Available on-line at www.prace-ri.eu

Partnership for Advanced Computing in Europe

PETSc4FOAM: A Library to plug-in PETSc into the OpenFOAM Framework

Simone Bnà[a], Ivan Spisso[a,*], Mark Olesen[b], Giacomo Rossi[c]

[a]SuperCompunting Application and Innovation Department, Cineca, Via Magnanelli 6/3, 40133, Casalecchio di Reno, Bologna, Italy
[b]ESI-OpenCFD, Engineering System International GmbH, Einsteinring 24, 85609 Munich, Germany
[c]Intel Corporation Italia SpA, Milanofiori Palazzo E 4, Milano, Italy

Abstract

OpenFOAM acts as a major player in the Open Source CFD arena, due to its flexibility, but its complexity also makes it more difficult to correctly define performance figure and scaling. One of the main bottlenecks for a full enabling of OpenFOAM for massively parallel cluster is the limit in its MPI-parallelism paradigm, embodied in the Pstream library, which limits the scalability up to the order of few thousands of cores. The proposed work aims to creating an interface to external linear algebra libraries for solving sparse linear system such as PETSc/Hypre thus providing to the users a greater choice and flexibility when solving their cases, and to utilise their respective Community's knowledge which has been developed over decades and not currently accessible within the OpenFOAM framework.

1. Introduction

OpenFOAM acts as a major player in the Open Source CFD arena, due to its flexibility, but its complexity also makes it more difficult to correctly define performance figures and scaling in an HPC environment.
From previous works, it is known that the scalability of the linear solvers restricts the parallel usage up to the order of few thousand of cores [1–3]. Meantime, the technological trends of exascale HPC is moving towards the use of order of millions of cores as reported in the November 2019 Top 500 List. The actual trend of the HPC cluster is to use nodes with high-density of cores, with order of O(10) cores per node, in conjunction with accelerators (GPUs, FPGAs, Xeon PHI, RISCV), with more cache level and with the order of O(100,000) of interconnected nodes and O(1,000,000) of available cores [4]. The current work aims to improve the HPC performance of the OpenFOAM, by means of a library to plug-in the PETSc into the OpenFOAM framework. PETSc stands for **P**ortable, **E**xtensible **T**oolkit for **S**cientific **c**omputation toolkit for advanced computation and it is pronounced PET-see (the S is silent).
The paper is structured as follows: Section 1. briefly reviews the activity related to HPC optimization of the OpenFOAM problem solving component. In Section 2. the OpenFOAM framework is introduced with a focus on its parallel aspects and the actual HPC bottlenecks. Section 3. sets forth the concept of PETSc as OpenFOAM plug-in for the linear algebra solvers. The relevant works of coupling linear solver algebra packages into OpenFOAM are described and the PETSc framework with its main components is introduced. Further on, the actual sparse matrix storage system of OpenFOAM (LDU), of PETSC (CSR) and the converter (ldu2csr) are presented. The structure of the *Petsc4FOAM* library is listed. Section 4. describes the test-case used with reference to the OpenFOAM HPC benchmark suite. Section 5. shows the numerical results and quantifies the benefits of the library for the used test-case. Finally, Section 6. concludes the article with an outlook to the further work.

1.1. HPC optimization of OpenFOAM problem solving

Since CFD simulation is usually a computationally intensive procedure, its performance has attracted much attention, and the optimization towards the CFD framework has always been a hot issue. In the past decades, lots of efforts have been made to improve the HPC performances of OpenFOAM, specifically for the problem solving component (see Figure 1). Historically, these activities can be divided into the following categories:

*Corresponsing author
tel. (+39) 051-6171445 e-mail. ivan.spisso@cineca.it

1. Improvement of the actual parallelism paradigm (`Pstream` library)
2. Multi-threaded-hybrid MPI/OpenMP parallelization
3. Use of accelerators as plug-in

This categorization does not include the activities related to the pre- and post-processing work-flows, such as domain decomposition, I/O, visualization, etc. ....
Regarding the first approach, past activities [5–7] focused on the improvement of the data exchange method in the inter-process communication protocol *Pstream*, to minimize data transfers based on the adjacent data exchange form. It successes to reduce the communication buffer size and the communication time and improve the performance of the entire application in large-scale parallel simulation. It has been already exploited and implemented into the main release since version $v1606$ [8], mainly due to a series of enhancements suggested by the contribution of Japanese Research organisation for Information Science and Technology (RIST) [6].
Concerning the second approach, a lot of effort has been spent in the past, mainly under the PRACE initiative [3,9–13]. Among them, the seminal work of Culpo [3] shows that the bottlenecks preventing scalability beyond order of O(100) cores resides in the linear algebra solvers. In particular, the scalar product proved to be the most communication intensive function for large number of tasks, due to the necessity of an `MPI_Allreduce` call for each scalar. The strategy of a multi-threaded hybridization of the linear algebra core libraries has been investigated. The work of Dagna et al. [11] stems from the previous one, by extending the hybrid multi-threaded solution to four OpenFOAM solvers, and tested them on HPC clusters. They found that when communication becomes the limiting factor to scalability, the benefits obtainable from a hybrid implementation are heavily mitigated by the "not hybridisable parts" where MPI communication increases drastically with the number of core used. In spite of the efforts and the in-depth analysis carried out in these works, no substantial benefits have been found; this is mainly due to the inherit and structural HPC bottlenecks of OpenFOAM (see Section 2.4.).
The third approach aims to use customized hardware platforms or architectures. As a representative example of this approach, Intel has released in the past `libhbm`, a library written specifically for running OpenFOAM in "flat" memory mode using Intel Xeon Phi product family (Knights Landing aka KNL [14]). The main objective of such library is to improve the memory bandwidth [15], one of the HPC bottlenecks of OpenFOAM.
Recently, Posey [16] gave an overview of works done by porting OpenFOAM into GPUs. Several efforts done by the communities use NVIDA-developed libraries such as `cuBLAS, cuSPARSE, AmgX`. This aids code maintainability, ongoing performance gains, and NVIDIA support. Among this libraries, `AmgX`, an NVIDIA library implementing algebraic multi-grid methods over unstructured grids, has capabilities well comparable with OpenFOAM. NVIDIA expectations relies upon a likely modification in matrix storage format from current LDU. Notably, two main groups have modernized their GPUs-enabled libraries with latest CUDA and Nvidia $V100$ GPUS [17]: Symscape by RapidCFD [18] and SpeedIT by Vratis [19].
It is noteworthy the work of Piscaglia et al. [20] on fast algorithms for highly underexpanded reactive spray simulations; they have ported to GPUs the ODEs deriving from the active chemistry. Taouil [21] implemented a matrix-vector multiplication accelerator based on field programmable gate arrays (FPGAs) and gained a speedup of about 2.7× to 7.3× for the SpVM kernel considering four PEs (Processing Elements). The increase in performance at the kernel level is nearly linear to the number of available PEs. Recently, a collection of algorithms, optimized for Xilinx Alveo FPGA, are developed by third party organizations [22].
All these efforts lack of generality and no inclusion or buy-in of such libraries have been done from OpenFOAM development organizations; this is required for successful code maintenance and repository distribution. Therefore, it is necessary to explore some general optimization strategies for the OpenFOAM framework.

1.2. Proposed work

The current work proposes a direct approach to improve the performances by reducing the time for solving linear equation systems. Specifically, *PETSc4FOAM* has been developed to use PETSc (and other linear algebra solver packages) as external library into the OpenFOAM framework.
The contributions of this work are:

i Release of an HPC test-cases repository made public available [23] by means of the OpenFOAM HPC Technical Committee [24]

ii Release of an interface to the PETSc linear solver toolkit, available on a repository to be publicly released in the near future [25]

   (a) A converter modifies the matrix format from the LDU to one of the matrix data structure available in PETSc, default is CSR

   (b) The whole bunch of solvers and preconditioners available can be invoked with a dictionary or a rc type file, for usability purpose

iii Possibility to use any liner algebra solver packages within OpenFOAM, compatible with PETSc structure such as `SuperLU_DIST`, `SuperLU_MCDT`, `MUMPS` (direct solvers) or `Trilinos`, `Hypre` (algebraic preconditioners), or to include a custom one by means of the *PETSc4FOAM* library.

2. The OpenFOAM framework

2.1. History of OpenFOAM

OpenFOAM (for "**Open**-source **F**ield **O**peration **A**nd **M**anipulation") is the free, open source CFD software developed primarily by OpenCFD Ltd since 2004 [26,27]. Currently, there are three main variants of OpenFOAM software that are released as free and open-source software under the GNU General Public License Version 3 [28–30].

- OpenFOAM® variant by OpenCFD Ltd, with the name trademarked since 2007 [31], first released as open-source in 2004. Since 2012, OpenCFD Ltd is an affiliate of ESI Group.
- FOAM-Extend variant by Wikki Ltd (since 2009).
- OpenFOAM Foundation Inc. variant, released by The OpenFOAM Foundation Inc. (since 2012), and transferred in 2015 to the English company The OpenFOAM Foundation Ltd.

The development model has a "*cathedral*"style [32], where code contributions from researchers are accepted back into the main distribution only after a very strict control of code base. The source code is available with each software release, but the code developed between releases is restricted to an exclusive group of software developers.

2.2. The execution flow of OpenFOAM

First and mostly, OpenFOAM consists of a C++ CFD toolbox for customized numerical solvers (over sixty of them) that can perform simulations of basic CFD, combustion, turbulence modeling, electromagnets, heat transfer, multi-phase flow, stress analysis, and even financial mathematics.
It has a large user base across most areas of engineering and science, from both commercial and academic organisations. The number of OpenFOAM users has been steady increasing over past years. It is now estimated to be of the order of thousands, with the majority of them being engineering in Europe. It is one of the most used open-source CFD code in HPC environment.
As a general CFD framework, the simulation with OpenFOAM is composed of three common procedures of CFD simulation: pre-processing, problem solving, and postprocessing, as shown in Figure 1.
Preliminary work, such as geometric modeling, mesh generation, and parallel partition, are done in the pre-processing procedure. Problem solving is the main procedure of OpenFOAM, which includes physical modeling, equation discretization, and numerical solving. The reconstitution and visualization of data are managed in post-processing. The current work focuses on the problem-solving part of the OpenFOAM framework.

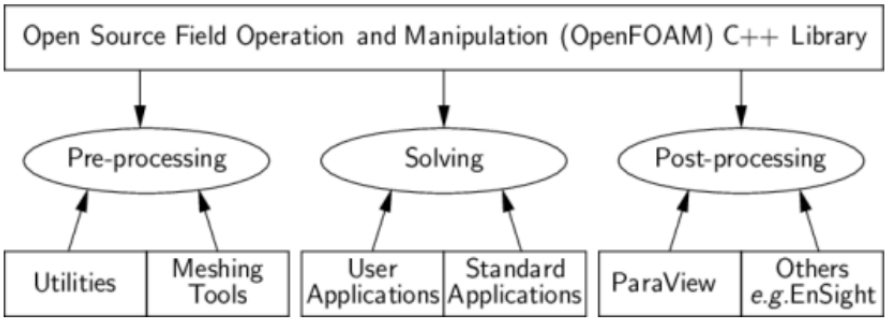

Fig. 1: Overview of OpenFOAM structure, from [33].

The main features of the numerical method OpenFOAM relies upon are: segregated, iterative solution of linear algebra solvers, unstructured finite volume method (FVM), co-located variables, equation coupling (e.g. SIMPLE, PIMPLE, etc.). It uses $C++$ and object-oriented programming to develop a syntactical model of *equation mimicking* and scalar-vector-tensor operations [33]. Figure 2 shows a code snippet for the Navier-Stokes equations. Equation mimicking is quite obvious: `fvm:laplacian` means an implicit finite volume discretization for the Laplacian operator, and similarly `fvm::div` for the divergence operator. On the other hand, `fvc::grad` means an explicit finite volume discretization for the gradient operator. The parentheses `(,)` means a product of the enclosed quantities, including tensor products.
The discretization is based on low order (typically second order) gradient schemes. It is possible to select different choices of iterative solvers for the solution of the linear solver algebra based on SpMVM (Sparse Matrix-Vector Multiplication), ranging from Conjugate Gradient up to Multi-Grid Methods. The temporal schemes can be selected among first and second order implicit CrankNicholson and Euler schemes [33].
Figure 3 is the dynamic flow execution of OpenFOAM's problem-solving procedure. First of all, the governing equation in mathematical form is converted to an OpenFOAM format. Each of the operators is discretized by the Finite Volume Method (FVM), then generates an *lduMatrix*, which is the matrix format used in OpenFOAM (see Section 3.5.). At last, all *lduMatrices* are integrated into a final linear equation system that can be solved by linear solvers like PCG (Preconditioned Conjugate Gradient) in OpenFOAM [34].

```
Solve
(
fvm::ddt(rho,U)
+ fvm::div(U,U)
- fvm::laplacian(mu,U)
==
- fvc::grad(p)
+ f
);
```

Fig. 2: Equation mimicking: OpenFOAM code for Navier-Stokes equations

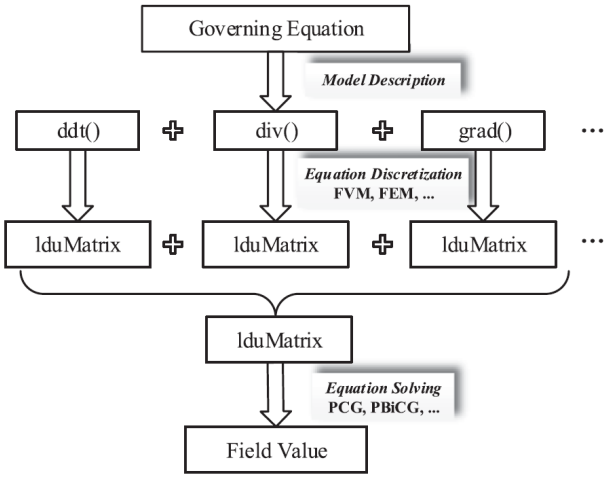

Fig. 3: Flow chart of OpenFOAM

2.3. Parallel aspect of OpenFOAM

The method of parallel computing used by OpenFOAM is based on domain decomposition in which the geometry and associated fields are broken into pieces and allocated to separate processors for solution. The parallel paradigm is based on the standard MPI. A convenient interface, named `Pstream`, that is a stream-based parallel communication library, is used to plug in any MPI library into OpenFOAM. It is a light wrapper around the selected MPI interface [35].
Traditionally, FVM parallelization uses the *halo layer* approach: data for cells next to a processor boundary is duplicated. Halo layer covers all processor boundaries and is explicitly updated through parallel communication cells. Instead, OpenFOAM operates in *zero halo layer approach*, which gives flexibility in the communication pattern. FVM operations "look parallel" without data dependency [36].

2.4. Actual HPC bottlenecks of OpenFOAM

From previous works, it is known that the scalability of the linear solvers restricts the parallel usage up to the order of few thousand of cores [1,3].
Up to date, the well known bottlenecks for a full enabling of OpenFOAM for massively parallel clusters are:

i The limit in the parallelism paradigm (`Pstream` Library)

ii Sub-optimal sparse matrices storage format (LDU) that does not enable any cache-blocking mechanism (SIMD, vectorization)

iii The I/O data storage system

The recently formed HPC Technical Committee (TC) [37] aims to work together with the Community to overcome the aforementioned HPC bottlenecks. The priorities of the HPC TC Committee are:

- Improve linear algebra solver limitation
- Create an open and shared repository of HPC Benchmarks

- GPUs enabling of OpenFOAM
- Parallel I/O: Adios 2 I/O is now available as a regular OpenFOAM module [38].

The current work will focus on the first priority of the list, which in turn will affect the second and third HPC bottleneck.
Specifically, a library named *PETSc4FOAM* is implemented to use PETSc [39], and other linear algebra solver packages, as external libraries into the OpenFOAM framework.
As a representative example, Figure 4 reports the profiling of S test-case of Lid Driven cavity flow, described later on, in Sec. 4.1. From the profiling of Intel Advisor it is observed that most of the computational time, namely $67,6\%$, is spent in the pressure solver part (that is `Foam:fvMatrix<double>::solveSegragated` function), and around 27% is spent in the velocity solver section (that is `Foam:solve<Foam::Vector<double>>` function).

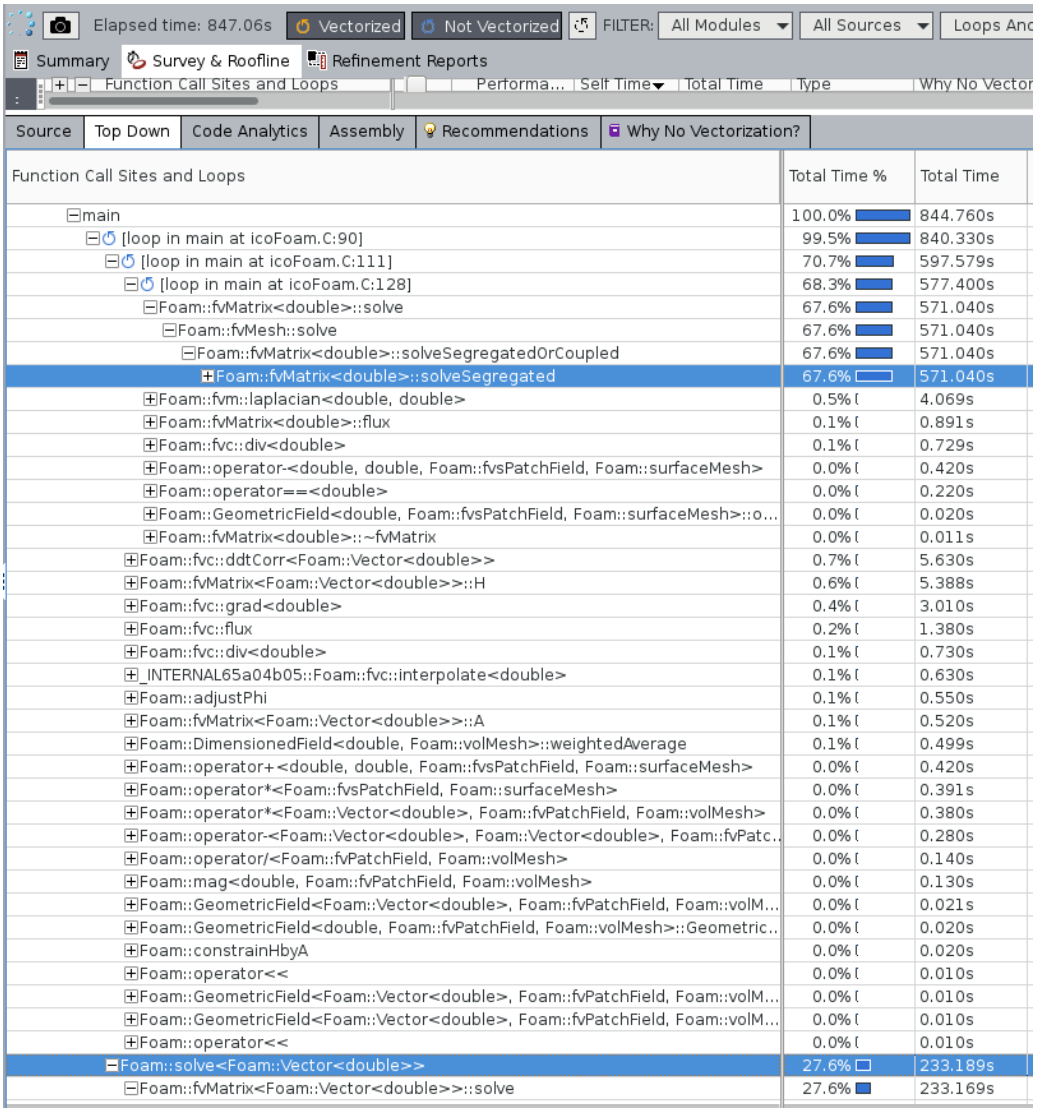

Fig. 4: Profiling of S Test-case with Intel Advisor (example picture)

3. Plug-in of PETSc into OpenFOAM

3.1. Related work: coupling linear solver algebra packages into OpenOFOAM

HPC parallel aspects of OpenFOAM are well-known and described in many works [1,3,12]. There is an overall agreement in the community that the linear algebra solving process is the most-consuming task and that affects its parallel scalability.
Previous examples of plug-in of a linear solver algebra library into OpenFOAM are detailed in [40,41]. Duran et. al [40] studied bio-medical fluid flow simulation by embedding the incompressible, laminar solver *icoFoam* with other direct solvers (kernel class) such as `SuperLU_DIST` 3.3 and `SuperLU_MCDT` (Many-Core Distributed) [42] for the large penta-diagonal and hepta-diagonal matrices coming from the simulation of blood flow in arteries with a structured mesh domain. They observed speed-up up to $16,384$ cores for the largest matrix of sizes up to 64 million $\times$ 64 million.
The authors support the idea of decoupling the effort in the linear algebra solver from the effort spent in the numerical solution of PDEs using the finite volume algorithm on unstructured arbitrary mesh. This approach is beneficial for the OpenFOAM framework, not only for a performance gain but also for the software quality improvement and the reduction of time spent in software development and maintenance. Furthermore, the plug-in of PETSc into OpenFOAM provides the open-access to the state-of-the art linear algebra solvers and preconditioners (e.g. the BoomerAMG preconditioner provided by the Hypre package for massively parallel computations [43]), developed and maintained by the PETSc community or from other Public Research Institutes.
PETSc is a library well-known for its high performance and good scalability. For these reasons it has been adopted in many areas (CSM, CFD, Aerodynamics, two-phase flow, porous flow, to name a few) and in many frameworks [44–46] or application codes [47–50]. Success stories of performance improvement by switching to PETSc, see [47,51–53], and the recent work [41] have inspired this work.
Li et al. [41] report an optimization of the OpenFOAM framework by inserting PETSc library to speed up the process of solving linear equation systems. Numerical results on a HPC cluster showed that such optimization can reduce the time for solving PDEs by about 27% in the 2-D lid-driven cavity flow and more than 50% in a 2-D flow past a circular cylinder in a uniform stream, when compared to the best solution with the original OpenFOAM. Both cases consists of a fixed mesh size of 192,000 cells; strong scaling tests have been performed from 6 up the 786 cores. The analysis have been profiled with *valgrind* tool. Final test-case reported is a 3-D incompressible flow around a submarine, with mesh size of 1 millions of cells. The minimal execution time of OpenFOAM-PETSc is almost 40% less than that with the original OpenFOAM.
The current work relies upon this precursor effort, and aims to push forward the activities in terms of:

- Public release of a library under the supervision of the main developers of OpenFOAM
- Very large test-cases (order of ten of millions of cells) running on massively parallel cluster (order of thousands of cores)
- Use of HPC profiling tools to spot bottlenecks in HPC scenario
- Systematic comparison among same or equivalent solvers and/or preconditioners to solve the sparse linear algebraic system
- Evaluation of the residual norm in the same way as done by OpenFOAM, that is the scaled L1 norm
- Enabling cache in memory of the coefficient matrix A and its preconditioner
- Possibility to extend the library in order to use GPUs through CUDA or OpenCL in PETSc [54]

3.2. The PETSc framework

PETSc is a suite of data structures and routines for the scalable (parallel) solution of scientific applications modeled by partial differential equations and sparse matrix computations. PETSc includes a large suite of parallel linear, nonlinear equation solvers and ODE integrators that are easily used in application codes written in C, C++, Fortran and Python.
PETSc provides many of the mechanisms needed within parallel application codes, such as simple parallel matrix and vector assembly routines that allow the overlap of communication and computation. It supports MPI, and GPUs through CUDA or OpenCL, as well as hybrid MPI-GPU parallelism [39,55].
PETSc consists of a variety of libraries (similar to classes in C++), which manipulate a particular class of objects and the operations one would like to perform on the objects. Some of the PETSc modules includes: index sets, vectors, matrices, over thirty Krylov subspace methods, dozens of preconditioners, including multigrid, block solvers, and sparse direct solvers. Each module consists of an abstract interface and one or more implementations using particular data structures. The library enables easy customization and extension of both algorithms and implementations. Figure 5 is a diagram of the interrelationships among different pieces of PETSc, which shows the library's hierarchical organization [55].

3.3. The execution flow of OpenFOAM-PETSc

After the insertion of the PETSc library into the OpenFOAM framework, the execution flow of OpenFOAM is unchanged except for the solving part.

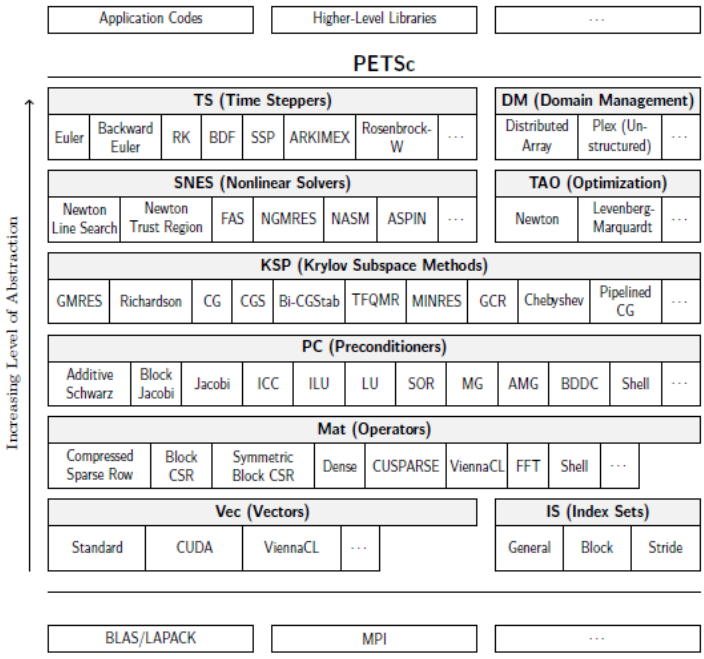

Fig. 5: Numerical libraries of PETSc, taken from [55]

Instead of calling one of the built-in solver based on the LDU matrix format, the final lduMatrix is converted into a PETSc matrix, which can be digested by a PETSc solver, made of a KSP (Krylov SubsPace) and a PC (PreConditioner) object. The matrix conversion introduces a slight overhead (see Fig. 14 later on), but it has the advantage to have a null impact on the OpenFOAM source code.

3.4. Iterative solution of sparse linear systems: preconditioning techniques

The solution of large sparse linear systems of the form

$$A\,x = b \tag{1}$$

where $A = [a_{ij}]$ (for symmetric matrices $a_{ij} = a_{ji}\,\forall\, i \neq j$) is an $n \times n$ matrix and b is a given right-hand-side vector, derives typically from the discretization of the PDEs of elliptic and parabolic type, such as the governing equation OpenFOAM relies upon (eq. (6) later on).
Direct methods are based on the factorization of the coefficient matrix A into easily invertible matrices. It is possible to efficiently solve, in a reasonable amount of time, linear systems of fairly large size particularly when the underlying problem is two dimensional. Unfortunately, direct methods scale poorly with problem size in terms of operation counts and memory requirements, especially on problems arising from the discretization of PDEs in three space dimensions. Detailed, three-dimensional multiphysics simulations lead to linear systems comprising hundreds of millions or even billions of equations as many unknowns. For such problems, in the majority of cases, iterative methods are the only option available [56].
As widely reported in literature [57], the term preconditioning refers to transforming the system of eq. (1) into another system with more favorable properties for iterative solution; a *preconditioner* is a matrix that effects such a transformation. Mathematically, preconditioning attempts to improve the spectral properties of the coefficient matrix. For symmetric positive definite (SPD) problems, the rate of convergence of the conjugate gradient method depends on the distribution of the eigenvalues of A. Hopefully, the transformed (preconditioned) matrix will have a smaller spectral condition number, and/or eigenvalues clustered around 1. For nonsymmetric (nonnormal) problems the situation is more complicated, and the eigenvalues may not describe the convergence of nonsymmetric matrix iterations like GMRES (Generalized Minimal Residual Algorithm) for solving nonsymmetric linear systems.
If M is a non-singular matrix that approximates A (in some sense), then the linear system

$$M^{-1}\,A\,x = M^{-1}\,b \tag{2}$$

has the same solution as eq. (2) but may be easier to solve. Here M is the *preconditioner*. In cases where M^{-1} is explicitly known (as with polynomial preconditioners or sparse approximate inverses), the *preconditioner* is M^{-1} rather than M. System of eq. (2) is preconditioned from the left, but one can also precondition from the

right:

$$\begin{aligned} A\,M^{-1}\,z &= b\,, \\ M\,x &= z\,. \end{aligned} \tag{3}$$

When Krylov subspace methods are used, it is not necessary to form the preconditioned matrices $M^{-1}\,A$ or $A\,M^{-1}$ explicitly (this would be too expensive, and we would lose sparsity). Instead, matrix–vector products with A and solutions of linear systems of the form $M\,z = r$ are performed (or matrix–vector products with M^{-1} if this is explicitly known). In addition, split preconditioning is also possible, i.e.:

$$M_1^{-1}\,A\,M_2^{-1}\,y = M_1^{-1}\,b, \quad x = M_2^{-1}\,y \tag{4}$$

where the *preconditioner* is now $M = M_1\,M_2$.
Which type of preconditioning to use depends on the choice of the iterative method, problem characteristics, and so forth. Generally speaking, the linear system must be preconditioned to achieve convergence. Convergence is accelerated if M^{-1} resembles, in some way, A^{-1}; conversely, M^{-1} must be sparse at the same time, so as to keep the cost for the preconditioner computation, storage and application to a vector as low as possible. Generally speaking, preconditioning is "the art of transforming a problem that appears intractable into another whose solution can be approximated rapidly" [58].

3.5. Sparse matrix storage formats: ldu2csr converter

While for dense matrices it is usually reasonable to store all matrix entries in consecutive order, sparse matrices are characterized by a large number of zero elements, and storing those is not only unnecessary but would also result in significant storage overhead. Different storage layouts exist, that aim for reducing the memory footprint of the sparse matrix by storing only a fraction of the elements explicitly, and anticipating all other elements to be zero.
The matrix A, showed on the right-hand side of Fig. 6, is a representative example that derives from numerical discretization of a Poisson solver over the two-dimensional numerical grid 3×3 displayed to the left hand-side of Fig. 6 (the numerical discretization consists of a finite difference method with a five-point stencil with second order accuracy $O(h^2)$ [59]). Matrix A will be used later on to describe different storage formats.

lduMatrix format

The coefficients of the matrices in OpenFOAM are stored in the LDU format. In such format, the matrix A is splitted into an upper triangle U, a diagonal D and a lower triangle L matrix, i.e. ($A = U + D + L$). Such matrices are stored in three vectors named, respectively, *upper*, *diag*, and *lower*, filled following the left-right top-down order. Two additional vectors are provided by the lduMatrix class implementation, named respectively *upperAddr* and *lowerAddr* (stands for *upper* and *lower Address*), in order to visit the index of the lower and upper triangle of the matrix A.
For the upper triangular matrix U, the *lowerAddr* provides the index of the rows (globally ordered) while the *upperAddr* provides the index of the columns (locally ordered); for the lower triangular matrix L the index of the *lowerAddr* and *upperAddr* are opposite to the one in the upper triangle matrix U. The number of elements in the *diag* vector is equal to the number of cells (mesh size), and the elements are indexed by the cell id. The number of elements in the *lower* and *upper* vectors are equal to the number of internal faces, and the elements are indexed by the face id.

csrMatrix format

The default matrix format in PETSc is the Compressed Sparse Row (CSR), but the toolkit implements many other matrix storage format (ELL, SELL, Hybrid, to name a few). CSR format has a high storage efficiency. Fig. 7 shows the CSR representation of the same two-dimensional computational grid and the corresponding matrix A shown in Fig. 6. The non-zero elements of the matrix A are stored in a single contiguous array, named *value*. Two auxiliary vectors, named respectively *col_index* and *row_index*, are needed to visit the matrix coefficients: *col_index* provides the column indices of those values; while *row_index* has one element per row of the matrix A and encodes the index in *value* where the given row starts. The size of *value* and *col_index* is the number of cells, while the size of *row_index* is the number of rows plus 1.

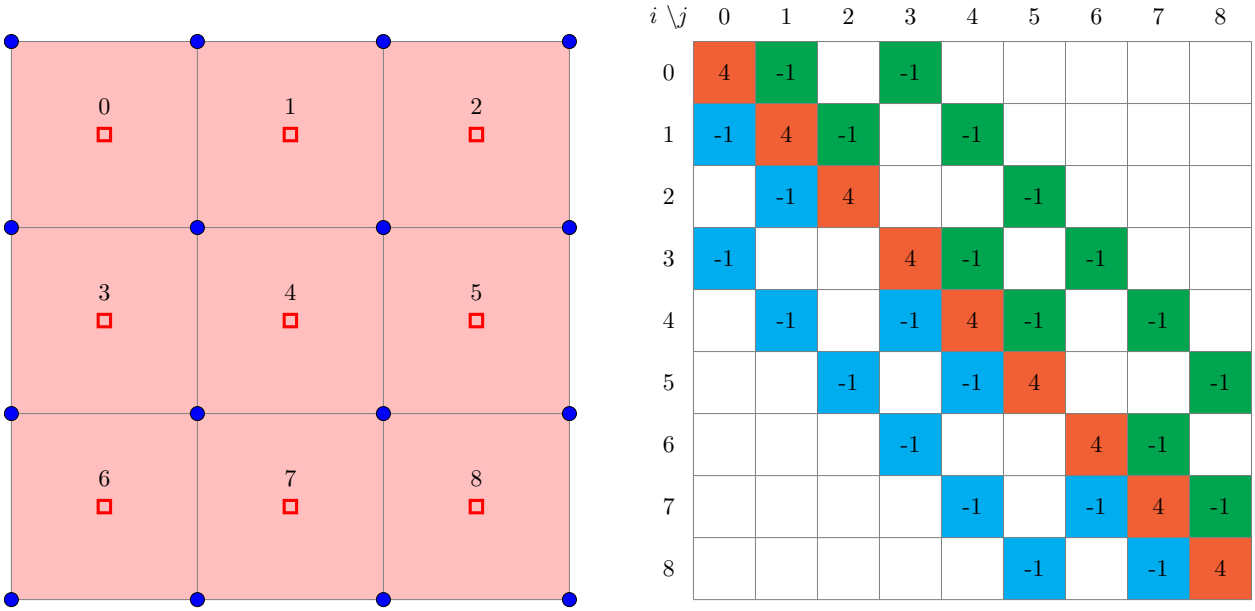

Fig. 6: Representation of the LDU matrix storage format: left hand-side, 2-D computational grid 3×3 with cell id at the cell center; right hand-side: corresponding matrix $A = U + D + L$. Different colours are used for the different matrices U, D, and L.

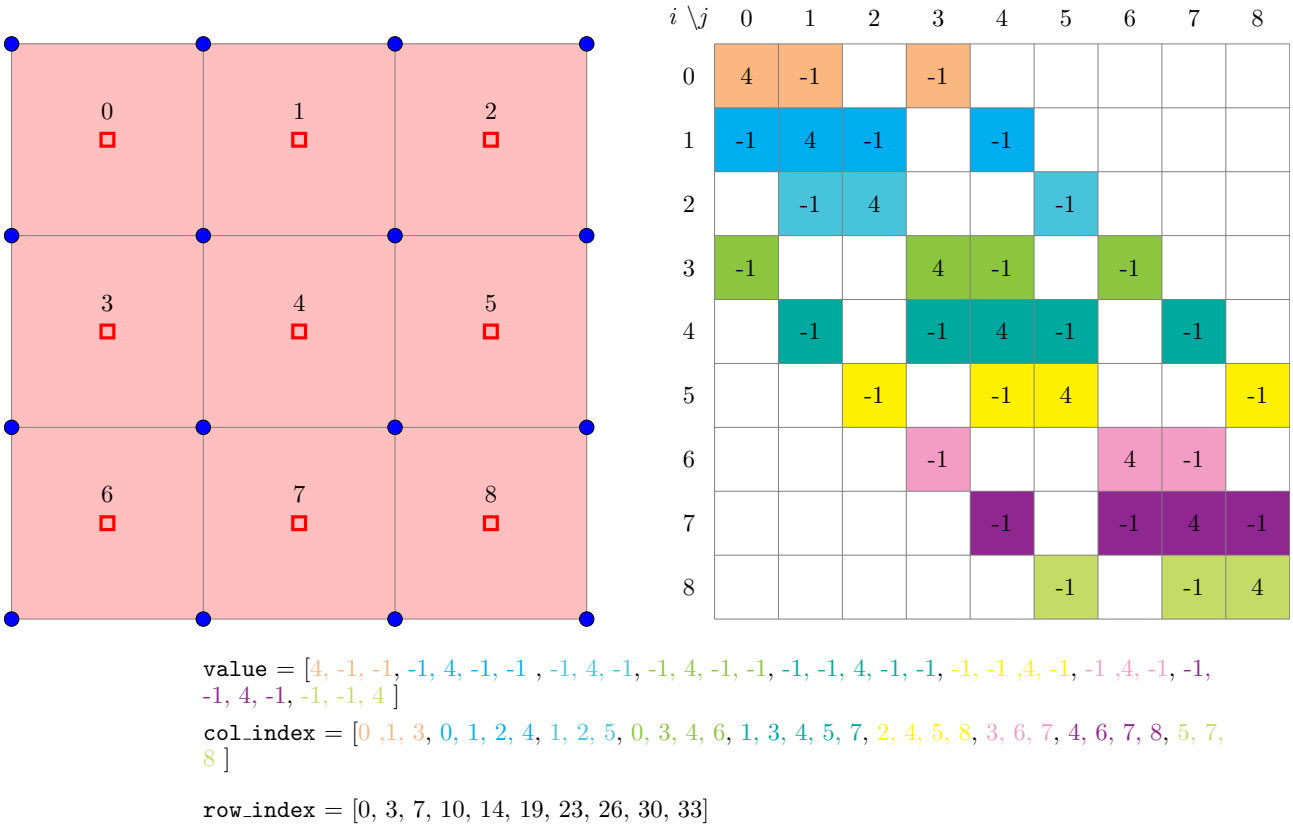

Fig. 7: Representation of the CSR matrix storage format: left hand-side, 2-D computational grid 3×3 with cell id at the cell center; right hand-side: corresponding matrix A. Different colours are used for the different rows of the matrix A.

ldu2csr converter

PETSc provides the abstract type *Mat* and the *MatSetValue()* API to set the a_{ij} coefficient for every type of matrix. According to the value set with *MatSetType*, a different storage format is used.
Some of the *MatTypes* available in PETSc are reported in Table 1. For sake of completness: `MATMPIAIJMKL` creates a sparse parallel matrix whose local portions are stored as `SEQAIJMKL` matrices (a matrix class that inherits from `SEQAIJ` but uses some operations provided by Intel `MKL`); `MATMPISBIAJ` matrix type is based on block compressed sparse row format and only the upper triangular portion of the matrix is stored; in the `MATSELL` format the block size makes the connection between the slim CSR format and the GPU-friendly ELLPACK format, for cross-platform efficient use [60].

MatTypes	Description
`MATAIJ`	"*aij*" type for sparse matrices
`MATSBAIJ`	"*sbaij*" type for symmetric block sparse matrices
`MATMPIAIJ`	"*mpiaij*" type for parallel sparse matrices
`MATMPIAIJMKL`	type for parallel sparse matrices with Intel MKL support
`MATMPISBIAJ`	"*mpisbaij*" type for distributed symmetric sparse block matrices
`MATHYPRE`	"*hypre*" type sequential and parallel sparse matrices based on the hypre *ij* interface
`MATSELL`	"*sell*" type for sparse parallel matrix in SELL format

Table 1: List of commonly used Mat types available in PETSc [61].

To convert an LDU matrix to a CSR matrix, we propose a traversal-based algorithm, as described in [41] and shown in the Algorithm 1 in pseudo-algorithm form. For good matrix assembly performance, PETSc requires to pre-allocate the matrix storage by setting the number of non-zero per rows both in the diagonal and in the off-diagonal portion of the local submatrix. In order to reduce the overhead of the conversion, we implemented the same optimizations proposed by [41]. Firstly, a special treatment for symmetric matrices is introduced, in which the a_{ij} element is equal to the corresponding a_{ji} element. Therefore, we can double the conversion efficiency by converting only the diagonal and upper triangular portion of A (lines `4-14` of Algorithm 1). Secondly, we can reduce the access frequency of the matrix elements; as described above, the non-zero values of the matrix are stored, in CSR format, in a contiguous array `value` row by row, ordered left-right and top-down, which is the same order used to store the upper triangular matrix U in the vector *upper* for the LDU format. Thus, we can just copy row by row from the LDU into the CSR matrix, rather than copy by element (lines `21-29` of Algorithm 1).
The LDU matrix is traversed once for all to compute the exact number of non-zero values per rows, as motivated above, and one time per time step (except for caching, see Sec. 3.6. later on) to set the coefficients in the PETSc matrix.

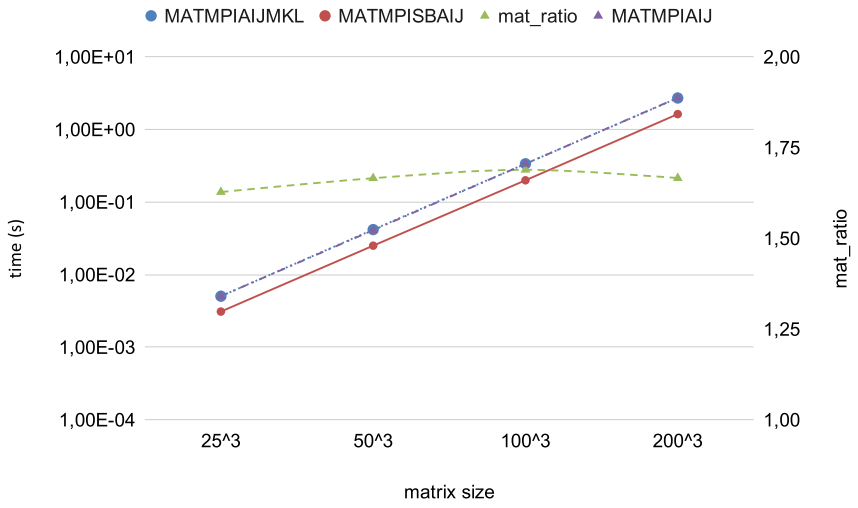

Fig. 8: Conversion times and mat_ratio versus matrix size for the *Mat* types reported in Tab. 2.

Algorithm 1: Listing of the `LDU2CSR` pseudo-algorithm

```
   /* initialization                                                    */
 1 Mat Amat;
 2 MatCreate(PETSC_COMM_WORLD, &Amat);
 3 MatSetFromOptions(Amat);
   /* Symmetric optimization                                           */
 4 if LDU matrix is symmetric then
 5 |  MatSetOption(Amat, MAT_SYMMETRIC, PETSC_TRUE);
 6 end
 7 else
   |  /* Loop the lower part of the LDU matrix                        */
 8 |  for idx = 0 to lower.size() do
 9 |  |  row = upperAddr[idx];
10 |  |  col = lowerAddr[idx];
11 |  |  val = lower[idx];
12 |  |  MatSetValue(Amat, row, col, val, INSERT_VALUES);
13 |  end
14 end
   /* Loop the elements in the diagonal part of LDU matrix             */
15 for idx = 0 to diag.size() do
16 |  row = idx;
17 |  col = idx;
18 |  val = diag[idx];
19 |  MatSetValue(Amat, row, col, val, INSERT_VALUES);
20 end
   /* Loop the upper part of the LDU matrix, row by row                */
21 kidx=0;
22 for idx = 0 to matrix.size() do
23 |  row = lowerAddr[idx];
24 |  for jidx = 0 to line[idx].size() do
25 |  |  cols[jidx] = upperAddr[kidx];
26 |  |  vals[jidx] = upper[kidx];
27 |  |  ++kidx;
28 |  end
29 |  MatSetValues(Amat, 1, row, line[idx].size(), cols, vals, INSERT_VALUES);
30 end
```

		MATMPIAIJMKL		MATMPISBAIJ		
run ID	**matrix size**	**time** $[s]$	**ratio**	**time** $[s]$	**ratio**	**mat_ratio**
1	15.625	5.05e-03	–	3.10e-03	–	1.63
2	125.000	4.18e-02	8.28	2.51e-02	8.09	1.67
3	1.000.000	3.37e-01	8.07	2.00e-01	7.96	1.69
4	8.000.000	2.72	8.06	1.63	8.17	1.67

Table 2: LDU2CSR conversion time for a 3D poisson problem over a uniform cartesian mesh, using three different *Mat* types.

The performance of the algorithm have been tested on a 3D Poisson problem (representative of the pressure solver solved in the Navier-Stokes equations) over a uniform mesh, see Table 2 and Fig. 8. The matrix size, equal to the cube of the number of rows, is increased by doubling the initial number of rows, up to a value of 8 millions. The average time (**time**) is the time averaged of different tests of the same configuration (run ID), by using two different *Mat* types: `MATMPISBAIJ` and `MATMPIAIJMKL`. The single test runs a kernel code which consists essentially of the matrix conversion. **ratio** is the ratio between the time employed to convert the matrix in the corresponding run ID over the previous one. **mat_ratio** is the ratio between the average time of the two *Mat* type analyzed for the same run ID.
Figure 8 shows that the algorithm scales linearly (with the factor of 8) with run ID from 1 up to 4, for both the *Mat* type analyzed. Moreover, it is possible to gain a speed-up of $\simeq 1.7$ if the symmetry property of the matrix is exploited, when using `MATMPISBAIJ` type in place of `MATMPIAIJMKL` one. For sake of completeness, Fig. 8 reports also the results for the `MATMPIAIJ` type. The trend is overlapping with the `MATMPIAIJMKL` type one.

3.6. The PETSc4FOAM library

The PETSc4FOAM library has been designed to be decoupled from the compilation of OpenFOAM; it is released as a standalone git submodule [25]. The dependencies of the library are obviously OpenFOAM and PETSc. The code structure of the library is shown in Figure 9.
The code is splitted in three main folders: *etc*, *tutorials* and *src*. We collect in *etc* all the utility files, in *src* the source files of the library and in *tutorials* some examples that show how to use it. All the *Allwclean* and *Allwmake* files are required by the wmake build system.

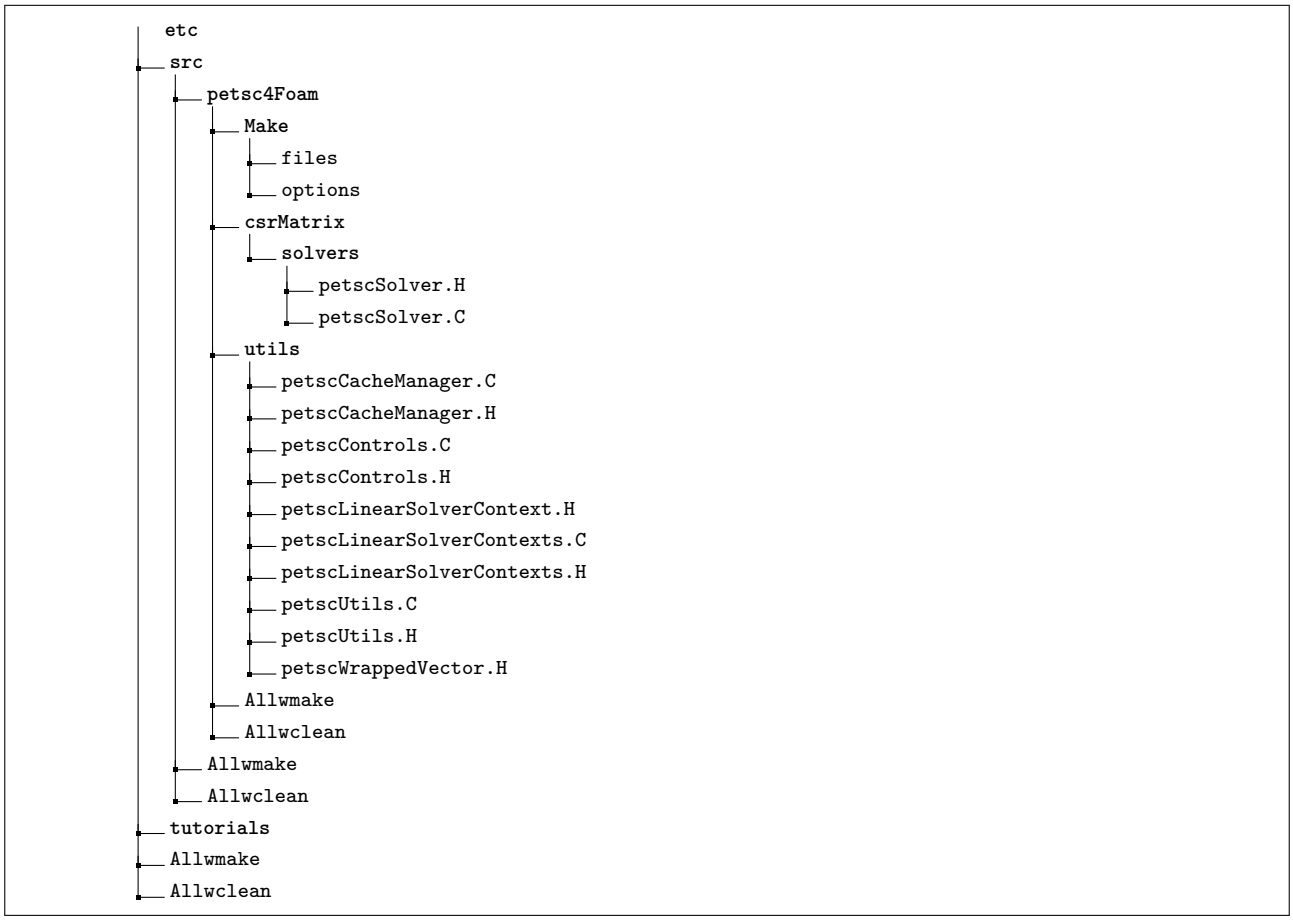

Fig. 9: Directory tree of the PETSc4FOAM library

```
solvers
{
    p
    {
        solver          petsc;
        preconditioner  petsc;

        petsc
        {
            options
            {
                ksp_type cg;
                pc_type bjacobi;
                sub_pc_type icc;
                ksp_cg_single_reduction true;
                mat_type mpiaijmkl;
            }

            caching
            {
                matrixCaching
                {
                    update always;
                }

                preconditionerCaching
                {
                    update always;
                }
            }

        tolerance       1.e-04;
        relTol          0;
        maxIter         3000;
    }

    pFinal
    {
        $p;
        relTol          0;
    }

    U
    {
        solver          PBiCGStab;
        preconditioner  DILU;
        tolerance       0;
        relTol          0;
        maxIter         5;
    }
}
```

Listing 1: Example of a fvSolution with PETSc options

The source files are splitted in three folders, *Make*, *csrMatrix/solvers* and *utils*. In *Make* there are two files needed by the wmake build system. In *csrMatrix/solvers* folder we added a *petscSolver.C* file that contains the implementation of the *petscSolver* class, a C++ wrapper of the petsc KSP solver object. This class, that inherits from *lduMatrix::solver*, is responsible for the following steps:

- Run the LDU2CSR algorithm in order to convert the *LDU matrix* into a *Mat CSR matrix*, if needed
- Set-up a *PC* and *KSP* object according to the user options, if needed
- Wrap the *rhs* and *solution* vector with a *Petsc Vec*
- Run the *KSPSolve* function

In *utils* we collect all the classes that are needed by the *petscSolver* class. In particular:

- *petscControls.C*: a utility class which is responsible to initialize and finalize correctly the PETSc library
- *petscLinearSolverContexts.C*: a utility class which is responsible to keep alive all the PETSc objects needed by *petscSolver.C* between successive iterations
- *petscCacheManager.C*: a manager class to cache the CSR matrix and its preconditioner between successive iterations
- *petscWrappedVector.H*: a utility class to wrap a OpenFOAM vector (e.g rhs) with a PETSc vector using a shallow copy paradigm

- *petscUtils.C*: collection of utility functions useful by all the classes of the PETSc4FOAM library

The user has two viable ways to set-up the PETSc options:

i the PETSc approach by setting the options in a *petscrc* file

ii the OpenFOAM approach by setting them in a *petsc/options* subdictionary of the *fvSolution* file.

All the options, not related to PETSc, are set using the OpenFOAM approach. An example is reported in the code of Listing 1.

caching update

In Listing 1 we have shown an example of usage of the caching algorithm implemented in the library for the solver matrix and its preconditioner. This algorithm stores in the RAM memory the matrix's coefficient and its preconditioner in the CSR format at a given time step for reusing in the subsequent time-steps according to the frequency set by the user. The user can select the cache update frequency among the following choices

- always (default)
- never
- periodic
- adaptive

For the first option "always", the caching update is done at each time step of the time dependent simulation; this is the library's default configuration. In the case of the second option "never", the storing of the matrix's coefficients is done once at the beginning of the time dependent simulation. The coefficients are re-used for all the time steps. In the case of constant coefficients' matrix, no error is introduced and a gain in term of performance is obtained by avoiding to convert and store the matrix at each time step.
The third options "periodic" means that the cache is updated every n steps.
The fourth option "adaptive" is the more complex one; it implements an adaptive procedure that computes the number of iterations after which the cache is updated. It is based on the following formula, proposed in [62]:

$$nsteps = min\left(1\mathrm{e}^{+05}, \frac{T_0}{T} \cdot \frac{1}{|1 - \frac{T_1}{T} + 1\mathrm{e}^{-06}|}\right) \tag{5}$$

where T_0 is the solver time after the matrix cache update, T_1 is the solver time after the first step from the matrix cache update and T is the current solver time.

4. OpenFOAM HPC Benchmark suite

The code repository for the HPC TC [23] is a shared repository with relevant data-sets and information created in order to:

- provide user guides and initial scripts to set-up and run different data-sets on different HPC architectures
- provide to the community an homogeneous term of reference to compare different hardware architectures, software environments, configurations, etc.
- define a common set of metrics/KPI (Key Performance Indicators) to measure performances

4.1. 3-D Lid-driven cavity flow

The first test-case chosen is a 3-D version of the lid driven cavity flow tutorial [63]. This test-case has simple geometry and boundary conditions, involving transient, isothermal, incompressible laminar flow in a three-dimensional box domain. The *icoFoam* solver, which solves the Navier Stokes equations:

$$\begin{gathered} \nabla \cdot \mathbf{u} = 0 \\ \frac{\partial \mathbf{U}}{\partial t} + \nabla \cdot (\mathbf{U}\mathbf{U}) = f - \nabla p + \mu \nabla^2 \mathbf{U}, \end{gathered} \tag{6}$$

using the *PISO* algorithm, is adopted [64,65].

Definition of geometrical and physical properties

In such simple geometry all the boundaries of the square are walls. The top wall moves in the x-direction at the speed of 1 m/s while the other five are stationary.
Three different sizes have been selected: S, M and XL for Small, Medium and eXtra-Large test-case, respectively. Table 3 and Fig. 10 show the geometrical and physical properties of the different test-cases, which have an increasing number of cells: 1 million (m) (S), 8 m (M) and 64 m (XL) of cells, obtained by halving Δx when moving from the smaller to the bigger test-case. The Courant number $Co = \frac{U\Delta t}{\Delta x}$ is kept under stability limit, and it is halved when moving to bigger test-cases. The time step Δt is reduced, proportionally to Co and Δx, by 4 times. The physical time to reach the steady state in laminar flow [1] is $T = 0.5$.
For the results shown in following Sec. 5., the test-case XL is used with a number of iteration equal to 100 to reduce the computational time.

Parameters	**Test-case**		
	S	M	XL
Δx (m)	0.001	0.0005	0.00025
N. of cells tot. (millions)	1	8	64
N. of cells lin.	100	200	400
ν (m^2/s)	0.01	0.01	0.01
d (m)	0.1	0.1	0.1
Co	1	0.5	0.25
Δt $(sec.)$	0.001	0.00025	0.0000625
N. of Reynolds	10	10	10
U (m/s)	1	1	1
n. of iter.	-	-	100

Table 3: Geometrical and physical parameters

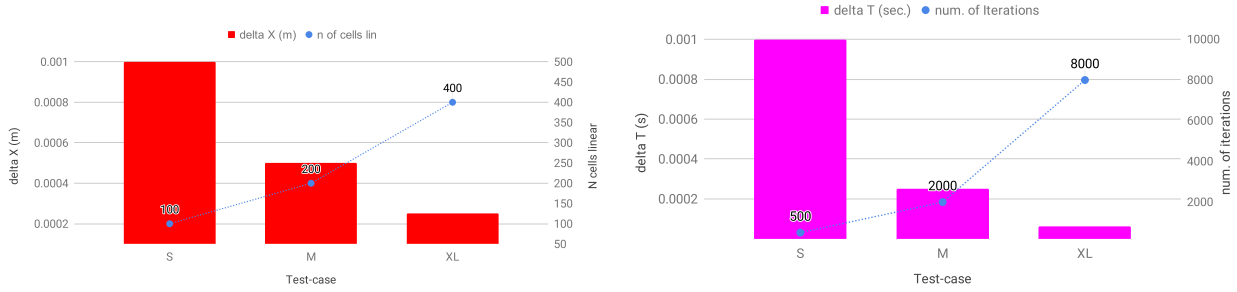

Fig. 10: Geometrical parameters for test-cases of Tab. 3: spatial (left) and temporal (right) parameters.

5. Numerical results

5.1. Methodology

The algorithm implemented in the icoFoam solver consists mainly of two parts, one is the solution of the momentum equation and the other one is the solution of the pressure equation (or continuity equation). The momentum equation is discretized by the Finite Volume method (FVM) using the under-relaxation to ensure the matrix diagonal dominance. This leads to a linear system that can be solved in a high efficient way.
The pressure equation in icoFoam is essentially the discrete form of the Poisson equation and it has been shown by many studies [66] that can be solved efficiently by the conjugate gradient (CG) algorithm if a proper choice of preconditioner is made. Since profiling analysis (see Fig. 4) has shown that most of the computational time is spent during the solution of the pressure equation (around 67%), we focus our effort in the comparison of different preconditioner, keeping fixed the solver/preconditioner pair for the momentum equation. Comparison is performed between the pressure solvers reported in Table 4. FOAM-DIC-PCG is an iterative solver provided by the OpenFOAM package that combines a diagonal-based incomplete-Cholesky preconditioner (DIC) with a Conjugate Gradient solver (PCG). DIC is a common preconditioner in OpenFOAM for its easy configuration

[1] The fluid should pass through the domain 10 times to reach steady state in laminar flow. In this case the flow does not pass through this domain as there is no inlet or outlet, so instead the end time can be set to the time taken for the fluid to travel ten times across the cavity, i.e. 1 s; ... we discover that 0.5 s is sufficient so we shall adopt this value. [63].

Method	Preconditioner	Solver
FOAM-DIC-PCG	Diagonal-based incomplete-Cholesky	Conjugate Gradient
PETSc-ICC-CG	Incomplete-Cholesky	Conjugate Gradient
FOAM-GAMG-PCG	Geometric-algebraic multigrid	Conjugate Gradient
PETSc-AMG-CG	Classic algebraic multigrid (BoomerAMG)	Conjugate Gradient

Table 4: Preconditioner/solver pairs used by OpenFOAM and OpenFOAM-PETSc

and high efficiency.
PETSc-ICC-CG is the PETSc counterpart of the FOAM-DIC-PCG method. PETSc provides a CG implementation that performs both inner products needed in the algorithm with a single `MPIU_Allreduce` call. ICC is the Incomplete Cholesky factorization implementation of PETSc. A variant that does not apply the Manteuffel shift is used [67].
FOAM-GAMG-PCG is an iterative solver implemented in OpenFOAM that uses a Conjugate Gradient (CG) accelerated by a generalized geometric-algebraic multigrid (GAMG) method that supports both geometrical and algebraic multigrid. Its parameters are set in Table 5 and chosen according to our experience.
PETSc-AMG-CG is the PETSc counterpart of the FOAM-GAMG-PCG method. In this solver we use an algebraic multigrid method, named BoomerAMG, provided by the third-party library of PETSc named Hypre. All parameters of BoomerAMG are set in Table 5. These parameter values have been chosen empirically, after a series of test aiming to minimize the single iteration time and maximizing the scalability in the range 8-128 nodes.
PETSc-AMG-CG + caching is the same solver described above but the matrix is converted only once at the beginning and cached together with the preconditioner for all the time-steps. In this particular case of constant-coefficients matrix, the numerical solution is equivalent to the case without caching.
The iterative methods can be run in two ways, by fixing the number of iterations (FIXEDITER) or the same

`GAMG`		`BoomerAMG`	
Parameter	**Value**	**Parameter**	**Value**
Agglomeration	faceAreaPair	max_iter	1
Smoother	GaussSeidel	strong_threshold	0.7
nPreSweeps	0	grid_sweeps_up	1
nPostSweeps	2	grid_sweeps_down	1
nFinestSweeps	2	agg_nl	2
mergeLevels	1	agg_num_paths	1
cacheAgglomeration	on	max_levels	25
agglomerator	faceAreaPair	coarsen_type	HMIS
processorAgglomerator	none	interp_type	ext+i
nCellsInCoarsestLevel	512	P_max	2
		truncfactor	0.2

Table 5: Parameters of GAMG (left) and BoomerAMG (right) preconditioners used, respectively, in FOAM-GAMG-PCG and FOAM-AMG-PCG.

residual norm (FIXEDNORM).
In the first case, FIXEDITER, the computational load is fixed; two solving methods with a different rate of convergence will exit with a different norm. This case is useful for comparing different hardware configurations by keeping constant the computational load.
In the second case, FIXEDNORM, the exit norm is the same, whereas the number of iterations is generally different. Unfortunately, the norm implemented in the standard release of OpenFOAM is not implemented in PETSc preventing from using the second criterion.
The residual norm implemented in OpenFOAM is a L1-norm scaled by the following factor:

$$normFactor = \|\mathbf{Ax} - \sum_j \mathbf{A}_j \bar{x}\|_1 + \|\mathbf{b} - \sum_j \mathbf{A}_j \bar{x}\|_1 \tag{7}$$

In formula we have

$$\|\mathbf{b} - \mathbf{Ax}\|_{foam\_1} = \frac{\|\mathbf{b} - \mathbf{Ax}\|_1}{\|\mathbf{Ax} - \sum_j \mathbf{A}_j \bar{x}\|_1 + \|\mathbf{b} - \sum_j \mathbf{A}_j \bar{x}\|_1} \tag{8}$$

The FOAM-L1 norm has been implemented in the PETSc4FOAM package as a new convergence criteria. A new function, named *foamKSPConverge*, computes the custom residual norm as in eq. (8). This function is handled in PETSc as an argument of the *KSPSetConvergenceTest* function. [2]
For the results shown in Sec. 5., all the tests are performed using the FIXEDNORM criterion, with a exit norm value of $1.0e^{-04}$.

5.2. Platform and software stack

All of our experiments are tested on the Tier-1 GALILEO machine hosted by CINECA, the italian super computing center. The cluster is made of 1022 compute nodes without GPUs and 60 compute nodes equipped with 1 nVidia K80 GPUs that are connected via an Intel OmniPath high-performance network of 140 GB/s. Each compute node contains two sockets made of 18 2.3 GHz Intel Xeon E5-2697 v4 (Broadwell) CPUs and 128 GB of memory. The operating system of the cluster is the CentOS 7.4
The software stack used is made of OpenFOAM 1906, PETSc 3.12 and Hypre 2.17. All the software are compiled with Intel 2019 and linked to Intel-MPI 2019.

5.3. Performance and scalability results

Firstly, we investigate the strong scalability of the PISO [3] algorithm keeping fixed the preconditioner/solver pair of the momentum equation and using the solvers for the pressure equation described in Table 4. The mesh size is kept constant (case XL of Table 3) and the process number ranges from 288 (8 nodes) to 4608 cores (128 nodes). At the same time, the number of cells per cores varies from $\approx 220,000$ down to $\approx 13,000$. Let's consider that the value of $\approx 20,000$ cells per core is considered the minimum threshold to get good scalability when running OpenFOAM in parallel mode.
Total (wall clock) time results are shown in Table 6 and in Figure 11. The average time per time step is shown is Figure 12. Speed-up is shown in Figure 13. Table 7 reports the PCSetUp time, the single iteration time, the number of iterations and the total time using 8 nodes (288 cores).
The results show clearly that using few nodes (e.g. 8 nodes) the multigrid preconditioners outperform the incomplete factorization ones. Table 7 reports that, even if the cost of the set-up of the Multigrid preconditioner is much higher respect to Incomplete Factorizations, both the single iteration time and the number of iterations to converge for the multi-grid preconditioners are lower. The final result is a faster preconditioner/solver combination for the PISO-FOAM-GAMG-PCG and PISO-PETSc-AMG-CG in the 8 nodes case. A further improvement is obtained with the caching (i.e. PISO-PETSc-AMG-CG + caching)) due to the null time for the PCSetUp in this case.
The same result does not hold using much more nodes. The CG preconditioned by the Incomplete Factorizations scales super-linearly with respect to the Multigrid, as shown in Figure 13, while PETSc-AMG-CG has an excellent linear scalability up to 64 nodes (2304 cores) and FOAM-GAMG-PCG only up to 16 nodes (576 cores). Therefore, this difference in terms of total time between the CG preconditioned by the Incomplete Factorizations and the two multi-grid methods is less pronounced at 32 nodes with respect to 8 nodes. The final result is that PISO-PETSc-AMG-CG is 6 times faster at 8 nodes with respect to PISO-FOAM-DIC-PCG but only twice at 128 nodes. Better results can be achieved if the caching is turned on in the PISO-PETSc-AMG-CG solver, both in terms of scalability and performance.
From this analysis it is clear that PISO-PETSc-AMG-CG is the best solver. Even if it is not the faster solver using 8 nodes [4], its excellent scalability and rate of convergence outperforms all the other solvers in all the other cases (from 16 to 128 nodes). As we already said, PETSc-AMG-CG is the pressure solver of the PISO algorithm. Figure 14 shows how much time in percentual is spent in the pressure solver respect to the momentum solver and LDU2CSR conversion in the range of $8 - 128$ nodes. The plot shows that the overhead of the conversion is around 2% and decreases slightly by increasing the number of nodes. The ratio between pressure solver and momentum solver is not constant and decreases from 79% to 55% when moving from 8 to 64 nodes. In particular we found a bigger jump moving from 64 to 128 nodes.
Figure 15 shows the average time per time step of the PETSc functions PCSetUp, PCApply and KSPSolve in the interval of $8 - 128$ nodes. The plot shows the same trend of Figure 11, that is an excellent scalability up to 64 nodes. The worsening of the scalability using 128 nodes is justified by a very low number of cells per core, below the threshold of 20,000 cells per processor (and as consequence a very little average time per time step to be measured). We are more than confident that an excellent scalability up to ten of thousands of core can be achieved with this solver set-up if a bigger case is used, keeping the number of cells per core always above the threshold of 20,000 cells per processor. Moreover, the add-on of a PETSc solver for the momentum equations has to be investigated, and it is expected a further time reduction (see Fig. 14).

Remark 1 *In the simulations reported in this work, the computational set-up used is at full computational density, that is using the whole cores availables per node (full populated node). In other words, we performed strong scaling test by using a number of MPI processes per node equal to the number of cores per node. This*

[2] PETSc implements the NORM_1 but it is not available among the KSP supported norms. For the tests we used a patched version that enables the NORM_1 in the KSP solver convergence criteria.

[3] In all the simulations we use one orthogonal correction. In other words, one PISO iteration is made of three velocity and two pressure linear systems to be solved.

[4] Better performance can be achieved if the parameters of Table 5 are tuned for the 8 nodes case

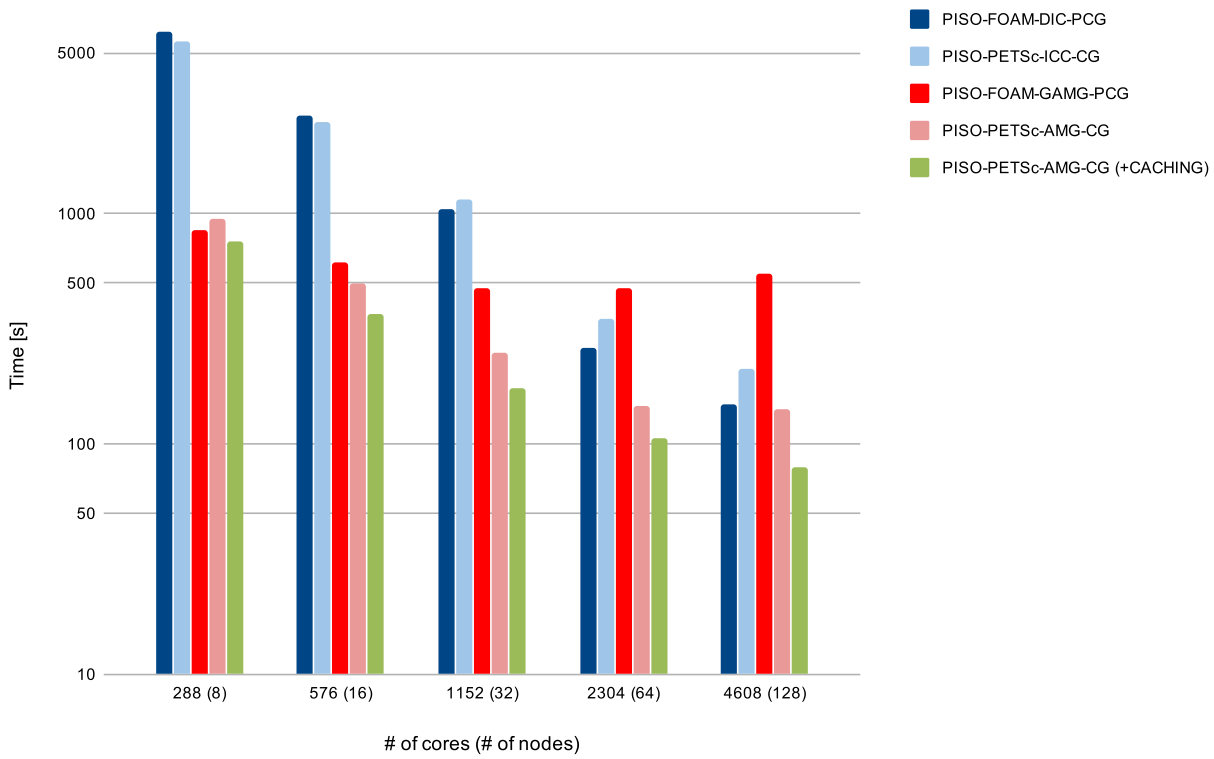

Fig. 11: Total time for solving PISO with different Linear Algebra solvers

configuration is adequate for tests but not the best one for production runs, being such configuration highly memory-bounded.

In fact, Fig. 16 reports the Bound in percentage[5] *and the memory bandwidth versus the computational density (i.e. number of cores per node) for the FOAM-DIC-PCG configuration. It is observed that the average and the peak memory bandwidth slightly decreases passing from the full to half populated node configuration. The peak memory bandwidth is near to the theoretically maximum value. A dramatically decrease of these quantities occurs in the quarter of a node configuration. The bound quantity goes to zero and the code switches from a MEMORY BOUND to a MPI-BOUND configuration.*

	Solution time [s] with different number of cores				
Solver	288 (8)	576 (16)	1152 (32)	2304 (64)	4608 (128)
PISO-FOAM-DIC-PCG	6,184.09	2,686.74	1,040.45	261.19	**147.82**
PISO-PETSc-ICC-CG	5,616.21	2,511.51	1,149.15	351.31	**212.3**
PISO-FOAM-GAMG-PCG	854.58	611.8	**473.88**	477.75	553.27
PISO-PETSc-AMG-CG	952.29	498.82	248.51	146.01	**140.97**
PISO-PETSc-AMG-CG (+CACHING)	755.03	367.42	173.97	106.62	**79.40**

Table 6: Time for solving PISO with different Linear Algebra solvers

[5]The metric represents the percentage of elapsed time spent heavily utilizing system bandwidth, from Intel Vtune profiler [68]

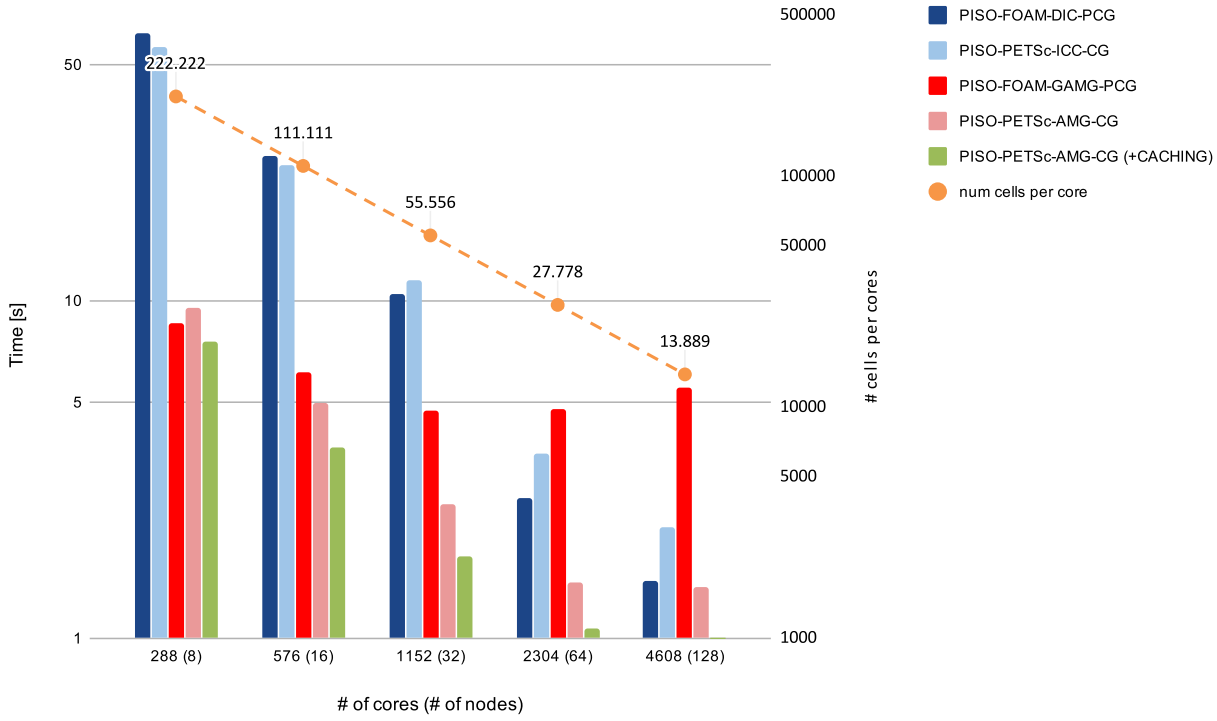

Fig. 12: Average time for solving PISO with different Linear Algebra solvers

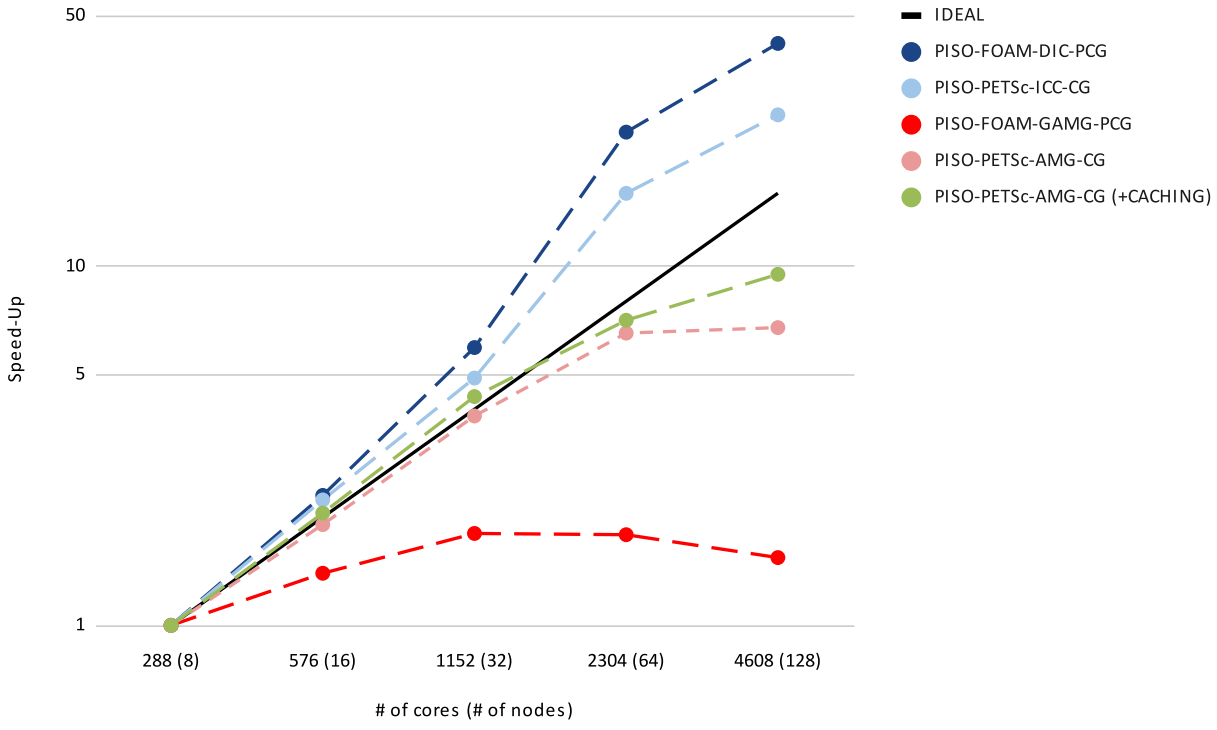

Fig. 13: Speed up of Linear Algebra solvers described in Table 4

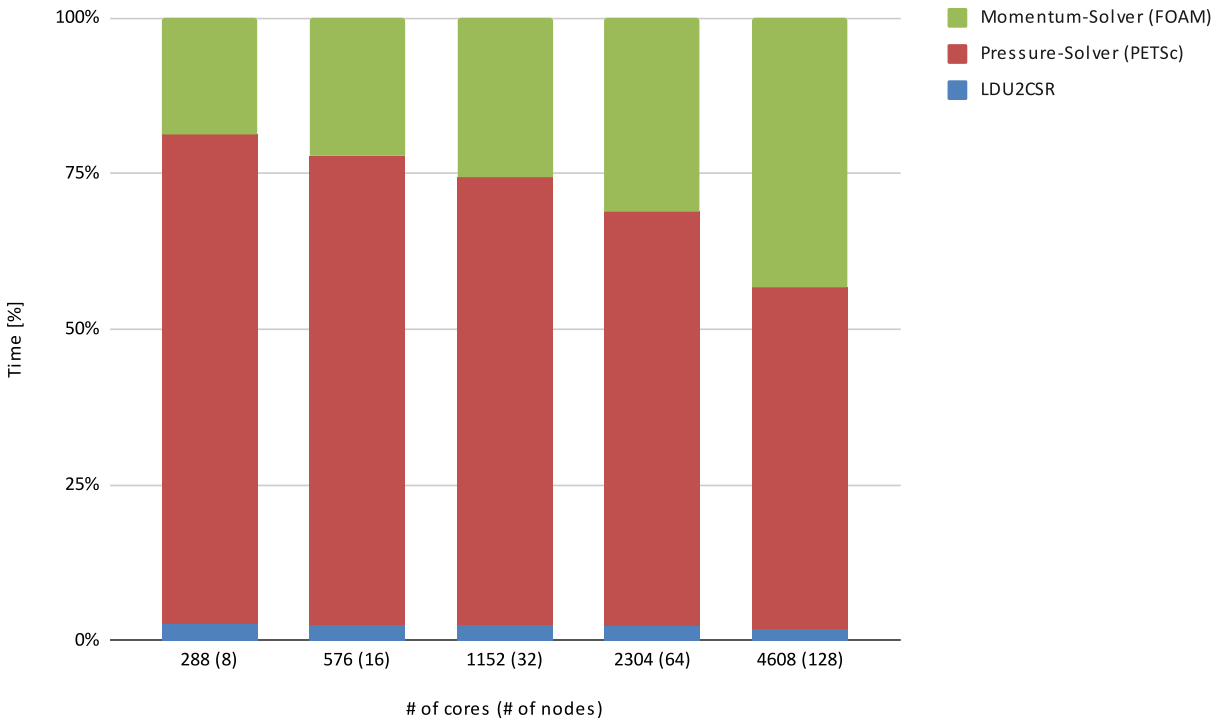

Fig. 14: PETSc-AMG-CG: total time repartition for solving PISO

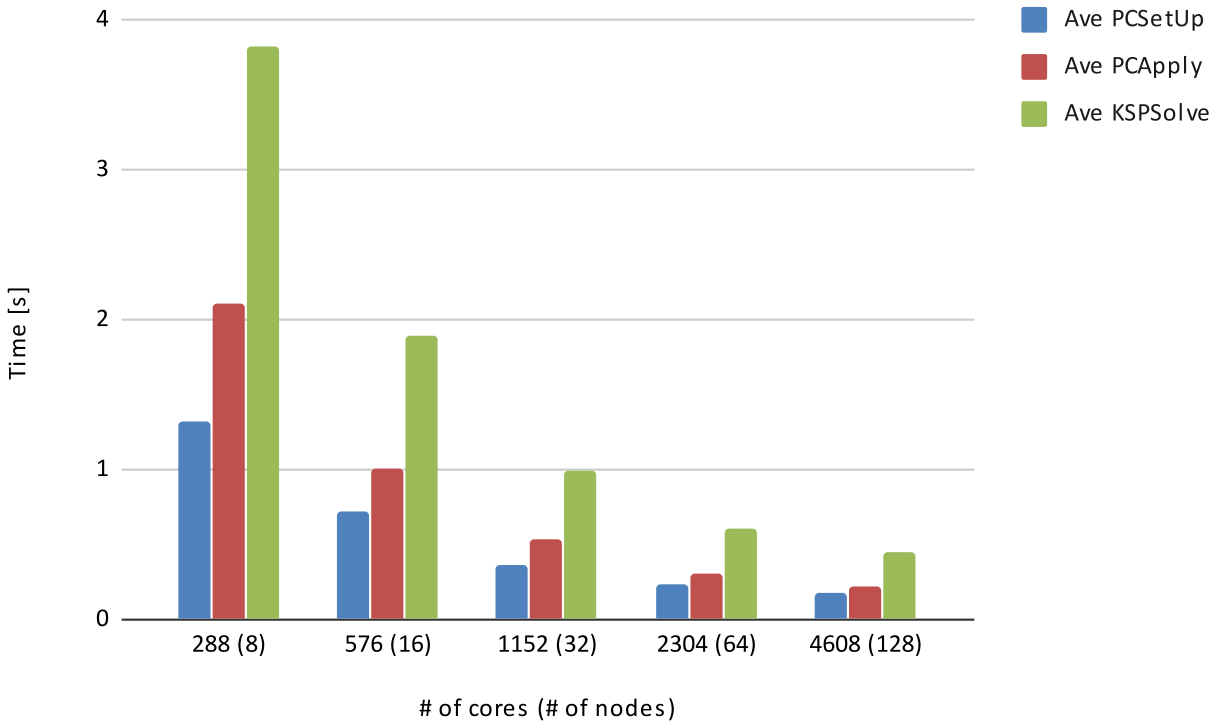

Fig. 15: PETSc-AMG-CG: average time for PCSetUp, PCApply and KSPSolve functions execution

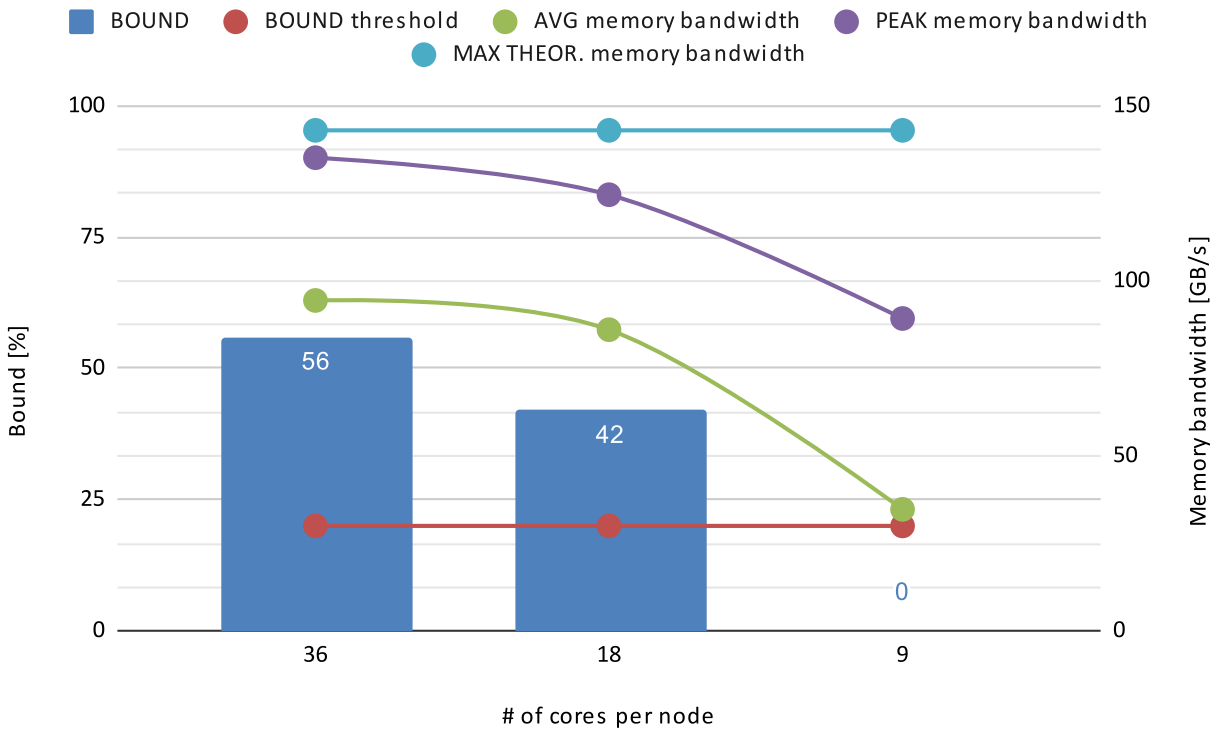

Fig. 16: FOAM-DIC-PCG: memory bandwidth and bound variation using a full, half and a quarter of a node and keeping fixed the problem size

Solver	PCSetUp Time [s]	PISO Iteration Time [s]	CG Iterations	Total Time [s]
PISO-FOAM-DIC-PCG	N.A.	41.27	603	6,184.09
PISO-PETSc-ICC-CG	0.018	38.14	603	5,616.21
PISO-FOAM-GAMG-PCG	N.A.	13.32	13	854.58
PISO-PETSc-AMG-CG	1.30	11.99	35	952.29
PISO-PETSc-AMG-CG (+CACHING)	0	10.13	34	755.03

Table 7: Preconditioner set-up time, single PISO iteration time, CG iterations (at the second PISO iteration) and total time using 288 cores (8 nodes)

6. Conclusion

This work has presented a direct approach to improve the performance of OpenFOAM in a HPC environment by reducing the time for solving linear equation systems. Specifically, a library *PETSc4FOAM* has been developed to use PETSc, and other linear algebra solver packages, as external libraries into the OpenFOAM framework. The *PETSc4FOAM* library relies upon a `ldu2csr` converter, to pass from the *LDU* format to *CSR*. The performances of such algorithm have been tested using two different *Mat* type of PETSc (`MATMPISBAIJ` and `MATMPIAIJMKL`). It scales linearly by increasing the matrix size, and the overhead of the conversion in the reported tests is around 2% decreasing slightly when increasing the number of nodes.
The test-case investigated is a 3-D version of the lid-driven cavity flow with 64 millions of cells (test-case XL of Tab. 3). The `icoFoam` solver has been used for the numerical tests. The pressure solver is the most computational intensive part of the code (typically around 70% of the total time). Our numerical tests focus on the comparison of different preconditioners, keeping fixed the solver/preconditioner pair for the momentum equations. The iterative methods have been run in the FIXEDNORM case, that is by using the same exit norm for the two different class of iterative methods.
The results show clearly that using few nodes (e.g. 8 nodes) the multi-grid preconditioners outperform the incomplete factorization ones. In fact, the PISO-PETSc-AMG-CG is 6 times faster at 8 nodes (64 cores) with respect to PISO-FOAM-DIC-PCG, but only twice at 128 nodes (4608 cores). The reason is that the CG preconditioned by the Incomplete Factorizations scales super-linearly with respect to the Multi-grid in the full-populated node configuration. Conversely, the FOAM-GAMG-PCG scales well only up to 16 nodes (576 cores) while PETSc-AMG-CG has an excellent linear scalability up to 64 nodes (2304 cores). Better results can be achieved if the caching is turned on in the PISO-PETSc-AMG-CG solver, both in terms of scalability and performance.
From this analysis, we can conclude that the PISO-PETSc-AMG-CG+caching is the best solver, due to its excellent scalability and rate of convergence.

Acknowledgments

This work was financially supported by the PRACE project funded in part by the EU's Horizon 2020 Research and Innovation programme (2014-2020) under grant agreement 823767. The work was achieved using the PRACE Research Infrastructure resources at the GALILEO (CINECA) supercomputer.

References

1. S. Bnà, I. Spisso, G. Rossi, M. Olesen, HPC Performance improvements for OpenFOAM linear solvers, Tech. rep., 7th ESI OpenFOAM Conference, Berlin (2019).
URL `http://tiny.cc/ry47kz`

2. I. Spisso, G. Amati, V. Ruggiero, C. Fiorina, Porting, optimization and bottleneck of OpenFOAM in KNL environment, Tech. rep., Intel eXtreme Performance Users Group (IXPUG) (2018).
URL `http://tiny.cc/ry47kz`

3. M. Culpo, Current Bottlenecks in the Scalability of OpenFOAM on Massively Parallel Clusters, Tech. rep., PRACE White paper (2012).
URL `http://tiny.cc/7b57kz`

4. November 2019 — TOP500 Supercomputer Sites (2019).
URL `https://www.top500.org/lists/2019/11/`

5. T. Ponweiser, P. Stadelmeyer, T. Karásec, Fluid-Structure Simulations with OpenFOAM for Aircraft Designs, PRACE White paper (5 2014). `doi:10.5281/ZENODO.823039`.

6. P. Van Phuc, Y. Inoue, A. Azami, M. Uchiyama, S. Chiba, EVALUATION OF MPI OPTIMIZATION OF C++ CFD CODE ON THE K COMPUTER, SIG Technical Reports HPC-151 (No.19 2015/10/01) (2015).

7. P. Van Phuc, Large Scale Transient CFD Simulations for Buildings using OpenFOAM on a World's Top-class Supercomputer, Tech. rep., 4th ESI OpenFOAM Conference (2016).
URL https://www.esi-group.com/it/resources/abstract-keynote-shimizu-vanphuc-buildingsimulation

8. OpenFOAM® v1606+: New Parallel Functionality (2016).
URL https://www.openfoam.com/releases/openfoam-v1606+/parallel.php

9. White Papers - Application Scalability - PRACE (2020).
URL https://prace-ri.eu/training-support/technical-documentation/white-papers/application-scalability/#CFD

10. M. Moyles, Nash Peter, I. Girotto, Performance Analysis of Fluid-Structure Interactions using OpenFOAM, Tech. rep., PRACE White paper (2012).
URL http://www.prace-project.eu/IMG/pdf/Performance_Analysis_of_Fluid-Structure_Interactions_using_OpenFOAM.pdf

11. P. Dagna, J. Hertzer, Evaluation of Multi-threaded OpenFOAM Hybridization for Massively Parallel Architectures, Tech. rep., PRACE White paper (2013).
URL http://tiny.cc/a84flz

12. O. Rivera, K. Fürlinger, Parallel Aspects of OpenFOAM with Large Eddy Simulations, 2011 IEEE International Conference on High Performance Computing and Communications (2011) 389–396.

13. M. Manguoglu, Parallel solution of sparse linear systems in OpenFOAM, Tech. rep., PRACE White paper (2017).
URL https://prace-ri.eu/wp-content/uploads/WP-A-General-Sparse-Sparse-Linear-System-Solver-and-Its pdf

14. Intel® Xeon Phi™ Processor 7285 (16GB, 1.3 GHz, 68 Core) Product Specifications (2017).
URL https://ark.intel.com/content/www/us/en/ark/products/128691/intel-xeon-phi-processor-7285-16gb-1-3-ghz-68-core.html

15. OpenFOAM-Intel/libhbm at master · OpenFOAM/OpenFOAM-Intel · GitHub (2020).
URL https://github.com/OpenFOAM/OpenFOAM-Intel/tree/master/libhbm

16. S. Posey, F. Pariente, Opportunities for GPU Acceleration of OpenFOAM, Tech. rep., 7th ESI OpenFOAM Conference, Berlin (2019).
URL http://tiny.cc/4moalz

17. NVIDIA V100 — NVIDIA (2020).
URL https://www.nvidia.com/en-us/data-center/v100/

18. RapidCFD GPU - OpenFOAM® on GPU (2020).
URL https://sim-flow.com/rapid-cfd-gpu/

19. What is SpeedIT? – Vratis (2020).
URL http://vratis.com/what-is-speedit/

20. F. Ghioldi, F. Piscaglia, Fast algorithms for highly underexpanded reactive spray simulations, Ph.D. thesis, Dept. of Aerospace Science and Technology (DAER), Politecnico di Milano, Italy (2019).
URL https://www.politesi.polimi.it/handle/10589/146075

21. M. Taouil, A Hardware Accelerator for the OpenFoam Sparse Matrix-Vector Product, Ph.D. thesis, TUDelft (2009).
URL http://resolver.tudelft.nl/uuid:ce583533-45ea-4237-b18d-fe31272ea1ee

22. byteLAKE's CFD Suite (byteLAKE) (2020).
URL https://www.bytelake.com/en/bytelake-products/bytelakes-cfd-suite/

23. Committees / HPC · GitLab (2020).
URL https://develop.openfoam.com/committees/hpc

24. High Performance Computing (HPC) Technical Committee - OpenFOAM Wiki (2020).
URL https://wiki.openfoam.com/High_Performance_Computing_(HPC)_Technical_Committee

25. Community / external-solver · GitLab (2020).
URL https://develop.openfoam.com/Community/external-solver

26. G. Chen, Q. Xiong, P. J. Morris, E. G. Paterson, A. Sergeev, Y.-C. Wang, OpenFOAM for Computational Fluid Dynamics, Notices of the American Mathematical Society 61 (4) (2014). doi:10.1090/noti1095.

27. Wiklipedia.org, OpenFOAM (2020).
URL https://en.wikipedia.org/wiki/OpenFOAM

28. CFD Direct — The Architects of OpenFOAM (2020).
URL https://cfd.direct/

29. OpenFOAM® - Official home of The Open Source Computational Fluid Dynamics (CFD) Toolbox (2020).
URL https://www.openfoam.com/

30. foam-extend download — SourceForge.net (2020).
URL https://sourceforge.net/projects/foam-extend/

31. OpenFOAM® History (2020).
URL https://www.openfoam.com/history/

32. E. S. Raymond, The cathedral & the bazaar : musings on Linux and open source by an accidental revolutionary, O'Reilly, 1999.

33. ESI-OpenCFD, 1 Introduction (2020).
URL http://tiny.cc/hx57kz

34. ESI-OpenCFD, User Guide: Solution and algorithm control (2020).
URL http://tiny.cc/z657kz

35. H. Jasak, Handling Parallelisation in OpenFOAM, Tech. rep., Cyprus Advanced HPC Workshop Winter (2012).

36. H. Jasak, HPC Deployment of OpenFOAM in an Industrial Setting, Tech. rep., PRACE, Stockholm (2011).
URL https://prace-ri.eu/wp-content/uploads/IndSem2011_Jasak.pdf

37. OpenFOAM® Technical Committees (2020).
URL https://www.openfoam.com/governance/technical-committees.php

38. Community / adiosFoam · GitLab (2020).
URL https://develop.openfoam.com/Community/adiosfoam

39. Satish Balay and Shrirang Abhyankar and Mark F. Adams and Jed Brown and Peter Brune, and Kris Buschelman and Lisandro Dalcin and Alp Dener and Victor Eijkhout and William D. Gropp, and Dmitry Karpeyev and Dinesh Kaushik and Matthew G. Knepley and Dave A. May and Lois Curfman McInnes, and Richard Tran Mills and Todd Munson and Karl Rupp and Patrick Sanan, and Barry F. Smith and Stefano Zampini and Hong Zhang and Hong Zhang, PETSc Web page (2019).
URL https://www.mcs.anl.gov/petsc

40. A. Duran, M. S. Celebi, S. Piskin, M. Tuncel, Scalability of OpenFOAM for bio-medical flow simulations, Journal of Supercomputing 71 (3) (2015) 938–951. doi:10.1007/s11227-014-1344-1.
URL http://link.springer.com/10.1007/s11227-014-1344-1

41. H. Li, X. Xu, M. Wang, C. Li, X. Ren, X. Yang, Insertion of PETSc in the OpenFOAM framework, ACM Transactions on Modeling and Performance Evaluation of Computing Systems 2 (3) (8 2017). doi:10.1145/3098821.
URL http://tiny.cc/7d57kz

42. X. S. Li, J. W. Demmel, SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems, ACM Transactions on Mathematical Software 29 (2) (2003) 110–140. doi:10.1145/779359.779361.

43. HYPRE: Scalable Linear Solvers and Multigrid Methods — Computing (2020).
URL https://computing.llnl.gov/projects/hypre-scalable-linear-solvers-multigrid-methods

44. W. Bangerth, R. Hartmann, G. Kanschat, deal.II - a General Purpose ObjectOriented Finite Element Library, ACM Trans. Math. Softw. 33 (4) (2007) 24/1 – 24/27.

45. G. R. Mirams, C. J. Arthurs, M. O. Bernabeu, R. Bordas, J. Cooper, A. Corrias, Y. Davit, S. J. Dunn, A. G. Fletcher, D. G. Harvey, M. E. Marsh, J. M. Osborne, P. Pathmanathan, J. Pitt-Francis, J. Southern, N. Zemzemi, D. J. Gavaghan, Chaste: An Open Source C++ Library for Computational Physiology and Biology, PLoS Computational Biology 9 (3) (2013). doi:10.1371/journal.pcbi.1002970.

46. C. J. Permann, D. R. Gaston, D. Andrš, R. W. Carlsen, F. Kong, A. D. Lindsay, J. M. Miller, J. W. Peterson, A. E. Slaughter, R. H. Stogner, R. C. Martineau, {MOOSE}: Enabling massively parallel multiphysics simulation, SoftwareX 11 (2020) 100430. doi:https://doi.org/10.1016/j.softx.2020.100430.
URL http://www.sciencedirect.com/science/article/pii/S2352711019302973

47. D. Hathorn, Y. Wu, Z. Chen, TOUGH2-PETSc: A Parallel Solver for TOUGH2, in: 2014 15th International Conference on Parallel and Distributed Computing, Applications and Technologies, 2014, pp. 174–179. doi:10.1109/PDCAT.2014.35.

48. S. Arens, H. Dierckx, A. V. Panfilov, GEMS: A Fully Integrated PETSc-Based Solver for Coupled Cardiac Electromechanics and Bidomain Simulations, Frontiers in Physiology 9 (2018) 1431. doi:10.3389/fphys.2018.01431.
URL https://www.frontiersin.org/article/10.3389/fphys.2018.01431

49. P. C. Lichtner, G. E. Hammond, C. Lu, S. Karra, G. Bisht, B. Andre, R. Mills, J. Kumar, PFLOTRAN User Manual: A Massively Parallel Reactive Flow and Transport Model for Describing Surface and Subsurface Processes, {} United States (2015). doi:10.2172/1168703.

50. F. Bassi, L. Botti, A. Colombo, A. Crivellini, A. Ghidoni, A. Nigro, S. Rebay, Time Integration in the Discontinuous Galerkin Code MIGALE - Unsteady Problems, in: N. Kroll, C. Hirsch, F. Bassi, C. Johnston, K. Hillewaert (Eds.), Notes on Numerical Fluid Mechanics and Multidisciplinary Design, Vol. 128, Springer, Cham, 2015, pp. 179–204. doi:10.1007/978-3-319-12886-3{_}10.

51. H. V. N. Paul R. Eller, Jing-Ru C. Cheng, R. S. Maier, Improving parallel performance of large-scale watershed simulations, Procedia Computer Science 1 (2012) 801–808.

52. L. D'Amore, A. Murli, V. Boccia, L. Carracciuolo, Insertion of PETSc in the NEMO stack software driving NEMO towards exascale computing, in: 2014 International Conference on High Performance Computing Simulation (HPCS), 2014, pp. 724–731. doi:10.1109/HPCSim.2014.6903761.

53. D. May, M. G. Knepley, M. Gurnis, CitcomSX: Robust preconditioning in CitcomS via PETSc, in: AGU Fall Meeting Abstracts, Vol. 2009, 2009, pp. P31A–1241.

54. PETSc: Features: GPU support (2020).
URL https://www.mcs.anl.gov/petsc/features/gpus.html

55. Satish Balay and Shrirang Abhyankar and Mark F. Adams and Jed Brown and Peter Brune, and Kris Buschelman and Lisandro Dalcin and Alp Dener and Victor Eijkhout and William D. Gropp, and Dmitry Karpeyev and Dinesh Kaushik and Matthew G. Knepley and Dave A. May and Lois Curfman McInnes, and Richard Tran Mills and Todd Munson and Karl Rupp and Patrick Sanan, and Barry F. Smith and Stefano Zampini and Hong Zhang and Hong Zhang, PETSc Users Manual, Tech. rep., Argonne National Laboratory (2019).
URL https://www.mcs.anl.gov/petsc

56. M. Benzi, Preconditioning techniques for large linear systems: A survey, Journal of Computational Physics 182 (2) (2002) 418–477. doi:10.1006/jcph.2002.7176.

57. C. Janna, Preconditioning techniques for large linear systems, Tech. rep., School on Numerical Methods for Parallel CFD, Rome (2019).
URL https://learn.cineca.it/pluginfile.php/9075/mod_resource/content/1/CFD10_Janna_UniPadova.pdf

58. L. N. L. N. Trefethen, D. Bau, Numerical linear algebra, Society for Industrial and Applied Mathematics, 1997.

59. K. Atkinson, W. Han, Theoretical numerical analysis: A functional analysis framework, Vol. 39, Springer Verlag, 2009.

60. H. Anzt, S. Tomov, J. Dongarra, Implementing a Sparse Matrix Vector Product for the SELL-C/SELL-C-σ formats on NVIDIA GPUs, Tech. rep., University of Tennessee (2014).
URL http://www.cise.ufl.edu/research/

61. Mat Manual Pages (2020).
URL https://www.mcs.anl.gov/petsc/petsc-current/docs/manualpages/Mat/index.html

62. C. Fernandes, V. Vukčević, T. Uroić, R. Simoes, O. S. Carneiro, H. Jasak, J. M. Nóbrega, A coupled finite volume flow solver for the solution of incompressible viscoelastic flows, Journal of Non-Newtonian Fluid Mechanics 265 (2019) 99–115. doi:10.1016/j.jnnfm.2019.01.006.

63. Lid-driven cavity flow (2020).
URL https://www.openfoam.com/documentation/tutorial-guide/tutorialse2.php

64. IcoFoam - OpenFOAMWiki (2020).
URL https://openfoamwiki.net/index.php/IcoFoam

65. OpenFOAM: User Guide: icoFoam (2020).
URL http://tiny.cc/9h57kz

66. Y. Saad, Iterative Methods for Sparse Linear Systems Second Edition, 2nd Edition, Society for Industrial and Applied Mathematics, USA, 2003.

67. PCICC (2020).
URL https://www.mcs.anl.gov/petsc/petsc-current/docs/manualpages/PC/PCICC.html

68. Intel® VTune™ Profiler — Intel® Software (2020).
URL https://software.intel.com/en-us/vtune