Computers and Mathematics with Applications 74 (2017) 74–88

Contents lists available at ScienceDirect

Computers and Mathematics with Applications

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

ParMooN—A modernized program package based on mapped finite elements

Ulrich Wilbrandt [a], Clemens Bartsch [a], Naveed Ahmed [a], Najib Alia [a], Felix Anker [a], Laura Blank [a], Alfonso Caiazzo [a], Sashikumaar Ganesan [b], Swetlana Giere [a], Gunar Matthies [c], Raviteja Meesala [b], Abdus Shamim [b], Jagannath Venkatesan [b], Volker John [a,d,*]

[a] Weierstrass Institute for Applied Analysis and Stochastics, Leibniz Institute in Forschungsverbund Berlin e.V. (WIAS), Mohrenstr. 39, 10117 Berlin, Germany

[b] Department of Computational and Data Sciences, Indian Institute of Science, Bangalore - 560012, India

[c] Department of Mathematics, Institute of Numerical Mathematics, TU Dresden, 01062 Dresden, Germany

[d] Department of Mathematics and Computer Science, Free University of Berlin, Arnimallee 6, 14195 Berlin, Germany

ARTICLE INFO

Article history:
Available online 16 January 2017

Keywords:
Mapped finite elements
Geometric multigrid method
Parallelization

ABSTRACT

ParMooN is a program package for the numerical solution of elliptic and parabolic partial differential equations. It inherits the distinct features of its predecessor MooNMD (John and Matthies, 2004): strict decoupling of geometry and finite element spaces, implementation of mapped finite elements as their definition can be found in textbooks, and a geometric multigrid preconditioner with the option to use different finite element spaces on different levels of the multigrid hierarchy. After having presented some thoughts about in-house research codes, this paper focuses on aspects of the parallelization for a distributed memory environment, which is the main novelty of ParMooN. Numerical studies, performed on compute servers, assess the efficiency of the parallelized geometric multigrid preconditioner in comparison with some parallel solvers that are available in the library PETSc. The results of these studies give a first indication whether the cumbersome implementation of the parallelized geometric multigrid method was worthwhile or not.

1. Introduction

MooNMD, a C++ program package for the numerical solution of elliptic and parabolic partial differential equations based on mapped finite elements, is described in [1]. A modernized version of this package, called ParMooN, has recently been developed to be used as a research code in the future.

The core of MooNMD was designed more than 15 years ago and this code has been successfully used in many scientific studies. There are almost 90 research articles citing MooNMD via [1], see [2]. Recent advances in computing hardware and

* Corresponding author at: Weierstrass Institute for Applied Analysis and Stochastics, Leibniz Institute in Forschungsverbund Berlin e. V. (WIAS), Mohrenstr. 39, 10117 Berlin, Germany.

E-mail addresses: ulrich.wilbrandt@wias-berlin.de (U. Wilbrandt), clemens.bartsch@wias-berlin.de (C. Bartsch), naveed.ahmed@wias-berlin.de (N. Ahmed), najib.alia@wias-berlin.de (N. Alia), felix.anker@wias-berlin.de (F. Anker), laura.blank@wias-berlin.de (L. Blank), alfonso.caiazzo@wias-berlin.de (A. Caiazzo), sashi@cds.iisc.ac.in (S. Ganesan), swetlana.giere@wias-berlin.de (S. Giere), gunar.matthies@tu-dresden.de (G. Matthies), raviteja@cmg.cds.iisc.ac.in (R. Meesala), shamim@cmg.cds.iisc.ac.in (A. Shamim), jagan@cmg.cds.iisc.ac.in (J. Venkatesan), volker.john@wias-berlin.de (V. John).

http://dx.doi.org/10.1016/j.camwa.2016.12.020
0898-1221/

language standards necessitated a re-design and re-implementation of some of the core routines. With the new core and the new features, the code was renamed to ParMooN (Parallel Mathematics and object-oriented Numerics).

The general aims of this paper are to report on the development of the existing research code towards the new package ParMooN in order to accomplish the desired features of an in-house research code that will be formulated in Section 2 and to assess the parallelized geometric multigrid method by comparing it with solvers that are available in an external library. The original code possesses some distinct features that should be transferred to ParMooN, like the strict decoupling of geometry and finite element spaces, the implementation of mapped finite elements as their definition can be found in textbooks, and a multiple discretization multilevel (MDML) preconditioner. This paper focuses on the most relevant aspect concerning the development of ParMooN, namely the distributed memory parallelization. In particular, the technically most cumbersome part, the parallelization of the geometric multigrid, is discussed.

A main contribution of this paper is a first assessment of the resulting parallel geometric multigrid method in comparison with parallel solvers for linear systems of equations that can be called from the library PETSc. The numerical studies were performed on compute servers as the available in-house hardware. We think that this assessment is also of interest for other groups who develop their own codes in order to get an impression whether it is worthwhile to implement a parallelized geometric multigrid method or not. Two main problem classes supported in ParMooN are considered in the numerical studies: scalar convection–diffusion–reaction equations and the incompressible Navier–Stokes equations.

The paper is organized as follows. Section 2 contains an exposition of our thoughts about in-house research codes, in particular about their advantages and their goals. Mapped finite elements, as they are used in MooNMD/ParMooN, are described in Section 3. Section 4 presents main aspects of the parallelization. The parallelization of the geometric multigrid method is briefly discussed in Section 5. Numerical studies that compare this method with solvers available in PETSc are presented in Section 6. The paper concludes with a summary.

2. Some considerations about in-house research codes

Nowadays, several academic software packages for solving partial differential equations exist in the research community. They are usually developed and supported by research groups for whom software development is one of the main scientific tasks. Such software packages include, among others, deal.II [3], FEniCS [4], DUNE [5,6], OpenFOAM [7], or FreeFem++ [8]. These packages have advanced functionality and support features like adaptive mesh refinement, parallelism, etc.

Naturally, a research code developed in-house possesses less functionality than these large packages. In view of their availability, the following questions arise: Why is it worth to develop an own research code? In particular, is it worth to develop a code within a research group that focuses primarily on numerical analysis? In the following, some arguments, mainly based on our own experience, are presented.

In-depth knowledge of details of the software. The first key aspect of working with a code developed and maintained within the research group is the detailed knowledge of the software structure. In fact, applied mathematicians often work at the development of numerical methods. These methods have to be implemented, assessed, and compared with popular state-of-the-art methods for the same problem. A meaningful assessment requires the usage of the methods in the same code. In this respect, it is important to have access to a code where one knows and can control every detail.

For brevity, just one example will be mentioned to show the importance of knowing the details of a software package. This example concerns the clarification of appropriate interface conditions in subdomain iterations for the Stokes–Darcy problem, see [9]. Standard Neumann interface conditions can be used only for viscosity and permeability coefficients that are unrealistically large. For realistic coefficients, appropriate Robin boundary conditions have to be used. The implementation of the Robin interface conditions was performed in a straightforward way in MooNMD/ParMooN.

Flexibility. Further advantages of an own research code are the possibility of controlling its core parts and flexibility. In particular, for our research it is very important that the code supports the use of different discretization strategies. As an example, MooNMD was designed for finite element methods. But for the investigation of discretizations of time-dependent convection–diffusion equations in [10], finite difference methods were implemented as well. Because these methods performed very well, they were later used in the context of simulating population balance systems defined in tensor-product domains, e.g., see [11].

Testing of numerical methods. MooNMD was used in the definition of benchmark problems in [12]. The list of examples could be extended. In addition, a number of numerical methods have been developed and implemented in our research code which turned out to be not (yet) competitive, like the optimization of stabilization parameters in SUPG methods in [13]. Having a known and flexible research code at disposal allows to test and support methods that, at the time of the implementation, have not been benchmarked in detail.

Certainly, also the large packages mentioned above allow the implementation of different methods and discretization strategies. However, we think that a successful implementation often requires a very close interaction with core developers of the packages. This effort might not be feasible for both the user and the developer. Therefore, an own code might reduce the time from the development to the assessment of numerical methods.

Benefits for students. A further aspect, related to the interaction of the core development team, concerns the students and the Ph.D. students who are involved in the development and in the usage of the code. Since the core developers of an own research code are readily available and they are experts in the focused research topics of the group, these students can be supported efficiently. In addition, students working at the code stated several positive effects: the work at details of the

implementation facilitates the insight into the methods and algorithms, which is important for analyzing their properties, and it enhances the skills in software design and management.

Of course, incorporating students into code development requires that there is an easy use of the code and an easy access to basic routines, such as, in the case of a finite element solver, assembling of matrices and solving linear systems of equations. This issue touches already the next question: What should be expected from an own research code?

Easy usage. In order to support students starting to work with the code, an easy installation and basic testing setup are essential. There are even successful attempts for designing complex codes that can be used for teaching students in basic courses on numerical methods for partial differential equations, like the so-called computational laboratory for Investigating Incompressible Flow Problems (IFISS), see [14], which uses MATLAB, and the open source software FREEFEM++ [8], which is based on an own language.

Modularity. The code should be modular. In particular, there should be a general core and individual projects are attached to this core. Of course, the projects use routines from the core. But using an own code, it seems to be easier than with a large package, which is developed somewhere else, to incorporate contributions from the projects into the core.

Stability. With respect to the required stability of the code, there are, in our opinion, no fundamental differences between own research codes and large packages.

Efficiency. However, there are different expectations with respect to the efficiency. A research code should be flexible in many respects, since its main tasks include supporting the development of numerical methods and results from numerical analysis. For instance, in the code MooNMD/ParMooN, the concept of mapped finite elements is implemented, see Section 3. In this way, the code supports currently around 170 finite elements in two dimensions and 75 finite elements in three dimensions. Consequently, all routines are implemented for the general situation. For certain finite elements, this might be less efficient than using tailored routines. However, also for a research code, efficiency is a key property that should not be neglected. For instance, the simulation of standard academic benchmark problems for turbulent incompressible flows requires the computation of large time intervals to collect temporal averages of statistics of interest. In our opinion, an own research code should be reasonably efficient on the available in-house hardware, which, in our case, are usually laptops, compute servers, or small clusters.

3. Mapped finite element spaces

The implementation of finite element methods in MooNMD/ParMooN is based on a rather abstract definition of a finite element space and on the mapping of each mesh cell to a reference cell.

Let $\Omega \subset \mathbb{R}^d$, $d \in \{2, 3\}$, be a bounded domain and let $\mathcal{T}^h$ be an admissible triangulation of Ω consisting of compact, simply connected mesh cells. For each mesh cell K, a local finite element space $P(K) \subset C^s(K)$, $s \geq 0$, is given by some finite-dimensional space of functions spanned by a basis $\{\phi_{K,i}\}_{i=1}^{N_K}$. Furthermore, a set of local linear functionals $\{\Phi_{K,i}\}_{i=1}^{N_K}$ is given. The space $P(K)$ is unisolvent with respect to the functionals. Often, a so-called local basis is chosen, i.e., a basis that satisfies $\Phi_{K,i}(\phi_{K,j}) = \delta_{ij}$ for $i, j = 1, \ldots, N_K$. The local linear functionals might be values of the functions or their derivatives at certain points, integrals on K or on faces of K.

Let $\Phi_1, \ldots, \Phi_N : \{v \in L^\infty(\Omega) : v|_K \in P(K)\} \to \mathbb{R}$ be given continuous linear functionals, where $v|_K \in P(K)$ has to be understood in the way that v is a polynomial in the interior of K and it is extended continuously to the boundary of K. The restriction of each functional to $C^s(K)$ defines the set of local functionals. The union of all mesh cells K_j, for which there is a $p \in P(K_j)$ with $\Phi_i(p) \neq 0$, will be denoted by ω_i. Now, the global finite element space is defined as follows. A function $v(\boldsymbol{x})$ defined on Ω with $v|_K \in P(K)$ for all $K \in \mathcal{T}^h$ is called continuous with respect to the functional Φ_i if $\Phi_i(v|_{K_1}) = \Phi_i(v|_{K_2})$ for all $K_1, K_2 \in \omega_i$. The space

$$S = \left\{ v \in L^\infty(\Omega) : v|_K \in P(K) \text{ and } v \text{ is continuous with respect to } \Phi_i,\ i = 1, \ldots, N \right\}$$

is called finite element space. The global basis $\{\phi_j\}_{j=1}^N$ of S is defined by the conditions $\phi_j \in S$ with $\Phi_i(\phi_j) = \delta_{ij}$ for $i, j = 1, \ldots, N$.

Using this definition for the implementation of a finite element space requires

(1) the definition of the local basis and linear functionals for each K,
(2) the implementation of a method that assures continuity with respect to the functionals stated in the definition of S.

The first requirement can be achieved in two different ways, via a mapped or an unmapped implementation. In the unmapped approach, the local basis and linear functionals are defined directly on K. In contrast, mapped finite elements are closely connected to a standard way of analyzing finite element discretizations. This analysis consists of three steps:

- Map an arbitrary mesh cell K to a compact reference mesh cell $\hat{K}$.
- Prove the desired properties on $\hat{K}$, which is the core of the analysis.
- Map the reference mesh cell $\hat{K}$ back to K to get the final result.

Hence, this approach has two main features:

- All considerations have to be done on $\hat{K}$ only.
- Information about neighbor mesh cells of K is neither available nor needed.

Mapped and unmapped finite element methods possess the same analytical properties if the reference map $F_K : \hat{K} \to K$ is affine for every mesh cell K of the given triangulation, e.g., compare [15, Chap. 2.3]. In the case of non-affine maps, occurring, e.g., for a triangulation consisting of arbitrary quadrilateral or hexahedral mesh cells, mapped and unmapped finite element spaces might be different. In MooNMD/ParMooN, the concept of mapped finite elements is implemented in the following way. Reference mesh cells are the unit simplices, e.g., in two dimensions with the vertices (0, 0), (1, 0), (0, 1), and the unit cubes $\hat{K} = [-1, 1]^d$. Affine maps are available for all reference mesh cells. To account for arbitrary quadrilaterals and hexahedra, d-linear maps are also implemented for the unit cubes. Based on the different reference cells, local spaces on $\hat{K}$, linear functionals, and reference maps, MooNMD/ParMooN currently supports about 170 finite elements in two dimensions and 75 finite elements in three dimensions.

The use of mapped finite element spaces essentially requires the implementation of finite elements on the reference cells. The quadrature rules for numerical integration have to be implemented only on these cells, since the integrals on physical cells are transformed to integrals on the reference cells. Note that the same strategy works also for the handling of cell faces, which are mapped onto lower-dimensional reference cells by corresponding reference maps.

Concerning (2) above, a finite element space S is represented by a map $\mathcal{F}$ called d.o.f.-manager which maps local, i.e., within a cell, indices of degrees of freedom (d.o.f.) to global ones. For this purpose define $M(K)$ to be the set of local d.o.f.s denoted by (K, i) on the cell K. Then define the set of all local degrees of freedom

$$M := \bigcup_{K \in \mathcal{T}^h} M(K).$$

The local-to-global map $\mathcal{F}$ now surjectively maps M to $\{1, \ldots, N\}$ such that $\mathcal{F}((K, i)) = \mathcal{F}((K', j))$ whenever the local degrees of freedom (K, i) and (K', j) belong to the same global degree of freedom. The number $N \leq |M|$ is then the number of global degrees of freedom. In other words, $\mathcal{F}$ describes a partition P of the set M (i.e., an equivalence relation) together with a global numbering. Computing such a map $\mathcal{F}$ is done via Algorithm 1 in ParMooN.

Algorithm 1: Computation of $\mathcal{F}$ that maps local degrees of freedom to global ones.
The input is a mesh $\mathcal{T}^h$ whose cells K are ordered by increasing integers $id(K)$.

```
1  P ← finest partition of M
2  for all mesh cells K ∈ T^h
3      determine M(K)
4      for all neighbors K' of K with id(K) < id(K')
5          determine M(K')
6          find local partition P_loc of M(K) ∪ M(K')
7          P ← finest partition coarser than P and P_loc
8  assign increasing integers to each subset in the partition P
```

Initially the partition is set to be the finest partition of the set M, i.e., it consists of disjoint single-element subsets, so that no two degrees of freedom in M are identified yet. Step 6 in Algorithm 1 consists of finding identical degrees of freedom in two neighboring cells K and K', i.e., a local partition P_{loc} of $M(K) \cup M(K')$. This step is done using the information of positions of degrees of freedom in the reference cell and tailored mapper classes. Step 7 updates the partition P such that all sets in the previous set P and in P_{loc} are contained in one set in the updated partition P. Finally, to each set of the resulting partition of M, a unique integer is assigned that will serve as (the index of) the global degree of freedom. Steps 6 and 7 are explained in detail in the following example.

Example 1. Consider a 2×2 mesh $\mathcal{T}^h$ consisting of four cells A, B, C, and D, each associated with a Q_1-finite element, see Fig. 1. The set of all local degrees of freedom is therefore $M = \{(A, 1), \ldots, (A, 4)\} \cup \{(B, 1), \ldots, (B, 4)\} \cup \{(C, 1), \ldots, (C, 4)\} \cup \{(D, 1), \ldots, (D, 4)\}$. Algorithm 1 modifies a partition P, which is initialized to be the finest partition of M. During the algorithm, the condition $id(K) < id(K')$ is true exactly four times and steps 6 and 7 are depicted in Fig. 2.

(i) $K = A, K' = B$: Identify $(A, 2)$, $(B, 1)$ (red) and $(A, 4)$, $(B, 3)$ (green).
(ii) $K = A, K' = C$: Identify $(A, 3)$, $(C, 1)$ (blue) and $(A, 4)$, $(C, 2)$ (green).
(iii) $K = B, K' = D$: Identify $(B, 3)$, $(D, 1)$ (green) and $(B, 4)$, $(D, 2)$ (yellow).
(iv) $K = C, K' = D$: Identity $(C, 2)$, $(D, 1)$ (green) and $(C, 4)$, $(D, 3)$ (teal).

Now the set P defines a partition with nine sets, each describing one global degree of freedom. The map $\mathcal{F}$ identifies M with its image space $\{1, \ldots, 9\}$:

$$\begin{aligned}
&\mathcal{F}(A, 1) = 1, && \mathcal{F}(A, 2) = \mathcal{F}(B, 1) = 2,\\
&\mathcal{F}(A, 3) = \mathcal{F}(C, 1) = 3, && \mathcal{F}(A, 4) = \mathcal{F}(B, 3) = \mathcal{F}(C, 2) = \mathcal{F}(D, 1) = 4,\\
&\mathcal{F}(B, 2) = 5, && \mathcal{F}(B, 4) = \mathcal{F}(D, 2) = 6,\\
&\mathcal{F}(C, 3) = 7, && \mathcal{F}(C, 4) = \mathcal{F}(D, 3) = 8,\\
&\mathcal{F}(D, 4) = 9.
\end{aligned}$$

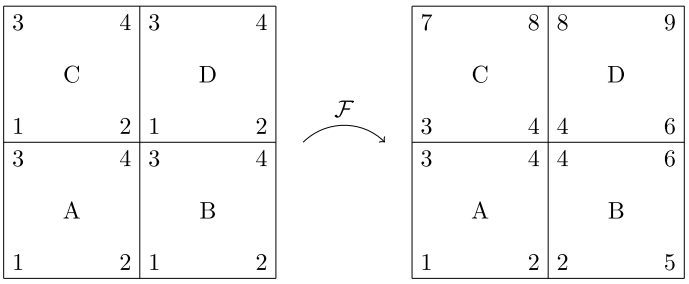

Fig. 1. A 2×2 mesh with Q_1 finite elements. Left: Local degrees of freedom in each cell. Right: Global degrees of freedom after the application of Algorithm 1.

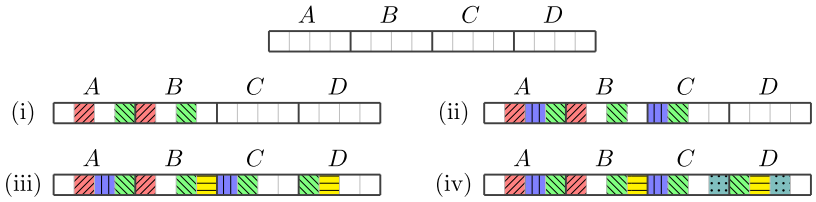

Fig. 2. The partition P of M during the four relevant steps in Algorithm 1. Each cell corresponds to one local degree of freedom. Cells with the same color/pattern are considered equal, only empty (white) cells are not equal to each other. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

The image of $\mathcal{F}$ is illustrated in Fig. 1 on the right.

The implementation of the concepts described in this section has been adapted from MooNMD to ParMooN in a straightforward way. It is clear that the definition of the local basis and functionals is completely local and therefore not affected by the parallelization. The required continuity in the definition of finite element spaces is performed in ParMooN on each process separately. The computed results are visualized with the software package ParaView [16,17] that does not require global numbers for the d.o.f.s. However, if needed, global numbers for d.o.f.s across all processes can be assigned, compare [18]. Altogether, the used concepts turned out to be applicable in the same way for the sequential as well as for the parallel code.

4. Parallel data structures in ParMooN

In this section, the main steps in our approach to parallelize a finite element code are described. ParMooN supports a single program, multiple data (SPMD) approach on parallelism using the Message Passing Interface (MPI) standard [19]. It relies on a decomposition of the domain, which is the standard for parallelized finite element codes. Decomposing the computational domain and distributing it among the processes naturally leads to a parallelization of matrix–vector operations. The local aspect of the finite element method, which is reflected in the sparsity of the arising matrices, limits the communication overhead.

The main steps to be described in this section are domain decomposition, d.o.f. classification, consistency levels for distributed vectors, and technical details on the implementation of communication.

4.1. Decomposing the domain—own cells and halo cells

In order to distribute the domain among the participating MPI processes, ParMooN makes use of the METIS graph partitioning tool [20]. At program start, all processes read the same geometry and perform the same initial domain refinement steps. Upon reaching the first refinement level on which computations will be performed, the root process (process number 0) calls the METIS library to compute a disjoint domain decomposition, i.e., to determine which process is going to be in charge of which mesh cells.

Root then communicates the METIS output to the other processes. Each process is informed about the cells it will be responsible for. These cells are called *own cells* of the process. Each process then keeps only its own cells plus those cells that share a boundary face, edge, or vertex with an own cell. In domain decomposition methods these cells are commonly referred to as *halo cells*. The sketches in Fig. 3 clarify that expression—the halo cells form a one-layer thick halo around the set of own cells. Each process P deletes all cells which are neither own nor halo cells of P.

The own cells are further divided into *dependent* and *independent cells*. Therefore one defines the *interface* as the set of those faces, edges, and vertices which are shared by own cells and halo cells. All own cells that contain a piece of interface are called dependent cells, while the remaining own cells are called independent cells.

The requirements on an efficient domain decomposition are twofold: the computational load must be balanced, i.e., there should be a comparable number of cells on each process, and the needed amount of communication must be small, i.e., the

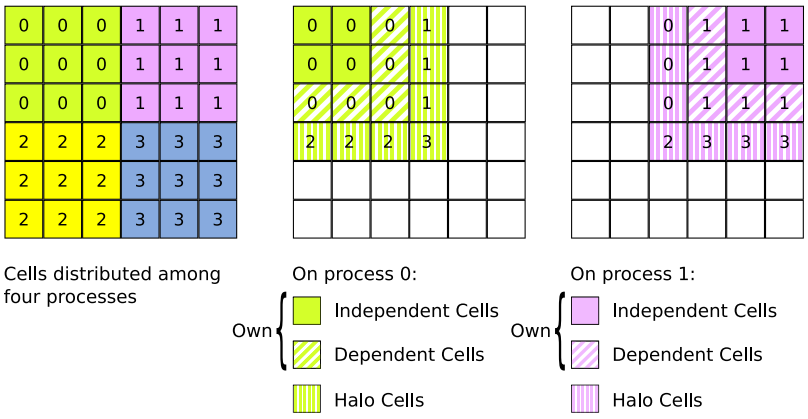

Fig. 3. Different cell types due to a domain decomposition.

interfaces should be as small as possible. Due to the deletion of cells, each process stores only a part of the entire problem, but all processes execute the same program code. This was initially referred to as the single program, multiple data approach.

With its domain reduced to own cells plus halo cells, each process sets up a finite element space and performs all further computations only on its part of the domain.

4.2. Types of degrees of freedom

In the case of parallelized finite element methods, communication is necessary to transmit values of d.o.f.s across processes. This section describes the classification of d.o.f.s which is applied for this purpose.

A d.o.f. with index i is defined by the finite element basis function ϕ_i and the associated global linear functional Φ_i. It is represented, e.g., by the ith entry in the vectors for the solution and the right-hand side. Each process is responsible for the classification of its d.o.f.s. The class to which a certain d.o.f. i belongs to depends on its location and on the classes of those d.o.f.s with which it is coupled. That is to say, two degrees of freedom of a finite element in d dimensions are said to be *coupled* if the supports of the corresponding basis functions intersect on a set of non-zero d-dimensional measure. Note that, with this definition, coupling only occurs for d.o.f.s that are located in the same mesh cell. This notion of coupling transfers directly to a property of the finite element matrix A: the coupling of d.o.f.s i and j will (potentially) lead to non-zero entries a_{ij} and a_{ji}, and the needed memory for these entries has to be allocated.

In the following, the classes of d.o.f.s in ParMooN will be described shortly. All d.o.f.s that are localized in an own cell known to process P are called *known* d.o.f.s and their set will be denoted by D^P_{known} (D^P_* stands in the following for the set of d.o.f.s on P of type $*$). This set is then divided into *masters* and *slaves*, i.e.,

$$D^P_{\text{known}} = D^P_{\text{master}} \dot\cup D^P_{\text{slave}},$$

where $\dot\cup$ denotes the disjoint union. A d.o.f. i is said to be a master on P if P is responsible for the value of i, in a way that will be clarified in Section 4.3. All known d.o.f.s which are not master on P are called slaves. It is worth noting that every d.o.f. in the entire problem is a master on exactly one process. In contrast, a d.o.f. can be slave on more than one process.

An even finer classification of D^P_{master} and D^P_{slave} will be used in the following. Note that the names for the classes correspond only loosely to the types of cells they are located in. Care must be taken of those d.o.f.s that lie on the intersection of different cell types. The classes are:

- *Independent d.o.f.*, i.e., all d.o.f.s which lie in P's own cells but not in its dependent cells. All P's independent d.o.f.s are set as masters, since they are not even known to any other process. They only couple to other masters of P.
- *Dependent d.o.f.*, i.e., those d.o.f.s lying in P's dependent cells, but not in its halo cells. Process P is the master of all its dependent d.o.f.s. The notation is motivated by the fact that the dependent d.o.f.s are in a vicinity to the domain interface and therefore possess a certain dependency on other processes.
- *Interface d.o.f.*, i.e., all d.o.f.s that lie on the intersection of dependent cells and halo cells. These d.o.f.s are known to all adjacent processes as interface d.o.f.s, too. Only one of these processes will take master responsibility for each interface d.o.f. In particular, on a process P, one distinguishes between *master interface d.o.f.s* (the interface d.o.f.s which are master on P) and *slave interface d.o.f.s* (all other d.o.f.s, for each of which a neighboring process takes master responsibility).
- *Halo d.o.f.*, i.e., all d.o.f.s which lie in halo cells but not on the interface. Since all of them are dependent d.o.f.s to neighboring processes, one of these will take master responsibility for them. On P all halo d.o.f.s are slaves.

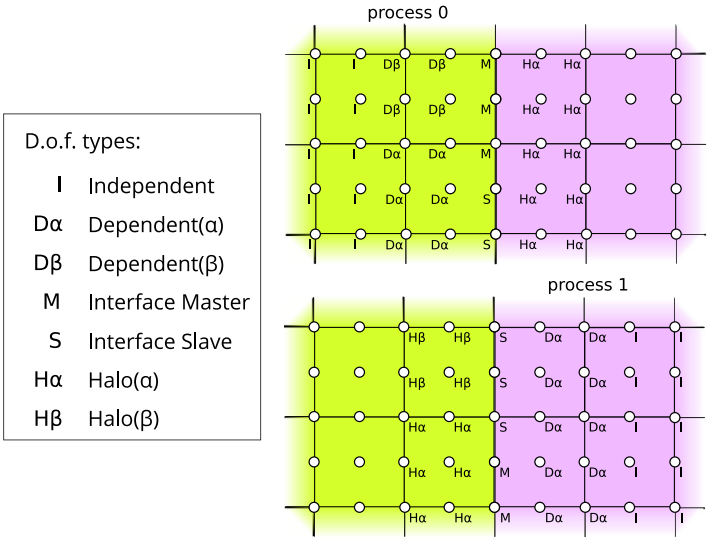

Fig. 4. Types of d.o.f.s at the interface for two-dimensional Q_2 finite elements, from the point of view of process 0 and process 1, where only the named d.o.f.s are known to the respective process.

Hence, the sets of masters and slaves can be divided as

$$D^P_{\text{master}} = D^P_{\text{independent}} \dot\cup D^P_{\text{dependent}} \dot\cup D^P_{\text{interfacemaster}},$$

$$D^P_{\text{slave}} = D^P_{\text{interfaceslave}} \dot\cup D^P_{\text{halo}}.$$

Moreover, it is convenient to further refine the d.o.f. classification in order to reduce the communication overhead, see Section 4.3. To this aim, the halo and the dependent d.o.f.s of P are further divided into Halo(α) and Halo(β), and Dependent(α) and Dependent(β), respectively:

- *Halo(α) and Halo(β) d.o.f.s*: The Halo(α) d.o.f.s are those that are coupled with at least one (interface) master of P, while Halo(β) d.o.f.s are coupled solely with other slaves, i.e., with interface slave d.o.f.s and other halo d.o.f.s.
- *Dependent(α) and Dependent(β) d.o.f.s*: Dependent(α) d.o.f.s are those connected to at least one (interface) slave, while Dependent(β) d.o.f.s are all those that are connected to masters only, i.e., to interface master, other dependent, or independent d.o.f.s. Note that all Dependent(β) d.o.f.s of process P will be Halo(β) on all other processes where they are known. For Dependent(α) d.o.f.s the situation is not as simple. Each of them is Halo(α) to at least one neighboring process, but can be Halo(β) to others.

In general, the d.o.f. classification depends on the finite element spaces and the decomposition of the domain. Fig. 4 sketches the different types for two-dimensional Q_2 finite elements at the interface between two processes.

4.3. Consistency levels

In parallel computations, values and information, e.g., finite element vectors, can be stored either in a consistent way or in an additive way. In the case of a finite element vector, which is of major interest here, *consistent* storage means that all processes have the same and correct value at all respective d.o.f.s as in the sequential environment. If the vector is present in *additive* storage, each global value is the sum of the values over all processes where it is known. In ParMooN, certain concepts of weakened consistent storage of finite element vectors and matrices are used.

A finite element vector belongs to one of the following four levels of consistency.

- *Level*-0-*consistent*. Consistency holds only with regard to master degrees of freedom. Each master on each process holds the same value as it would be in a sequential computation. The values of slaves are in an undefined storage state. In the implementation of operations care must be taken not to lose Level-0-consistency—all master values must be kept as they would be in a sequential execution.
- *Level*-1-*consistent*. All masters and all interface slave d.o.f.s are stored consistently. The values of all halo d.o.f.s are in an undefined storage state.
- *Level*-2-*consistent*. Consistency is established for all but Halo(β) d.o.f.s. The values of Halo(β) d.o.f.s. are in an undefined storage state while all other values are consistent.
- *Level*-3-*consistent*. All d.o.f.s are stored consistently. This situation is called the "real" consistent storage.

Table 1
Master–slave relationship of d.o.f. types.

Relation (shorthand)	Master type	Updates	Slave type
Interface (IMS)	Interface master	$\longrightarrow$	Interface slave
Dependent(α)–Halo(α) (DHα)	Dependent(α)	$\longrightarrow$	Halo(α)
Dependent(β)–Halo(β) (DHβ)	Dependent(β)	$\longrightarrow$	Halo(β)

The main motivations behind introducing this classification are that different operations require a different state of consistency of their input data and that restoring a certain state of consistency requires a certain amount of communication—the lower the required state of consistency, the lower the required amount of communication.

After the domain has been decomposed, each process P assembles a finite element matrix on all its known cells. The use of halo cells assures that all information to assemble the rows belonging to masters is available on P. The finite element matrix assembled in this way will have the property that all rows and columns which belong to masters are correctly assembled, i.e., consistent. A matrix with this property is called Level-0-consistent, too. In fact, even the rows belonging to interface slave d.o.f.s are correct, but this is only a by-product. Hence, the finite element matrix is even Level-1-consistent. Note that it is not possible to extend this concept to Level-2- or even Level-3-consistency of a matrix. The reason is that some of the d.o.f.s which would be needed to store the entire matrix row associated with a halo d.o.f. are not in D^P_{known}.

Regarding operations, multiplication of a Level-0-consistent matrix with a Level-2-consistent vector gives a Level-0-consistent vector. Level-3-consistency of the input vector is not needed. If the input is Level-3-consistent, together with the matrix being Level-1-consistent, the result will be Level-1-consistent.

Multiplying a vector with a constant scalar will maintain the current consistency level, as will vector–vector addition. In the latter case, vectors with different consistency level might be added and the result has the lower consistency level. Scalar products require Level-0-consistency of both vectors, where all slaves will be skipped, and a globally additive reduce operation is required to get a consistent result.

Level-3-consistency of a finite element vector in ParMooN is only enforced if operations require knowledge of the represented finite element function even on the halo cells. Such operations include the matrix assembling with an input finite element function, e.g., for the convective term of the Navier–Stokes equations, interpolating initial conditions in a time-dependent problem, or gradient recovery by averaging gradients over a patch of mesh cells.

Enforcing certain consistency levels is a matter of communication. For each d.o.f. which needs an update, the responsible master process communicates its value to all processes where it is slave. These processes simply reset its value to the received value. The required infrastructure for this communication is set up just once for a certain finite element space and can be reused whenever an update is necessary.

4.4. Organizing communication

When setting up the communication structure, one has to find, for each non-independent master i, all those slaves on other processes that are globally identical to i. Certain master types match with certain slave types, see Table 1, forming three distinct pairs of master–slave relations. To restore a certain consistency level, at least one of these relations requires an update.

Note that it is generally not possible to immediately identify the global number of a d.o.f., since each process creates finite element spaces only on its known (own and halo) cells and it numbers its d.o.f.s locally, unaware of the other processes. In ParMooN, the global identification of a d.o.f. is defined according to the global number of the mesh cell in which the d.o.f. is located. These global cell numbers are assigned to each cell before decomposing the domain. Performing only uniform refinements after domain decomposition, such a globally unique cell number can easily be given to children cells, too. Hence, the global cell number and a consistent local numbering of the d.o.f.s within each cell enable to identify each d.o.f. globally.

The communication structure is stored in a class called `ParFEMapper`, while the communication itself is performed by a class named `ParFECommunicator`. Setting up the `ParFEMapper` and `ParFECommunicator` requires some communication itself, and a detailed description of this task is presented in [18]. In what follows, only an overview of it is given, including a short description of those data fields of `ParFEMapper` that are relevant when updating the d.o.f.s of a certain master–slave relation, see Table 1. The interface (IMS) relation is presented as an example. For the two other relations, the data fields are defined in a similar way.

For the IMS update, the `ParFECommunicator` wraps a call to the MPI function `MPI_Alltoallv`, where every process may send a different set of values of the same type (`MPI_DOUBLE` in this case) to each other process. To control the `MPI_Alltoallv` call, the `ParFEMapper` stores the following data, where `mpi_size` is the total number of processes and `nInterfaceSlaves` is the number of interface slaves local to process P.

- `int* sendBufIMS` is the send buffer, filled with the values of all interface masters, each one possibly appearing more than once, which will then be sent to the other processes. Its total length equals the sum over all values of `sendCountsIMS`.
- `int* sendCountsIMS` is an array of size `mpi_size`. It lists how many values P has to send to each process.

- `int* sendDisplIMS` is the send displacement, an array of size `mpi_size`. It stores where in the array `sendBufIMS` the message for a certain process starts. In ParMooN there are neither overlaps nor gaps, so `sendDisplIMS[i]` holds the sum of `sendCountsIMS[0]` to `sendCountsIMS[i-1]`.
- `int* recvBufIMS` is the receive buffer, which will be filled with sent values from the other processes in the communication routine. Its size equals `nInterfaceSlaves`.
- `int* recvCountsIMS` is an array of size `mpi_size`. It lists how many values are to be received from each process. The sum of its values is `nInterfaceSlaves`.
- `int* recvDisplIMS` is the receive displacement. Like for `sendDisplIMS` there are neither gaps nor overlaps.

Besides the data, which is needed in the immediate control of `MPI_Alltoallv`, the `ParFEMapper` contains two arrays that allow to interpret the sent and received data, by mapping between send or receive buffer and the local d.o.f.s:

- `int* sentDofIMS` interprets `sentDofIMS[i]` $= d$ as: The ith place in `sendBufIMS` has to be filled with the value of the local d.o.f. i.
- `int* rcvdDofIMS` interprets `rcvdDofIMS[i]` $= d$ as: The ith value in `recvBufIMS` should update the local d.o.f. i.

To change a vector from Level-0-consistency to Level-1-consistency, only an IMS update is required. For restoring Level-2-consistency, additionally a DHα update is necessary, while Level-3-consistency requires even a DHβ update on top.

5. The parallel geometric multigrid method

Geometric multigrid methods are an appealing option to be used as preconditioners in problems where the necessary hierarchy of grids can be provided. These methods have been used in the simulations performed with MooNMD in particular for three-dimensional problems and for linear saddle point problems arising in the linearization and discretization of equations modeling incompressible flow problems [21].

The components of a geometric multigrid method are the following: function prolongation, defect restriction, function restriction, smoother, and coarse grid solver. Concerning details of the algorithms and implementation of the first three components in MooNMD/ParMooN, it will be referred to [1] for details. It shall be only noted that multilevel methods are supported that allow different finite element spaces on different levels of the multigrid hierarchy. In particular, the so-called multiple discretization multilevel (MDML) method for higher order discretizations can be used. This method possesses the higher order discretization on the highest multigrid level and a low order discretization on all coarser levels, where more multigrid levels might be defined than geometric levels exist. The motivation for this approach is the experience that multigrid methods often work very efficiently for low order discretizations. Numerical studies of the efficiency of the MDML method can be found, e.g., in [22]. The grid transfer operations are performed with a local operator, taking values only on the mesh cells of the current level, proposed in [23], that can handle different finite element spaces on different levels of the grid hierarchy, see [21,1] for details.

The implementation and parallelization of geometric multigrid methods require a considerable amount of work. The geometric data structures need to be equipped with parent–child information and the grid transfer operations have to be implemented. In the current version of ParMooN, each process is responsible for a part of the coarsest grid, compare Section 4.1, and refines this part uniformly. Consequently, all parent–child information is available on the process. More technical details on constructing the grid hierarchy can be found in [18].

As usual in parallel geometric multigrid methods, block-Jacobi smoothers are applied, where the blocks correspond to the master and interface slave degrees of freedom of a process. Within the blocks, the actual smoother, like SSOR or the Vanka smoother, is used. After each smoothing iteration, the values at the interface are updated by computing their arithmetic average. As already mentioned above, the grid transfer operators need as potential input all values that are connected to a mesh cell. Since it is sufficient to perform the grid transfer only on own cells, the input vectors for the grid transfer have to be Level-1-consistent.

6. Numerical studies

The performed numerical studies are a first step of assessing the efficiency of the parallelized geometric multigrid method in comparison with parallel solvers that can be used by linking an external library to the code. The underlying question is whether it was worthwhile to perform the complex parallelization of this method. We think that this question arises also in other groups that maintain an in-house research code.

In this paper, the numerical studies concentrate on the standard multigrid method (same number of geometric and multigrid levels, same discretization on each level) since we think that this method is of most interest for the community. It was used as preconditioner in the flexible GMRES (FGMRES) method [24]. The system on the coarsest grid was solved with a sparse direct solver. Moreover, the V-cycle was applied because of its better efficiency on parallel computers compared with the more stable F- and W-cycle (the F-cycle is our standard approach in sequential simulations). The V-cycle approaches less often coarser grids, which possess an unfavorable ratio of computational work and necessary communications, than the other cycles.

The efficiency of the geometric multigrid preconditioner has been compared with the efficiency of the sparse direct solver MUMPS [25,26] and the FGMRES method preconditioned with SSOR (for scalar problems), the BoomerAMG [27] (for scalar

problems), or the LSC preconditioner [28] (for linear saddle point problems). These solvers were used as they are provided in the library PETSc, version 3.7.2, [29–31]. The restart parameter in FGMRES was set to be 50.

The numerical studies were performed on a hardware platform that can be usually found in universities and academic institutes, in our case, on compute servers HP BL460c Gen9 2xXeon, Fourteen-Core 2600 MHz. We think that the performance on a hardware platform that is widely available is of interest for the community. The results will consider only the computing times for the different solvers of the linear systems of equations. We could observe some variations of these times for the same code and input parameters across different runs. To reduce the influence of these variations, all simulations were performed five times, the fastest and the slowest computing time were neglected and the average of the remaining three times is presented below.

Example 2 (*Steady-State Convection–Diffusion Equation*)**.** This example is a three-dimensional extension of a benchmark problem for two-dimensional convection–diffusion equations—the so-called Hemker example [32,33]. The domain is defined by

$$\Omega = \left\{\{(-3, 9) \times (-3, 3)\} \setminus \left\{(x, y) : x^2 + y^2 \leq 1\right\}\right\} \times (0, 6)$$

and the equation is given by

$$-\varepsilon \Delta u + \boldsymbol{b} \cdot \nabla u = 0 \quad \text{in } \Omega,$$

with $\varepsilon = 10^{-6}$ and $\boldsymbol{b} = (1, 0, 0)^T$. Dirichlet boundary conditions $u = 0$ were prescribed at the inlet plane $\{x = -3\}$ and $u = 1$ at the cylinder. At all other boundaries, homogeneous Neumann boundary conditions were imposed. This example models, e.g., the heat transport from the cylinder. The solution exhibits boundary layers at the cylinder and internal layers downwind the cylinder, see Fig. 5.

It is well known that stabilized discretizations have to be employed in the presence of dominant convection. In the numerical studies, the popular streamline-upwind Petrov–Galerkin (SUPG) method [34,35] was used with the standard parameter choice given in [36, Eqs. (5)–(7)]. Simulations were performed with Q_1 finite elements. The initial grid is depicted in Fig. 5.

Computational results are presented in Figs. 6 and 7 for refinement levels 4 (1 297 375 d.o.f.s) and 5 (10 388 032 d.o.f.s). The PETSc solvers were called with the flags `-ksp_type fgmres -pc_type sor` and `-ksp_type fgmres -pc_type hypre -pc_hypre_type boomeramg`, respectively. For the geometric multigrid preconditioner, the V(2,2)-cycle was used and the overrelaxation parameter of the SSOR smoother within the block-Jacobi method was set to be $\omega = 1$. This approach is certainly not optimal since with an increasing number of processes the number of blocks of the block-Jacobi method increases, which in turn makes it advantageous to apply some damping, i.e., a somewhat smaller overrelaxation parameter. In order not to increase the complexity of the numerical studies, we decided to fix a constant overrelaxation parameter that worked reasonably well for the whole range of processor numbers which was used. The iterative solvers were stopped if the Euclidean norm of the residual vector was less than 10^{-10}. Parameters like the stopping criterion, the overrelaxation factor, and the restart parameter were the same in all iterative methods.

It can be seen in Fig. 6 that the sparse direct solver performed less efficient by around two orders of magnitude than the other solvers. This behavior was observed for all studied scalar problems and no further results with this solver for scalar problems will be presented. Among the iterative solvers, PETSc FGMRES with SSOR preconditioner and ParMooN FGMRES with geometric multigrid preconditioner (MG) performed notably more efficient than PETSc FGMRES with BoomerAMG. The latter solver did not even converge on the finer grid in the simulations on more than two processors. With respect to the computing times and the stability, similar results were obtained by applying two V(1,1) BoomerAMG cycles (`-pc_hypre_boomeramg_max_iter 2`). With the V(2,2) cycle (`-pc_hypre_boomeramg_grid_sweeps_all 2`), the solver did not converge. FGMRES with SSOR required considerably more iterations than ParMooN FGMRES with MG: around 125 vs. 25 on level 4 and 320 vs. 45 on level 5. A notable decrease of the computing time for the iterative solvers on the coarser grid can be observed only until 8 processors. On this grid, PETSc FGMRES with SSOR was a little bit faster than ParMooN FGMRES with MG. On the finer grid, a decrease of the computing times occurred until 16 processors and ParMooN FGMRES with MG was often a little bit more efficient.

Example 3 (*Time-Dependent Convection–Diffusion–Reaction Equation*)**.** This example can be found also in the literature, e.g., in [10], and it models a typical situation which is encountered in applications. A species enters the domain $\Omega = (0, 1)^3$ at the inlet $\Gamma_{\text{in}} = \{0\} \times (5/8, 6/8) \times (5/8, 6/8)$ and it is transported through the domain to the outlet $\Gamma_{\text{out}} = \{1\} \times (3/8, 4/8) \times (4/8, 5/8)$. In addition, the species is diffused somewhat and in the subregion where the species is transported, also a reaction occurs. The ratio of diffusion and convection is typical for many applications.

The underlying model is given by

$$\begin{aligned}
\partial_t u - \varepsilon \Delta u + \boldsymbol{b} \cdot \nabla u + cu &= 0 && \text{in } (0, 3) \times \Omega,\\
u &= u_{\text{in}} && \text{in } (0, 3) \times \Gamma_{\text{in}},\\
\varepsilon \frac{\partial u}{\partial \boldsymbol{n}} &= 0 && \text{on } (0, 3) \times \Gamma_{\text{N}},\\
u &= 0 && \text{on } (0, 3) \times (\partial\Omega \setminus (\Gamma_{\text{N}} \cup \Gamma_{\text{in}})),\\
u(0, \cdot) &= u_0 && \text{in } \Omega.
\end{aligned}$$

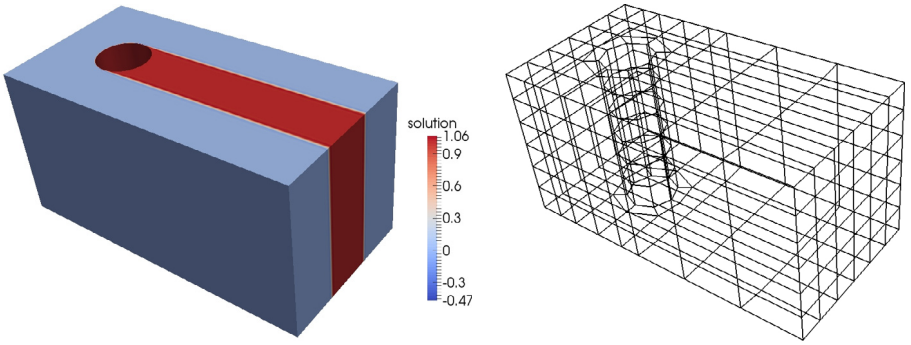

Fig. 5. Example 2, numerical solution (left) and initial grid (level 0, right). The color bar shows that the numerical solution computed with the SUPG method possesses under- and over-shoots. These spurious oscillations occur in particular in a vicinity of the cylinder, compare [13, Fig. 14] for the two-dimensional situation. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

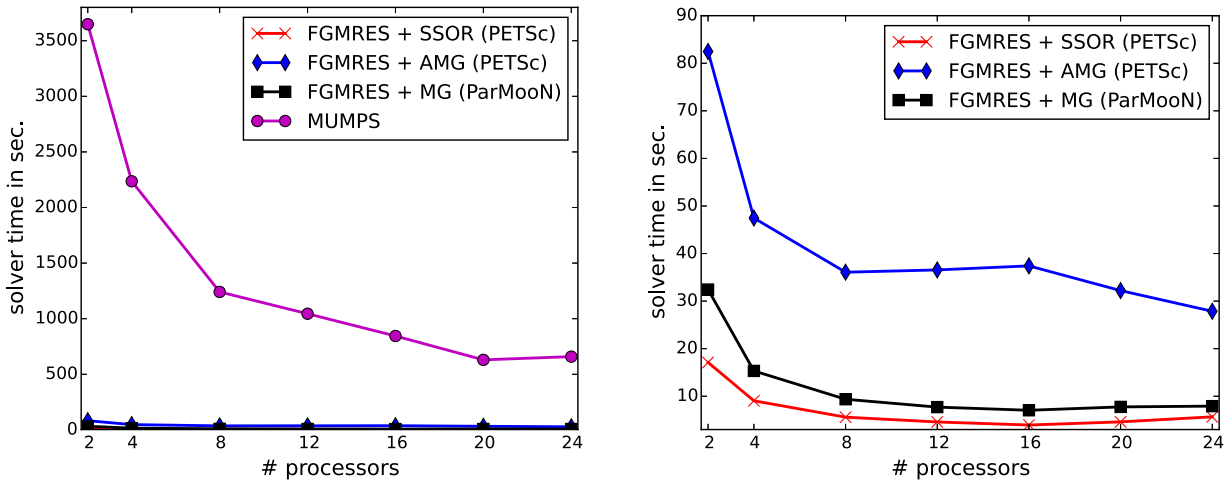

Fig. 6. Example 2, solver times on refinement level 4.

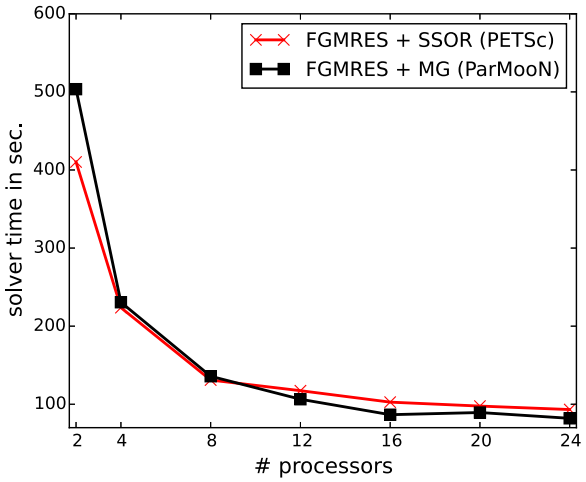

Fig. 7. Example 2, solver times on refinement level 5. PETSc FGMRES with BoomerAMG converged only with two processors (1519 s).

The diffusion parameter is given by $\varepsilon = 10^{-6}$, the convection field is defined by $\boldsymbol{b} = (1, -1/4, -1/8)^T$, and the reaction by

$$c(\boldsymbol{x}) = \begin{cases} 1 & \text{if } \operatorname{dist}(\boldsymbol{x}, g) \leq 0.1, \\ 0 & \text{else,} \end{cases}$$

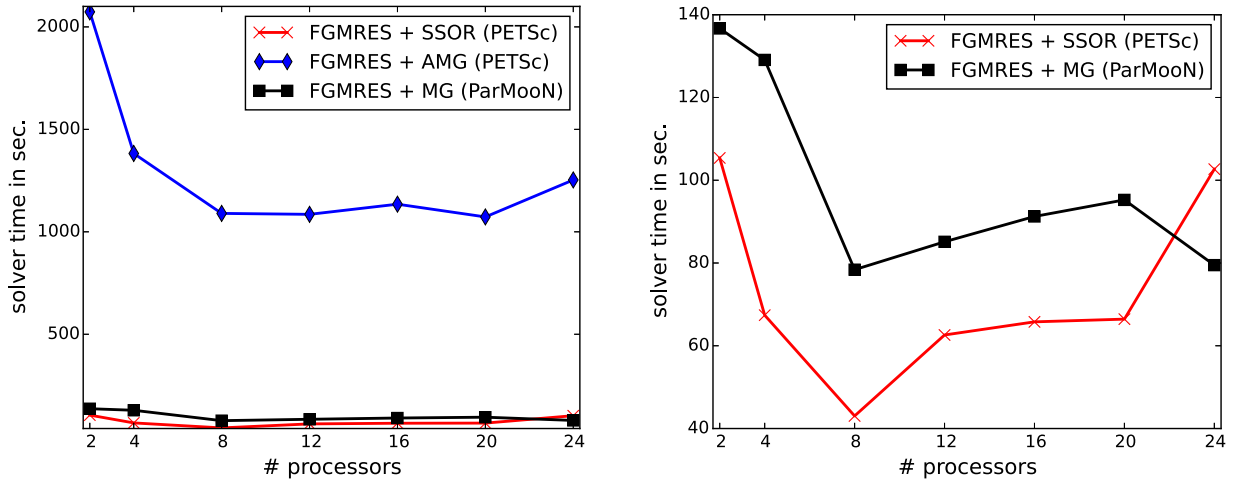

Fig. 8. Example 3, solver times on the 64^3 cubed mesh.

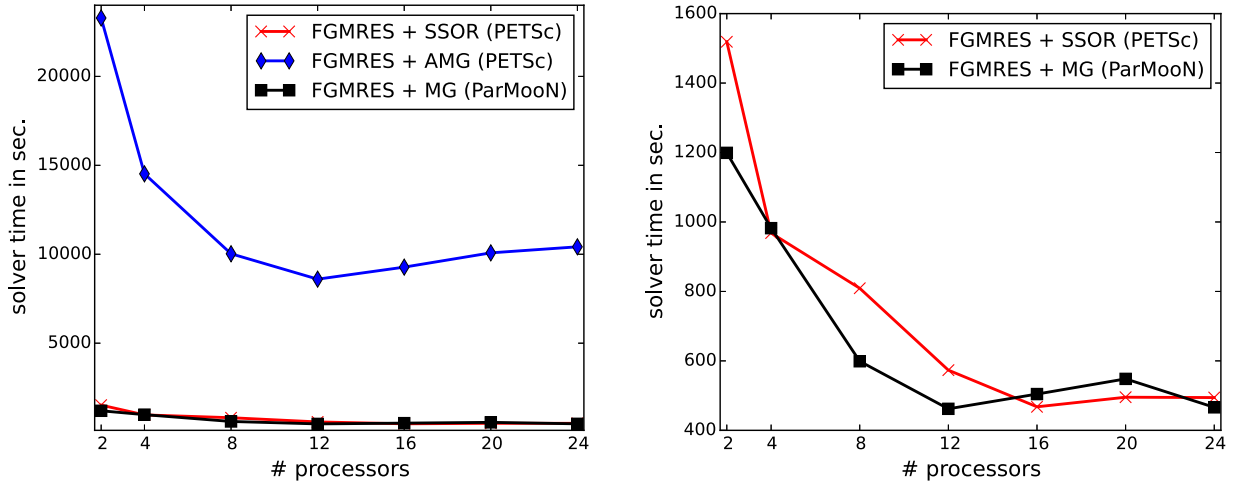

Fig. 9. Example 3, solver times on the 128^3 cubed mesh.

where g is the line through the center of the inlet and the center of the outlet and $\mathrm{dist}(\boldsymbol{x}, g)$ denotes the Euclidean distance of the point $\boldsymbol{x}$ to the line g. The boundary condition at the inlet is prescribed by

$$u_{\text{in}} = \begin{cases} \sin(\pi t/2) & \text{if } t \in [0, 1], \\ 1 & \text{if } t \in (1, 2], \\ \sin(\pi(t-1)/2) & \text{if } t \in (2, 3]. \end{cases}$$

The initial condition is set to be $u_0(\boldsymbol{x}) = 0$. Initially, in the time interval $[0, 1]$, the inflow increases and the injected species is transported towards the outlet. Then, there is a constant inflow in the time interval $(1, 2]$ and the species reaches the outlet. At $t = 2$, there is almost a steady-state solution. Finally, the inflow decreases in the time interval $(2, 3]$, compare [10].

The SUPG stabilization of the Q_1 finite element method was used and as temporal discretization, the Crank–Nicolson scheme with the equidistant time step $\Delta t = 10^{-2}$ was applied. Simulations were performed on grids with 64^3 and 128^3 cubic mesh cells. The coarsest grid for the geometric multigrid method possessed 4^3 cubic mesh cells. The SSOR method was applied with the overrelaxation parameter $\omega = 1.25$. As initial guess for the iterative solvers, the solution from the previous discrete time was used. The availability of a good initial guess and the dominance of the system matrix by the mass matrix are the main differences to the steady-state case.

Computing times for the iterative solvers are presented in Figs. 8 and 9. On the coarser grid, a speed-up can be observed only until 8 processors and on the finer grid until 12–16 processors. With respect to the efficiency of the iterative solvers, the same observations can be made as in steady-state Example 2. Using the V(2,2) cycle or two V(1,1) cycles in the BoomerAMG gave the same order of computing times as the presented times of the V(1,1) cycle. The same statement holds true for the number of iterations per time step, e.g., on the finer grid there were up to 45 for PETSc FGMRES with SSOR and around 1–3 for PETSc FGMRES with BoomerAMG and ParMooN FGMRES with MG.

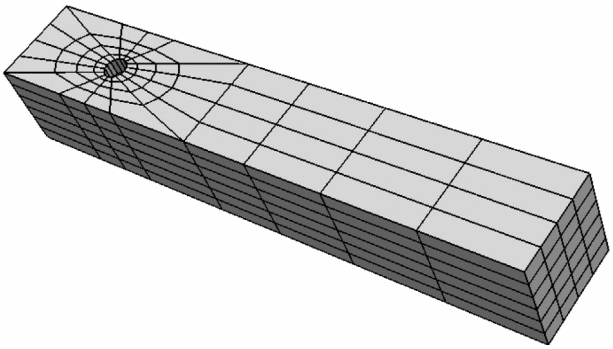

Fig. 10. Example 4, initial grid (level 0).

Example 4 (*Steady-State Incompressible Navier–Stokes Equations*)**.** This example considers the benchmark problem of the flow around a cylinder defined in [37]. The steady-state Navier–Stokes equations are given by

$$\begin{aligned} -\nu\Delta\boldsymbol{u} + (\boldsymbol{u}\cdot\nabla)\boldsymbol{u} + \nabla p &= \boldsymbol{0} \quad \text{in } \Omega, \\ \nabla\cdot\boldsymbol{u} &= 0 \quad \text{in } \Omega, \end{aligned}$$

where $\boldsymbol{u}$ is the velocity, p the pressure, $\nu = 10^{-3}$ is the dimensionless viscosity, and Ω is the domain given by

$$\Omega = \left\{(0, 2.55) \times (0, 0.41) \setminus \left\{(x-0.5)^2 + (y-0.2)^2 \leq 0.05^2\right\}\right\} \times (0, 0.41).$$

At the inlet $x = 0$, the velocity was prescribed by

$$\boldsymbol{u} = \left(\frac{7.2}{0.41^4} yz(0.41-y)(0.41-z), 0, 0\right)^T,$$

at the outlet $x = 2.55$, the do-nothing boundary condition was imposed and on all other boundaries, the no-slip boundary condition was used. The flow field exhibits vortices behind the cylinder.

The Navier–Stokes equations were discretized with the popular Q_2/P_1^{disc} (continuous piecewise triquadratic velocity, discontinuous piecewise linear pressure) pair of finite element spaces on a hexahedral grid, see Fig. 10 for the initial grid. Simulations were performed on level 2 (776 160 velocity d.o.f.s, 122 800 pressure d.o.f.s) and level 3 (6 052 800 velocity d.o.f.s, 983 040 pressure d.o.f.s). The nonlinear problem was linearized with a Picard iteration (fixed point iteration). In each step of this iteration, the iterative solvers reduced the Euclidean norm of the residual vector by at least the factor 10 before performing the next Picard iteration. The Picard iteration was stopped if the Euclidean norm of the residual vector was less than 10^{-8}.

In the geometric multigrid preconditioner, the so-called mesh-cell oriented Vanka smoother was used, e.g., see [22,21]. This smoother is a block Gauss–Seidel method that solves a local system in each mesh cell. The multigrid preconditioner was applied with the V(2,2)-cycle.

The linearization and used discretization of the incompressible Navier–Stokes equations require the solution of a linear saddle point problem in each Picard iteration. We tried several options for calling an iterative solver from PETSc for such problems, as well based on the coupled system as on Schur complement approaches. Only with a Schur complement approach and the least square commutator (LSC) preconditioner (`-ksp_type fgmres -pc_type fieldsplit -pc_fieldsplit_type schur -pc_fieldsplit_schur_factorization_type upper -fieldsplit_1_pc_type lsc -fieldsplit_1_lsc_pc_type lu -fieldsplit_0_ksp_type preonly -fieldsplit_0_pc_type lu`), reasonable computing times could be obtained, at least for the coarser grid, see Fig. 11. Like for the scalar problems, the sparse direct solver performed by far the least efficient. The solver with the geometric multigrid method was more efficient by around a factor of 20 compared with the iterative solver called from PETSc. For the finer grid, the numerical studies were restricted to the geometric multigrid preconditioner, see Fig. 12. It can be observed that increasing the number of processors from 2 to 20 reduces the computing time by a factor of around 6.6.

7. Summary

This paper presented some aspects of the remanufacturing of an existing research code, in particular with respect to its parallelization. All distinct features of the predecessor code could be incorporated in a straightforward way into the modernized code ParMooN. The efficiency of the most complex method in the parallel implementation, the geometric multigrid method, was assessed against some parallel solvers that are available in external libraries. The major conclusions of this assessment are twofold. For scalar convection–diffusion–reaction equations, the geometric multigrid preconditioner was similarly efficient as an iterative solver from the PETSc library. The larger the problems became, the better was its efficiency in comparison with the external solver. For linear saddle point problems, arising in the simulation of the

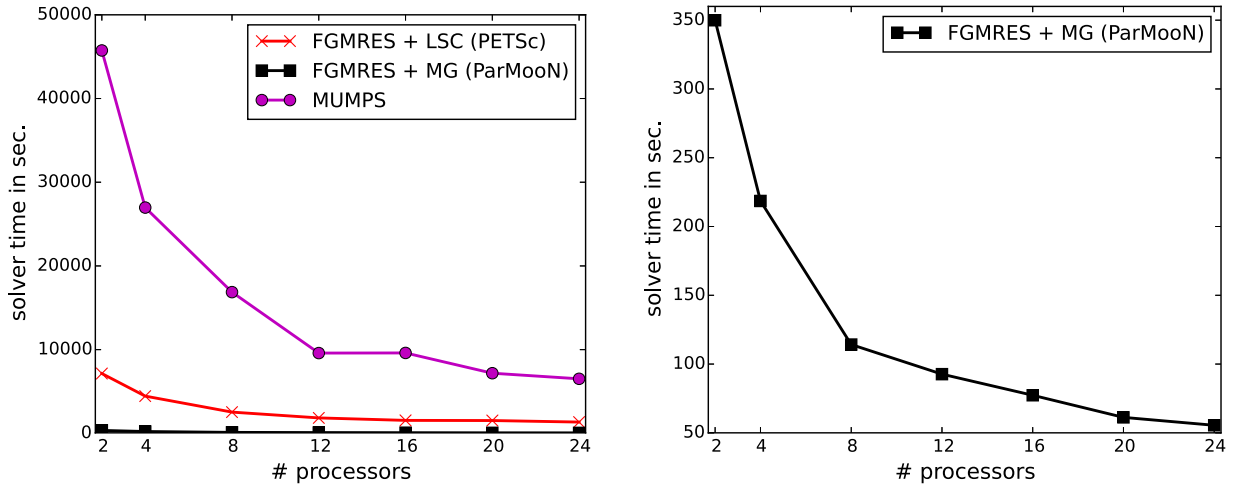

Fig. 11. Example 4, solver times on refinement level 2.

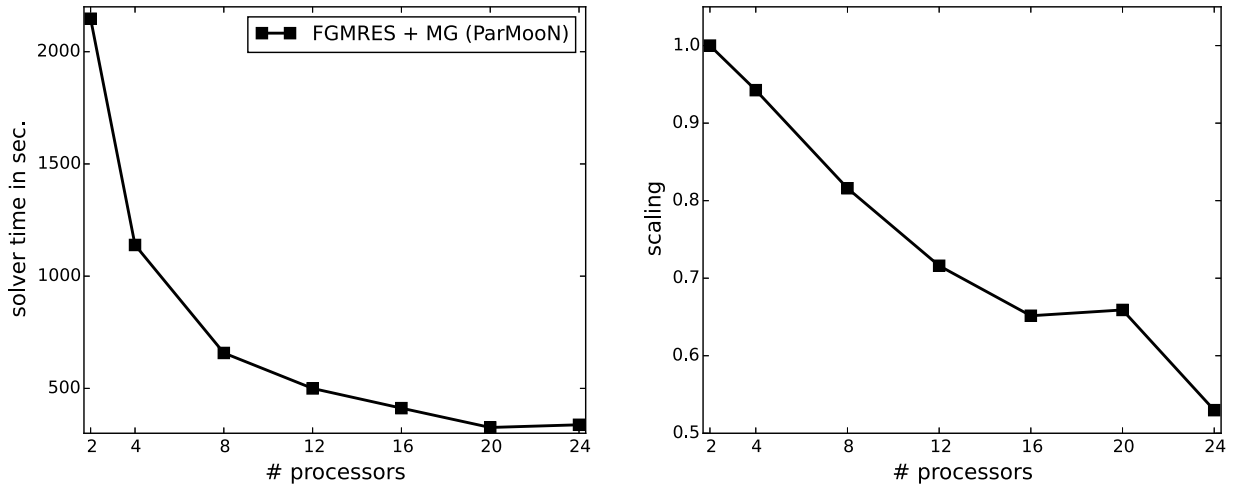

Fig. 12. Example 4, solver times and scaling on refinement level 3. The scaling is computed by $2 \cdot t_2/(p \cdot t_p)$, where p is the number of processors and t_p the corresponding time from the left picture.

incompressible Navier–Stokes equations, we could not find so far any external solver that proved to be nearly as efficient as the geometric multigrid preconditioner.

On the one hand, we keep on working at improving the efficiency of the geometric multigrid preconditioner. On the other hand, we continue to assess external libraries with respect to efficient solvers for linear saddle point problems, which can be used in situations where a multigrid hierarchy is not available. In addition, an assessment as presented in this paper on another widely available hardware platform, namely small clusters of processors, is planned.

Acknowledgments

The work of Najib Alia has been supported by a funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie Grant Agreement No. 675715 (MIMESIS). The work of Felix Anker has been supported by grant Jo329/10-2 within the DFG priority programme 1679: Dynamic simulation of interconnected solids processes. The work of Sashikumaar Ganesan has partially been supported by the Naval Research Board, DRDO, India through the grant NRB/4003/PG/368. The work of Volker John has partially been supported by grant Jo329/10-2 within the DFG priority programme 1679: Dynamic simulation of interconnected solids processes.

References

[1] Volker John, Gunar Matthies, MooNMD—a program package based on mapped finite element methods, Comput. Vis. Sci. 6 (2–3) (2004) 163–169.
[2] Scopus. https://www.scopus.com/record/display.uri?src=s&origin=cto&ctoId=CTODS_701493876&stateKey=CTOF_701493878&eid=2-s2.0-7444244347 (accessed 13.09.16).
[3] W. Bangerth, D. Davydov, T. Heister, L. Heltai, G. Kanschat, M. Kronbichler, M. Maier, B. Turcksin, D. Wells, The `deal.II` library, version 8.4, J. Numer. Math. 24 (2016).
[4] Martin Alns, Jan Blechta, Johan Hake, August Johansson, Benjamin Kehlet, Anders Logg, Chris Richardson, Johannes Ring, Marie Rognes, Garth Wells, The fenics project version 1.5, Arch. Numer. Softw. 3 (100) (2015).
[5] Andreas Dedner, Robert Klöfkorn, Martin Nolte, Mario Ohlberger, A generic interface for parallel and adaptive discretization schemes: abstraction principles and the DUNE-FEM module, Computing 90 (3–4) (2010) 165–196.
[6] Markus Blatt, Ansgar Burchardt, Andreas Dedner, Christian Engwer, Jorrit Fahlke, Bernd Flemisch, Christoph Gersbacher, Carsten Gräser, Felix Gruber, Christoph Grüninger, Dominic Kempf, Robert Klöfkorn, Tobias Malkmus, Steffen Müthing, Martin Nolte, Marian Piatkowski, Oliver Sander, The distributed and unified numerics environment, version 2.4, Arch. Numer. Softw. 4 (100) (2016) 13–29.
[7] OpenFOAM. OpenFOAM Web page, 2016. http://openfoam.org/.
[8] F. Hecht, New development in freefem++, J. Numer. Math. 20 (3–4) (2012) 251–265.
[9] Alfonso Caiazzo, Volker John, Ulrich Wilbrandt, On classical iterative subdomain methods for the Stokes-Darcy problem, Comput. Geosci. 18 (5) (2014) 711–728.
[10] Volker John, Julia Novo, On (essentially) non-oscillatory discretizations of evolutionary convection–diffusion equations, J. Comput. Phys. 231 (4) (2012) 1570–1586.
[11] Ellen Schmeyer, Róbert Bordás, Dominique Thévenin, Volker John, Numerical simulations and measurements of a droplet size distribution in a turbulent vortex street, Meteorol. Z. 23 (4) (2014) 387–396. 09.
[12] S. Hysing, S. Turek, D. Kuzmin, N. Parolini, E. Burman, S. Ganesan, L. Tobiska, Quantitative benchmark computations of two-dimensional bubble dynamics, Internat. J. Numer. Methods Fluids 60 (11) (2009) 1259–1288.
[13] Volker John, Petr Knobloch, Simona B. Savescu, A posteriori optimization of parameters in stabilized methods for convection–diffusion problems—Part I, Comput. Methods Appl. Mech. Engrg. 200 (41–44) (2011) 2916–2929.
[14] Howard C. Elman, Alison Ramage, David J. Silvester, IFISS: a computational laboratory for investigating incompressible flow problems, SIAM Rev. 56 (2) (2014) 261–273.
[15] Philippe G. Ciarlet, The Finite Element Method for Elliptic Problems, in: Studies in Mathematics and its Applications, vol. 4, North-Holland Publishing Co., Amsterdam, 1978.
[16] James Ahrens, Berk Geveci, Charles Law, ParaView: An end-user tool for large data visualization, in: Visualization Handbook, Elsevier, 2005.
[17] Utkarsh Ayachit, The ParaView Guide: A Parallel Visualization Application, Kitware, Inc., 2015.
[18] S. Ganesan, V. John, G. Matthies, R. Meesala, S. Abdus, U. Wilbrandt, An object oriented parallel finite element scheme for computing PDEs: Design and implementation, in: IEEE 23rd International Conference on High Performance Computing Workshops, (HiPCW), IEEE, Hyderabad, 2016, pp. 106–115.
[19] MPI-Forum. Mpi: A message-passing interface standard, version 3.1. Technical Report, University of Tennessee, Knoxville, Tennessee, 2015.
[20] George Karypis, Vipin Kumar, MeTis: Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 4.0, 2009. http://www.cs.umn.edu/~metis.
[21] Volker John, Finite Element Methods for Incompressible Flow Problems, in: Springer Series in Computational Mathematics, vol. 51, Springer-Verlag, Berlin, 2016.
[22] Volker John, Higher order finite element methods and multigrid solvers in a benchmark problem for the 3D Navier–Stokes equations, Internat. J. Numer. Methods Fluids 40 (6) (2002) 775–798.
[23] F. Schieweck, A general transfer operator for arbitrary finite element spaces. Preprint 00-25, Fakultät für Mathematik, Otto-von-Guericke-Universität Magdeburg, 2000.
[24] Youcef Saad, A flexible inner-outer preconditioned GMRES algorithm, SIAM J. Sci. Comput. 14 (2) (1993) 461–469.
[25] Patrick R. Amestoy, Iain S. Duff, Jean-Yves L'Excellent, Jacko Koster, A fully asynchronous multifrontal solver using distributed dynamic scheduling, SIAM J. Matrix Anal. Appl. 23 (1) (2001) 15–41. (electronic).
[26] Patrick R. Amestoy, Abdou Guermouche, Jean-Yves L'Excellent, Stéphane Pralet, Hybrid scheduling for the parallel solution of linear systems, Parallel Comput. 32 (2) (2006) 136–156.
[27] Van Emden Henson, Ulrike Meier Yang, BoomerAMG: a parallel algebraic multigrid solver and preconditioner, Appl. Numer. Math. 41 (1) (2002) 155–177.
[28] Howard C. Elman, David J. Silvester, Andrew J. Wathen, Finite Elements and Fast Iterative Solvers: With Applications in Incompressible Fluid Dynamics, second ed., in: Numerical Mathematics and Scientific Computation, Oxford University Press, Oxford, 2014.
[29] Satish Balay, Shrirang Abhyankar, Mark F. Adams, Jed Brown, Peter Brune, Kris Buschelman, Lisandro Dalcin, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Karl Rupp, Barry F. Smith, Stefano Zampini, Hong Zhang, Hong Zhang, PETSc Web page, 2016. http://www.mcs.anl.gov/petsc.
[30] Satish Balay, Shrirang Abhyankar, Mark F. Adams, Jed Brown, Peter Brune, Kris Buschelman, Lisandro Dalcin, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Karl Rupp, Barry F. Smith, Stefano Zampini, Hong Zhang, Hong Zhang, PETSc users manual. Technical Report ANL-95/11 - Revision 3.7, Argonne National Laboratory, 2016.
[31] Satish Balay, William D. Gropp, Lois Curfman McInnes, Barry F. Smith, Efficient management of parallelism in object oriented numerical software libraries, in: E. Arge, A.M. Bruaset, H.P. Langtangen (Eds.), Modern Software Tools in Scientific Computing, Birkhäuser Press, 1997, pp. 163–202.
[32] Matthias Augustin, Alfonso Caiazzo, André Fiebach, Jürgen Fuhrmann, Volker John, Alexander Linke, Rudolf Umla, An assessment of discretizations for convection-dominated convection–diffusion equations, Comput. Methods Appl. Mech. Engrg. 200 (47–48) (2011) 3395–3409.
[33] P.W. Hemker, A singularly perturbed model problem for numerical computation, J. Comput. Appl. Math. 76 (1–2) (1996) 277–285.
[34] Alexander N. Brooks, Thomas J.R. Hughes, Streamline upwind/Petrov-Galerkin formulations for convection dominated flows with particular emphasis on the incompressible Navier-Stokes equations, Comput. Methods Appl. Mech. Engrg. 32 (1–3) (1982) 199–259.
[35] T.J.R. Hughes, A. Brooks, A multidimensional upwind scheme with no crosswind diffusion, in: Finite Element Methods for Convection Dominated Flows (Papers, Winter Ann. Meeting Amer. Soc. Mech. Engrs., New York, 1979), in: AMD, vol. 34, Amer. Soc. Mech. Engrs. (ASME), New York, 1979, pp. 19–35.
[36] Volker John, Petr Knobloch, On spurious oscillations at layers diminishing (SOLD) methods for convection–diffusion equations. I. A review, Comput. Methods Appl. Mech. Engrg. 196 (17–20) (2007) 2197–2215.
[37] M. Schäfer, S. Turek, Benchmark computations of laminar flow around a cylinder. (With support by F. Durst, E. Krause and R. Rannacher), in: Flow simulation with High-Performance Computers II. DFG Priority Research Programme Results 1993–1995, Wiesbaden: Vieweg, 1996, pp. 547–566.