quad l = 0, \dots, n_\ell,$$

where ω is a damping parameter. The overall procedure thus amounts essentially to using four sweeps of the damped block-Jacobi method on the matrix of the current level while solving the blocks with the forward, respectively backward, Gauss–Seidel method.

To build the prolongation (and thus the restriction) matrices, we employ the *coarsening based on compatible weighted matching* strategy; a full account of the derivation and detailed theoretical analysis may be found in [13, 15, 16]. This is a recursive procedure that starts from the adjacency graph $G = (V, E)$ associated with the sparse matrix A; this is the graph in which the vertex set V consists of either the row or column indices of A and the edge set E corresponds to the indices pairs (i, j) of the nonzero entries in A. The method works by constructing a *matching* $\mathcal{M}$ in the graph G to obtain a partition into subgraphs. We recall that a graph matching is a subset of the graph's edges such that no two of them are incident on the same vertex. Specifically, we consider more than a purely topological matching by taking into account the *weights* of the edges, i.e., the values of the entries of the matrix A and of a suitable vector. In the first step, we associate an edge weight matrix C, computed from the entries $a_{i,j}$ in A and an arbitrary vector $\mathbf{w}$; then, we compute an approximate maximum product matching of the whole graph to obtain the aggregates defining the coarse spaces. We define $C = (c_{i,j})_{i,j}$ as

$$c_{i,j} = 1 - \frac{2a_{i,j}w_i w_j}{a_{i,i}w_i^2 + a_{j,j}w_j^2}; \tag{4}$$

then, $\mathcal{M}$ is an *approximate maximum product matching* of G with edge weight matrix C, i.e.,

$$\mathcal{M} \approx \arg\max_{\mathcal{M}'} \prod_{(i,j)\in\mathcal{M}'} c_{i,j}. \tag{5}$$

The aggregates are then the subsets of indices $\{\mathcal{G}_p\}_{p=1}^{|\mathcal{M}|}$ of the whole index set $\mathcal{I}$ of A made of pairs of indices matched by the algorithm, where we denote with $|\mathcal{M}|$ the cardinality of the graph matching $\mathcal{M}$. In other terms, we have obtained the decomposition

$$\mathcal{I} = \{1, \dots, n\} = \bigcup_{p=1}^{n_\mathcal{M}} \mathcal{G}_p, \quad \mathcal{G}_p \cap \mathcal{G}_r = \emptyset \text{ if } p \neq r;$$

see, e.g., Fig. 2 in which the matching of a test graph is computed–in more detail, Fig. 2a has a black dot corresponding to a non-zero element of the adjacency matrix; Fig. 2b shows the corresponding graph obtained from it; while Fig. 2c highlights the aggregated nodes, i.e., the $\mathcal{G}_p$ sets. In most cases, we will end up with a sub-optimal matching, i.e., not all vertices will be endpoints of matched edges; thus, we usually have *unmatched* vertices. To each unmatched vertex, we associate an aggregate G_s

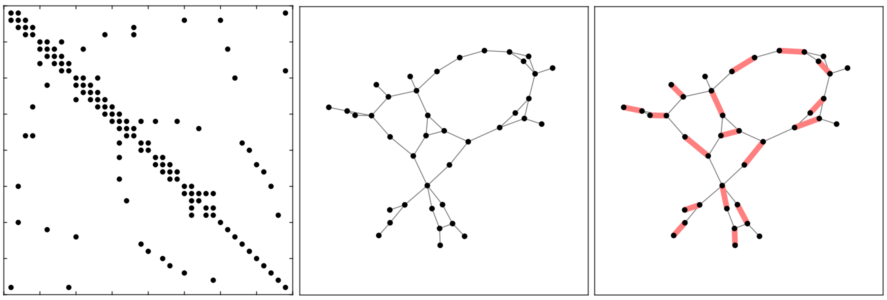

(a) Adjacency matrix (b) Original graph (c) Matched edges

Fig. 2 Matching of the graph `bcspwr01` from the Harwell-Boeing collection. The matched nodes in the graph are highlighted by a bold red edge

that is a singleton, and we denote with $n_{\mathcal{S}}$ the total number of singletons. The main computational cost of this phase is represented by the computation of the *approximate graph matching* on a graph that is distributed across thousands of processors. The parallel coarsening implemented in `AMG4PSBLAS` uses the `MatchBox-P` software library [11]; this implements a distributed parallel algorithm for the computation of *half-approximate maximum weight matching* with complexity $\mathcal{O}(|E|\Delta)$, where $|E|$ is the cardinality of the graph edge set and Δ is the maximum vertex degree, i.e., the maximum number of edges incident on any given node of the graph. The procedure guarantees a solution that is at least half of the optimal weight, i.e., the approximation in (5) holds within 1/2 of the optimum. The message aggregation and overlapping between communication and computation employed by this strategy reduces the impact of the data communication on parallel efficiency; we refer the reader to [16] for a complete set of experiments showcasing this feature. Finally, to build the prolongator matrices, the last ingredients we need are the vectors $\mathbf{w}_e$ identifying for each edge $e_{i\mapsto j} \in \mathcal{M}$ the orthonormal projection of $\mathbf{w}$ on the non-singleton aggregate G_p. For the sake of the explanation, we consider an ordering of the indices in which we move all the unknowns corresponding to unmatched vertices at the bottom,[2] and thus define a *tentative prolongator*

$$\hat{P} = \begin{pmatrix} \tilde{P} & 0 \\ 0 & W \end{pmatrix} \in \mathbb{R}^{n \times n_c}, \tag{6}$$

where:

$$\tilde{P} = \mathrm{blockdiag}(w_{e_1}, \dots, w_{e_{n_{\mathcal{M}}}}),$$

$W = \mathrm{diag}(w_s/|w_s|)$, $s = 1, \dots, n_{\mathcal{S}}$, corresponds to unmatched vertices. The resulting number of coarse variables is then given by $n_c = n_{\mathcal{M}} + n_{\mathcal{S}}$. The matrix $\hat{P}$ we have just built is a piecewise constant interpolation operator whose range includes, by construction, the vector $\mathbf{w}$. The actual prolongator P is then obtained from $\hat{P}$ as

[2] This ordering is for explanatory purposes only, and is not actually enforced in practice.

$P = (I - \omega D^{-1}A)\hat{P}$, where $D = \text{diag}(A)$ and $\omega = 1/\|D^{-1}A\|_\infty \approx 1/\rho(D^{-1}A)$, with $\rho(D^{-1}A)$ the spectral radius of $D^{-1}A$. Indeed, the P we have built is an instance of *smoothed aggregation*. Please observe that the procedure we have described produces, at best, a halving of the size of the system at each new level of the hierarchy. Given the size of the systems we are interested in, this may be unsatisfactory since the number of levels in the hierarchy and thus the operational cost needed to cross it would be too large. Fortunately, it is rather easy to overcome this issue: to obtain aggregates of size greater than two, we just have to collect them together by multiplying the corresponding prolongators (restrictors). This permits us to select the desired size of the aggregates (2, 4, 8, and so on) as an input parameter of the method.

To conclude the description of the preconditioners, we need to specify the choice for the coarsest solver. While using a direct solver at the coarsest level is the easiest way to ensure that the coarsest grid is resolved to the needed tolerance, such an approach for an AMG method running on many thousands of parallel cores can be very expensive. If the matching strategy has worked satisfactorily, the coarsest-level matrix will tend to have both a small global size and a small number of rows per core: in this case, the cost of data communication will dominate the local arithmetic computations causing a deterioration of the method efficiency. We use here a dual strategy: on the one hand, we employ a *distributed coarsest solver* running on all the parallel cores, whilst on the other, we limit the maximum size of the coarsest-level matrix to around 200 unknowns per core. Specifically, we use the Flexible Conjugate Gradient (FCG) method with a block-Jacobi preconditioner on which we solve approximately the blocks by an incomplete LU factorization with one level of fill-in, ILU(1), the stopping criterion is based on the reduction of the relative residual of 3 orders of magnitude or a maximum number of iterations equal to 30.

To have a comparison with the preconditioner just discussed, we also consider the same construction but with a different aggregation procedure: the decoupled version of the classic smoothed aggregation of Vaněk et al. [27]. This is an aggregation option that was already available in previous versions of the library [9, 14], and was already successfully used in CFD applications [2, 3]. The basic idea is to build a coarse set of indices by grouping unknowns into disjoint subsets (the aggregates) by using an affinity measure and defining a simple tentative prolongator whose range contains the so-called near null space of the matrix of the given level, i.e., a sample of the eigenvector corresponding to the smallest eigenvalue. The strategy is implemented in an embarrassingly parallel fashion, i.e., each processor produces aggregates by only looking at local unknowns, i.e., the aggregation is performed in a decoupled fashion, in contrast to the previous matching procedure that instead crosses the boundary of the single process.

Table 1 summarizes the different preconditioners we have discussed here and that are used in the experiments of Sect. 6.1.

Remark 1 The `AMG4PSBLAS` library provides interfaces to some widely used parallel direct solvers, such as `SuperLU` [23] and `MUMPS` [1]. Thus, we could have used

Table 1 Summary of the described preconditioners, the labels are used to describe the results in Sect. 6.1

Pre-smoother	4 iterations of hybrid forward Gauss–Seidel		
Post-smoother	4 iterations of hybrid backward Gauss–Seidel		
Coarsest solver	FCG preconditioned by block-Jacobi with ILU(1) block solvers		
Cycle	V-cycle		
Aggregation	Coupled smoothed based on matching		Decoupled classic
	$\lvert\mathcal{G}\rvert \leq 8$	$\lvert\mathcal{G}\rvert \leq 16$	smoothed
Label	*MLVSMATCH3*	*MLVSMATCH4*	*MLVSBM*

any of those within the damped block-Jacobi method, either on the smoother or on the coarsest solvers. For what concerns the smoothers, it has been observed in the literature [4, 16] that the combination with the Gauss–Seidel method delivers better smoothing properties for the overall method. In the coarsest solver case, the size of the local matrices is small enough to not usually show a significant performance increase when using a direct solver. We also stress that the preconditioner described in this section depends only on native `PSCToolkit` code, i.e., the user does not have to install optional third-party libraries to use it.

4 Interfacing Alya to `PSCToolkit`

The Alya code is organized in a modular way, and its architecture is split into *modules*, *kernel*, and *services*, which can be separately compiled and linked. Each module represents a physical model, i.e., a set of partial differential equations which can interact for running a multi-physics simulation in a time-splitting approach, while Alya's kernel implements the functionalities for dealing with the discretization mesh, the solvers and the I/O functionalities. As already mentioned, the governing equations of a physical model are discretized in space by using FE methods and all the functionalities to assemble the global stiffness matrix and right-hand-side (RHS) of the corresponding set of equations, including boundary conditions and material properties are the responsibility of the module. Instead, all the functionalities needed to solve the algebraic linear systems are implemented in the kernel. Some work on data structures and distribution of matrices and RHS was necessary to interface Alya with libraries from `PSCToolkit`, as described in the following.

Alya uses the compressed sparse row matrix scheme for the internal representation of sparse matrices. This scheme is supported by `PSCToolkit`, so no significant difficulty was met from this perspective. The main difficulty in the interfacing process was how the data, i.e. the discretization mesh and the corresponding unknowns, are distributed among the parallel processes and the way the related sparse matrix rows and RHS are locally assembled. The Alya code is based on a domain decomposition where the discretization mesh is partitioned into disjoint subsets of elements/nodes, referred to as subdomains. Then, each subdomain is assigned to a parallel process that carries out all the geometrical and algebraic operations

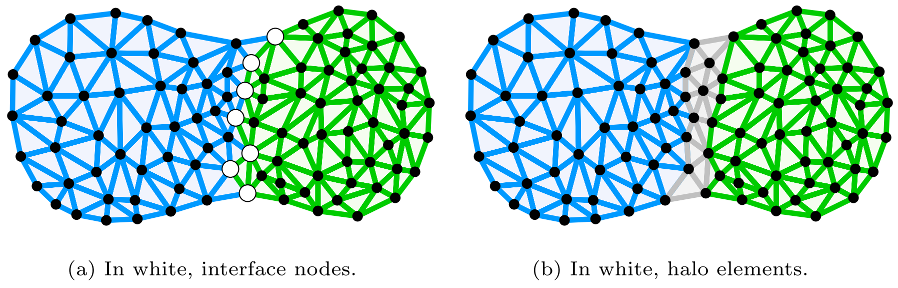


Fig. 3 Mesh partitioning into (3a) disjoint sets of nodes, and (3b) disjoint sets of elements

corresponding to that part of the domain and the associated unknowns. The interface elements/nodes on the boundary between two subdomains are assigned to one of the subdomains (see Fig. 3).

The sparse matrices expressing the linear couplings among the unknowns are distributed in such a way that each parallel process holds the entries associated with the couplings generated on its subdomain. Two different options are possible for sparse matrix distribution: the partial row format and the full row format [20], respectively. In the full row format, if a mesh element/node and the corresponding unknown belong to a process, all row entries related to that unknown are stored by that process. In the partial row format, the row of a matrix corresponding to an unknown is not full and needs contributions from unknowns belonging to different processes. Alya uses a partial row format for storing the matrix.

The libraries from PSCToolkit build the preconditioners and apply the Krylov methods on the assumption of a full row format; nevertheless, support for partial row format was added to the libraries' pre-processing stage so that the interfacing can be as transparent as possible. The pre-processing support implies the retrieval of remote information for those matrix contributions that correspond to elements on the boundary; the data communication is split between the discovery of the needed entries (which needs only be executed when the discretization mesh changes) and the actual retrieval of the matrix entries, which must happen at any time step where the matrix coefficients and/or vector entries may be rebuilt, prior to an invocation of the solvers. When the topology of the mesh does not change, and there is only an update in the coefficients, it is also possible to reuse the same preconditioner; this may be full reuse of the overall matrices hierarchy, or partial reuse, employing the same prolongators/restrictors to rebuild the AMG hierarchy and smoothers.

We developed a software module in Alya's kernel for declaration, allocation, and initialization of the library's data structures as well as for using solvers and preconditioners. PSBLAS makes available some of the widely used iterative methods based on Krylov projection methods through a single interface to a driver routine, while preconditioners for PSBLAS Krylov solvers are available through the AMG4PSBLAS package. The main functionalities for selecting and building the chosen preconditioner are the responsibility of the software module included in the Alya's kernel, while the functionalities for applying it within the PSBLAS Krylov solver are completely transparent to the Alya code and are the responsibility of the library.

Fig. 4 Photograph of the Bolund hill [7]

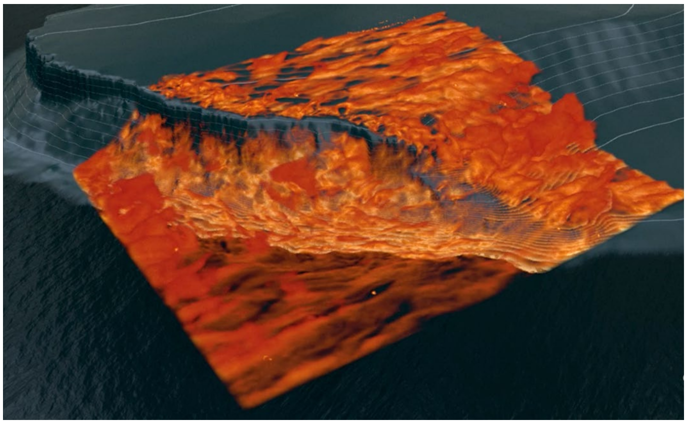


Fig. 5 Volume rendering of the velocity over Bolund obtained with Alya

5 The Bolund test case

Our main aim was to test the libraries for systems stemming from fluid dynamics simulation of incompressible flow arising in the study of wind-farm efficiency. The test case is based on the Bolund experiment (Figs. 4 and 5), a classical benchmark for microscale atmospheric flow models over complex terrain [5, 6]. An incompressible flow treatment is used because the Mach number, i.e., the ratio of the speed of the flow to the speed of sound, is much smaller than 0.3. The test case is based on a small (12 m) isolated steep hill at Roskilde Fjord in Denmark having a significantly steep escarpment in the main wind direction and uniformly covered by grass so that the resulting flow is not influenced by individual roughness elements. This is considered the ideal benchmark for the validation of neutral flow models and, hence a most relevant scenario for the analysis of software modeling for wind energy. Though relatively small, its geometrical shape induces complex 3D flow. Bolund was equipped with several measurement masts with conventional meteorological instruments and remote sensing Lidars to obtain detailed information of mean wind, wind shear, turbulence intensities, etc. A publicly available database for evaluating currently available flow models and methodologies for turbine siting in complex terrain regarding wind resources and loads is available at [7].

We discretize the incompressible Navier–Stokes Eq. (1) as described in Sect. 2. At each time step of the LES procedure, we solved the SPD linear systems arising from the pressure equation employing the preconditioned flexible version of the CG method (FCG) method by PSBLAS. Starting from an initial guess for pressure from the previous time step, we stopped linear iterations when the Euclidean norm of the relative residual was no larger than $TOL = 10^{-3}$. The Reynolds number based on the friction velocity for this test case is approximately $RE_\tau = Uh/\nu \approx 10^7$ with $U = 10\,\mathrm{m\,s^{-1}}$. As discussed in [6, Section 2.1], we can neglect Coriolis force in the horizontal direction and use the formulation (1) since the Rossby number $R_O = 667 \gg 1$.

The next Sect. 6 details the scalability result obtained for this test case with the new solvers and preconditioners from PSCToolkit described in Sect. 3.

6 Parallel performance results

In the following, we discuss the results of experiments run on two of the most powerful European supercomputers. The first set of experiments aimed to analyze the behavior of different AMG preconditioners available from AMG4PSBLAS and run on the Marenostrum-4 supercomputer up to 12288 CPU cores. Marenostrum-4 is composed of 3456 nodes with 2 Intel Xeon Platinum 8160 CPUs with 24 cores per CPU. It is ranked 121th in the November 2023 TOP500 list[3], with more than 10 petaflops of peak performance and is operated by the Barcelona Supercomputer Center. 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. The second set of experiments aimed to reach very large scales and run by using only one of the most promising preconditioners by AMG4PSBLAS on the Juwels supercomputer, up to 23551 CPU cores. Juwels is composed of 2271 compute nodes with 2 Intel Xeon Platinum 8168 CPUs, of 24 cores each. It is ranked 127th in the November 2023 TOP500 list, with more than 9 petaflops of peak performance, and is operated by the Jülich Supercomputer Center. The simulations have been performed with the Alya code interfaced to the same versions of the solvers libraries mentioned above, built with GNU compilers 10.3.

6.1 Comparison of AMG preconditioners

In this section, we discuss results obtained on Marenostrum-4 and compare the behavior of FCG coupled to the preconditioners described in Sect. 3.1 and

[3] Available at www.top500.org.

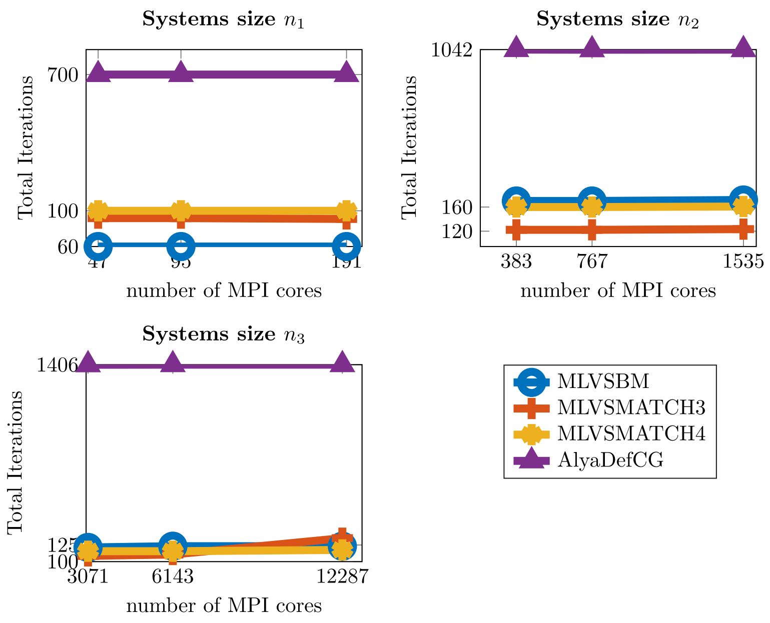


Fig. 6 Strong scalability: total iteration number of the linear solvers

summarized in Table 1. We run both strong scalability analysis for unstructured meshes of tetrahedra of three fixed sizes as well as weak scalability analysis, obtained by fixing different mesh sizes per core and linearly increasing both mesh size and the number of cores. A general row-block matrix distribution based on the Metis 4.0 mesh partitioner [21] was applied for the parallel runs.

6.1.1 Strong scalability

We first focus on strong scalability results obtained on the Bolund experiment for three fixed size meshes (small, medium and large) including $n_1 = 5570786 \approx 6 \times 10^6$, $n_2 = 43619693 \approx 4.4 \times 10^7$ and $n_3 = 345276325 \approx 0.35 \times 10^9$ dofs, respectively. Three different configurations of the number of cores, obtained by doubling each time the number of MPI cores with respect to the minimum number of cores (nodes) needed to run at full load, were employed for the three different mesh sizes: from $\min_p = 48$ to $\max_p = 192$ cores in the case of the small mesh, from $\min_p = 384$ to $\max_p = 1536$ cores for the medium mesh, and finally from $\min_p = 3072$ to $\max_p = 12288$ cores for the large mesh. We analyze the parallel efficiency and convergence behavior of the linear solvers for 20 time steps after a pre-processing phase so that we focus on the solvers' behavior 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.

In Figs. 6 and 7, 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

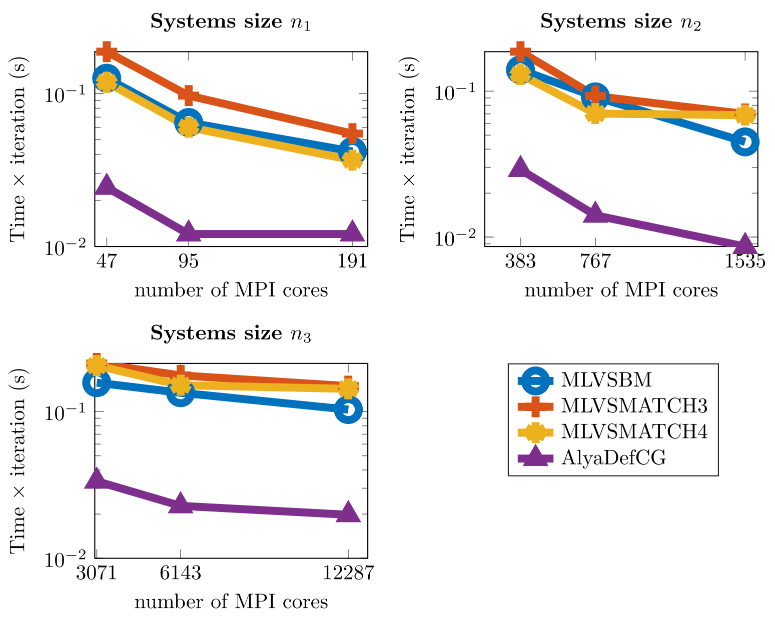


Fig. 7 Strong scalability: time per iteration of the linear solvers

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 much smaller than that with the original *AlyaDefCG*, for all three meshes, when `AMG4PSBLAS` multilevel preconditioners are applied. For the small mesh, the minimum number of linear iterations is obtained by *MLVSBM* which shows a fixed number of 60 iterations for all core counts, while *MLVSMATCH3* requires 90 iterations for all core counts except on 192 cores, where 1 less iteration was needed, and *MLVSMATCH4* requires 100 iterations; in this case, the original *AlyaDefCG* requires 700 iterations for all core counts.

In the case of the medium mesh, we observe a larger number of iterations of the solvers employing `AMG4PSBLAS` preconditioners with respect to the large mesh. We have a minimum number of iterations with *MLVSMATCH3* ranging from 122 to 123 for all number of cores, while *MLVSMATCH4* requires a range from 160 to 161 iterations and *MLVSBM* requires a range from 172 to 174 iterations. The original *AlyaDefCG* requires a number of iterations ranging from 1040 to 1042 for the medium mesh.

In the case of the large mesh, 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; a similarly stable behavior is observed for *MLVSBM* which requires a number of iterations ranging from 121 to 123. *AlyaDefCG* requires a number of iterations ranging from 1404 to 1406 for the large mesh.

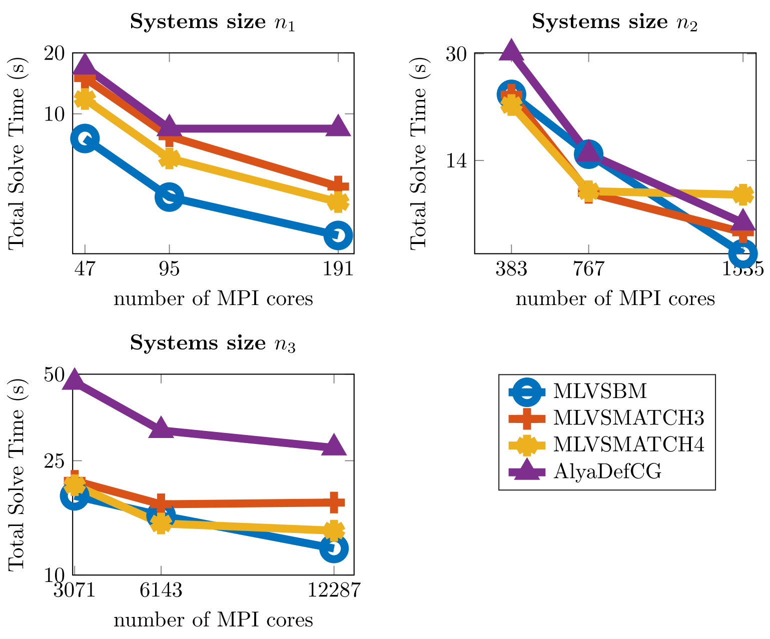


Fig. 8 Strong scalability: total solve time of the linear solvers

The oscillations in the number of iterations seem to be mostly dependent on the data partitioning obtained by Metis, which in turn, appears to have a larger impact on the *MLVSMATCH3* preconditioner in the case of the large mesh. A deeper analysis of the impact of the data partitioner on the solver behavior, albeit interesting, is out of the scope of our current work and would require a significant amount of computing resources.

In all cases, the time needed per iteration decreases for an 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 mesh size and number of cores, the AMG preconditioners show very similar behavior, although *MLVSBM* always requires a smaller time per iteration for the large mesh and for the medium mesh when 1536 cores are used.

In Figs. 8 and 9, 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 mesh size.

We observe that the AMG preconditioners from `AMG4PSBLAS` generally achieve shorter execution times than the original *AlyaDefCG*; indeed, the expected longer time per iteration is more than compensated by the large reduction in the number of iterations especially for the small and large mesh. In good

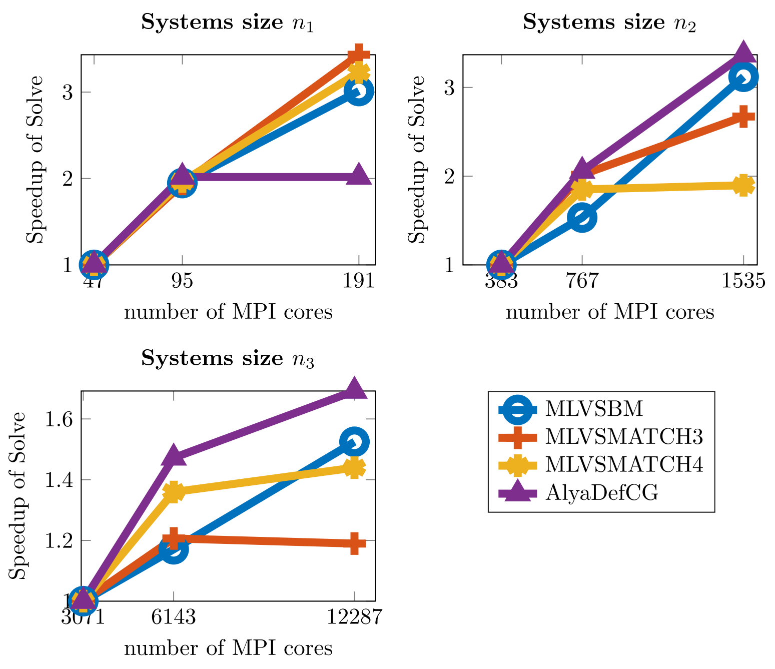


Fig. 9 Strong scalability: speedup of the linear solvers. We note that ideal values for speedups in all three configurations are 1, 2 and 4, respectively.

agreement with the behavior in terms of iterations and time per iteration, we observe that *MLVSBM* generally shows the shortest execution time for the small mesh, especially for small number of cores, while for the medium and large mesh, *MLVSMATCH3* and *MLVSMATCH4* show some better or comparable behavior with respect to *MLVSBM*. The best speedups are generally obtained, except for the small mesh, by the original *AlyaDefCG*, while in the case of AMG preconditioners, the very good convergence behavior and solve time on the smallest number of cores limit the speedup for the increasing number of cores. For the `AMG4PSBLAS` preconditioners, speedups are in good agreement with the total solve times, showing that *MLVSMATCH3* and *MLVSMATCH4* are generally better or comparable with respect to *MLVSBM* for all meshes when the small and medium number of cores are used, while *MLVSBM* is better for medium and large mesh when the largest number of cores is used.

In conclusion, the selected solvers from the `PSCToolkit` generally outperform the original Alya solver for the employed test case, and the choice of the better preconditioner from *AMG4PBLAS* depends on target mesh size and number of employed parallel cores. This appears as an advantage for Alya's users that having available a large set of parallel preconditioners through the interface to `PSCToolkit`, can select the best one for their specific aims.

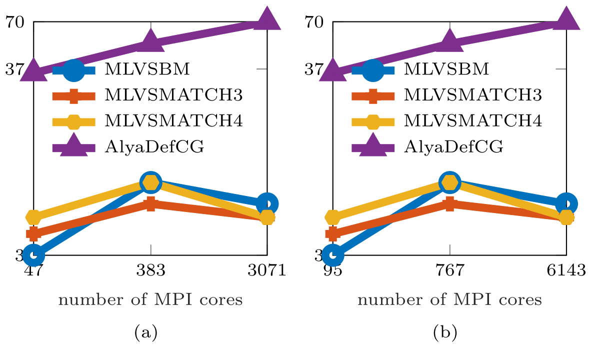


Fig. 10 Weak scalability: average number of linear iterations per time step. nxcore$_1$ dofs per core (**a**), nxcore$_2$ dofs per core (**b**), nxcore$_3$ dofs per core (**c**)

6.1.2 Weak scalability

In this section, we analyze the weak scalability of the AMG4PSBLAS preconditioners, i.e., we observe the solvers looking at their behavior when we fix the mesh size per core and increase the number of cores.

We considered the same test case and the three meshes 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. The different configurations of cores correspond to three different (decreasing) mesh sizes per core equal to nxcore$_1$ = 1.1e5, nxcore$_2$ = 5.9e4, and nxcore$_3$ = 2.9e4, respectively. Note that the medium and the large mesh correspond to scaling factors of 8 and 64, respectively, with respect to the small mesh; therefore in the same way, we scaled 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 execution times and scaled speedup for solve. In Fig. 10, we report the average number of iterations for each time step. We can observe a general increase, ranging from 35 to 70 for an increasing number of cores when the original *Alya-DefCG* is employed. On the other hand, when AMG preconditioners from AMG4PSBLAS coupled with FCG by PSBLAS are applied, we observe a constant average number of iterations equal to 5 for *MLVSMATCH4* both for the small and the large mesh, independently of the number of cores, while *MLVSBM* requires 3 iterations for the small mesh and 6 for the large mesh. *MLVSMACTH3* ranges from 4 to 6 iterations on the small mesh and the large mesh, respectively. In the case of medium mesh, in agreement with what was observed for the strong scalability analysis, all the preconditioners require a larger average number of iterations, which is 8 for *MLVSBM* and *MLVSMATCH4*, and 6 for *MLVSMATCH3*. This behavior indicates a very promising algorithmic scalability of *MLVSMATCH4*. In Figs. 11 and 12, we can see the total solve time and the corresponding scaled speedup. We can observe that, as expected from the previous sections, all preconditioners from AMG4PSBLAS generally lead to a smaller increase ratio in the solve times with respect to the

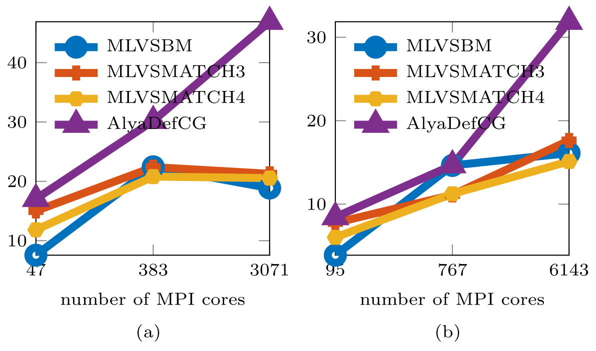


Fig. 11 Weak scalability: total solve time (s) of the linear solvers. nxcore_1 dofs per core (**a**), nxcore_2 dofs per core (**b**), nxcore_3 dofs per core (**c**)

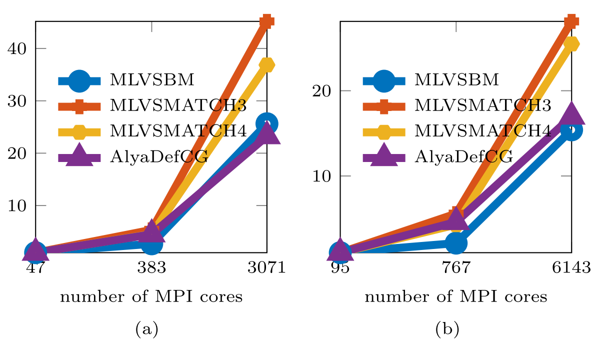


Fig. 12 Weak scalability: the scaled speedup of the linear solvers. nxcore_1 dofs per core (**a**), nxcore_2 dofs per core (**b**), nxcore_3 dofs per core (**c**)

original *AlyaDefCG*, when the mesh size goes from the small to the large one. In more detail, we observe that, for all mesh sizes per core, smaller increase ratios in the execution time are generally obtained with *MLVSMATCH3* and *MLVSMATCH4*. This is better observed by looking at the scaled speedup. It is defined as scalfactor $\times\, T_{\mathrm{min}_p}/T_p$, 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 mesh 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 mesh size per core. We observe that the best values are obtained with the *MLVSMATCH3* and *MLVSMATCH4* preconditioners when nxcore_1 and nxcore_2 dofs per core are used. In detail, for nxcore_1 dofs per core, *MLVSMATCH3* reaches the best value of about 71% of scaled efficiency on 3072 cores and about 44% of scaled efficiency on 6144 core when nxcore_2 dofs per core are employed. This shows that the scalability of *MLVSMATCH3* and *MLVSMATCH4* are 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 nxcore_3

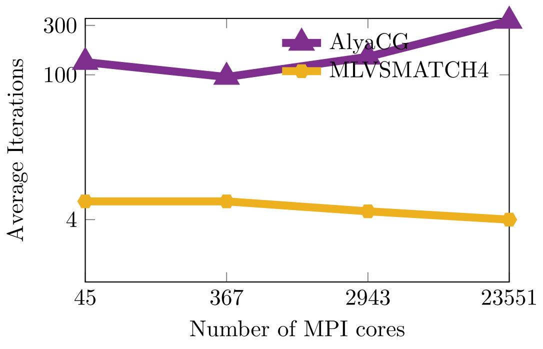


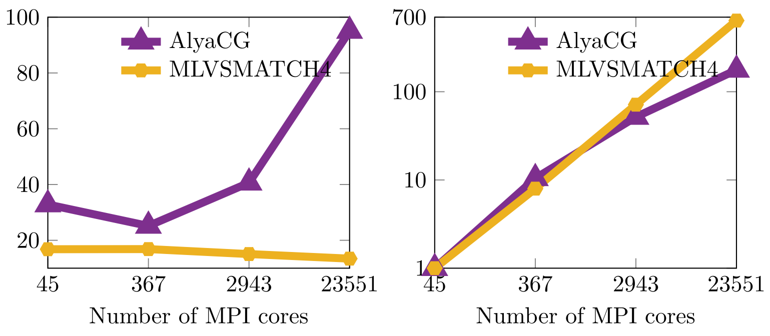
Fig. 13 Weak scalability: average number of linear iterations per time step. Systems size from n_1 to n_4

dofs per core (12c), the scaled speedup of *AlyaDefCG* is better; this is essentially due to the very large solve time spent by this solver on 192 cores.

6.2 Results at extreme scales

In this section, we discuss some results obtained on the Juwels supercomputer by increasing the number of dofs till to $n_4 \approx 2.9 \times 10^9$. We limit our analysis to the weak scalability results of one of the most promising solvers in `PSCToolkit`. Indeed, due to the limited access to the Juwels resources and taking into account the above preconditioners comparison, we only run experiments by using the *MLVSMATCH4* preconditioner. A general row-block data distribution based on a parallel geometric partitioning using Space Filling Curve (SFC) [8] was applied for these experiments. As in the previous experiments, we analyze the parallel efficiency and convergence behavior of the linear solver for 20 time steps after a pre-processing phase so that we focus on the solver behavior in the simulation of a fully developed flow for all the meshes but the largest one, where we were not able to skip the transient phase due to long simulation time. In this last case, we considered a total number of time steps equal to 1379 and analyzed solver performance in the last 20 time steps. Note that increasing mesh size imposes a decrease in time step due to stability constraints of the explicit time discretization that is preferred for LES simulations. Therefore, the total simulated time depends on the mesh size. Furthermore, to reduce observed operating oscillations associated with the full node runs, we used only a total of 46 cores per node.

As already mentioned, we analyze the weak scalability of the solvers; we considered a mesh size per core equal to nxcore_1 and used a scaling factor of 8 for going up to the largest mesh size; therefore in the same way, we scaled the number of cores for our weak scalability analysis. We can limit our analysis to observing the average number of linear iterations of the solver per each time step and analyzing execution times and scaled speedup for the solve phase. We compare the results obtained by using the `PSCToolkit`'s solver against Alya's Conjugate Gradient solver (hereby *AlyaCG*). Observe that in these experiments, we also tried to use the Deflated CG implemented in Alya, but it does not work for the two larger test cases, and *AlyaCG* appears better in the case of smaller size meshes. In Fig. 13, we report the average



(a) Total solve time of the linear solvers (s). (b) Scaled speedup of the linear solvers.

Fig. 14 Weak scalability: systems size from n_1 to n_4

number of iterations per each time step. We can observe a general increase, ranging from 133 to 331 for an increasing number of cores, but on 368 cores where 95 average iteration count is obtained, when the original *AlyaCG* is employed, while very good algorithmic scalability, with an average number of linear iterations per each time step ranging from 4 to 6, when the `PSCToolkit`'s solver is applied. In Fig. 14a,b, we can see the total solve time and the corresponding scaled speedup. We can observe that the good algorithmic scalability of *MLVSMATCH4* leads to an almost flat execution time for solving when the first three meshes are employed, while a decrease is observed for the simulation carried out with the largest mesh, depending on a smaller average number of iterations per time step. On the contrary, the original *AlyaCG* generally shows a huge increase for increasing number of cores and mesh size, but in the second one, where a decrease in the average number of iterations per time step is observed. Then we look at the scaled speedup, defined as scalfactor $\times\, T_{45}/T_p$, where scalfactor $= 1, 8, 64, 512$, for increasing number of cores, T_{45} is the total time for solving linear systems when 45 cores are involved in the simulation, and T_p is the total time spent in linear solvers for all the increasing number of cores. We observe that for the two larger meshes, *MLVSMATCH4* has a super-linear scaled speed-up of about 71 (up from the ideal speedup of 64) and 640 (up from the ideal speedup of 512), respectively, showing that its very good algorithmic scalability is coupled with excellent implementation scalability of all the basic computational kernels. This scalability is very promising in facing the exascale challenge.

7 Conclusions

In this paper, we presented our work on improving the linear solver capabilities of a large-scale CFD code by interfacing it with a software framework, including new and state-of-the-art algebraic linear solvers, specifically designed to exploit the very large potential of current petascale supercomputers and aimed at the early exascale supercomputers. Our activities were carried out in the context of the European Center of Excellence for Energy applications, where one of the lighthouse codes was the Alya code, developed at the Barcelona Supercomputing Center (BSC) and applied to wind flow studies for renewable production. However, this work has a wider impact, and confirms the benefits of using third-party software libraries developed by specialists, in complex, multi-component and multi-physics simulation codes.

From Alya's perspective, the most significant achievement has been obtaining excellent algorithmic scalability thanks to multigrid preconditioners, as shown in the weak scalability studies. This allows us to solve much bigger problems efficiently. Our first objective for the future is to test the GPU version of `PSCToolkit`. During EoCoE, we have significantly optimized the FE assembly on GPUs, making it four times more energy efficient than the CPU version; integrating a competitive linear algebra GPU package is now the next priority. After that, having the entire workflow for incompressible flow problems on GPUs should be relatively straightforward. We expect to have a much higher number of unknowns for problems running for MPI process when GPU accelerators are exploited. Therefore, strong scalability should be much less critical. While we have focused on a wind energy problem in this work, we wish to test the solver in other incompressible flow problems in the future. Moreover, since the solver is fully interfaced with Alya, it will be interesting to test the suitability of `PSCToolkit` for other problems, such as solid mechanics or heat transfer.

Acknowledgements We thank the two anonymous reviewers whose suggestions helped improve and clarify this manuscript.

Author Contributions The first two authors are involved in the development of the Alya code and worked on the integration of the PSCToolkit into Alya. Herbert Owen did main contribution in setup and running of experiments. The last three authors are the developers of PSCToolkit and experts in Linear Algebra. They have equally contributed in setup of suitable linear solvers, analyzing results and writing the paper.

Funding Open access funding provided by Consiglio Nazionale Delle Ricerche (CNR) within the CRUI-CARE Agreement. The research received funding and PRACE grants for supercomputers access from Horizon 2020 Project "Energy oriented Centre of Excellence: toward exascale for energy" (EoCoE–II), Project ID: 824158. This work has been partially supported by the Italian Research Center on High-Performance Computing, Big Data and Quantum Computing (ICSC) funded by MUR Missione 4 Componente 2 Investimento 1.4:Potenziamento strutture di ricerca e creazione di "campioni nazionali di R\&amp;S(M4C2-19)" - Next Generation EU (NGEU). F.D. acknowledges the MUR Excellence Department Project awarded to the Department of Mathematics, University of Pisa, CUP I57G22000700001. The last three authors are members of the INdAM Research Group GNCS.

Availability of data and materials PSCToolkit is available at https://psctoolkit.github.io/ The data that support the findings of this study are available from the corresponding author upon reasonable request.

Declarations

Ethical Approval N/A

Conflict of interest N/A

References

1. Amestoy P, Duff I, L'Excellent JY (2000) Multifrontal parallel distributed symmetric and unsymmetric solvers. Comput Methods in Appl Mech Eng 184(2):501–520. https://doi.org/10.1016/S0045-7825(99)00242-X, https://www.sciencedirect.com/science/article/pii/S004578259900242X
2. Aprovitola A, D'Ambra P, Denaro F, et al (2011) Scalable algebraic multilevel preconditioners with application to CFD. Lecture Notes in Computational Science and Engineering 74 LNCSE:15 - 27. https://doi.org/10.1007/978-3-642-14438-7_2
3. Aprovitola A, D'Ambra P, Denaro FM et al (2015) SParC-LES: Enabling large eddy simulations with parallel sparse matrix computation tools. Comput Math Appl 70(11):2688–2700. https://doi.org/10.1016/j.camwa.2015.06.028
4. Baker AH, Falgout RD, Kolev TV et al (2011) Multigrid smoothers for ultraparallel computing. SIAM J Sci Comput 33(5):2864–2887. https://doi.org/10.1137/100798806
5. Bechmann A, Sorensen NN, Berg J et al (2011) The Bolund experiment, Part II: Blind comparison of microscale flow models. Bound -Layer Meteorol 141(2):245–271
6. Berg J, Mann J, Bechmann A et al (2011) The Bolund experiment, Part I: flow over a steep. Three-Dimensional Hill. Bound-Layer Meteorol 141(2):219. https://doi.org/10.1007/s10546-011-9636-y
7. Bolund (2022) The Bolund experiment. https://www.bolund.vindenergi.dtu.dk, accessed: 2022-09-27
8. Borrell R, Cajas JC, Mira D et al (2018) Parallel mesh partitioning based on space filling curves. Comput Fluids 173:264–272. https://doi.org/10.1016/j.compfluid.2018.01.040
9. Buttari A, D'Ambra P, Di Serafino D et al (2007) 2LEV-D2P4: A package of high-performance preconditioners for scientific and engineering applications. Appl Algebra Eng Commun Comput 18(3):223–239. https://doi.org/10.1007/s00200-007-0035-z
10. Capuano F, Coppola G, Chiatto M et al (2016) Approximate projection method for the incompressible Navier–Stokes equations. AIAA J 54(7):2179–2182. https://doi.org/10.2514/1.J054569
11. Catalyürek UV, Dobrian F, Gebremedhin A, et al (2011) Distributed-memory parallel algorithms for matching and coloring. In: 2011 IEEE international symposium on parallel and distributed processing workshops and Phd forum, pp 1971–1980, https://doi.org/10.1109/IPDPS.2011.360
12. Codina R (2001) Pressure stability in fractional step finite element methods for incompressible flows. J Comput Phys 170:112–140
13. D'Ambra P, Vassilevski PS (2013) Adaptive AMG with coarsening based on compatible weighted matching. Comput Vis Sci 16(2):59–76. https://doi.org/10.1007/s00791-014-0224-9
14. D'Ambra P, di Serafino D, Filippone S (2010) MLD2P4: a package of parallel algebraic multilevel domain decomposition preconditioners in Fortran 95. ACM Trans Math Software 37(3):23. https://doi.org/10.1145/1824801.1824808
15. D'Ambra P, Filippone S, Vassilevski PS (2018) BootCMatch: a software package for bootstrap AMG based on graph weighted matching. ACM Trans Math Software 44(4):25. https://doi.org/10.1145/3190647
16. D'Ambra P, Durastante F, Filippone S (2021) AMG preconditioners for linear solvers at extreme scale. SIAM J Sci Comp. https://doi.org/10.1137/20M134914X
17. Filippone S, Buttari A (2012) Object-oriented techniques for sparse matrix computations in Fortran 2003. ACM TOMS 38(4):23:1-23:20

18. Filippone S, Colajanni M (2000) PSBLAS: a library for parallel linear algebra computations on sparse matrices. ACM TOMS 26(4):527–550
19. Houzeaux G, de la Cruz R, Owen H et al (2013) Parallel uniform mesh multiplication applied to a Navier–Stokes solver. Comput Fluids 80:142–151. https://doi.org/10.1016/j.compfluid.2012.04.017
20. Houzeaux G, Borrell R, Fournier Y, et al (2018) High-Performance Computing: Dos and Don'ts. In: Ionescu A (ed) Computational Fluid Dynamics - Basic Instruments and Applications in Science. IntechOpen, pp 3–41, https://doi.org/10.5772/intechopen.72042
21. Karypis G, Kumar V (1998) A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM J Sci Comput 20(1):359–392. https://doi.org/10.1137/S1064827595287997
22. Lehmkuhl O, Houzeaux G, Owen H et al (2019) A low-dissipation finite element scheme for scale resolving simulations of turbulent flows. J Comput Phys 390:51–65. https://doi.org/10.1016/j.jcp.2019.04.004
23. Li XS (2005) An overview of SuperLU: algorithms, implementation, and user interface. ACM Trans Math Software 31(3):302–325. https://doi.org/10.1145/1089014.1089017
24. Owen H, Chrysokentis G, Avila M et al (2020) Wall-modeled large-eddy simulation in a finite element framework. Int J Numer Meth Fluids 92(1):20–37. https://doi.org/10.1002/fld.4770
25. PRACE (accessed May 2020) Unified European Application Benchmark Suite. https://repository.prace-ri.eu/git/UEABS/ueabs/
26. PRACEBS (accessed May 2020) PRACE benchmark-suite. https://prace-ri.eu/training-support/technical-documentation/benchmark-suites/
27. Vaněk P, Mandel J, Brezina M (1996) Algebraic multigrid by smoothed aggregation for second and fourth order elliptic problems. Computing 56(3):179–196. https://doi.org/10.1007/BF02238511. (**international GAMM-Workshop on Multi-level Methods (Meisdorf, 1994)**)
28. Vassilevski PS (2008) Multilevel block factorization preconditioners: matrix-based analysis and algorithms for solving finite element equations. Springer, New York
29. Vazquez M, Houzeaux G, Koric S et al (2016) Alya: multiphysics engineering simulation toward exascale. J Comput Sci 14:15–27. https://doi.org/10.1016/j.jocs.2015.12.007
30. Vreman AW (2004) An eddy-viscosity subgrid-scale model for turbulent shear flow: Algebraic theory and applications. Phys Fluids 16(10):3670–3681

Authors and Affiliations

Herbert Owen[1] · Oriol Lehmkuhl[1] · Pasqua D'Ambra[2] · Fabio Durastante[2,3] · Salvatore Filippone[2,4]

✉ Pasqua D'Ambra
pasqua.dambra@cnr.it

Herbert Owen
herbert.owen@bsc.es

Oriol Lehmkuhl
oriol.lehmkuhl@bsc.es

Fabio Durastante
fabio.durastante@unipi.it

Salvatore Filippone
salvatore.filippone@uniroma2.it

[1] Barcelona Supercomputing Centre (BSC), Plaça d'Eusebi Güell, 08034 Barcelona, Spain

[2] Institute for Applied Computing, National Research Council (CNR), Via P. Castellino, 111, 80131 Naples, NA, Italy

[3] Department of Mathematics, University of Pisa, Largo Bruno Pontecorvo, 5, 56127 Pisa, PI, Italy

[4] Department of Civil and Computer Engineering, University of Rome "Tor Vergata", Via del Politecnico, 1, 00133 Rome, RM, Italy