Adv Comput Math (2015) 41:1145–1177
DOI 10.1007/s10444-015-9405-4

Software concepts and numerical algorithms for a scalable adaptive parallel finite element method

T. Witkowski · S. Ling · S. Praetorius · A. Voigt

Received: 6 December 2013 / Accepted: 7 January 2015 /
Published online: 29 January 2015

Abstract An efficient implementation of an adaptive finite element method on distributed memory systems requires an efficient linear solver. Most solver methods, which show scalability to a large number of processors make use of some geometric information of the mesh. This information has to be provided to the solver in an efficient and solver specific way. We introduce data structures and numerical algorithms which fulfill this task and allow in addition for an user-friendly implementation of a large class of linear solvers. The concepts and algorithms are demonstrated for global matrix solvers and domain decomposition methods for various problems in fluid dynamics, continuum mechanics and materials science. Weak and strong scaling is shown for up to 16.384 processors.

Keywords Software concepts · Adaptive finite elements · High performance computing

Mathematics Subject Classification (2010) 65M60 · 65Y05 · 65Y20

1 Introduction

General purpose finite element toolboxes have become more and more important in computational science. This results from an increase in complexity of the mathe-

Communicated by: Charlie Elliott

Present Address:
T. Witkowski · S. Ling · S. Praetorius · A. Voigt (✉)
Institute of Scientific Computing, TU Dresden, 01062 Dresden, Germany
e-mail: axel.voigt@tu-dresden.de

T. Witkowski
e-mail: thomas.witkowski@gmx.de

matical models to be solved, an increase in complexity of the numerical algorithms to be used and a decrease in the available time for a specific research project. All together make it more and more inefficient to develop simulation software only for a specific problem. Various general purpose simulation software packages exist. We refer to deal.II [9], FEniCS/DOLFIN [32], DUNE [16], Hermes [44], libMesh [26] or AMDiS [45]. They all provide a parallel implementation of an adaptive finite element method and offer a more or less user friendly way to solve general partial differential equations on massively parallel hardware systems. To fulfill both requirements: to be easy to use and to be efficient, leads to several problems and several compromises, either on the user-friendliness or the efficiency are often made. We try to overcome this discrepancy and demonstrate how appropriate data structures can be used to implement an efficient massively parallel adaptive finite element method in a user friendly way and demonstrate its efficiency on various examples.

From a software point of view the finite element method is a composition of weakly coupled algorithms. It requires data structures for the finite elements, the mesh and some data on it, algorithms to modify the mesh, assembling procedures, solvers for the resulting systems of linear equations and error estimators. Some of these algorithms require only local information, as assembling which can be done on all elements independently of each other. It thus can be perfectly parallelized and will not be considered here. Error estimation is likewise an element local procedure. Only the marker strategy, which chooses a subset of the elements to be refined or coarsenes requires some global information and thus must be adjusted for parallel computations. This is already considered, e.g., in [12] and will also not be discussed here. The other algorithms are more crucial to ensure scalability, which strongly depends on the distributed matrix and vector data structures. We will make use of PETSc [6, 7], a widely used package that supports distributed linear algebra data structures and algorithms, and are especially concerned with scalability of the linear solver. In [8] it was shown that deal.II has good weak and strong scaling of up to 16,384 processors. However, the main bottleneck in the shown benchmarks is the linear solver, which shows a break down for the largest of the presented simulations. The situation is similar to the results of [23], which show parallel scaling of DOLFIN up to 1,024 processors. The algorithms for error estimation and mesh adaptivity scale very well, but the linear solver does not for a larger number of processors. The problem in both examples is the communication between the mesh data structure and the solver.

A solution to this problem is to create data structures that allow information flow from the mesh to the solver in an efficient and general way. As we consider a general purpose finite element toolbox with different requirements for the linear solver, depending on the specific problem to be solved, we must allow for a general solver interface that makes it possible to implement a large class of linear solvers. All the concepts presented here are implemented in the finite element toolbox AMDiS [45].

The paper is structured as follows: Section 2 provides information of the software concepts that are used in our approach for the implementation of parallel distributed meshes. We describe in detail how specific information required by the solver can be generated. The data structures and algorithms are used in Section 3 for a general interface for global matrix and domain decomposition methods. Various examples demonstrate the efficiency of the approach. We first show scalability of the mesh

adaptation on an example of dendritic growth, in which the mesh has to be adapted in every time step and has to be redistributed frequently. In addition, a specific global matrix Navier-Stokes solver [24, 40] is used to demonstrate the flexibility of the introduced concepts on a channel flow problem with dissolved particles. Besides global matrix solvers an efficient implementation of the FETI-DP method is used to exemplify the concepts. The efficiency of all these methods is demonstrated by showing weak and strong scaling on up to 16,384 processors. Finally Section 4 gives a detailed discussion and shows limitations of the presented algorithms.

2 Software concepts for distributed meshes

A specific partial differential equation, possible initial and boundary conditions, appropriate basis functions, and a mesh representing the geometry are the essential input data for any finite element toolbox. In this work we assume that the geometry can be sufficiently represented by a *coarse mesh*. We further assume mesh refinement to be necessary to solve the problem with a discretization error below a given error bound. Throughout this paper we use meshes consisting of triangles and tetrahedrons, which are refined by *bisectioning*.

2.1 Parallel adaptive finite element method

Figure 1 illustrates the basic building blocks of a h-adaptive finite element method. In the first step the *assembler* creates a matrix-vector representation of the equation on the given mesh using the predefined basis functions.

This is a local integration process. An appropriate *solver* is used to solve the resulting system of equations. An *error estimator* is used to estimate the error of the discrete solution. If it is above a threshold, a *marker strategy* is used to identify parts of the mesh to be adapted in order to decrease the error in the next iteration. The

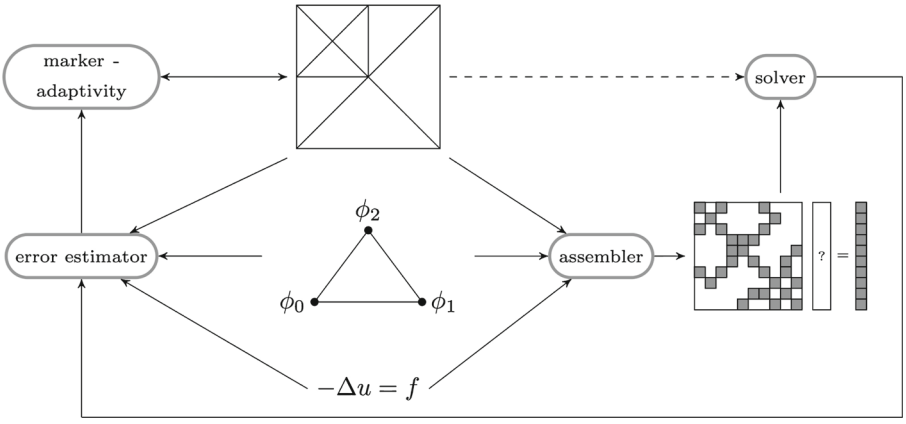


Fig. 1 Sketch of the finite element method with local mesh adaptivity. The dashed line, representing the information flow from the mesh to the linear solver, is optional

same concept might be used to mark parts of the mesh to be coarsened if the error estimate is below a threshold in this region to further reduce the computational cost. This loop is continued until the error estimate drops below a given threshold everywhere. Other adaptive concepts, such as p- or hp-adaptivity will also fit (with small modifications) into this abstract illustration.

The building blocks can easily be implemented on a single processor or shared memory system but require modifications on distributed memory systems. We need to consider a distributed mesh and distributed matrices and vectors. Parallelization of data structures and algorithms from linear algebra is considered, e.g., in [21]. As already mentioned assembler, error estimator and marker strategy are quite simple to parallelize, as all of them are mostly local mesh procedures and require almost no communication between processors, see e.g. [8, 26]. The situation is totally different for solving the system of equations, as the local solution of one processor is potentially governed by the data of all other processors. During the last decades, many sequential linear solver methods have been redefined for parallel computing, such as iterative Krylov subspace methods [22, 39], or multigrid methods [43] or have been developed specifically for parallel computing, such as domain decomposition methods [37]. Most solver methods, which show scalability to a large number of processors make use of some geometrical information of the mesh. Geometric multigrid methods for example require information on the hierarchical decomposition of the mesh, iterative substructuring methods require geometrical information of the degree of freedoms (DOFs) that composite the interior boundaries between subdomains. This information is available in the data structure storing the mesh and needs to be provided to the solver in an efficient and solver specific way.

2.2 Formal definitions

We provide some formal definitions: In what follows, $\Omega \subset \mathbb{R}^d$ with $d = 2, 3$, is an arbitrary domain and $\partial\Omega$ denotes its boundary. The boundary splits into a Dirichlet boundary part Γ_D and a Neumann boundary part Γ_N, with $\Gamma_D \cup \Gamma_N = \partial\Omega$ and $\Gamma_D \cap \Gamma_N = \emptyset$. We do not consider Robin and periodic boundary conditions explicitly, as they can be handled in the same way. In this work, we restrict to non-overlapping decompositions:

Definition 1 A set $\Omega_1, \ldots, \Omega_p$ of open subregions of Ω is a *non-overlapping decomposition* of the domain Ω, if $\overline{\Omega} = \cup_{i=1}^{p} \overline{\Omega_i}$ and $\Omega_i \cap \Omega_j = \emptyset$ for all $1 \leq i < j \leq p$.

Each subdomain is handled by exactly one processor, and each processor handles exactly one subdomain. Non-overlapping domain decomposition naturally leads to the splitting of each subdomain into subdomain's *interior* part and *interior boundaries*, i.e., element segments which intersect with other subdomain boundaries. Data is communicated only along the interior boundaries, which are defined as follows:

Definition 2 The boundary of a subdomain Ω_i is denoted by $\partial\Omega_i = \Gamma_{D_i} \cup \Gamma_{N_i} \cup \mathcal{I}_i$ with $\Gamma_{D_i} \subset \Gamma_D$, $\Gamma_{N_i} \subset \Gamma_N$ and $\mathcal{I}_i$ is called the *interior boundary* of subdomain i. Furthermore, $\mathcal{I}_{ij} = \mathcal{I}_i \cap \mathcal{I}_j$ denotes the interior boundary between the subdomains i

and j and $\mathcal{I} = \bigcup_{i=1}^{p} \mathcal{I}_i$ is the set of all interior boundaries in Ω. In many situations we are interested in the decomposition of the sets of interior boundaries into sets of vertices, edges, and faces:

$$\mathcal{I}_i = \mathcal{I}_i^V \cup \mathcal{I}_i^E \cup \mathcal{I}_i^F \text{ and } \mathcal{I}_{ij} = \mathcal{I}_{ij}^V \cup \mathcal{I}_{ij}^E \cup \mathcal{I}_{ij}^F,$$

where $\mathcal{I}_i^F$ and $\mathcal{I}_{ij}^F$ are empty in 2D.

Figure 2 illustrates this concept on a simple 2D example with three subdomains. Note that also subdomain 1 and 2 share an interior boundary that consists of one vertex. Based on the concept of interior boundaries, we define neighborhood relations in the natural way as:

Definition 3 Subdomain j is called to be a *neighbor* of subdomain i if $\mathcal{I}_{ij} \neq \emptyset$. The set of neighbors for a subdomain i is defined by:

$$neigh_i = \{j \mid 1 \leq j \leq p, j \neq i, \mathcal{I}_{ij} \neq \emptyset\}$$

For both, formal definitions and the implementation, it is required to identify for every substructure, e.g. a vertex, an edge or a face, of all coarse mesh elements the subdomains which contain this substructure. Therefore, we establish the following auxiliary definitions, which are exemplified in Fig. 2:

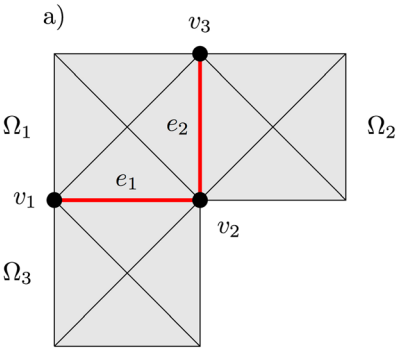
Definition 4 Let b be an arbitrary vertex, edge or face of a coarse mesh element in Ω. Then we define $\mathcal{W}b$ to be an index set defined as

$$\mathcal{W}b = \{i \mid b \in \Omega_i\},$$

and the *degree* of b is defined by

$$degree(b) = |\mathcal{W}b|.$$

a)



b)

$\mathcal{I}_{ij}$	$i=1$	$i=2$	$i=3$
$j=1$	-	e_2,v_2,v_3	e_1,v_1,v_2
$j=2$	e_2,v_2,v_3	-	v_2
$j=3$	e_1,v_1,v_2	v_2	-

c)

	$W(\cdot)$	$degree(\cdot)$	$owner(\cdot)$
e_1	1,3	2	2
e_2	1,2	2	1
v_1	1,3	2	2
v_2	1,2,3	3	2
v_3	1,2	2	1

Fig. 2 **a**) Non-overlapping domain decomposition in 2D with three subdomains **b**) interior subdomain decomposition **c**) definition of the degree and ownership on the interior boundary segments

As two or more subdomains may intersect at some interior boundary segments, we have to define ownership for these segments:

Definition 5 We call a subdomain i to be the *owner* of a boundary segment $b \in \mathcal{I}_i$, if there is no other subdomain with a higher index number that contains this boundary segment:

$$owner(b) = i, \forall j : 1 \leq j \leq p \wedge j \neq i \wedge b \in \mathcal{I}_j \Rightarrow j < i$$

This definition is somehow arbitrary and other could be possible. The only requirements are that it is unique, consistent and simple to compute. Figure 2 exemplifies this concept.

2.3 Class structure

In the previous section we have identified the need for an appropriate general information flow between the mesh structure and the solver method. The following mesh information could be required:

- hierarchical decomposition of the mesh: mostly used by multigrid methods to project and prolongate the solution (or the residual) between fine and coarse meshes.
- communication pattern: most parallel solvers iterate between a local and a global solution procedure, thus it must be known which subdomain DOFs (degrees of freedom) are shared with some other subdomains and must be therefore communicated/synchronized.
- restricted sets of DOFs: especially in iterative substructuring methods [37] it is common to split the set of DOFs in multiple subsets and to define continuous global indices for these subsets. For example, this can be the set of all DOFs which relate to cross points of interior boundaries (see Section 2.6).
- geometrical information of interior boundary DOFs: defining subsets of DOFs is usually done based on geometrical information. For example, the FETI-DP method, which we describe in Section 3.2.1 to exemplify the general concepts presented in this section, must differ between DOFs that belong to vertices, edges and faces of the coarse mesh elements.

This mesh information should be created only on demand when requested by a specific solver method. Figure 3 shows a general overview of all classes that make it possible to create exactly the data required. First, the initial coarse mesh is passed to the *mesh partitioner*, which is responsible to assign each coarse mesh element to one processor. This information is used by the *mesh distributor* to move the coarse mesh elements, together with their possible adaptive refinement structure and all values which are defined on it, to the corresponding processors. As indicated by Definition 5, not only an assignment of mesh elements to processors is required, but also the ownership definition for vertices, edges and faces of all coarse mesh elements. This information is computed by the *element object database* (EL object DB), which can be used to query for the following questions: given a geometrical entity,

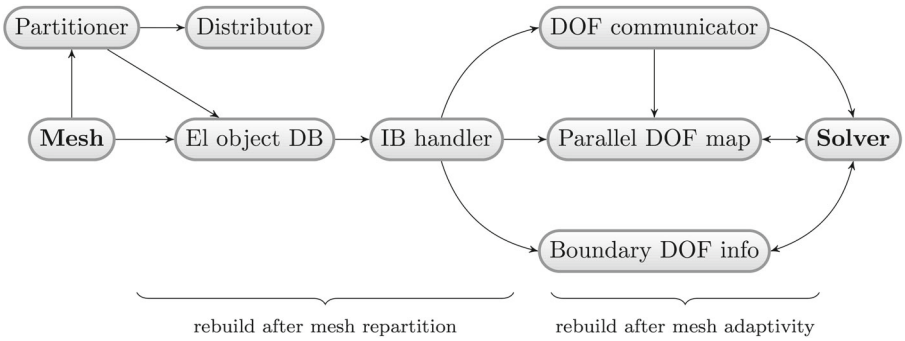


Fig. 3 Information flow between the mesh data structure and the parallel solver.

i.e., a vertex, edge or face, of a specific element: What are all elements which contain this entity? Which and how many processors contain this entity (Definition 4)? Which processor is the owner of this entity (Definition 5)? Thus, the *element object database* takes the information of the mesh partitioner and breaks it down to the level of vertices, edges and faces. This database works only on the level of the coarse mesh and no information is stored about refined elements. Furthermore, the database is stored on all processors for the whole initial mesh. We refer to Section 4 for a discussion of the limitation of this approach and how to circumvent them.

The *element object database* is mainly used to initialize the *interior boundary handler* (IB handler), which stores on each processor all the geometric entities that form its interior boundaries with other subdomains, see Fig. 2. These boundary elements are subdivided into two sets: the boundary elements that are owned by the processor and boundary elements that are part of processor's subdomain but owned by another processor. All the information is again stored on the level of the coarse mesh and thus does not change due to local mesh adaptivity. It must be rebuilt only after mesh redistribution. To establish the interior boundary handler, all processors traverse all elements of the coarse mesh and pick up all their entities which are part of an interior boundary, i.e. $degree(\cdot) > 1$, and which are owned by the processor, see Algorithm 1. Each processor then sends its list of own boundary segments to all neighboring processors, which share the same interior boundary. This ensures, that the list of boundary segments is the same, and especially in the same order, on all processors that share this interior boundary.

Up to this point, the initial coarse mesh is partitioned, the corresponding subdomains are created and all processors know which geometric entities form their interior boundaries. If the mesh is adapted new DOFs are introduced, which might be located on the interior boundaries. These DOFs are shared by at least two subdomains, and we define ownership of these DOFs in the same way as we have done it for the interior boundaries. All communication between subdomains is done on the basis of these common DOFs. To store all common DOFs, we introduce the concept of *DOF communicators*, that describe the DOF communication pattern between all subdomains. Once they have been established, they can be used for very efficient point-to-point communication. Assuming that an interior boundary handler is already initialized,

Algorithm 1 Creation of information about interior boundaries.

```
input  : db: element object database, mesh: initial coarse mesh, mpiRank: unique
         identification number of the current processor
output: intBoundOwn: set of element entities, which form the interior boundary
         that is owned by the processor, intBoundOther: set of element entities,
         which form the interior boundary that is not owned by the processor
foreach macroElement ∈ mesh do
  if db.elInSubdomain(macroElement) then
    foreach entity ∈ macroElement do
      if db.degree(entity) > 1 and db.owner(entity) == mpiRank then
        foreach rank ∈ db.W(entity) do
          if rank ≠ mpiRank then
            intBoundOwn.add(entity, rank)
        end
    end
end
foreach index ∈ neigh do
  if index < mpiRank then
    intBoundOwn.send(index)
  else
    intBoundOther.recv(index)
end
```

DOF communicators can be easily created without further communication. Each subdomain just traverses all geometric entities of all interior boundaries and collects the corresponding DOFs. As the interior boundary handler ensures that the boundary elements are in the same order on all neighboring processors, the collected DOFs directly fit together on the interior boundaries and can be used for communication. As the *DOF communicators* must be reinitialized after each mesh adaptivity, it is quite important that this procedure can be done fast and without further communication.

The *DOF communicator* has just the knowledge how to exchange data with neighboring subdomains, but it has no global DOF view. This is the main task of the *parallel DOF mapper*. It creates a mapping from local DOF indices to global indices. This mapping must be consistent, i.e., if two local DOFs in two different subdomains represent the same global DOF they must also map to the same global index. The parallel DOF mapping is described in Section 2.6.

The last concept is the *boundary DOF info* object. It can be used by a specific solver method to get geometrical information about interior boundary DOFs. This can be necessary, if, e.g., a domain decomposition method must decompose the set of interior boundary DOFs into DOFs which are part of a boundary vertex, edge or face.

2.4 Mesh structure codes for parallel mesh adaptivity

Parallel adaptive mesh refinement is e.g. considered in the distributed mesh library `p4est` [11, 13], which shows excellent weak and strong scaling for over 224,000 processors. Due to its internal mesh representation based on octrees, it is not directly usable for triangle and tetrahedron meshes. In [23] a method for tetrahedral mesh is presented which shows parallel scaling up to 1,024 processors with an

efficiency of around 80 %. However, the mesh quality is influenced by the partitioning. All these algorithms ensure coincident interior boundaries which requires communication between neighboring processors. We present a method for parallel mesh adaptivity where the communication is minimized and parallel scaling efficiency is only limited by the efficiency of the used MPI library. One of our key concepts for efficient distributed adaptive meshes are *mesh structure codes*, see [38]. The main idea is to traverse the binary trees, that represent refined elements, in a unique way, e.g. using pre-order traverse, and to denote each leaf element by 0 and a non-leaf element, which is further refined, by 1. Thus, a sequence of 0 and 1 uniquely represents the refinement structure of an element. Figure 4 shows the construction of a binary code for a refined triangle. Here, the code 1101000 (decimal value 104) can be used to reconstruct the refinement structure of this element. The main advantage of mesh structure codes is that they can easily be created and their communication between processors is very cheap. In this way, mesh structure codes are used by the mesh distributor to transfer the refinement structure of elements from one processor to another. See Section 2.5 for more details.

We extend the concept to *substructure codes*. A substructure code does not store the refinement structure of one coarse mesh element but only of one of its substructures, i.e. an edge of a triangle. Substructure codes are used to check if two elements have the same refinement structure along their substructures, i.e., if they fit together on interior boundaries between two subdomains.

The creation of substructure codes is based on a modified pre-order traverse that works on element's substructures. The algorithm, see Algorithm 2, starts on a coarse mesh element and traverses recursively only the children that intersect with a given substructure of the coarse mesh element. We must care about the order of the traversed children. As it is shown in Fig. 5, the left children of $\mathcal{T}_1$ is the neighbor of the right children of $\mathcal{T}_2$. Thus, to compare the substructure codes of both triangles

Algorithm 2 *preOrderTraverse*(el, t, r): Pre-order traverse on substructures such as vertex, edge or face.

```
input  : element el, substructure t, reverse mode r
output: substructure code c
if isLeaf(el) then
    c += 0
else
    c += 1
    el0 = getChild(el, 0)
    el1 = getChild(el, 1)
    if r then
        swap(el0, el1)
    if contains(el0, t) then
        c += preOrderTraverse(el0, t, r)
    if contains(el1, t) then
        c += preOrderTraverse(el1, t, r)
end
```

along their neighboring edge, one of these codes must be created in reverse order. This means, while traversing the element hierarchy, left and right child of an element are swapped.

Using the *interior boundary handler* it is straightforward to define an algorithm that iteratively adapts the local subdomains until all of them coincide along their interior domains. This procedure is defined in Algorithm 3. Every processor creates substructure codes for all interior boundary segments it owns. These codes are sent to the neighboring processors where they can be directly used to check if the mesh structures are the same at this edge or face. If this is not the case, the substructure code can directly be used to refine the corresponding element. This loop must be repeated as long as all processors accept the received substructure code. In all of our simulations, even in cases which require the mesh to be changed in every timestep, the mesh adaption algorithm terminates in a few iterations. For an optimal scaling the number of iterations should be independent of the number of subdomains. Theoretically this cannot be achieved as the number of iterations increase with $\mathcal{O}(\log(p))$, with p the number of subdomains. A situation where this might be observed is when a very localized mesh refinement is done within only one subdomain. In general, however we see good scaling, see Section 3.1.2.

Algorithm 3 Parallel mesh adaption. The only global communication is the MPI reduction on the variable `changeOnRank` to synchronize the loop.

```
bound = ownBound ∪ otherBound
repeat
    changeOnRank = false
    foreach (obj, rank) ∈ bound do
        if isEdge(obj) or isFace(obj) then
            c0 = preOrderTraverse(obj)
            send c0 to rank
            recv c1 from rank
            if c0 isequal c1 then
                adapt (obj, c1)
                changeOnRank = true
            end
        end
    end
    mpi reduce on changeOnRank
until changeOnRank == false
```

2.5 Mesh distribution

A *mesh distributor* is used to move coarse mesh elements, and all data on them, from one processor to another. Once a coarse mesh element has been moved from

processor A to processor B, processor B must reconstruct the refinement structure. Therefore processor A creates mesh structure codes for all coarse mesh elements that must be reconstructed on processor B. When processor B receives these mesh structure codes, it has first to create the corresponding coarse mesh element on its local subdomain and use the mesh structure code to reconstruct its refinement structure. Besides reconstruction of the coarse mesh element refinement structure, reconstruction of the DOF vectors is also required and handled by the mesh distributor. Not only the element structure must be communicated, but also the values that are defined on them. For this, we make use of *value mesh structure codes* that are used for both, the reconstruction of elements and DOF vectors defined on them.

As the mesh structure code defines in a unique way not only the final structure of the element but also the order of newly created DOFs on it, a vector of values can be used to restore a DOF vector on this element. The length corresponds to the number of vertices of the initial element plus the number of 1s in the mesh structure code. This functionality of reconstructing an element refinement structure and a DOF vector is shown in Algorithm 4.

Algorithm 4 $reconstruct(el, code, valueCode, dofVec)$: Reconstruction of elements and DOF vectors using value mesh structure codes

```
input: element el, mesh structure code code, value structure code
       valueCode, DOF vector dofVec
if isMacroElement(el) then
    foreach dof ∈ el do
        dofVec[dof] = valueCode.next()
    end
end
if code.next() == 1 then
    bisect(el, newDof)
    dofVec[newDof] = valueCode.next()
    reconstruct(getChild(el, 0), code, valueCode, dofVec)
    reconstruct(getChild(el, 1), code, valueCode, dofVec)
end
```

After mesh redistribution, the following data structures must be rebuilt: interior boundary data, DOF communicators and all requested parallel DOF mappings. All algorithms for mesh redistribution require only point-to-point communication. The same holds for rebuilding the interior boundary data and DOF communicators. The only global communication required in mesh redistribution is hidden in the creation of parallel DOF mappings, see Section 2.6.

2.6 Parallel DOF mapping

DOFs in domain Ω_i are enumerated with a continuous index set $1, \ldots, d_i$. For the solver method the subdomain matrices and vectors must be related such that local

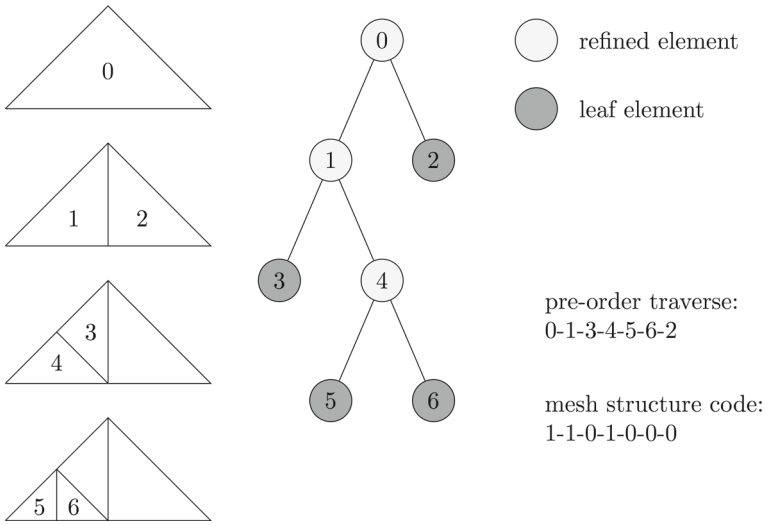


Fig. 4 Mesh structure code of an adaptively refined triangle

DOF indices of two different subdomains which correspond to the same global DOF are related to each other. This is the main task of the *parallel DOF mapping*.

For a general solver method it is important that these mappings can also be established on subsets of local and global DOFs. In Fig. 6 four subdomains are shown. Each subdomain has five DOFs and there are 13 global DOFs. A parallel DOF mapping for all DOFs would map on each processor from the set of local DOF indices $1, \ldots, 5$ to the global set $1, \ldots, 13$. Figure 6 shows the situation when a solver requires a local to global DOF mapping only for the interior boundary DOFs. In this case, a parallel DOF mapping is a partial mapping from local DOF indices $1, \ldots, 5$ to the global set of all interior boundary DOF indices $1, \ldots, 5$. Figure 6 shows this mapping for subdomain Ω_3.

We consider a node as a container for DOFs. A vertex can be a node, but nodes can also occur on element edges, faces or in the interior of an element. For scalar valued partial differential equations each node contains exactly one DOF. In this case both terms are equivalent. But for vector valued equations nodes may contain more than one DOF. The number of DOFs per node can also vary, if different finite element spaces are used for the variables.

For each subdomain we define $\mathcal{D}_i = \{1, \ldots, d_i\}$ to be the set of all DOF indices in subdomain Ω_i. The subset $\overline{\mathcal{D}}_i$ contains all DOF indices that are owned by processor i. We denote with $\overline{n}_i = |\overline{\mathcal{D}}_i|$ the number of DOF indices owned by processor i. To simplify the following definitions and the implementation of the corresponding algorithms we assume that $\mathcal{D}_i$ is sorted containing first all DOFs owned by the processor of subdomain Ω_i and followed by the other DOF indices. Consequently, $\overline{\mathcal{D}}_i$ is also a continuous set of indices starting with 1. To relate DOFs on interior boundaries, we define the mapping $\mathcal{R}$ as follows:

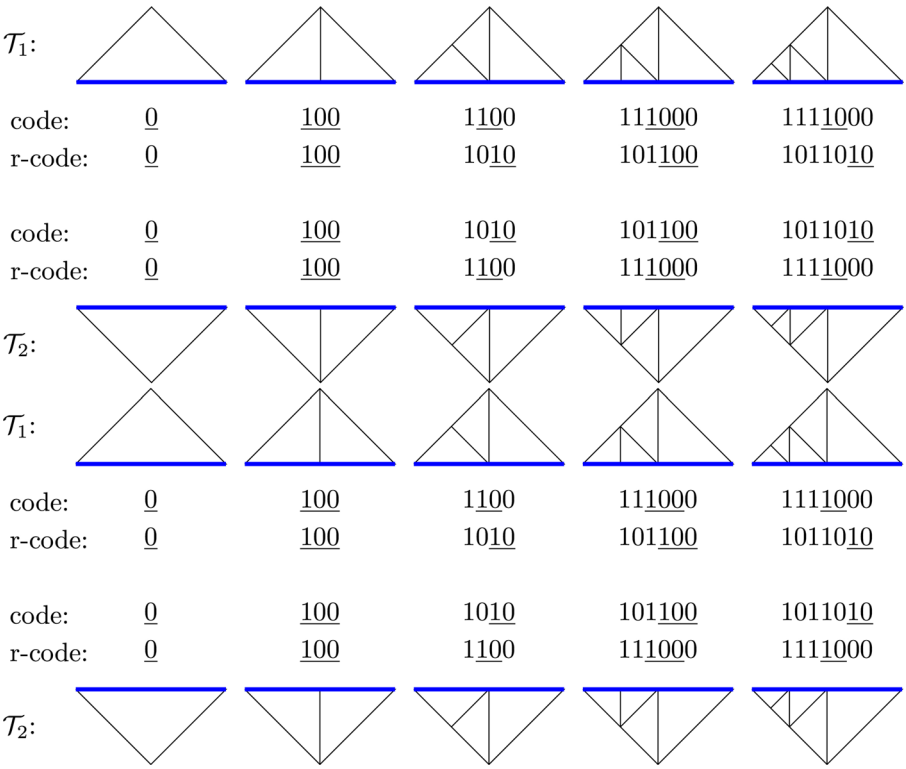


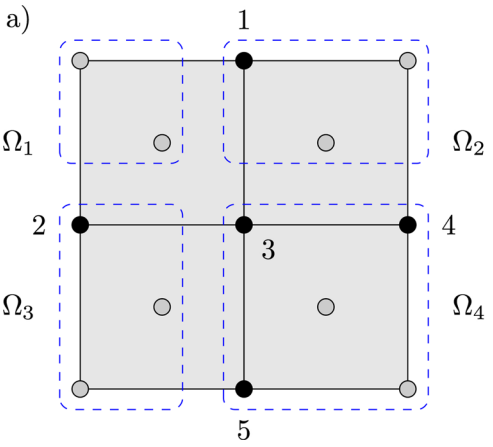
Fig. 5 Creation of a substructure code and the corresponding reverse code for the longest edge of one triangle. The underlined part of a code represents its new part caused by one refinement, and always replaces here one 0 in the code before

Definition 6 Let $d \in \mathcal{D}^i$ and $e \in \mathcal{D}^j$, with $1 \leq i, j \leq p$ and $i \neq j$, be DOF indices in subdomains Ω_i and Ω_j respectively. If d and e correspond to the same global DOF index, we relate them with $\mathcal{R}^i_j(d) = e$.

The *parallel DOF mapping* provides a global index for a set of local DOF indices. This index must be continuous and consistent on all subdomains. Thus, a DOF on an interior boundary must have the same global index on all subdomains that include this DOF. We define the global index to be a mapping from local DOF indices to the set $\mathcal{D}$ of global indices:

$$g_i : \mathcal{D}_i \mapsto \mathcal{D}$$
$$g_i(d) = \begin{cases} \sum_{j=1}^{i-1} \overline{n}_j + d & \text{if } d \in \overline{\mathcal{D}}_i \\ g_j(d') & \text{if } d \notin \overline{\mathcal{D}}_i, \mathcal{R}^i_j(d) = d' \text{ and } j = owner(d') \end{cases} \quad (1)$$

In the implementation of the local to global DOF mapping, two communications are necessary. In the first one, all ranks must compute the global index offset, i.e., the first global DOF index owned by the rank. This offset is denoted by the sum of global indices in all ranks having a smaller rank number. Computation



local:	1	2	3	4	5
global:	2	3	-	-	5

Fig. 6 **a**) Creation of global index for a subset (*dark colored*) of local DOFs on four subdomains; dotted line indicate DOFs which are owned by the corresponding processor **b**) local DOF numbering in each subdomain **c**) mapping g_3 from local DOF indices of subdomain Ω_3 to the global index of the selected DOFs

of this value can be implemented efficiently with using the parallel prefix reduction operation `MPI::Scan`. When all ranks have computed the global index for all rank owned DOFs, neighboring ranks must communicate the global DOF indices along interior boundaries. As the number of neighboring subdomains is bounded independently of the overall number of subdomains, also this communication is scalable. Note that the communication pattern to interchange global DOF indices does not need to be computed, as it is already defined by the interior boundary database.

Most parallel solver and domain decomposition methods require the set of global DOFs to be splitted in multiple subsets that must not necessarily be disjoint. Usually the global DOFs are splitted into the set of all DOFs on interior boundaries and the DOFs of subdomain's interior. Many domain decomposition methods split the set of interior boundary DOFs, e.g., to create a global coarse space that is defined on some interior boundary DOFs with special properties. All these subsets require a local and global continuous index.

Global mappings, defined on subsets of DOFs, are mostly used to create distributed matrices and vectors. The definition of a global mapping allows directly for subassembling local matrices to global ones. This becomes more complicated when mixed finite elements are used. Then, there exist multiple finite element spaces which define different sets of DOFs on the mesh. Thus, also local and global mappings have to be defined for each finite element space. There are two different assembling strategies when using multiple solution components, that may possibly be defined on different finite element spaces: the node-wise and the block-wise ordering. The node-wise ordering assigns to all DOFs at one node a continuous index, while the block-wise ordering considers all DOFs of the first component, then of the second, and so on. Both are just permutations of each other. In this

work, we make use of the block-wise ordering, as it is simple to implement in a parallel environment. Figure 7 shows the different views on the local and global numbering of DOFs with multiple finite element spaces. The global DOF indices for each component are sorted with respect to the rank number that own the DOF, which follows directly from the definition of the mapping g_i. The first global index of the j-th component finite element space on rank i is denoted with $rStart_i^j$. With this definition we can specify a function that maps on rank number i for each local DOF index d in component number j to a unique matrix row index:

$$matIndex_i(d, j) = \begin{cases} \sum_{k=1}^{j-1} rStart_i^k + \sum_{k=1}^{j-1} nRank_i^k + g_i^j(d) & \text{if } d \in \overline{\mathcal{D}}_i \\ matIndex_k(d, j) & \text{if } d \notin \overline{\mathcal{D}}_i, \mathcal{R}_k^i(d) = d', \ k = owner(d') \end{cases}$$

If there is only one component, or if the component number does not play any role, we will omit the second argument and just write $matIndex_i(d)$ for the matrix index of DOF d in rank number i.

3 Examples for efficient parallel methods

There exist two large classes of methods: *global matrix solvers* and *domain decomposition methods*.

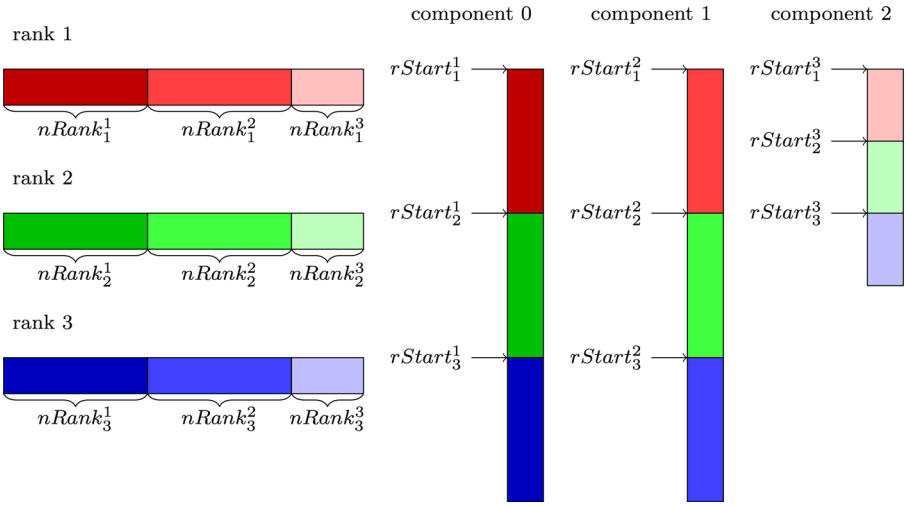


Fig. 7 Example for global mapping with three component defined on two different finite element spaces and three processors. For simplicity we assume the mesh to be equidistributed. (*left*) the rank view of the locally owned DOFs. The i-th rank contains in the finite element space of the j component $nRank_i^j$ DOFs. (*right*) the component view of enumerating DOFs

3.1 Global matrix solvers

The first one works on a distributed, globally assembled matrix and includes iterative Krylov subspace methods, parallel direct solvers and a large class of multigrid methods. To be efficient, iterative methods require for an efficient preconditioner. Typically, the performance and scaling of the preconditioner is the limiting factor of all iterative methods. The creation of an purely algebraic based, parallel, robust and optimal (w.r.t. scaling) preconditioner for iterative Krylov subspace methods is still an open research question. We will demonstrate the efficiency of the introduced data structures and algorithms on three examples. First, a constructed problem to show the scaling properties of mesh adaptation and repartitioning, second, a problem in dendritic solidification, which is solved using a phase-field model and a standard GMRES solver with Block-Jacobi preconditioning with local ILU and also requires frequent mesh adaptation and redistribution and third, a problem in fluid dynamics, which is solved using a diffuse domain approximation of an incompressible Navier-Stokes equation with a specific solver, as described in [24, 40]. Here a locally refined, but fixed mesh is used.

3.1.1 Mesh adaptivity according to a prescribed motion

We first address a problem of adaptive mesh refinement. Due to the local refinement the work distribution changes over time, making an initially good load balance useless. In order to sustain a good parallel efficiency, the mesh must be repartitioned and redistributed, which has to be rather efficient and should scale well for a large number of processors. To concentrate on this issue we use a simple geometrical problem of a prescribed moving sphere in a three-dimensional domain and require a prescribed fine mesh resolution within the sphere and a coarse mesh outside. This requires mesh adaptivity within each time step and serves as a worst case scenario for mesh adaptivity and load balancing. We measure the work load for each processor by counting the number of leaf elements associated with this processor. Performance of mesh adaptivity and load balancing is shown in Fig. 8 and Table 1 for strong scaling and in Fig. 9 and Table 2 for weak scaling.

While the local mesh adaption scales almost perfectly, as expected, also the parallel DOF mapping, with its global communication scales well. The number of iterations in the parallel adaption algorithm increase only slightly for both considered cases, the one with repartitioning in every 10th timestep and the one where repartitioning is done in every timestep. But the scaling behavior of the mesh adaption loop differs and better scaling is achieved for an appropriate load balancing. However, even if the mesh is repartitioned in each time step, perfect load balancing is not achieved. For large numbers of processors this is due to the underlying coarse mesh, which does not provide enough elements. METIS is used for repartitioning and does not scale well in our example and is the most expensive part of the algorithm. If repartitioning is done in each time step the overall algorithm is dominated by the repartitioning and the parallel efficiency for 4,096 processors drops to 13.6 %, if measured with respect to 256

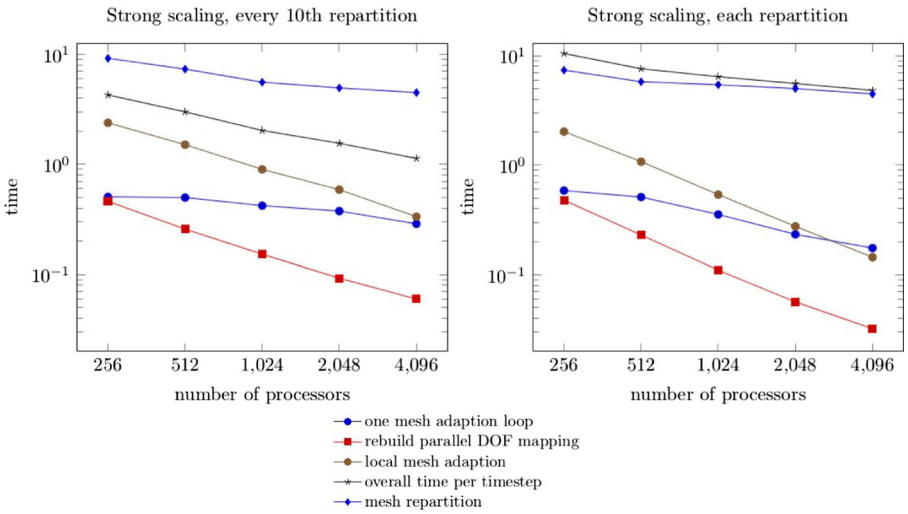


Fig. 8 Strong scaling of parallel mesh adaption: Shown are results for a simulation with repartitioning every 10th timestep and every time step. The problem setting in such that approximatly 1/4 of the processors are effected by the parallel mesh adaption in each time step.

processors. If load balancing is only done every 10th timestep, the overall time of the mesh adaption scales better, with an efficiency of 23.6 %. If we do not consider the costs for repartitioning the parallel efficiency is 55.1 %. However, for larger numbers of processors it is expected that also this number goes down

Table 1 Strong scaling of parallel mesh adaption: The upper part show the results with repartitioning at every 10th iteration and the lower part with reparitioning at every iteration. Shown is the average of 50 timesteps

processors	unbalancing	parallel adapt. iter	min.	avrg.	max. elements
256	56.9 %	3.97	26,165	41,612	65,290
512	95,9 %	4.00	11,934	21,243	25,968
1,024	137.8 %	4.03	5,377	10,919	25,968
2,048	173.8 %	4.36	2,740	5,651	15,468
4,096	206.8 %	4.75	1,218	2,940	9,018
256	12.3 %	3.86	33,161	41,619	46,728
512	16.7 %	4.08	14,792	21,244	24,797
1,024	18.9 %	4.19	7,726	10,926	12,994
2,048	20.8 %	4.33	4,325	5,654	6,833
4,096	24.4 %	4.53	2,263	2,942	3,658

The first colum shows the number of processors, the second the load unbalancing, which is defined w.r.t. the number of leaf elements and defined as $(max./avrg. - 1)100$ %. The iteration number of the parallel mesh adaption is shown in the third column and the forth column shows the minimal, average and maximal number of leaf elements per processor

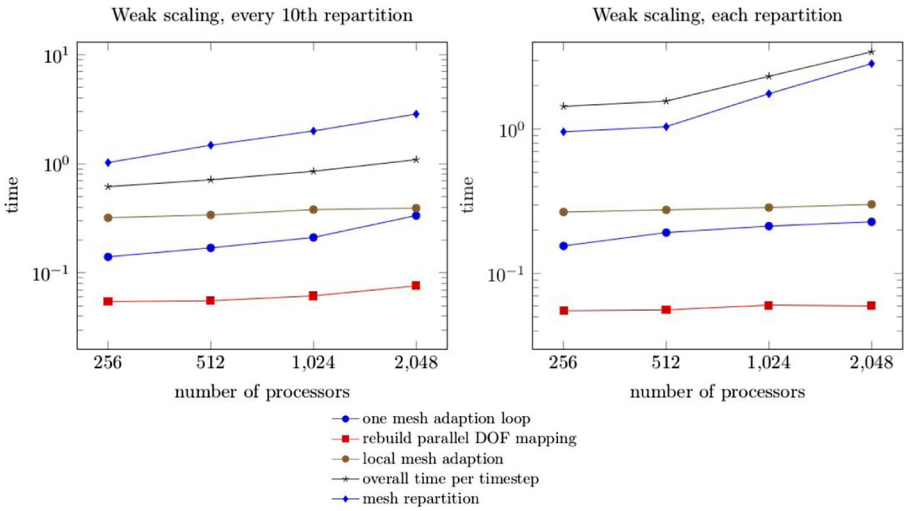


Fig. 9 Weak scaling of parallel mesh adaption: Shown are results for a simulation with repartitioning every 10th timestep and every time step. The previously considered example of a moving sphere which has to be refined is duplicated if the number of processors is doubled

as the costs for the mesh adaption loop, which accounts for the communication of domain boundaries, starts to dominate the local adaption and parallel DOF mapping.

Table 2 Weak scaling of parallel mesh adaption: The upper part shows the results with repartitioning at every 10th iteration and the lower part with reparitioning at every iteration. Shown is the average of 50 timesteps

processors	unbalancing	parallel adapt. iter	min.	avrg.	max. elements
256	50.6 %	4.0	4,162	5,807	8,799
512	60.8 %	4.0	5,329	5,849	9,404
1,024	71.8 %	4.1	3,357	5,852	10,053
2,048	65.5 %	4.2	3,172	5.838	9,684
256	14.9 %	4.0	4,929	5,807	6,672
512	17.6 %	4.3	3,545	5,849	6,874
1,024	18.5 %	4.5	3,346	5,855	6,940
2,048	19.6 %	4.5	3,149	5.853	7,005

The first colum shows the number of processors, the second the load unbalancing, which is defined w.r.t. the number of leaf elements. The iteration number of the parallel mesh adaption is shown in the third column and the forth column shows the minimal, average and maximal number of leaf elements per processor

We see similar bahavior for weak scaling. If the algorithm is not dominated by the repartitioning we observe good scaling properties for the mesh adaption up to 2,048 processors, which is 56.7 % for repartitioning every 10th timestep and 41.9 % if repartitioning is done every time step. Again, parallel efficiency is measured with respect to 256 processors. Not considering the costs for repartitioning gives an efficiency of 81.0 %. Here we do not see an indication for a lower efficiency if the number of processors is further increased.

Our results are still better than the theoretically predicted scaling properties for parallel mesh adaption and repartitioning in [23] but not as good as their reported computational examples up to 1.024 processors, which might be due to the considered configuration with shows a strong influence of the mesh adaption loop, which cannot be expected to scale well, as it only considers communication along domain boundaries.

As we use an external library for repartitioning, improving the scaling properties of this part is out of the scope of this paper and as long as mesh adaption does not turn out to be the dominating part in the adaptive finite element algorithm, we still expect good scaling properties of the overall algorithm. The next example thus considers a real application, which also requires frequent mesh adaption and repartitioning.

3.1.2 Strong scaling in dendritic growth simulations

A phase field equation is used to model dendritic growth. The model and the used discretization is described in detail in [47]. We use a standard GMRES solver with Block-Jacobi preconditioning with local ILU. While the solver is more or less standard, the challenge comes from the change and increase of the adaptively refined phase boundary, which requires frequent redistribution. The example is thus well-suited to demonstrate the prescribed scaling properties of the mesh adaptation in a real world example.

Figure 10 shows the phase field describing the dendrite at a specific time step as well as strong scaling results. The mesh is adapted in every timestep to resolve the phase boundary and redistributed if the unbalancing is above 20 %. The average of 50 timesteps is shown. The initial configuration was computed with 512 processors. Results with 128 up to 4,096 processors are shown. Because the coarse mesh is fix, the unbalancing factor increases for more processors as not enough coarse mesh elements can be redistributed. Table 3 shows more details.

The workload for mesh adaption and repartitioning is comparible with the previous example. However, due to a larger number of elements per processor and the refinement restricted only to the evolving interface, the adaption loop is dominated by the local mesh adaption and thus gives better scaling properties.

Compared with the overall time per timestep, which in addition include matrix assembly, error estimation, preconditioning, linear solver and redistribution the scaling properties for parallel mesh adaptivity play only a minor role. The overall algorithm shows an acceptable parallel efficiency for 4.096 processors of 52,3 % if compared with 128 processors.

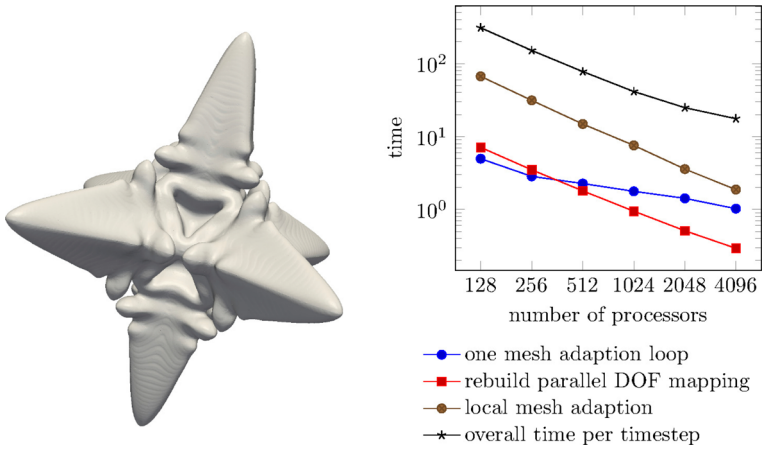


Fig. 10 (*left*) Dendritic structure at a specific time step. (*right*) The time is the average of 50 timesteps. As in the previous example, the time for local mesh adaptivity and rebuild of the parallel DOF mapping scales perfectly and only the mesh adaption loop shows weaker scaling properties. In addition the overall time per timestep is shown, which include in addition matrix assembly, error estimation, preconditioning, linear solver and redistribution and shows good scaling properties

3.1.3 Strong scaling for a parallel Navier-Stokes solver

To justify that our methods can be used for fast and simple implementation of problem specific solver methods, we show the results of our implementation of a parallel solver for the instationary Navier-Stokes equations, which was proposed in [40]. We briefly describe the solver. Consider the discrete system:

$$\begin{pmatrix} F & B^T \\ B & 0 \end{pmatrix} \begin{pmatrix} u \\ p \end{pmatrix} = \begin{pmatrix} f \\ g \end{pmatrix} \tag{2}$$

Table 3 Strong scaling of parallel mesh adaptivity in dendritic growth simulation. Shown is the average of 50 timesteps

processors	unbalancing	parallel adaption iter	ratio
128	4.44 %	3.31	3.9 %
256	4.56 %	3.54	4.1 %
512	6.03 %	3.94	5.2 %
1,024	10.05 %	3.98	6.5 %
2,048	18.14 %	4.92	7.6 %
4,096	28.32 %	4.82	7.5 %

The first column shows the number of processors, the second load unbalancing, which is defined w.r.t. the unknowns of the linear system. Redistribution is only done if the unbalancing is above 20. The third column shows an almost constant number of iterations for parallel mesh adaptivity, consistent with the previous example. The last column shows the ratio of time required in each timestep for parallel mesh adaptivity with the overall computing time for one timestep. We observe a slight increase, which probably continuous for larger processor numbers

We now consider the following block triangular preconditioner (see [10] for a general overview on preconditioning saddle point systems):

$$P^{-1} = \begin{pmatrix} F^{-1} & F^{-1}B^T S^{-1} \\ 0 & -S^{-1} \end{pmatrix} \tag{3}$$

with $S = BF^{-1}B^T$. When using exact solvers for computing F^{-1} and S^{-1}, GMRES converge in at most two iterations [10]. F^{-1} is replaced by the approximation F_*^{-1} which is obtained by one algebraic multigrid V-cycle on the matrix F. The Schur complement solution S^{-1} is approximated by $S_*^{-1} = Q_*^{-1} F_p H_*^{-1}$, where H_*^{-1} is the approximate solution of the pressure Laplace matrix with one algebraic multigrid V-cycle, Q_*^{-1} is the approximate solution of the pressure mass matrix with two CG iterations with diagonal preconditioning, and F_p the convection-diffusion operator discretized in pressure space. In [40] it was shown that this solver is independent of the mesh size and timestep, and that viscosity has only mild influence on the iteration count. The solver was extended in [24] for two-phase flow problems. Here, we use the same solver in a parallel environment for a diffuse domain model of the Navier-Stokes equation [1]. The results of [40] are not applicable in this situation, but the solver still provides an efficient method for the incompressible Navier-Stokes equation in complicated geometries.

We consider a flow channel with spherical particles. They are implicitly described by a phase-field variable, which in the current situation is fixed in time. No-slip boundary conditions at the particles are specified and incorporated into the diffuse domain approximation. A gravity force is used to drive the flow. The Reynolds number is $Re = 100$. The system to be solved in each timestep has $1.15 \cdot 10^7$ unknowns. 32 to 512 processors are used. The same initial mesh is provided, which is created in a sequential preprocessing step. The partitioning is computed using METIS and not changed during the computation. Figure 11 shows the runtimes of the individual sub-algorithms. The overall efficiency w.r.t. 32 processors is around 100% for all runs up to 256 processors and goes down to 82 % when using 512 processors. Here the sub-problems are already very small, with only $\sim$ 5000 elements per subdomain. Also load balancing can no longer be achieved due to the used coarse mesh, which does not provide enough elements. Details are shown in Table 4.

3.2 Domain decomposition methods

Domain decomposition methods decouple the problem in local subproblems, which can be solved independently of each other and only have to be coupled together once or in an iterative way by some global problem. Thus, an efficient domain decomposition method must balance between the parallelism it introduces and the size and complexity of the global problem. Most known domain decomposition methods are Schwarz iterative algorithms [17], Schur complement approaches and iterative substructuring algorithms [33, 34], and the family of FETI-DP (finite element tearing and interconnecting - dual primal) [19, 20, 27, 29] and BDDC (balancing domain decomposition by constraints) [35, 36] methods. FETI-DP and BDDC methods are well

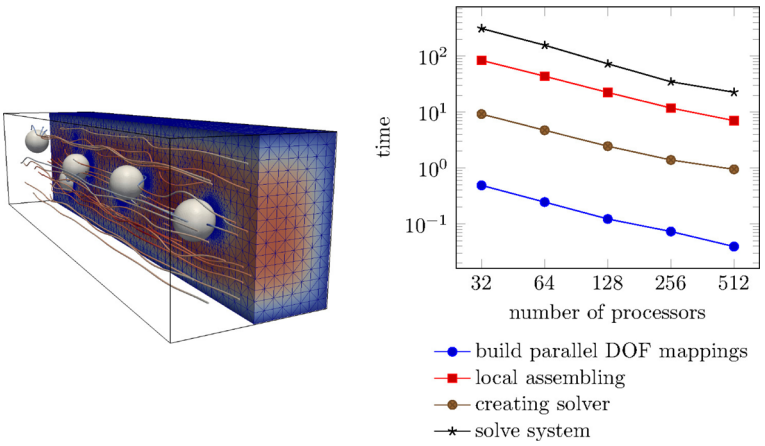


Fig. 11 (*Left*) solution at a specific time. The particles and the adaptive mesh are shown together with the stream lines. The color coding is according to the magnitude of the velocity field. (*Right*) strong scaling behavior of the solver for 32 to 512 processors

suited as black-box solvers for the finite element method. The methods are nearly free of parameters, only the null space of the linear system must be considered in some way. Both methods consider only a coarse mesh problem as the global problem and thus require less communication. The methods have been successfully applied in elasticity [27, 29], fluid dynamics [25, 46] and in electromagnetics [48]. Parallel and numerical scalability of the FETI-DP method for up to 65,536 processors was shown in [28]. We will use the FETI-DP method to examplify that the introduced algorithms and data structure allow for a simple, fast and scalable implementation. We demonstrate the algorithm on two examples. First a problem in grain growth, in which a phase field crystal model is used, and second a problem in biomechanics in which elasticity is considered within the diffuse domain approach. We start with a brief discussion of the method.

Table 4 Data for the Navier-Stokes solver for a diffuse domain configuration in 3D

processors	**avrg. unknowns**	**unbalancing**	**runtime [s]**	**efficiency**
32	359,144	5.3 %	405.5	100.0 %
64	179,572	13.3 %	205.8	98.4 %
128	89,786	14.2 %	98.1	103.3 %
256	44,893	14.1 %	48.2	105.0 %
512	22,446	40.9 %	30.6	82.6 %

The first column shows the number of processors, the second the average number of unknowns, the third the load unbalancing, which is defined w.r.t. the unknowns of the linear system. The fourth shows the overall runtime and efficiency in the last column is computed w.r.t the calculation with 32 processors

3.2.1 FETI-DP method

According to Definition 1 Ω is decomposed into p non-overlapping subdomains Ω_i, which are distributed to p processors. Each processor assembles local matrices A^i and local right-hand side vectors f^i. The vectors of unknowns are denoted by u^i. The basic idea of the FETI-DP method is to solve the local systems independently of each other and to ensure continuity of the solution across interior boundaries in some special way. For this, the unknowns u^i are first partitioned into the set u_I^i of interior unknowns and into the set u_Γ^i of unknowns on the interior boundaries. The interior boundary unknowns are further partitioned into the set of dual u_Π^i and primal interior boundary unknowns u_Δ^i. We further define the vector of local variables u_B^i:

$$u^i = \begin{bmatrix} u_I^i \\ u_\Gamma^i \end{bmatrix} = \begin{bmatrix} u_I^i \\ u_\Delta^i \\ u_\Pi^i \end{bmatrix} = \begin{bmatrix} u_B^i \\ u_\Pi^i \end{bmatrix} \tag{4}$$

All subdomains are directly coupled on the primal nodes, which play the role of the global coarse mesh problem. Along dual nodes, subdomains are only weakly coupled, see Fig. 12. The way how to choose the interior boundary nodes to be either dual or primal is crucial. To be consistent with Definition 2, we denote the set of all primal and dual interior boundary nodes by $\mathcal{I}_\Pi$ and $\mathcal{I}_\Delta$, respectively. Correspondingly, $\mathcal{I}_{i,\Pi}$ and $\mathcal{I}_{i,\Delta}$ denote the primal and dual nodes in subdomain Ω_i. As the primal nodes are non-local, there are always subdomains i and j, such that $\mathcal{I}_{i,\Pi} \cap \mathcal{I}_{j,\Pi} \neq \emptyset$. For dual nodes we have $\mathcal{I}_{i,\Delta} \cap \mathcal{I}_{j,\Delta} = \emptyset$ for all $1 \leq i, j \leq p$.

Continuity on the primal variables is enforced by global subassembly on the matrices and vectors restricted to the primal variables. To enforce continuity on the dual variables, we introduce a discrete jump operator J, such that the solution on the dual

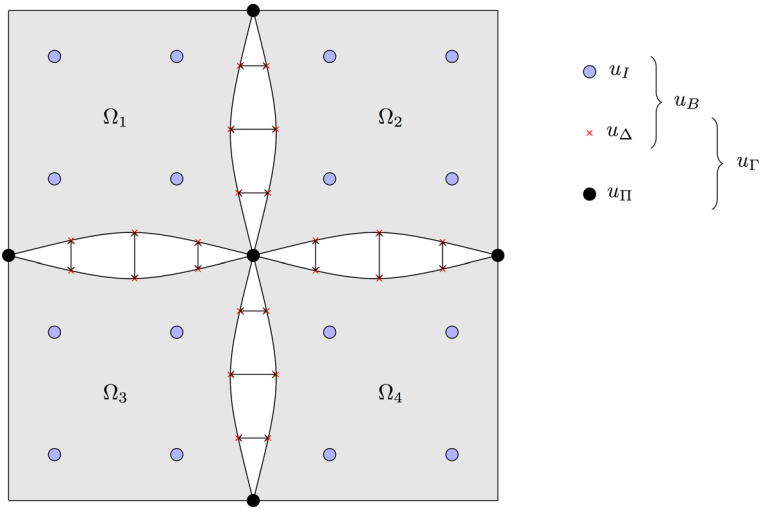


Fig. 12 Partitioning of the unknowns for the FETI-DP method

variables u_Δ is continuous across interior boundaries, when $Ju_\Delta = 0$. Each row of the matrix J, i.e., each constraint, must satisfy that the difference of two dual variables, that correspond to the same global node, is zero: $x_n - x_m = 0$, with $x_n \in \mathcal{I}_i$, $x_m \in \mathcal{I}_j$, $i \neq j$ and $\mathcal{R}^i_j(x_n) = x_m$. If $degree(b) \geq 3$, there is some choice in the number of constraints for vertex b. On the one hand, we cam take the whole set of redundant constraints. In this case, we will have $\frac{1}{2}degree(b)(degree(b) - 1)$ constraints for each vertex b and the matrix J will not be of full rank if $degree(b) \geq 3$ for at least one vertex b. On the other hand, we can choose a minimal, linear independent subset of constraints and obtain a matrix J of full rank. The first case is simpler to implement and will be used here. Let $v \in \mathcal{I}_\Delta$ be a dual node. We denote the ordered set of all constraints for vertex v with

$$v_C = \{(i, j) \mid \text{with } i, j \in \mathcal{W}v \text{ and } i < j\}. \tag{5}$$

The number of overall (possibly redundant) constraints is given by:

$$\mathcal{C}_n = \sum_{b \in \mathcal{I}_\Delta} \frac{1}{2} degree(b)(degree(b) - 1)$$

The overall number of dual variables is given by:

$$\Delta_n = \sum_{i=1}^{p} \mathcal{I}_{i,\Delta}$$

We denote by $\mathcal{C}(vij) \mapsto [1, \ldots, \mathcal{C}_n]$, with $i, j \in \mathcal{W}v$ and $i < j$, the global index of the constraint associated to the dual node v on subdomains Ω_i and Ω_j. Then, the jump operator matrix J is of size $\mathcal{C}_n \times \Delta_n$ and defined as follows:

$$J_{k,l} = \begin{cases} 1 & \text{, if } \mathcal{C}(vij) = k \text{ and } \widetilde{v}_i = l \\ -1 & \text{, if } \mathcal{C}(vij) = k \text{ and } \widetilde{v}_j = l \\ 0 & \text{, otherwise} \end{cases} \tag{6}$$

We partition the local matrices according to the partitioning of the unknown variables:

$$A^i = \begin{bmatrix} A^i_{BB} & A^i_{B\Pi} \\ A^i_{\Pi B} & A^i_{\Pi\Pi} \end{bmatrix}, A^i_{BB} = \begin{bmatrix} A^i_{II} & A^i_{I\Delta} \\ A^i_{\Delta I} & A^i_{\Delta\Delta} \end{bmatrix}, A^i_{B\Pi} = \begin{bmatrix} A^i_{I\Pi} \\ A^i_{\Delta\Pi} \end{bmatrix}, A^i_{\Pi B} = \begin{bmatrix} A^i_{\Pi I} & A^i_{\Pi\Delta} \end{bmatrix} \tag{7}$$

The right-hand side vectors are partitioned in the same way. To create the global coarse mesh problem of the primal variables, the primal variables are subassembled using the prolongation matrices ${R^i_\Pi}^T$:

$$\widetilde{A}_{\Pi\Pi} = \sum_{i=1}^{p} {R^i_\Pi}^T A^i_{\Pi\Pi} R^i_\Pi \tag{8}$$

and

$$\widetilde{A}^i_{B\Pi} = A^i_{B\Pi} R^i_\Pi, \widetilde{A}^i_{\Pi B} = {R^i_\Pi}^T A^i_{\Pi B} \tag{9}$$

$$\widetilde{A}^i_{BB} = \begin{bmatrix} A^1_{BB} & & 0 \\ & \ddots & \\ 0 & & A^p_{BB} \end{bmatrix}, \widetilde{A}^i_{B\Pi} = \begin{bmatrix} \widetilde{A}^1_{B\Pi} \\ \vdots \\ \widetilde{A}^p_{B\Pi} \end{bmatrix}, \widetilde{A}^i_{\Pi B} = \begin{bmatrix} \widetilde{A}^1_{\Pi B} & \dots & \widetilde{A}^p_{\Pi B} \end{bmatrix} \quad (10)$$

We now define the partially assembled matrix $\widetilde{A}$ and the corresponding right-hand side $\widetilde{f}$ as follows:

$$\widetilde{A} = \begin{bmatrix} \widetilde{A}_{BB} & \widetilde{A}_{B\Pi} \\ \widetilde{A}_{\Pi B} & \widetilde{A}_{\Pi\Pi} \end{bmatrix}, \widetilde{f} = \begin{bmatrix} f_B \\ \widetilde{f}_\Pi \end{bmatrix} \quad (11)$$

We introduce Lagrange multiplier λ for the continuity constraints on dual variables and formulate the FETI-DP saddle point problem as follows:

$$\begin{bmatrix} \widetilde{A}_{BB} & \widetilde{A}_{B\Pi} & J^T \\ \widetilde{A}_{\Pi B} & \widetilde{A}_{\Pi} & 0 \\ J & 0 & 0 \end{bmatrix} \begin{bmatrix} u_B \\ \widetilde{u}_\Pi \\ \lambda \end{bmatrix} = \begin{bmatrix} f_B \\ \widetilde{f}_\Pi \\ 0 \end{bmatrix} \quad (12)$$

By simple block Gaussian elimination on the variables u_B and $\widetilde{u}_\Pi$, we obtain the reduced linear system

$$F\lambda = d \quad (13)$$

with the FETI-DP operator F and the reduced right-hand side vector d defined by

$$\begin{aligned} F &= J\widetilde{A}^{-1}_{BB}(I + \widetilde{A}_{B\Pi}\widetilde{S}^{-1}_{\Pi\Pi}\widetilde{A}_{\Pi B}\widetilde{A}^{-1}_{BB})J^T \\ d &= J\widetilde{A}^{-1}_{BB}(f_B - \widetilde{A}_{B\Pi}\widetilde{S}^{-1}_{\Pi\Pi}(\widetilde{f}_\Pi - \widetilde{A}_{\Pi B}\widetilde{A}^{-1}_{BB}f_B)) \end{aligned} \quad (14)$$

with

$$\widetilde{S}_{\Pi\Pi} = \widetilde{A}_{\Pi\Pi} - \widetilde{A}_{\Pi B}\widetilde{A}^{-1}_{BB}\widetilde{A}_{B\Pi}. \quad (15)$$

Note that the matrix F is never built explicitly but is evaluated in every iteration of some Krylov subspace solver. The efficient parallel evaluation of the FETI-DP operator is discussed in the next Section.

With an appropriate preconditioner, which we do not describe here (see [29] for more information about this topic), the condition number of the preconditioned FETI-DP system was proven [20] to grow asymptotically as:

$$\mathcal{O}\left(1 + \log^2\left(\frac{H}{h}\right)\right), \quad (16)$$

where H is the subdomain size and h the mesh element size. Consequently, when the number of processors, and thus the number of subdomains, is fixed and the local meshes are refined, the condition number of the FETI-DP system grows asymptotically as $\log^2(h^{-1})$. If instead the problem size is fixed and the number of processors is increased, the condition number of the FETI-DP system decreases. For weak scaling, the problem size per processor is kept fixed and thus H/h and the condition number of the FETI-DP system remains constant.

The approach can be efficiently implemented using the algorithms and data structures described in Section 2. Required is:

1. creating all nodes on interior boundaries and deciding then to be either primal or dual nodes,

2. creation of matrices $\widetilde{A}_{BB}$, $\widetilde{A}_{B\Pi}$, $\widetilde{A}_{\Pi B}$, $\widetilde{A}_{\Pi\Pi}$ and J,
3. creation of vectors f_B and $\widetilde{f}_\Pi$,
4. defining some procedure for the solution of the Schur complement system $\widetilde{S}_{\Pi\Pi}$, and
5. solving the FETI-DP system in Eq. 13.

Our implementation of the FETI-DP method is based on the distributed data structures provided by the PETSc [6, 7] which are created using the algorithms and data structures from Section 2.

There are three different index sets that must be defined for the FETI-DP method: the set of indices of interior, dual and primal nodes. Partitioning of interior boundary nodes into the group of primal and dual nodes depends on the specific problem, or more precisely on its null space [30]. All selection algorithms have in common that they must distinguish between nodes on coarse mesh vertices, edges and faces. We use the interior boundary handler and the boundary DOF info class to collect all DOFs along the interior boundaries and to split them into sets on element substructures. A parallel DOF mapping for the primal nodes is established to directly create matrices and vectors defined on the space of primal nodes. As dual nodes are not shared by multiple processors, no parallel DOF mapping is required here. Instead, each processor create a local DOF mapping, which is also handled by the parallel DOF mapping class but restricted to one processor. The mapping is from local DOF indices to a continuous index set of non-primal DOFs.

We further have to create the matrix J. Each row of this matrix defines one constraint and thus has exactly two entries, 1 and -1, connecting two dual nodes in two different subdomains, see Algorithm 5. It requires one DOF communicator for the interior boundaries, which is used to get all subdomain indices of each dual node. Furthermore, the algorithm makes use of two parallel DOF mappings: one for the

Algorithm 5 Computation of the matrix J

```
input : Set of dual node indices I_{i,Δ}, DOF communicator object dofComm
output: Globally distributed matrix J representing the discrete jump operator
J = 0
use dofComm to create W for all nodes in I_{i,Δ}
foreach x ∈ I_{i,Δ} do
    matRowIndex = constraintMap.matIndex(x)
    for i = 0 to degree − 1 do
        for j = i + 1 to degree − 1 do
            if W(i) == mpiRank or W(j) == mpiRank then
                matColIndex = localDofMap.matIndex(x)
                if W(i) == mpiRank then
                    value = 1
                else
                    value = −1
                J[matRowIndex][matColIndex] = value
            end
            matRowIndex + +
        end
    end
end
```

local nodes, i.e., the interior and dual nodes, and one for the Lagrange constraints. The algorithm traverses for all dual nodes in its subdomain all constraint pairs. If the rank is part of this constraint, a corresponding entry to the matrix is set.

There are two different ways to implement the solution of the system with the operator $\widetilde{S}_{\Pi\Pi}$, see Eq. 15. Either its action on a vector is implemented and an iterative solution method is used, or the matrix is assembled explicitly and a direct solver is used. The first one requires at each iteration three matrix-vector multiplications, one solution with $\widetilde{A}_{BB}$ and one vector-vector addition. The solution with $\widetilde{A}_{BB}$ can be done quite efficiently, see the next section.

Schur complement operators are usually not explicitly assembled, as they are dense in most applications. We experimentally show, that this is not the case for $\widetilde{S}_{\Pi\Pi}$, which has some sparsity structure. We show that it might be beneficial to assemble it explicitly and to use parallel direct solvers to compute an LU factorization of this matrix, which allows to solve multiple systems with different right-hand side vectors efficiently. We have implemented both methods and will compare their efficiency.

The algorithm to apply the FETI-DP operator, see Eq. 14, on a vector is described in Algorithm 7. For the outer loop we use either the CG method for symmetric positive definite systems, MINRES for symmetric but indefinite systems or GMRES if the system is non-symmetric. The solver is stopped if the residual is reduced to less than 10^{-8}. The same solver and stopping criteria is used for the iterative Schur primal solver. In the case of the direct Schur primal solver, we use the parallel sparse direct solver MUMPS [2, 3]. For factorization of the local matrices, we use the multifrontal sparse LU factorization package UMFPACK [14, 15].

Algorithm 6 Explicit computation of matrix $\widetilde{S}_{\Pi\Pi}$

input : Matrices $\widetilde{A}_{\Pi\Pi}$, $\widetilde{A}_{\Pi B}$, $\widetilde{A}_{B\Pi}$ and A_{BB}
output: Matrix $\widetilde{S}_{\Pi\Pi}$
create matrix $\widetilde{K}_{B\Pi}$ of same size as $\widetilde{A}_{B\Pi}$
foreach $l \in \mathcal{I}_{i,\Pi}$ **do**
  $v = l$-th column of matrix $\widetilde{A}^i_{B\Pi}$
  $A^i_{BB} w = v$
  set w to be the l-th column of matrix $\widetilde{K}^i_{B\Pi}$
end
$\widetilde{S}_{\Pi\Pi} = \widetilde{A}_{\Pi\Pi} - \widetilde{A}_{\Pi B}\widetilde{K}_{B\Pi}$

Algorithm 7: Application of the FETI-DP operator

input : Matrices defined within the FETI-DP operator, Schur complement operator $\widetilde{S}_{\Pi\Pi}$, some vector λ'
output: $v = F\lambda'$
$t_0 = B^T\lambda'$
solve for $A_{BB}t_1 = t_0$
$v = \widetilde{A}_{\Pi B}t_1$
solve for $\widetilde{S}_{\Pi\Pi}t_1 = v$
$t_0 = t_0 + \widetilde{A}_{B\Pi}t_1$
solve for $A_{BB}t_1 = t_0$
$v = Bt_1$

3.2.2 *Weak scaling of FETI-DP for a phase field crystal model*

The phase field crystal (PFC) equation was introduced in [18] as a model for elasticity on atomic scales. It is a $6th$ order, nonlinear and time dependent partial differential equation which will be resolved on a globally refined mesh, see [5]. After refinement, each subdomain contains 66,049 DOFs for each of the three components. We run this configuration for 10 timesteps with 16 up to 16,384 processors with increasing domain size. On the largest domain, a system with more than $8 \cdot 10^8$ unknowns must be solved in each timestep.

For the coarse mesh, all four corner nodes of each subdomain are taken to be primal. The resulting system is indefinite and non-symmetric. It can be also formulated in a symmetric, but still indefinite way, but loses diagonal dominance and leads to higher computational costs for solving, even if an appropriate and efficient solver, e.g. MINRES, is used. Note that solving the resulting systems with FETI-DP is beyond its theory, which mostly assumes the matrices to be symmetric and positive definite. Nevertheless, FETI-DP is a robust and efficient solver for this case. Figure 13 shows runtime and weak scaling for 16 up to 16,384 processors using either the direct or the iterative Schur primal solver. The runtime is the average of ten timesteps. It includes the time for local subdomain assembling, creating the appropriate FETI-DP data structures and solving the resulting system. There is no error estimator used here and we have disabled all disk I/O.

The direct Schur primal solver performance is better for small size computations, but shows bad scaling for a larger number of processors. Using 4,096 processors, the direct solver was not even able to compute the very first timestep within 30 minutes. The main reason for this behaviour is the structure of the coarse grid. All processors contribute to the coarse mesh, but with a very low number of DOFs. For this benchmark, each processor contains only 4 coarse nodes. The sparsity structure of explicitly created $\widetilde{S}_{\Pi\Pi}$ decays from around 32 % by using 16 processors to less than 0.05 % in the case of 16,384 processor. The iterative Schur primal solver is around

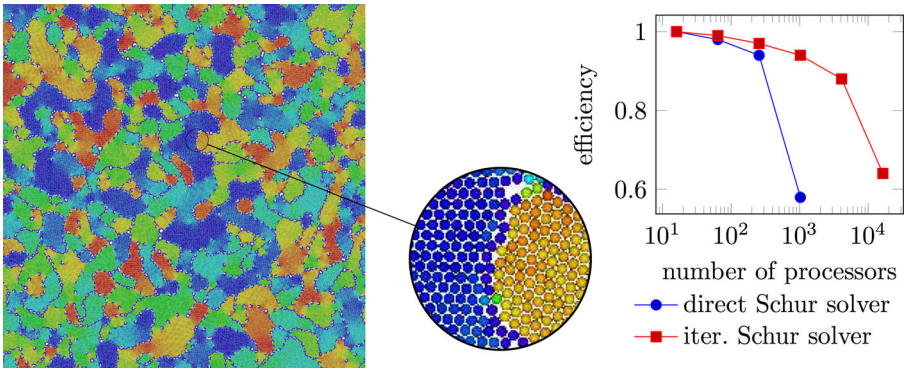


Fig. 13 (*Left*) Configuration for polycrystalline structure after post-processing, which is done with OVITO [41, 42]. Shown is the lattice structure, the color coding corresponds to the crystal orientation of a grain, see [4] for details. (*Right*) Computing time for one timestep of a 2D PFC computation using FETI-DP with two different solvers for the Schur primal system

10 % slower than the direct one for less than 1,024 processors but shows stable and good scaling also for the larger configurations. For the benchmark simulations, only 5 outer iterations are required to solve the FETI-DP system. The performance of the FETI-DP implementation is analyzed in more detail in Fig. 14.

Herein, we see that computing the explicit Schur primal matrix scales well, but the time for computing their LU factorization and using this factorization for solving a system highly increases from 16 processors to 1,024 processors. The iterative solution of this system scales very well up to 1,024 processors but shows a small breakdown for 4,096 processors and goes down to an efficiency of only 65 % for 16,384 processors. The very sparse coarse mesh is mostly responsible for the loss of computational efficiency of the FETI-DP implementation. FETI-DP's setup phase, i.e. creating primal, dual and interior node information and the corresponding parallel DOF mappings, shows a constant time on a very low level. Creating all required matrices shows a small increase in time for a larger number of processors. When further dropping the time down it can be shown that this is mainly due to the coarse mesh matrices $\widetilde{A}_{\Pi\Pi}$, $\widetilde{A}_{\Pi B}$ and $\widetilde{A}_{B\Pi}$.

3.2.3 Strong scaling of FETI-DP for linear elasticity

We consider a diffuse domain approximation [31] of a linear elasticity problem in biomechanics. The lamellar structure of a columnar cactus is analyzed using the FETI-DP method. Figure 15 shows the lamellar structure together with the deformation. The mesh has more than 55 million elements and around 11 millon vertices. Computations are performed on 256 to 2,048 processors. Scaling results are shown in Fig. 15. Here, creating FETI-DP data structures contains also the creation and factorization of local matrices. We have not broken down this number further, as more than 95 % of this time is related to LU factorization of the local matrices. Overall runtime, load unbalancing information and overall efficiency of the solver method are presented in Table 5. The super linear speedup for computations with 512 and 1,024 processors is related to the complexity of the direct solver UMFPACK in 3D,

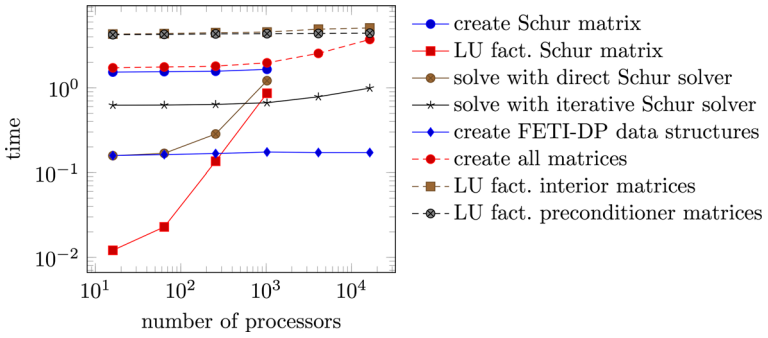


Fig. 14 Weak scaling of the FETI-DP method. The computing time is split into the subalgorithms. The direct Schur solver breaks down above 1.024 processors and shows a strong increase before. The most time is spent on the LU factorization, which however remains constant. The time needed to create the required matrices increases by a small amount for more than 1.024 processors

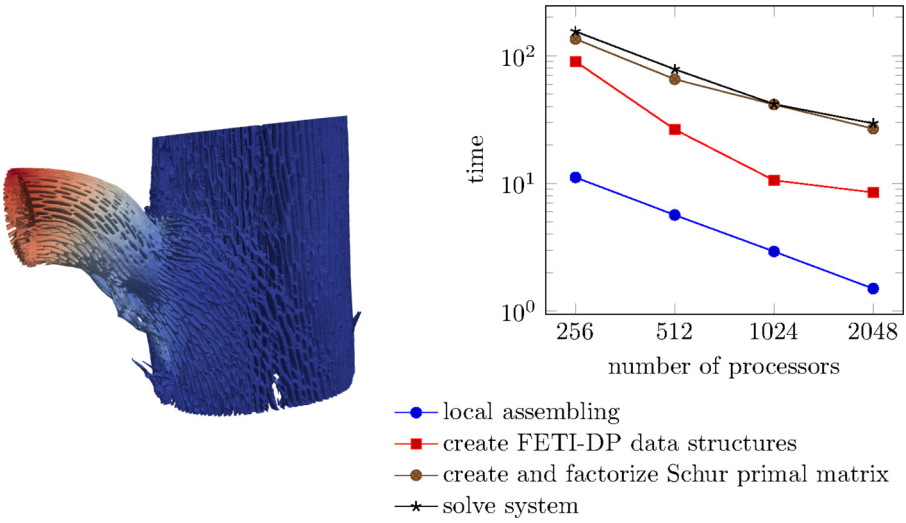


Fig. 15 Left part of the figure shows the result of computing linear elasticity in a cactus geometry defined by a diffuse domain approach. The result is colored with the magnitude of the displacement field. The right part of the figure shows strong scaling of the solver in the range of 256 up to 2,048 processors

which is of order $O(n^2)$, with n the number of unknowns. When the size of the subdomain is halved, time for local factorization is quartered. In opposite to this effect, the size of the coarse space problem, i.e. the Schur primal matrix, is increased.

3.3 Hardware details

All computations are performed on the HPC system JUROPA at the Jülich Supercomputing Centre (JSC) in Germany. This system consists of 3,288 nodes, each equipped with two Intel Xeon X5570 quad-core processors and 24 GB memory. The nodes are connected with an Infiniband QDR HCA network. Due to the hardware configuration, we do not perform benchmarks with less than 8 tasks as the effective cash-size and memory-bandwidth per task would be different from runs with more than 8 tasks.

Table 5 Data for benchmarking the FETI-DP solver of linear elasticity in a 3D diffuse domain configuration

processors	avrg. unknowns	unbalancing	runtime [s]	efficiency
256	40,934	6.7 %	380.5	100,0 %
512	20,934	11.0 %	170.2	111.7 %
1,024	10,740	18.5 %	85.7	110.9 %
2,048	5,532	25.6 %	64.7	73.4 %

The first column shows the number of processors, the second the average number of unknown, the thrird the load unbalancing, which is again defined w.r.t. the unknowns of the linear system. The fourth shows the runtimes and the efficiency in the last column is computed w.r.t the calculation with 256 processors

4 Discussion

In order to achieve our goal to provide a user-friendly efficient massively parallel adaptive finite element method we have presented software concepts and numerical algorithms for distributed meshes. They provide mesh information to a parallel solver in a solver specific way. The concepts and algorithms are general in order to allow to implement a broad range of efficient and scalable methods for the solution of linear systems. Several examples of weak and strong scaling up to 16,386 processors justify our approach for global matrix solvers and domain decomposition methods. As long as repartitioning is not an issue we obtain excellent parallel efficiency, see Sections 3.1.3, 3.2.2 and 3.2.3. In situations where the mesh has to be frequently adapted and repartitioned to achieve an appropriate load balancing the parallel efficiency goes down, which is primarily due to the bad scaling properties of the used library METIS for partitioning. But also for these examples a parallel efficiency above 50 % could be reached, see Section 3.1.2.

One of our assumptions is that the geometry can be sufficiently represented by a coarse mesh. Our implementation in the finite element toolbox AMDiS stores this coarse mesh on all processors, which will become critical if even larger HPC systems are used. We would like to point out, that this is not specific to the presented method but only to our implementation. Each processor requires only information on the coarse mesh elements of its subdomain plus all neighboring coarse mesh elements. Correspondingly, the element object database must only be stored for these coarse mesh elements communicated during mesh redistribution.

In most of our simulations working on the coarse mesh level for parallel mesh partitioning and distribution has the advantage of fast mesh partitioning, using mesh substructure and structure codes. There are still some very few scenarios, where this approach can limit parallel scaling. Especially, when the mesh has to be refined only very locally. It can be impossible to create a coarse mesh to distribute the leaf mesh to an appropriate number of processors. In such situations the assumption to start with a coarse mesh has to be weakened.

However, for most situations the described data structures and numerical algorithms provide an efficient way to implement problem specific global matrix solvers and domain decomposition methods for adaptive finite element discretizations on today's HPC systems.

Acknowledgments SL and AV acknowledge support from German Science Foundation within EXC CfAED. We further acknowledge computing resources at the HPC system JUROPA (JSC, Jülich, Germany) through grand HDR06. The FETI-DP implementation presented here is based on PETSc. We would like to thank PETSc's developer team, especially J. Brown, B. Smith and M. Knepley for their support. We further thank R. Backofen for providing the phase field crystal simulations, R. Gärtner for providing the linear elasticity simulations and S. Aland for providing the Navier-Stokes results.

References

1. Aland, S., Lowengrub, J., Voigt, A.: Two-phase flow in complex geometries: a diffuse domain approach. CMES **57**, 77–108 (2010)

2. Amestoy, P., Duff, I., L'Excellent, J., Koster, J.: A fully asynchronous multifrontal solver using distributed dynamic scheduling. SIAM Journal on Matrix Analysis and Applications **23**(1), 15–41 (2001)
3. Amestoy, P., Guermouche, A., L'Excellent, J.-Y., Pralet, S.: Hybrid scheduling for the parallel solution of linear systems. Parallel Comput. **32**(2), 136–156 (2006)
4. Backofen, R., Barmak, K., Elder, K., Voigt, A.: Capturing the complex physics behind universal grain size distributions in thin metallic films. Acta Mater. **64**, 72–77 (2014)
5. Backofen, R., Rätz, A., Voigt, A.: Nucleation and growth by a phase field crystal (PFC) model. Philos. Mag. Lett. **87**(11), 813–820 (2007)
6. Balay, S., Brown, J., Buschelman, K., Eijkhout, V., Gropp, W.D., Kaushik, D., Knepley, M.G., McInnes, L.C., Smith, B.F., manual, H.Zhang.P.ETSc.u.sers.: Technical Report ANL-95/11 - Revision 3.2. Argonne National Laboratory (2011)
7. Balay, S., Brown, J., Buschelman, K., Gropp, W.D., Kaushik, D., Knepley, M.G., McInnes, L.C., Smith, B.F., Zhang, H.: PETSc Web page. http://www.mcs.anl.gov/petsc (2011)
8. Bangerth, W., Burstedde, C., Heister, T., Kronbichler, M.: Algorithms and data structures for massively parallel generic adaptive finite element codes. ACM Trans. Math. Softw. **38**(2), 14/1–14/28 (2012)
9. Bangerth, W., Hartmann, R., Kanschat, G.: Deal.II – a general purpose object oriented finite element library. ACM Trans. Math. Softw. **33**(4), 24/1–24/27 (2007)
10. Benzi, M., Golub, G.H., Liesen, J.: Numerical solution of saddle point problems. Acta Numerica **14**, 1–137 (2005)
11. Burstedde, C., Ghattas, O., Gurnis, M., Isaac, T., Stadler, G., Warburton, T., Wilcox, L.C.: Extreme-scale AMR. In SC10: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis. ACM/IEEE (2010)
12. Burstedde, C., Ghattas, O., Stadler, G., Tu, T., Wilcox, L.C.: Towards adaptive mesh pde simulations on petascale computers. In TeraGrid'08 (2008)
13. Burstedde, C., Wilcox, L.C., Ghattas, O.: `p4est`: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees. SIAM J. Sci. Comput. **33**(3), 1103–1133 (2011)
14. Davis, T.: Algorithm 832: Umfpack v4.3—an unsymmetric-pattern multifrontal method. ACM Trans. Math. Softw. **30**(2), 196–199 (2004)
15. Davis, T.: A column pre-ordering strategy for the unsymmetric-pattern multifrontal method. ACM Trans. Math. Softw. **30**(2), 165–195 (2004)
16. Dedner, A., Klöfkorn, R., Nolte, M., Ohlberger, M.: A generic interface for parallel and adaptive discretization schemes: abstraction principles and the dune-fem module. Computing **90**, 165–196 (2010)
17. Dryja, M., Smith, B., Widlund, O.: Schwarz analysis of iterative substructuring algorithms for elliptic problems in three dimensions. SIAM J. Numer. Anal. **31**(6), 1662–1694 (1994)
18. Elder, K.R., Katakowski, M., Haataja, M., Grant, M.: Modeling elasticity in crystal growth. Phys. Rev. Lett. **245701**, 88 (2002)
19. Farhat, C., Lesoinne, M., LeTallec, P., Pierson, K., Rixen, D.: FETI-DP: a dual-primal unified FETI method-part I: A faster alternative to the two-level FETI method. Int. J. Numer. Methods Eng. **50**(7), 1523–1544 (2001)
20. Farhat, C., Lesoinne, M., Pierson, K.: A scalable dual-primal domain decomposition method. Numerical Linear Algebra with Applications **7**(7-8), 687–714 (2000)
21. Filippone, S., Colajanni, M.: PSBLAS: a library for parallel linear algebra computation on sparse matrices. ACM Trans. Math. Softw. **26**(4), 527–550 (2000)
22. Golub, G., van Van Loan, C.: Matrix Computations. The Johns Hopkins University Press (1996)
23. Jansson, N., Hoffman, J., Jansson, J.: Framework for massively parallel adaptive finite element computational fluid dynamics on tetrahedral meshes. SIAM J. Sci. Comput. **34**(1), 24–41 (2012)
24. Kay, D., Welford, R.: Efficient numerical solution of Cahn-Hilliard-Navier-Stokes fluids in 2D. SIAM J. Sci. Comput. **29**(6), 2241–2257 (2007)
25. Kim, H., Lee, C., Park, E.: A FETI-DP formulation for the stokes problem without primal pressure components. SIAM J. Numer. Anal. **47**(6), 4142–4162 (2010)
26. Kirk, B.S., Peterson, J.W., Stogner, R.H., Carey, G.F.: `libMesh`: A C++ Library for Parallel Adaptive Mesh Refinement/Coarsening Simulations. Engineering with Computers **22**(3–4), 237–254 (2006)
27. Klawonn, A., Rheinbach, O.: Robust FETI-DP methods for heterogeneous three dimensional elasticity problems. Comput. Methods Appl. Mech. Eng. **196**(8), 1400–1414 (2007)

28. Klawonn, A., Rheinbach, O.: Highly scalable parallel domain decomposition methods with an application to biomechanics. ZAMM **90**(1), 5–32 (2010)
29. Klawonn, A., Widlund, O.B.: Dual-primal FETI methods for linear elasticity. Commun. Pur. Appl. Math. **59**(11), 1523–1572 (2006)
30. Lesoinne, M.: A FETI-DP corner selection algorithm for three-dimensional problems. In: Proceedings of the 14th International Conference on Domain Decomposition Methods (2003)
31. Li, X., Lowengrub, J., Rätz, A., Voigt, A.: Solving PDEs in complex domains: a diffuse domain approach. Commun. Math. Sci. **7**, 81–107 (2009)
32. Logg, A., Mardal, K.-A., Wells, G.N., et al.: Automated Solution of Differential Equations by the Finite Element Method. Springer (2012)
33. Mandel, J.: Iterative solvers by substructuring for the p-version finite element method. Comput. Methods Appl. Mech. Eng. **80**(1-3), 117–128 (1990)
34. Mandel, J.: On block diagonal and schur complement preconditioning. Numer. Math. **58**, 79–93 (1990)
35. Mandel, J.: Balancing domain decomposition. Commun. Numer. Methods Eng. **9**(3), 233–241 (1993)
36. Mandel, J., Sousedík, B., Dohrmann, C.: Multispace and multilevel BDDC. Computing **83**, 55–85 (2008)
37. T. P: Matthew. Domain Decomposition Methods for the Numerical Solution of Partial Differential Equations. Springer (2008)
38. Ribalta, A., Stoecker, C., Vey, S., Voigt, A.: AMDiS - adaptive multidimensional simulations: Parallel concepts. In Domain Decomposition Methods in Science and Engineering XVII, volume 60 of Lecture Notes in Computational Science and Engineering, pp. 615–621. Springer, Berlin Heidelberg (2008)
39. Saad, Y.: Iterative Methods for Sparse Linear Systems. SIAM (2003)
40. Silvester, D., Elman, H., Kay, D., Wathen, A.: Efficient preconditioning of the linearized navierâĂŞstokes equations for incompressible flow. J. Comput. Appl. Math. **128**(1-2), 261–279 (2001)
41. Stukowski, A.: OVITO Web page. http://www.ovito.orf (2010)
42. Stukowski, A.: Visualization and analysis of atomistic simulation date with OVITO-the open visualization tool. Model. Simul. Mater. Sci. Eng. **128**, 261–279 (2010)
43. Trottenberg, U., Oosterlee, C.W., Schüller, A.: Multigrid: Basics, Parallelism and Adaptivity. Academic Press (2000)
44. Vejchodský, T., Šolín, P., Zítka, M.: Modular hp-FEM system HERMES and its application to Maxwell's equations. Math. Comput. Simul. **76**(1-3), 223–228 (2007)
45. Vey, S., Voigt, A.: AMDiS: adaptive multidimensional simulations. Comput. Vis. Sci. **10**, 57–67 (2007)
46. Šístek, J., Sousedík, B., Burda, P., Mandel, J., Novotný, J.: Application of the parallel BDDC preconditioner to the Stokes flow. Computers and Fluids **46**(1), 429–435 (2011)
47. Witkowski, T., Voigt, A.: A multi-mesh finite element method for Lagrange elements of arbitrary degree. J. Comput. Sci. **3**, 420–428 (2012)
48. Xue, M., Jin, J.: Nonconformal FETI-DP methods for large-scale electromagnetic simulation. IEEE Trans. Antennas Propag. **PP**(99), 1 (2012)