TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Mathematik und Naturwissenschaften, Institut für Wissenschaftliches Rechnen

SOFTWARE CONCEPTS AND ALGORITHMS FOR AN EFFICIENT AND SCALABLE PARALLEL FINITE ELEMENT METHOD

Der Fakultät Mathematik und Naturwissenschaften
der Technischen Universität Dresden

zur

Erlangung des akademischen Grades
Dr. rer. nat.
eingereichte

Dissertation

von

Thomas Witkowski
geboren am
24. Mai 1982 in Loslau/Polen.

Die Dissertation wurde in der Zeit von 07/2007 bis 12/2012
am Institut für Wissenschaftliches Rechnen angefertigt.

Tag der Einreichung: ...
Tag der Verteidigung: ...

Gutachter: Prof. Dr. rer. nat. habil. A. Voigt
Technische Universität Dresden
...
...

Contents

1 Introduction

The finite element method (FEM) is one of the most used methods for the numerical solution of partial differential equations (PDE). Over the last decades many codes and software packages have been developed to solve specific PDEs, e.g. for the Navier Stokes, the Helmholtz or the Maxwell equation. These codes are highly specialized and allow to efficiently solve the corresponding PDE. But usually they cannot easily be used to solve another PDE. These highly specialized and very efficient codes are contrasted with general purpose finite element toolboxes which can be used to solve some class of PDEs. General purpose finite element toolboxes become more and more important for a large part of the science community, as the models to be solved include more and more physics from different areas of research. The need for these general PDE solver methods is emphasized by the success of widely used general software packages for the finite element method like deal.II [17, 16], FEniCS/DOLFIN [83, 84], DUNE [18, 33], Hermes [119], libMesh [69] or AMDiS [124].

In most cases, solving a PDE with a general finite element toolbox requires more computing time compared to a highly specialized software package. Modern programming paradigms, for example meta programming [11, 51, 57], and new compiler techniques allow to shrink this gap. Furthermore, as computing time becomes cheaper and available computing systems become faster, the development time for a PDE solver becomes more costly and more important than the pure runtime of a code. These are the main reasons why we see a trend over the years towards using general finite element packages in the scientific community. In this work we consider software concepts and numerical algorithms for their efficient implementation. Many different approaches are possible, but all of them can be assigned to one of the following three categories:

- increasing sequential performance of the code by using programming techniques, modern compilers and code optimization to make best use of the available hardware resource, i.e. CPU and memory
- parallelization approaches to make use of a multitude of computing nodes
- providing broad support for different discretization schemes such that the user can choose the optimal one; this may include, for example, the support of different mesh structures or different finite elements

In this thesis, we do not consider the first category but instead focus on the latter two. Parallelization of finite element software, is a very broad research field. Firstly, there are many different parallel systems: multi-core CPUs, large HPC systems with several hundreds of thousands of cores or GPUs. One may consider parallelization techniques that are special to one of these architectures, or consider some general approaches. Furthermore, the finite

element method is not only a single algorithm, but it is formed from a set of numerical algorithms which must be considered independently of each other for parallelization. In this work, we present software concepts and algorithms for parallel distributed meshes that allow to exchange information with a solver for the resulting systems of linear equations. This allows to implement solvers, either specific to some PDE or in a black box approach, that are efficient and scalable for a large number of cores. To exemplify the presented concepts, we derive and implement a new multilevel FETI-DP method. For a broad range of PDEs, this solver method is not only optimal from a mathematical, but also from a computational point of view. This means, that this method reliably scales to a large number of processors.

Independently of parallelization, and in consideration of the last category in the list above, we present the application of a multi-mesh finite element method for discretization of systems of PDEs with Lagrange finite elements of arbitrary degree. When a system of PDEs consists of multiple variables with different solution singularities, using independently refined meshes for the different components can be superior to using a single mesh for all of them.

Both, the presented parallelization concepts and the multi-mesh method are implemented in the finite element toolbox AMDiS (adaptive multidimensional simulations) [124, 123]. AMDiS is an open source C++ library which supports the solution of a large class of stationary and instationary systems of PDEs.

1.1 Overview

Chapter 2 gives a very quick and broad introduction to adaptive meshes. It introduces all the terminology that is used later for defining parallel distributed meshes. Furthermore, the concept of mesh structure and substructure codes is introduced, which is of high importance for the efficient implementation of the software concepts and algorithms for distributed meshes, that are described in Chapter 3. There we show that the link between distributed meshes and the solver method for the resulting systems of linear equations is necessary to eventually develop an efficient and scalable solver method. We present an efficient method to establish this link. The concepts are exemplified by numerical results of two totally different solver methods: an efficient implementation of the FETI-DP method is presented which can serve as a block box solver in finite element codes, and we present the implementation of a parallel solver for the 3D instationary Navier-Stokes equation. In Chapter 4, a multi-mesh method is presented which allows to use different mesh refinements for different components of a PDE. Two examples are presented where the multi-mesh method leads to a much faster solution procedure than a standard single-mesh method. Finally, Chapter 5 summarizes the results and gives an outlook for further research of the topics presented in this thesis.

1.2 Technical notes

The benchmarks that are used to show efficiency of the software concepts, algorithms and their implementation are always more or less synthetic. We tried to make them not too

simple to make sure that they are representative for real applications. But eventually, they have to be chosen such that they directly show efficiency and scaling of the implementation of some basic algorithms. As all these algorithms are implemented in the finite element toolbox AMDiS, they are also used to solve real problems from different research areas. The author of this thesis made use of the presented methods in the following works: [9], [6], [129].

This work addresses both, readers who are interested in efficient and scalable finite element methods, and users of the finite element toolbox AMDiS. The latter are usually more interested in technical details, which are also important for an efficient implementation but are not necessary for the reader interested in a general concept description. To cope with both, technical details are pushed to the end of each section and are illustrated using algorithm descriptions in pseudo code. In general, their understanding is not necessary to understand the general concepts. Though AMDiS is implemented in C++, non of the presented data structures or algorithms are limited to a specific imperative programming language. To simplify the notation, all indices in figures, definitions and algorithms start with 1. This differs from the 0-starting indexing of arrays in C++.

The landscape of today's HPC (high performance computing) systems is very heterogeneous. Most systems contain computing nodes consisting of multiple CPUs which by themselves include multiple cores. There are computing nodes which additionally contain GPUs that consist of several hundreds or even thousands of relatively simple cores. Our presentation of software concepts and algorithms is not specific to some hardware configuration. Therefore, we settle with using either *processor* or *core* to speak about the smallest computing unit that processes on one subdomain. The term *rank* denotes a unique identifier, i.e. an integer value, of each processor participating in a computation. The presented software concepts and their implementation are based on distributed memory parallelization and we use MPI (message passing interface) [106] to implement the required communication between processors. Even though the MPI standard is defined such that it allows a specific MPI library to implement nearly all communication patterns in a scalable way, most MPI implementations lack scalability for some MPI commands for large numbers of processors. In [12], these effects are analyzed and it is discussed which scalability we can theoretically expect from a best possible MPI implementation. To understand scalability behavior on very large HPC systems, one has to also consider *system noise*, which can influence parallel scalability in a negative way. See [53], and reference therein, for a detailed analysis.

All computations presented here are performed on the HPC system JUROPA at the Jülich Supercomputing Centre (Germany). This system consists of 3,288 computing nodes, each equipped with two Intel Xeon X5570 quad-core processors and 24 GB memory. Overall, there are 26,304 cores available. The nodes are connected with an Infiniband QDR HCA network.

2 Adaptive meshes for finite element method

Using adaptively refined meshes is one of the key concepts for an efficient finite element method, and is used by most finite element codes today. The general idea is that the geometry, on which a PDE should be calculated, can be sufficiently represented by a *coarse mesh*, but must be refined in some way to solve the PDE with a discretization error below a given error bound. Especially when solving PDEs, that describe physical phenomena on multiple space scales, using uniformly refined meshes is not possible anymore. To control mesh adaptivity, some method is required to choose elements which should be refined. Therefore, the roots of developing adaptive mesh refinement methods are closely related to the development of a-posteriori error estimators, which make it possible to estimate the discretization error element wise and thus can be used to control mesh adaptivity. The history goes back to the late seventies and early eighties of the last century, see for example the pioneering work [4, 67, 32]. Today, mesh adaptivity in context of the finite element method is covered by many text books.

Section 2.1 gives a brief overview about data structures and algorithms which are used to store and work on adaptively refined meshes. This description is mainly based on [102, 123]. Section 2.3 first gives an introduction into mesh structure codes, which allow for a very efficient representation of the mesh structure. We then extend mesh structure codes to mesh substructure codes and to mesh value codes. Both are concepts which are elementary for our efficient and scalable implementation of distributed meshes, see Section 3.2.

2.1 Data structures of adaptive meshes

The meshes we consider in this work consist of triangles in 2D and tetrahedrons in 3D. Even though we restrict all of the algorithms in this work to these mesh elements, by appropriate changes of the underlying data structures all of them can be used also for rectangles and cuboids, which are also very popular for mesh discretization. For refining triangles and tetrahedrons, two different methods are widely used: bisectioning and red-green refinement. In this work, we restrict to the first one, as its implementation and analysis is simpler without having considerable disadvantages. More information about biscectioning in 2D and 3D can be found in [115]. Red-green refinement is considered, e.g., in [122].

When using bisectioning on an element T, we call the newly created elements T_1 and T_2 the left and right *children* of T. Correspondingly, T is called the *parent* of T_1 and T_2. To make the definition explicit, we have to define left and right children of triangles and tetrahedrons. For this, we first need to define a local numbering of vertices. For triangles, see also Figure 2.1, we start with the longest edge of a triangle and number the vertices counter clockwise. When a triangle is bisectioned, the new vertex is created as the

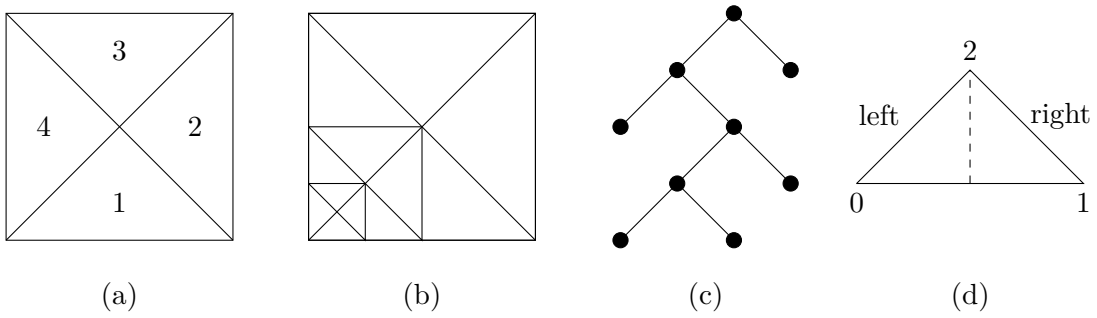


Figure 2.1: Basic concepts of adaptively refined meshes: (a) Coarse mesh consisting of four triangles. (b) Refined mesh. (c) Binary tree representing refinement structure of coarse mesh element 1. (d) Local numbering of vertex nodes of a triangle and the definition of left and right child.

mid-point of the line connecting local vertices 0 and 1 of the parent element. The newly created triangle, which contains the local vertex 0 of its parent, is called the left child, and the other one the right child, correspondingly. For the 3D case, the definition of left and right children is similar but requires for some more technical definition. We refer the interested reader to [102, Section 1.1.1].

The refinement structure of one coarse mesh element is stored within one binary tree. The refinement structure of a whole mesh is thus represented by a set of binary trees. See Figure 2.1 for an illustration of a refined 2D mesh and a binary tree representing the refinement structure of one coarse element. The *level* of a mesh element is equivalent to the depth of the corresponding node in the binary tree. Thus, coarse mesh elements have level 0. Furthermore, the union of all mesh elements, which are represented by leaf nodes of the binary trees, is named *leaf meshes* and its elements are named *leaf elements*.

Most of the element data are stored only on the level of coarse mesh elements, for example element coordinates. To get the information for an arbitrary element of the mesh, the concept of *mesh traverse* is introduced. The mesh traverse object allows to traverse a mesh in a predefined way and to calculate the required data on all traversed mesh elements. The mesh can be traversed either in pre-order (parent - left child - right child), post-order (left child - right child - parent) or in-order (left child - parent - right child) up to a given mesh level.

2.2 Error estimation and adaptive strategies

We consider an a posterior residual based error estimator as, for example, described by Verfürth [121] for general non-linear elliptic PDEs. If $\mathcal{T}$ is a partition of the domain into simplices, the error estimator defines for each element $T \in \mathcal{T}$ an indicator, depending on the finite element solution u_h, by

$$\eta_T(u_h) = C_o R_T(u_h) + C_1 \sum_{E \subset T} J_E(u_h) \tag{2.1}$$

where R_T is the element residual on element T, and J_E is the jump residual, defined on an edge E. C_0 and C_1 are constants used to adjust the indicator. The global error estimation of a finite element solution is then the sum of the local error indicators

$$\eta(u_h) = \left(\sum_{T \in \mathcal{T}} \eta_T(u_h)^p \right)^{1/p} \quad \text{with } p \geq 1 \tag{2.2}$$

If an error tolerance, tol, for the solution u_h is given, and $\eta(u_h) > tol$, the mesh must be refined in some way in order to reduce the error. We make use of the equidistribution strategy, as described by Eriksson and Johnson [40]. The basis for this strategy is the idea that the error is assumed to be equidistributed over all elements. If the partition consists of n elements, the element error estimates should fulfill

$$\eta_T(u_h) \approx \frac{tol}{n^{1/p}} \equiv \eta_{eq}(u_h) \tag{2.3}$$

Mesh adaption using the equidistribution strategy is controlled with the two parameters $\theta_R \in (0, 1)$ and $\theta_C \in (0, 1)$. An element is refined if $\eta_T(u_h) > \theta_R \eta_{eq}(u_h)$, and the element is coarsened if $\eta_T(u_h) \leq \theta_C \eta_{eq}(u_h)$. The parameter θ_R is usually chosen to be close to 1, and θ_C to be close to 0, respectively.

Thus for scalar PDEs, one needs to define constants for error estimation and mesh adaption, and to adjust both for a specific equation. If a single mesh is used to resolve all variables of a system of PDEs, multiple error estimators and different adaption strategies must be introduced. In this case, an element is refined if at least one strategy has marked the element to be refined. An element is coarsened if all strategies have marked it to be coarsened. Thus, the number of constants to be defined increases with the number of variables of the system to be solved. The situation is quite similar in the case of a multi-mesh method, which we introduce in Chapter 4. When using multiple, independently refined meshes an error estimator must be defined for each mesh to be adapted. Furthermore, it is possible to have different adaption strategies for the different meshes. As for the scalar case, the constants C_0 and C_1 must be defined for each component independently. In multiphysics problems the error tolerance and the error estimating constants may represent some meaningful parameters, making it somehow intuitive to find appropriate values. However, when the systems of PDEs arise from operator splitting of some higher order PDE, the specific variables may not have a physical significance. The constants for error estimation are especially arbitrary, making it hard to derive useful values. Determining values for the error estimation constants in the context of dendritic growth is discussed in [35, 95].

2.3 Mesh structure codes

The usage of *mesh structure codes* for efficient mesh representation goes back to [97]. The main idea is to traverse the binary trees, which represent the refinement structure of a mesh, in a unique way, e.g. with the 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 a coarse mesh element. Figure 2.2 shows

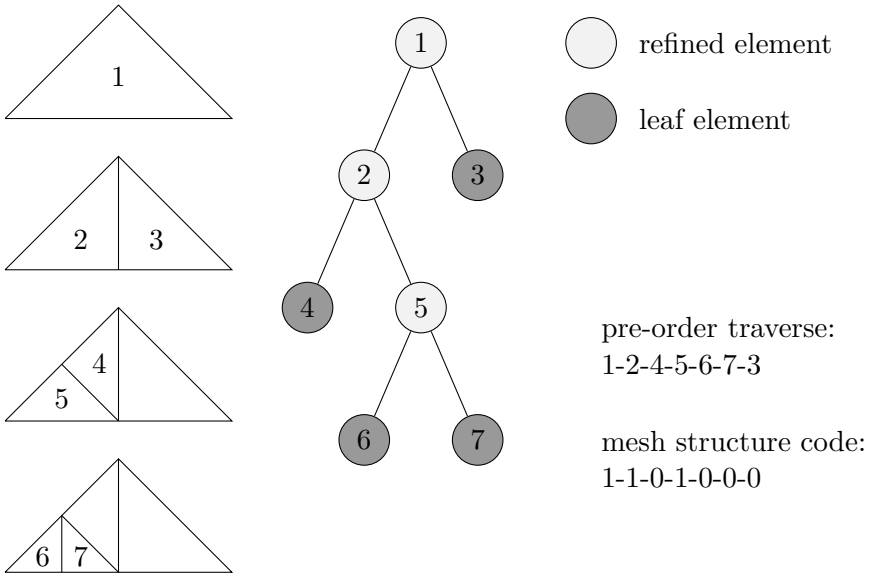


Figure 2.2: Mesh structure code of an adaptively refined triangle

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. If the code becomes larger than the word size of the system, which is typically 64 or 128 bit, vectors of decimal values are used to represent a mesh structure code.

We extend the concept of mesh structure codes to *substructure codes*. A substructure code does not store the refinement structure of a coarse mesh element but only of one of its substructures, i.e. an edge of a triangle, or an edge or a face of a tetrahedron. 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. Mesh structure codes are not appropriate for this purpose as they contain much more information leading to larger codes that must be communicated between processors. Furthermore, they do not allow for a simple and fast check, if two elements have the same refinement structure along an element substructure.

The creation of substructure codes is based on a modified pre-order traverse algorithm, which is reduced to work only on one substructure of an element. The function *preOrderTraverse*, see Algorithm 1, starts on a coarse mesh element and traverses recursively only the children that intersect with a given substructure of the coarse mesh element. In the algorithm, the += operator applied on substructure codes is not defined in the standard numerical way, but instead it appends a 0 or 1 to the substructure code. Furthermore, this function can create the substructure code in reverse mode. In that case, the left and right children of elements are swapped. This is required, as the left and right children on two neighbouring coarse mesh elements may not correspond to each other. In this case, to allow for a very fast check whether the refinement structure on two neighbouring elements fit together, the substructure code of one of these coarse mesh elements must be created in reverse mode. This scenario is shown in Figure 2.3, where

Algorithm 1: $preOrderTraverse(el, t, r)$: Pre-order traverse on substructures

```
input  : element el, substructure t, reverse mode r
output : substructure code c
if isLeaf(el) then
    c += 0
else
    c += 1
    elLeft = getLeftChild(el)
    elRight = getRightChild(el)
    if r then
        swap(elLeft, elRight)
    if contains(elLeft, t) then
        c += preOrderTraverse(elLeft, t, r)
    if contains(elRight, t) then
        c += preOrderTraverse(elRight, t, r)

end
```

substructure codes and the reverse codes are created along the refinement edge of two triangles.

Lastly, we introduce the concept of *mesh value codes*. Mesh structure codes and substructure codes allow to reproduce the refinement structure of either a whole mesh, a coarse mesh element or the refinement structure along an element's substructure. For mesh distribution, see Chapter 3.2.2, we need some functionality to create data structures which allow to reconstruct not only the refinement structure of a coarse mesh element, but also all value vectors which are defined on them. This is provided by *mesh value codes*. Each mesh value code corresponds to one mesh structure code and contains as many value entries as the mesh structure code contains 1-bits plus a constant offset. As each 1 of the mesh structure code corresponds to a bisectioning of an element, it also corresponds to exactly one mesh vertex. Therefore, if an element refinement structure is reconstructed using a mesh structure code, also the values defined on this element can be reconstructed using a mesh value code. The constant offset for the size of mesh value codes is 3 in 2D and 4 in 3D and corresponds to the vertices of the coarse mesh elements, whose value must also be set but which are not created by bisectioning of some coarser element.

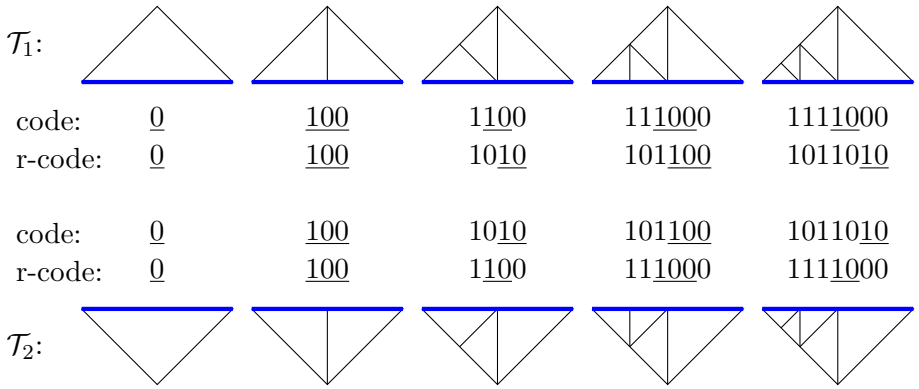


Figure 2.3: Creating a substructure code and the corresponding reverse code for the longest edge of one triangle. The underlined part of a code represents its new part caused by one refinement, and replaces one 0 in the code before.

3 Scalable parallelization

The finite element method (FEM) is a widely used scheme for the numerical solution of partial differential equations (PDEs). It is not only one algorithm, but instead it is an accumulation of several sub-methods which all require, directly or indirectly, the following input data: a PDE with some boundary conditions, appropriate basis functions, and a geometry. In this work we assume that the geometry can be sufficiently well represented by a *coarse mesh*, but must be refined in some way to solve the PDE with a discretization error below a given error bound. In this work, we use meshes consisting of triangles and tetrahedron. These are refined by *bisectioning*, i.e. an element is successively split into two smaller elements. Figure 3.1 illustrates the finite element method with local mesh adaptivity (h-FEM). In the first step, which is called the *assembler*, a local integration process creates a matrix-vector representation of the discretized PDE on the given mesh and using the predefined basis functions. An appropriate *solver* method is used to solve the large and sparse system of linear equations. For many PDEs a local mesh refinement, which leads to a satisfactory discretization error, is not known a priori. In this case, an *error estimator* is used to estimate the error of the discrete solution. For the case that the overall error estimate is too large, a *marker strategy* is used to identify some parts of the mesh to be adapted in order to decrease the estimated error. This loop is continued until the error estimate drops below a given threshold. Note that besides h-FEM, there also exist other finite element methods, like e.g., hp-FEM [5] or extended finite element method (XFEM) [19], which also fit with small modifications into this abstract illustration.

To make best use of the available computing platforms, parallelization of the finite element method become quite popular in the eighties [93, 78]. Many of the very first approaches are based on shared memory systems, where all processors have direct access to the same memory. Using a shared memory parallelization approach, all algorithms of the finite element method can work in parallel on the same input data to speed up the overall computation time. Today's high performance systems consist of several hundred thousands of processors and are mostly based on the concept of distributed memory: all processors can communicate among each other via some message protocol, but each processor cannot directly access the memory of all other processors. These systems allow to solve large problems that do not fit into the memory of one processor, but accordingly they lead to the new challenge to partition and distribute the input data and to redefine the algorithms of the finite element method to work on distributed data. Distributing data in the finite element method leads to two independent subproblems: providing a distributed mesh, and providing distributed matrices and vectors. In this work, we present concepts and algorithms for the former one. Parallelization of data structures and algorithms from linear algebra is considered, e.g., in [46]. For the algorithmic parallelization of the finite element method we must further distinguish between the different sub-methods: assembler, error estimator and marker strategy are quite simple to parallelize, as all of them are mostly

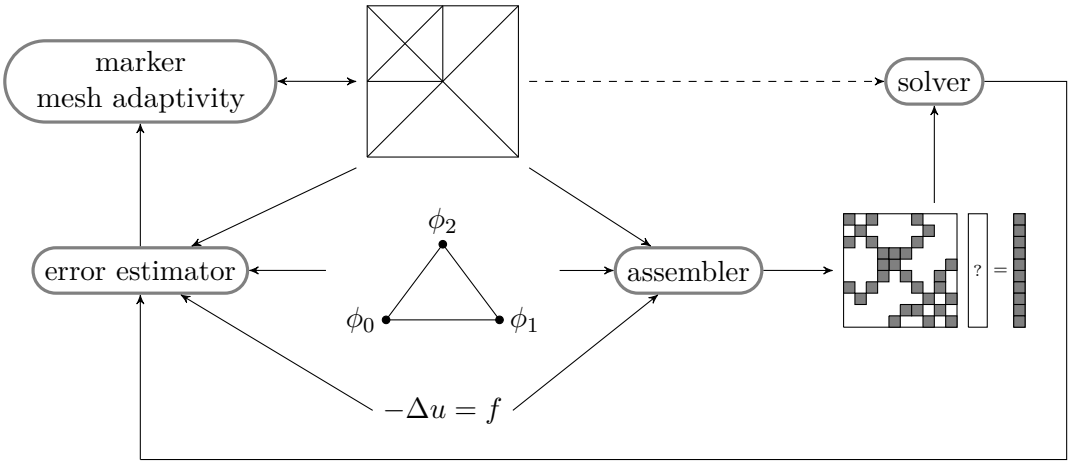


Figure 3.1: Sketch of the finite element method with local mesh adaptivity (h-FEM). The dashed line, representing the information flow from the mesh to the linear solver, is optional.

local mesh procedures and require only little communication between processors. For more information on this topic, we refer the reader to [15, 69]. The situation is totally different for solving linear 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 [58, 104, 98, 49], e.g. CG or GMRES, with different kinds of algebraic and geometric preconditioners, or multigrid methods [116]. Another approach for solving linear systems in parallel are domain decomposition methods, which are treated in Section 3.4. It is also quite common to combine different of these approaches to design an efficient solution procedure. Efficiency of parallel solver methods is defined by three points:

- strong parallel scaling: the problem size is fixed and the number of processors varies. A perfect scaling solver should halve the solving time when the number of processors is doubled.
- weak parallel scaling: the problem size per processor is fixed, thus when the number of processors is increased, also the overall problem size grows accordingly. In this case, a perfect scaling solver should keep the solving time constant.
- numerical scaling: the influence of discretization parameters, e.g. mesh size or some physical constants within the PDE, on the solver time should be as small as possible.

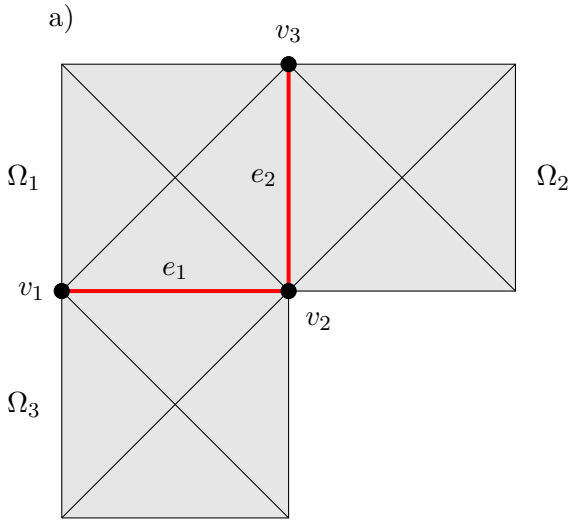
Many solver methods, which are able to show scalability to large number of processors, have one thing in common: they do not work on an algebraic level only but make use of some geometrical information of the mesh. For example, geometric multigrid methods require information about the hierarchical decomposition of the mesh, iterative substructuring methods require geometrical information of the degree of freedoms (DOFs) that composite the interior boundaries between subdomains. Therefore, we do not only present efficient

parallel algorithms and data structures for distributed meshes, but we also consider the link to solve large systems of linear equations. Thereby, we do not restrict ourselves to a specific linear solver, but instead we present a framework that allows for the implementation of a large class of highly scalable solver methods.

During the last ten years, several finite element packages have been created by different research groups which also allow to solve a large class of PDEs on distributed HPC systems. We shortly compare these packages and their software concepts with our implementation in the finite element toolbox AMDiS. The finite element package deal.II [17, 16] allows to solve a very large class of PDEs. It's mesh is based on the mesh library `p4est` [28] and supports distributed adaptive mesh consisting of rectangles and cubes. Parallelization of deal.II has been considered in [15]. Strong scaling at solving the Laplace equation in 2D has been shown up to 16,384 processors. For weak scaling, a thermal convection equation in a 2D domain was solved with up to 512 processors. In both numerical examples, mesh adaptivity is used. Linear solvers are not considered in a special way in this work. DUNE (Distributed and Unified Numerics Environment) [18] has the goal to abstract all parts of the numerical solving of PDEs. Thus, it is not restricted to a discretization scheme, and besides finite element methods also finite volume and finite difference methods can be used. Furthermore, it has an abstract mesh interface which allows to use any mesh library as long as it provides an implementation to this interface. In [33] parallelization issues are considered and some benchmarks with scaling up to 512 processors are presented. The FEniCS project [83] is a collection of several software components with the common goal to automate the solution of differential equations. More information about various aspects of this very interesting and promising project can by found e.g. in [84, 85]. An example for a parallelized finite element code based on FEniCS is Unicorn [55, 54], an adaptive finite element solver for fluid and structure mechanics. Parallelization issues of Unicorn are considered in [54, 59]. Some nontrivial scaling studies for 3D flow up to 8,192 processors are presented in [54]. Software concepts and algorithms for parallelization of Unicorn are described in more detail in [59]. As in our work, Unicorn's meshes consist of triangles and tetrahedrons. It allows for distributed meshes, but the parallel mesh adaptivity algorithm has some influence on the mesh quality. This is not the case for our concepts and implementation of distributed meshes, where the mesh structure is independent of the number of subdomains. The scaling shown in [59] up to 1,024 processors is sufficient but it is questionable whether this approach will scale for large number of processors. In particular, no special parallel linear solver is considered, but instead standard approaches of using for example Krylov subspace methods with block Jacobi and local ILU(0) preconditioners are used, which are known not to scale for large number of processors.

Section 3.1 introduces some terminology and gives some general formal definition of the concepts, which are required in the remaining part of this chapter. Section 3.2 describes software concepts and algorithms for efficient and scalable implementation of distributed meshes. Hereby, we already focus on providing mesh dependent data, which may be requested by a parallel solver method. In Section 3.3, we give a brief overview about parallel linear solver methods. To exemplify the software concepts, we show that they allow to implement a large class of different solver methods. Thereby, we present the implementation of a black box solver in Section 3.4 and of a highly specialized solver for the instationary Navier-Stokes equation in Section 3.6. In Section 3.4, we give first a brief

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	3
e_2	1,2	2	2
v_1	1,3	2	3
v_2	1,2,3	3	3
v_3	1,2	2	2

Figure 3.2: 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

overview on domain decomposition methods and present an efficient implementation of the FETI-DP method. To our best knowledge, this is the first presentation of the FETI-DP method not restricted to the solution of one specific PDE, but which considers this method for the solution of a broad range of PDEs. Even though the FETI-DP method has nearly optimal properties from a mathematical point of view, we show absence of computational scaling for large number of processors. In Section 3.5, we present two methods to overcome two computational drawbacks of the FETI-DP method. First, inexact FETI-DP methods are introduces, which allow to replace the exact subdomain solver by inexact ones. This introduction is mostly based on the work [71]. To deal with the distributed coarse space matrix, which sparsity pattern is mostly responsible for the breakdown of scalability for large number of cores, we introduce a new approach of a multilevel FETI-DP method. To certify that our approach can also be used for problem specific solver methods, Section 3.6 presents an implementation of a state-of-the-art iterative solver method for the instationary Navier-Stokes equation. Finally, Section 3.7 shows implementation specific problems of the concepts presented in this chapter. Hereby, we focus on software design concepts, which are necessary to implement a modular finite element toolbox.

3.1 Formal definitions

For the presentation of our parallel algorithms, data structures and software concepts, we need some formal definition: In what follows, $\Omega \subset \mathbb{R}^d$ with $d = 2$ or $d = 3$, is an arbitrary open domain and $\partial\Omega$ denotes its boundary. The boundary is divided 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, 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 it's *interior* part and *interior boundaries*, i.e., element segments which intersect with other subdomain boundaries. As all data is communicated along interior boundaries, they play a central role in all parallelization concepts. We define interior boundaries 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 3.2 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 a neighborhood relation in the natural way as

Definition 3 *Subdomain j is called to be a* neighbour *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 of all coarse mesh elements the subdomains which contain this substructure. Therefore, we establish the following auxiliary definitions, which are exemplified in Figure 3.2.

Definition 4 *Let b be an arbitrary vertex, edge or face of a coarse mesh element in $\overline{\Omega}$. We define $\mathcal{W}(b)$ to be an index set defined as*

$$\mathcal{W}(b) = \{i \mid b \in (\Omega_i \cup \mathcal{I}_i)\}$$

and the degree *of b is defined by*

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

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:*

$$owner(b) = \{i \in \mathcal{W}(b) \mid \forall j \in \mathcal{W}(b) : j \neq i \Rightarrow j < i\}$$

Figure 3.2 exemplifies this concept.

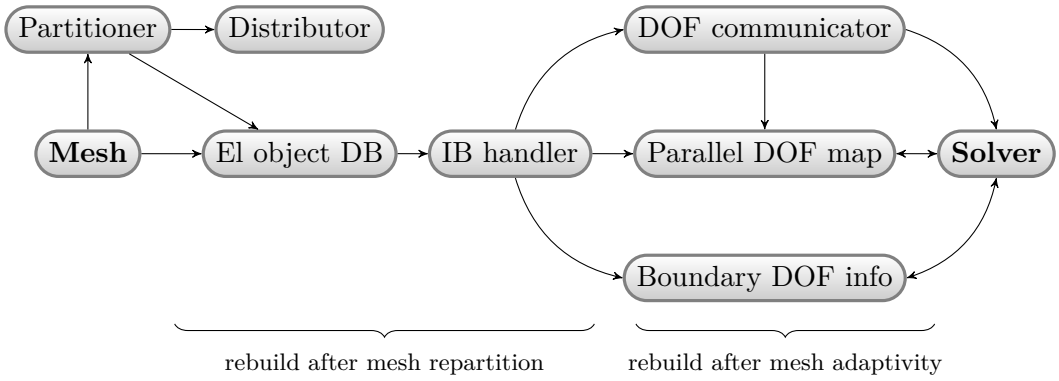


Figure 3.3: Information flow between the mesh data structure and the parallel solver for systems of linear equations.

3.2 Distributed meshes

In the introduction of this chapter, we have identified the need for an appropriate general information flow between the mesh structure and the solver method. The following mesh information is required by most parallel solver methods:

- hierarchical decomposition of the mesh: mostly used by multigrid methods to project the solution (or the residual) between the fine and some coarse mesh
- communication pattern: most parallel solvers iterate between some local and some global solution procedures, thus it must be known which subdomain DOFs are shared with some other subdomains and must be therefore communicated/synchronized
- restricted sets of DOFs: especially in iterative substructuring methods [91] 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 3.2.3).
- 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 3.4 to exemplify the general concepts presented in this section, must distinguish between DOFs that belong to vertices, edges and faces of the coarse mesh elements.

Only the first point of this list is also used by sequential solvers, and is thus not specific to the parallel case. Mesh information described by the other three points should be created only on demand when requested by a specific linear solver method. Figure 3.3 shows a general overview of all concepts that make it possible to create exactly the data required by some specific linear solver method. The notation directly corresponds to the class structure described in Section 3.7. First, the initial coarse mesh must be passed to the *mesh partitioner*, which is responsible for assigning each coarse mesh element to

one processor. This information is used by the *mesh distributor* to move the coarse mesh elements, together with their 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. These are computed by the *element object database* (EL object DB), which can be used to query for the following questions: given a geometrical entity, i.e., either 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 more refined elements. Furthermore, the database is stored on all processors for the whole initial mesh. We refer to Section 3.2.5 for a discussion of this approach and how to circumvent its limitations.

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 Figure 3.2. These boundary elements are subdivided into two sets: the boundary elements that are owned by the processor and boundary elements that are part of the processor's subdomain but owned by another processor. All this information is again stored on the level of the coarse mesh and thus do not change due to local mesh adaptivity. They must be rebuild 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 2. Each processor then sends its list of own boundary segments to all neighbouring 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. While computing the solution of some PDE, the mesh will be adapted. This introduces new DOFs, some of which may 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, DOF communicators can be easily created without further communication, see Algorithm 3. Each subdomain just traverses all geometric entities of all interior boundaries and collects the corresponding DOFs. As the interior boundary handler ensures that the

Algorithm 2: Creating 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 coarseElement ∈ mesh do
    if db.elInSubdomain(coarseElement) then
        foreach entity ∈ coarseElement 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
    end
    foreach index ∈ neigh do
        if index < mpiRank then
            intBoundOwn.send(index)
        else
            intBoundOther.recv(index)
        end
    end
end
```

boundary elements are in the same order on all neighbouring processors, the collected DOFs directly fit together on the interior boundaries and can be used for communication. Because DOF communicators must be reinitialized after each adaption of the mesh, 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 neighbouring 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 mapping of DOFs must also work for subsets of DOFs. For example, iterative substructuring methods need a global index of all interior boundary DOFs. The parallel DOF mapping is described in Section 3.2.3.

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.

Algorithm 3: Creating DOF communicators.

```
input  : intBoundOwn and intBoundOther: interior boundary handler for the processor
         owned boundary segments and for all other interior boundary segments
output : sendDofs: set of DOFs on processor owned boundaries, recvDofs: set of DOFs
         on other boundaries
foreach bound ∈ intBoundOwn do
    Element el = bound.el
    vector <DegreeOfFreedom> dofs = el.getAllDofs (bound)
    sendDofs.insert (dofs)
end
foreach bound ∈ intBoundOther do
    Element el = bound.el
    vector <DegreeOfFreedom> dofs = el.getAllDofs (bound)
    recvDofs.insert (dofs)
end
```

3.2.1 Mesh structure codes for parallel mesh adaptivity

For non-overlapping domain decomposition methods the data structures and algorithms are different dependent on whether hanging mesh nodes can be handled by the finite element code. If hanging nodes are possible, or some Mortar method [91] is used, the mesh refinement structure along interior boundaries does not need to fit together. Otherwise, an algorithm must be used that ensures coincident interior boundaries after local mesh refinement or coarsening in at least one subdomain has been performed. In this work, we consider a finite element that require matching interior boundaries. Several work has been done in the past to describe and implement parallel adaptive mesh refinement (AMR) on large number of processors. The distributed mesh library `p4est` [28, 27] has shown excellent weak and strong scaling to over 224,000 processors. Due to its internal mesh representation based on octrees, it is not directly usable for triangle and tetrahedron meshes. As a pure mesh library, it provides little support for implementing parallel solvers. Currently, in [59] an AMR method for tetrahedral meshes is presented. Parallel scaling up to 1,024 processors with an efficiency of around 80% is shown. One disadvantage of the this AMR method is that the mesh quality is influenced by the partitioning.

We present a method for parallel mesh adaptivity where the parallel scaling efficiency is mostly limited by the efficiency of the used MPI library. One of our key concepts for efficient distributed adaptive meshes are the so called *mesh structure codes* and *mesh substructure codes*, which are introduced in Section 2.3. The main advantage of these data structures 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 one. See Section 3.2.2 for more details.

Mesh substructure codes are used to create a parallel distributed mesh with coincident boundary segments on all subdomains. Based on the interior boundary handler it is straightforward to define an algorithm that iteratively adapts the local subdomains until

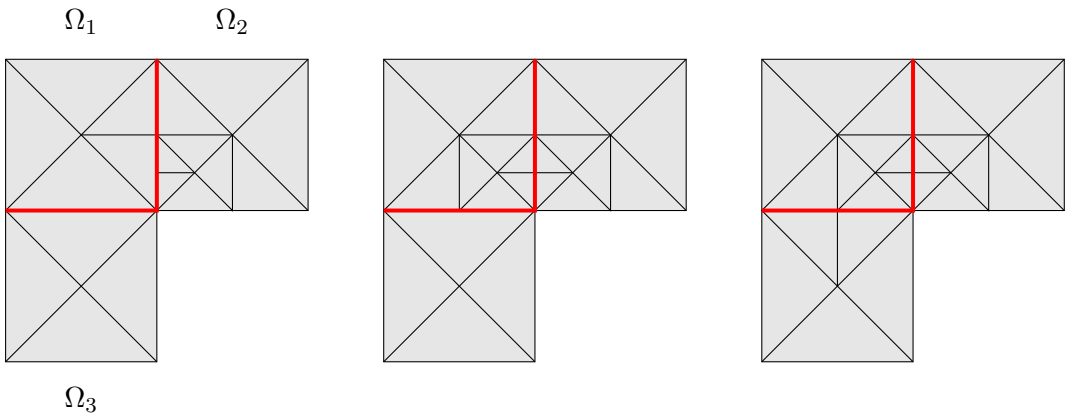


Figure 3.4: Iterative parallel mesh adaptivity on three subdomains. The left picture shows the initial situation with one hanging node. The central and the right picture are the results after the first and second refinement loop, respectively.

all of them coincide along their interior domains. This procedure is defined in Algorithm 4. Every processor creates substructure codes for all of its own interior boundary segments. These codes are send to the neighbouring processors where they can be directly used to check if the mesh refinement structure is the same at the corresponding edges or faces. If this is not the case, a received substructure code can directly be used to refine the corresponding element. This process must be repeated as the refinement in one subdomain may cause hanging nodes on the interior boundary with other subdomains, see Figure 3.4 for an example where this situation may happen. In the initial situation, there is one hanging node between subdomain Ω_1 and Ω_2, and there are no hanging nodes between Ω_1 and Ω_3. Thus, in a first step, Ω_1 adapts its subdomain to remove the only hanging node. But this introduce a new hanging node with Ω_3. This is removed within a second loop of the adaption algorithm. Therefore, this loop must be repeated until all processors accept the received substructure code without making local mesh changes. In all of our simulations, even for PDEs that require the mesh to be changed in every timestep, the mesh adaption algorithm terminates in a few iterations.

The parallel mesh adaption algorithm scales well for large number of processors, as it mostly contains point-to-point communication. Each processor communicates only with the directly neighbouring processors. The only global communication is the MPI reduction on the variable `changeOnRank`, to synchronize the loop. The most critical point is the question whether the number of iterations is independent of the number of subdomains. Theoretically, this scales at least in the order of $\mathrm{O}(\log(p))$, with p the number of subdomains. A situation where a logarithmic growth of the iteration number for parallel mesh adaptivity can be observed is when a very localized mesh refinement is done within only one subdomain. We will show in Section 3.2.4 that the iteration count only slightly increases with increasing number of subdomains, but is small (< 5) in all of our simulations.

Algorithm 4: Parallel mesh adaption

```
input : intBoundOwn and intBoundOther: interior boundary handler for the processor
        owned boundary segments and for all other interior boundary segments
bound = intBoundOwn ∪ intBoundOther
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 ≠ c1 then
                adapt (obj, c1)
                changeOnRank = true
            end
        end
    end
    mpi reduce on changeOnRank
until changeOnRank == false
```

3.2.2 Mesh partitioning and mesh distribution

A mesh partitioning is an assignment of process numbers to mesh elements, and thus it specifies all processor's subdomains to be used for the next calculations. In this work, we do not consider mesh partitioning techniques but assume that some efficient algorithm for parallel mesh partitioning are available. In our implementation, we use either ParMETIS [100, 99] or Zoltan [23, 29, 34]. Essential to our approach is that only coarse mesh elements are partitioned and therefore only information about the coarse mesh is forwarded to the mesh partitioner. In order to ensure good load balancing, each coarse mesh element is assign with a weight which is defined by the number of leaf elements within the coarse mesh element.

When a mesh partitioning is computed, a *mesh distributor* is used to move coarse mesh elements, and all data specified on them, from one processor's subdomain to another one. Once a coarse mesh element has been moved from processor A to processor B, processor B must also reconstruct the refinement structure that the coarse mesh element had before on processor A. For this, we make use of mesh structure codes, see Section 2.3. 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 must first create the corresponding coarse mesh elements on its local subdomain and use the mesh structure codes to reconstruct their refinement structure.

Besides reconstruction of the coarse mesh element refinement structure, reconstruction of the DOF vectors on these elements is the second main task of the mesh distributor. Not only the element structure must be communicated between ranks, but also the values that

are defined on them. For this, we make use of value mesh structure codes as described in Section 2.3. They are used for both, the reconstruction of elements and DOF vectors defined on them. This functionality of reconstructing an element refinement structure and a DOF vector is shown in Algorithm 5.

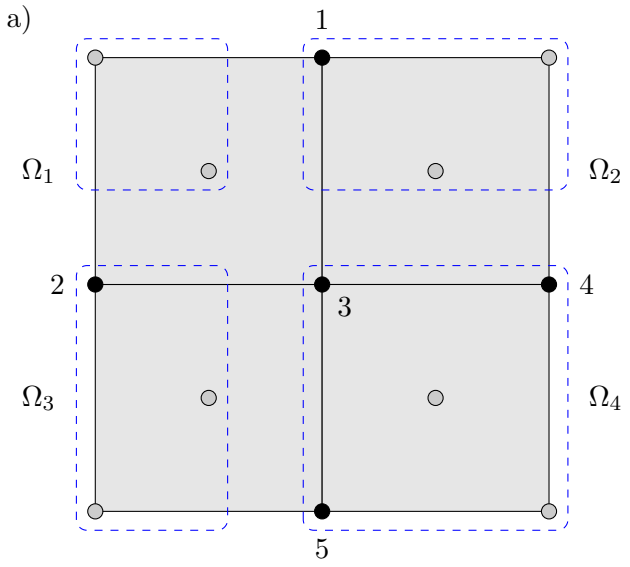
Algorithm 5: $reconstruct(el, code, valueCode, dofVec)$: Reconstruction of elements and DOF vectors using value mesh structure codes

```
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 rebuild: 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 3.2.3.

3.2.3 Parallel DOF mapping

When the domain is partitioned and distributed to all participating processors, the finite element method requires first to assemble local matrices and vectors. When there are d_i DOFs in domain Ω_i, the DOFs must be enumerated with a continuous local index set $1, \ldots, d_i$. For the solver method, which cannot be applied in a pure local way, the subdomain matrices and vectors must be related such that local DOF indices of two different subdomains which correspond to the same global DOF, also have the same global DOF index in both subdomains. This is the main work of the *parallel DOF mapping*, that maps from local DOF indices to global ones. For general solver methods it is important that these mappings can also be established on subsets of local and global DOFs. In Figure 3.5, 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 $1, \ldots, 13$. The figure 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 3.5 shows this mapping for subdomain Ω_3.



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

Figure 3.5: 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

Before going into details, let us recall the difference between the terms *vertex*, *node* and *Degree Of Freedom (DOF)*. A vertex is a geometric entity of the mesh. A node is a container for DOFs. For Lagrange basis functions, each vertex correlates with a node. When using higher order basis functions, nodes will occur also on element edges, faces or at element centers. When solving a PDE with one variable, each node contains exactly one DOF. In this case the terms are equivalent. If the PDE has more than one variable, nodes may contain more than one DOF. The number of DOFs per node can 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 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

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 main task of the parallel DOF handler is to provide a global index for a set of local DOF indices. For this, the local DOF indices must be sticked together to create a global index of all DOFs. 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

$$g_i : \mathcal{D}_i \mapsto \mathcal{D}$$
$$g_i(d) = \begin{cases} \sum_{j=1}^{i-1} \overline{n}_j + d & \text{if } d \in \overline{\mathcal{D}}_i \\ g_j(d') & \text{if } d \notin \overline{\mathcal{D}}_i,\ \mathcal{R}_j^i(d) = d' \text{ and } j = owner(d') \end{cases} \tag{3.1}$$

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 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, neighbouring ranks must communicate the global DOF indices along interior boundaries. As the number of neighbouring 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 the 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.

Index mappings from local to global DOF indices, which are defined only on subsets of DOFs, are mostly used to create distributed matrices and vectors. The definition of a global mapping allows directly for subassemling 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, so the solution of the system remains the same. In this work, we make use of the block-wise ordering, as it is simple to implement in a parallel environment. Figure 3.6 shows the different views on the local and global numbering of DOFs with multiple finite element spaces. In this example we assume that a PDE with three components is solved. The first two components are defined on the same finite element spaces. The third component is defined on a different one. For simplicity we assume the mesh to be equidistributed. Three computing nodes are used to solve this PDE. The left part of the Figure contains the rank view of the locally owned DOFs. The i-th rank contains in the finite element space of the j-th component $nRank_i^j$ DOFs. The

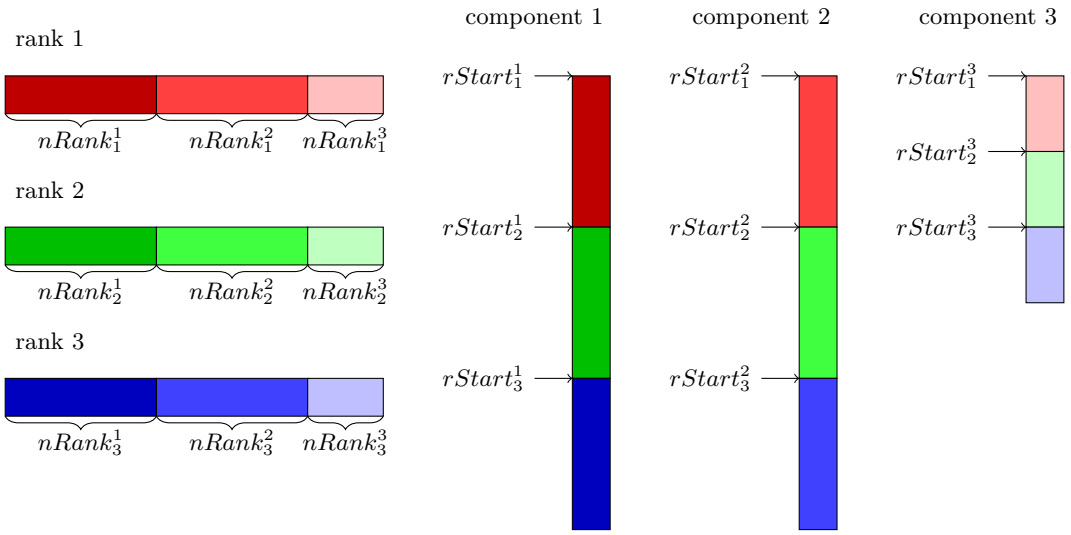


Figure 3.6: Example for global mapping with three components defined on two different finite element spaces

right part of the Figure shows the view of enumerating DOFs from the component view. 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 each local DOF index d in component number j to a unique matrix row index

$$matIndex_i(d,j) = \begin{cases} \sum_{k=1}^{j-1} rStart_i^k + \sum_{k=1}^{j-1} nRank_i^k + g_i^j(d) & \text{if } d \in \overline{\mathcal{D}}_i \\ matIndex_k(d',j) & d \notin \overline{\mathcal{D}}_i,\ \mathcal{R}_k^i(d) = d' \text{ and} \\ & k = owner(d') \end{cases} \tag{3.2}$$

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.

3.2.4 Efficiency and parallel scaling

To justify that our implementation of distributed meshes is efficient and scalable, we have chosen to simulate dendritic growth using a phase-field model in 3D, which today is the method of choice to simulate microstructure evolution during solidification. For a review we refer to [22]. From a technical point of view, using phase-field models in parallel computations is very challenging. The phase field is constant in most parts of the domain, where it can be discretized with a very coarse mesh. But to resolve the very thin transition between both phases, a highly refined mesh is required. As the phase moves during simulation time, the mesh is adapted at each timestep. In parallel computations, this leads to the following problems:

- As the mesh changes at each timestep on most domains, the parallel mesh adaptivity procedure to create a correct distributed mesh without hanging nodes must also be executed at each timestep.
- All parallel DOF mappings must be newly created at each timestep.
- Even if at one timestep the problem size is well balanced on all processors, it becomes unbalanced very fast due to the moving and growing phase field. Thus, mesh repartitioning and distribution must be executed every few timesteps to ensure good load balancing.

Altogether, these functions must show not only parallel scaling but their efficiency must be also comparable with the sequential running code for each subdomain.

A widely used model for quantitative simulations of dendritic structures was introduced in [62, 63], and reads in non-dimensional form

$$\begin{aligned} A^2(n)\partial_t\phi &= (\phi - \lambda u(1-\phi^2))(1-\phi^2) + \nabla\cdot(A^2(n)\nabla\phi) + \\ &\quad \sum_{i=1}^{d}\partial_{x_i}\left(|\nabla\phi|^2 A(n)\frac{\partial A(n)}{\partial_{x_i}\phi}\right) \\ \partial_t u &= D\nabla^2 u + \frac{1}{2}\partial_t\phi \end{aligned} \tag{3.3}$$

where $d = 2, 3$ is the dimension, D is the thermal diffusivity constant, $\lambda = \frac{D}{a_2}$, in which $a_2 = 0.6267$ is a coupling term between the phase-field variable ϕ and the thermal field u, and A is an anisotropy function. For both, simulation in 2D and 3D, we use the following anisotropy function

$$A(n) = (1-3\epsilon)\left(1 + \frac{4\epsilon}{1-3\epsilon}\frac{\sum_{i=1}^{d}\phi_{x_i}^4}{|\nabla\phi|^4}\right) \tag{3.4}$$

where ϵ controls the strength of the anisotropy and $n = \frac{\nabla\phi}{|\nabla\phi|}$ denotes the normal to the solid-liquid interface. In this setting the phase-field variable is -1 in the liquid and 1 in the solid domain, and the melting temperature is set to be zero. We set $u = -\xi$ as a boundary condition to specify an undercooling. For the phase-field variable we use zero-flux boundary conditions. The time integration is done using a semi-implicit Euler method, which yields a sequence of nonlinear stationary PDEs

$$\begin{aligned} \frac{A^2(n_n)}{\tau}\phi_{n+1} + f + g - \nabla(A^2(n_n)\nabla\phi_{n+1}) - \mathcal{L}[A(n_n)] &= \frac{A^2(n_n)}{\tau}\phi_n \\ \frac{u_{n+1}}{\tau} - D\nabla^2 u_{n+1} - \frac{1}{2}\frac{\phi_{n+1}}{\tau} &= \frac{u_n}{\tau} - \frac{1}{2}\frac{\phi_n}{\tau} \end{aligned} \tag{3.5}$$

with $f = \phi_{n+1}^3 - \phi_{n+1}$, $g = \lambda(1-\phi_{n+1}^2)^2 u_{n+1}$ and

$$\mathcal{L}[A(n_n)] = \sum_{i=1}^{d}\partial_{x_i}\left(|\nabla\phi_{n+1}|^2 A(n_n)\frac{\partial A(n_n)}{\partial_{x_i}\phi_n}\right)$$

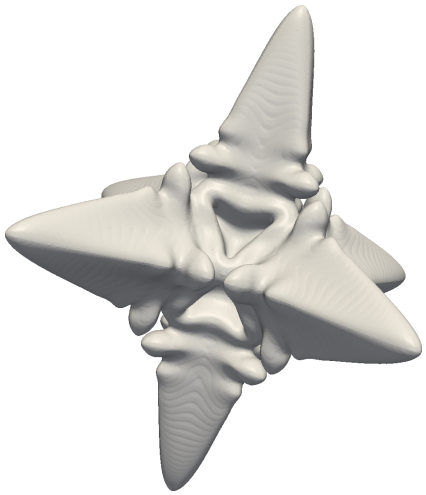


Figure 3.7: Dendritic structure at time $t = 3000$ for parameters $\xi = 0.55$, $D = 1.4$, and $\epsilon = 0.05$

We linearize the involved terms f and g with

$$\begin{aligned} f &\approx (3\phi_n^2 - 1)\phi_{n+1} - 2\phi_n^3 \\ g &\approx \lambda(1 - \phi_n^2)^2 u_{n+1} \\ \mathcal{L}[A(n_n)] &\approx \sum_{i=1}^{d} \partial_{x_i} \left(|\nabla \phi_n|^2 A(n_n) \frac{\partial A(n_n)}{\partial_{x_i} \phi_n} \right) \end{aligned} \tag{3.6}$$

to obtain a linear system for ϕ_{n+1} and u_{n+1} to be solved at each time step.

For the parallel 3D simulations, we choose the following parameters: $\xi = 0.55$, $D = 1.4$, $\epsilon = 0.05$. A constant timestep $\tau = 1.0$ is used, and the simulation is run for $7,500$ timesteps. For mesh adaptivity no error estimator is used, but instead we refine the mesh at each timestep along the interface such that there are at least 10 mesh vertices inside the phase transition. Outside of the phase transition, the mesh is coarsened as much as possible. As the dendritic growth is symmetric in all spatial dimensions, it is possible to make use of the solution's symmetry and restrict the computational domain to only one octant. As we want to create a 3D mesh that is as large as possible for benchmarking purposes, we abstain from this optimization. This simulation was performed with an increasing number of processors starting with 64 processors and increasing it up to 512 processors. The result at $t = 7,500$ was then restarted for further 50 timestep with 128 up to 4,096 processors. Figure 3.8 shows the average runtimes of these 50 timestep for the different number of processors. Time for local mesh adaption and the overall time for solving one timestep are also given to compare it with the time required for parallel mesh adaption. The rebuilding of parallel DOF mappings shows very good scalability, as it contains only a constant number of communication independently of the subdomain sizes. In contrast, the time for one parallel mesh adaption loop decreases only slightly between 512 and 4,096 processors. This is related to the increasing communication size as the number of interior boundaries

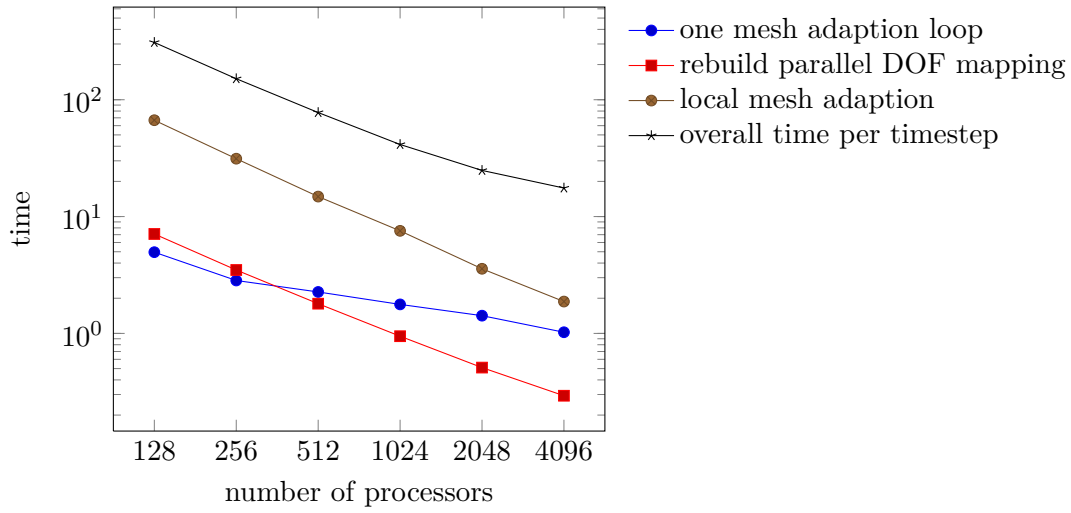


Figure 3.8: Strong scaling of parallel mesh adaptivity. The time is the average of 50 timesteps computing dendritic growth. Time for local mesh adaptivity and for the overall solution of one timestep are shown to compare with time needed for parallel mesh adaptivity.

increases when using more and more processors for a fixed problem size. Table 3.1 gives, among other information for this benchmark, the ratio of time required for parallel mesh adaptivity w.r.t. overall computational time for one timestep. Even though the time for parallel mesh adaptivity decreases slower than the overall computational time, it takes less than 10% of the overall time. Load unbalancing is measured w.r.t. the number of mesh vertices (which is proportional to the number of unknown in the resulting system of linear equations). If p processors are used, and the mesh of i-ith processor's subdomain contains V_i vertices, then *load unbalancing* of the overall problem is defined by the proportion of the maximal to the average load

$$\frac{\max_{i=1}^{p} V_i}{p^{-1} \sum_{i=1}^{p} V_i} \tag{3.7}$$

3.2.5 Limitations of coarse element based partitioning

All the concepts for distributed meshes that are presented in this work are based on partitioning and distribution of coarse mesh elements. This technique has many advantages: partitioning can be done much faster than on the leaf level of the mesh, it allows to use mesh structure codes for element redistribution and mesh substructure codes for local mesh adaption, pre- and post-processing steps for calculations are quite simple, as we have a direct correspondence between all mesh elements and processor numbers. In comparison with implementations that partition and distribute the mesh on leaf level, there are several disadvantages which are discussed in the following.

In our implementation, the coarse mesh is stored on all processors. Even for the case that the memory usage for storing the coarse mesh is considerably smaller than the available

cores	load unbalancing	parallel adaption iter	ratio	efficiency
128	4.44%	3.31	3.9%	100.0%
256	4.56%	3.54	4.1%	102.2%
512	6.03%	3.94	5.2%	99.5%
1,024	10.05%	3.98	6.5%	93.4%
2,048	18.14%	4.92	7.6%	77.8%
4,096	28.32%	4.82	7.5%	55.0%

Table 3.1: Strong scaling of parallel mesh adaptivity. All data is the average of 50 timesteps computing dendritic growth. The fourth column shows the ratio of time required in each timestep for parallel mesh adaptivity with the overall computing time for one timestep. The efficiency shown in the last column gives the efficiency of the overall computational time w.r.t using 128 cores.

memory of one processor, this limits the scaling for large number of processors, though we have not seen any problems in our benchmarks. The reader should note that this problem is not a conceptual one but depends on the specific implementation. Thus, it is possible to modify our implementation so that each processor must store only the coarse mesh element for its subdomain together with all neighbouring coarse mesh elements. During mesh redistribution, this information must also be communicated along with the refinement information of coarse mesh elements.

From a users point of view it is required to prepare a coarse mesh which is appropriate for the used number of processors and for the final refinement structure of the leaf level mesh. A general rule is to use around 10 to 25 coarse mesh elements for each processor to get a good load balancing. When the leaf level has a very localized refinement structure, e.g. because of a phase field as described in Section 3.2.4, more coarse mesh elements are required. As much information is stored on the level of coarse mesh elements, e.g. coordinates or interior boundaries, the memory footprint of a coarse mesh element is higher than for a refined mesh element. Therefore, it is always required to find a good balance between a coarse mesh that allows for good load balancing and one that does not require too much memory. In some very few configurations, when the leaf mesh has a singular refinement structure at one point in space, the situation can occur that it is not possible anymore to find an appropriate coarse mesh allowing to use a suitable number of processors.

3.3 Linear solver methods

In this and the following two sections, we consider parallel methods for the solution of large systems of linear equations resulting from the finite element method. Most of such methods can be classified to be either a *global matrix solver* or a *domain decomposition method*. A global matrix solver works on a distributed, globally assembled matrix and includes iterative Krylov subspace methods, parallel direct solvers and a large class of multigrid methods. Scalability of global matrix solvers is justified, for example, by the success of UNIC and PFLOTRAN. The UNIC neutron transport code [105, 64] scaled, with a problem size of more than 500 billion unknowns, to all 294,912 cores of JUGENE, a Blue Gene/P

system located at the Jülich Supercomputing Center (Germany). The PFLOTRAN code [92] scaled to 2 billion unknowns on 224,000 cores of a Cray XT5. Recently, [10] showed weak and strong scalability of algebraic multigrid methods on different systems up to 196,608 cores of the Cray XT5 "Jaguar" at the National Center for Computational Sciences at Oak Ridge National Laboratories (USA). Scaling of a geometric multigrid method on JUROPA Bkue Gene/P is considered in [48]. A comparison of scalability of geometric and algebraic multigrid up to 262,144 cores of the Cray XK6 "Jaguar" can be found in [113]. All of these works, which present scaling of parallel solver up to hundreds of thousands of cores, are highly tuned codes for one class of problems. Furthermore, the centerpiece of most of these codes is a solver for a discrete Laplace matrix. To the best of our knowledge, there exists no method or software package for the solution of large and sparse systems of linear equations, which has a broader application area and has shown scalability for more than 10^5 cores.

To decrease the iteration number, iterative methods require for an appropriate preconditioner. Usually, efficiency and scaling of the preconditioner are the limiting factors of many iterative methods. The creation of a purely algebraic based, parallel, robust and optimal (w.r.t. scaling) preconditioner for iterative Krylov subspace methods is still an open research question. In many applications, algebraic and geometric multigrid methods can be used as preconditioner of the original problem. Even though, these methods are not applicable in a black-box fashion, as they include several parameters and sub methods that must be adjusted to a specific equation. Besides pure algebraic motivated preconditioners, there is a lot of ongoing research to develop specialized preconditioners which work only for matrices resulting from discretization of a specific PDE. Most of them are based on block decomposition of the matrix, see e.g. [103] for the instationary Navier-Stokes equation, [24] for the Cahn-Hilliard equation, and [20] and references therein for a general overview on this topic.

In contrast to global matrix solvers, domain decomposition methods decouple the global problem in local subproblems, which can be solved independently of each other. The price for decoupling computations and thus introducing parallelism is the necessity to solve some additional 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, among others, Schwarz iterative algorithms [36], Schur complement approaches and iterative substructuring algorithms [87, 86], and the family of FETI-DP (finite element tearing and interconnecting - dual primal) [44, 43, 72, 75] and BDDC (balancing domain decomposition by constraints) [88, 90] methods.

FETI-DP and BDDC methods are well suited as block-box solvers for a the solution of equations leading from finite element discretization of a broad range of PDEs. The methods are nearly free of parameters, only the null space of the linear system must be considered in some way. Both methods include some global coarse mesh problem leading to optimal parallel scaling. The methods have been successfully applied to the solution of problems in elasticity [72, 75], fluid dynamics [68, 126] and in electromagnetics [130]. Parallel and numerical scalability of the FETI-DP method for up to 65,536 processors was show in [74]. Note that FETI-DP and BDDC methods are closely related to each other [80, 89], as they lead to linear systems that have the same eigenvalue spectrum, but for eigenvalues 0 and 1.

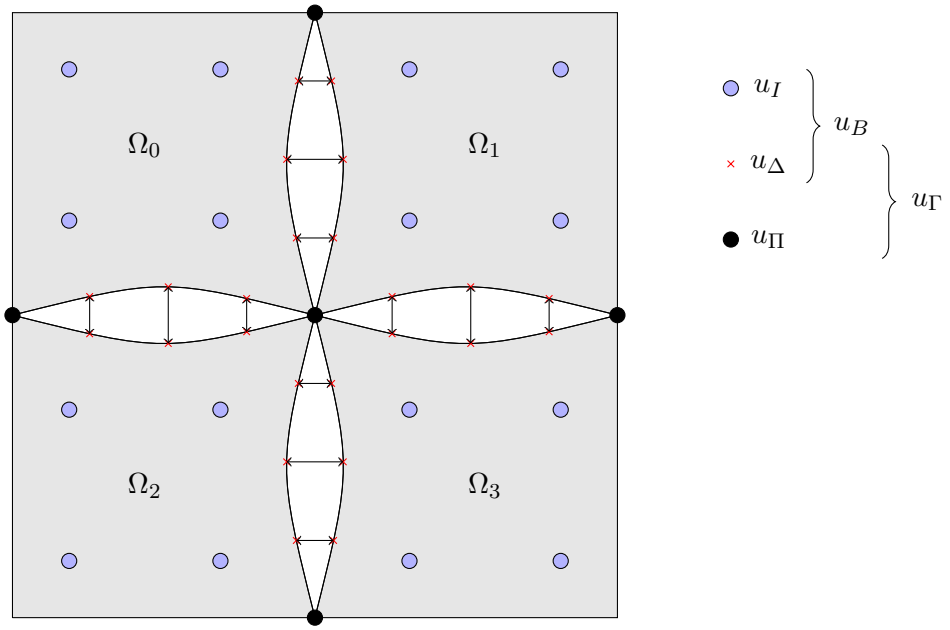


Figure 3.9: Partitioning of the unknowns for the FETI-DP method. Arrows between dual nodes u_Δ denote Lagrange multipliers which are used to enforce continuity of the solution at convergence.

3.4 FETI-DP

First, we give a comprehensive introduction to the FETI-DP method before we describe its efficient implementation in Section 3.4.1, based on the concepts presented in Section 3.2 before. Hereby we essentially follow the notation of the work [44, 72]. FETI-DP was originally established and proven to be optimal for symmetric positive definite matrices. Some work has been done to extend FETI-DP to more general matrices: In [45], a FETI-DP method for a class of complex-valued and indefinite systems is presented. Furthermore, FETI-DP was used in [79] to solve the stationary Navier-Stokes equation, that leads to non-symmetric and indefinite matrices. Optimality w.r.t. parallel scaling is shown experimentally in this work.

In what follows, we introduce the FETI-DP method for arbitrary real valued matrices, which may be non-symmetric and indefinite. Note that optimal parallel scaling cannot be proven in all cases, but we will experimentally show that for many PDEs, FETI-DP is a scalable and robust solution method.

Let a domain Ω be decomposed into non-overlapping subdomains Ω_i, with $i = 1, \ldots, p$, see Definition 1. Each processor assembles local matrices A^i and local right-hand side vectors f^i. On each subdomain, the vector of unknowns is denoted by u^i. For the first, let us assume that all A^i are non-singular. Then, all systems $A^i u^i = f^i$ can be uniquely solved independently of each other. To recover the solution of the original problem, we need to enforce continuity of the unknowns u^i across interior boundaries. The basic idea of the FETI-DP method is to split the unknowns on interior boundaries into the sets of *dual* and

primal unknowns and to ensure their continuity in different ways. Thus, 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_Δ^i and primal interior boundary unknowns u_Π^i

$$u^i = \begin{bmatrix} u_I^i \\ u_\Gamma^i \end{bmatrix} = \begin{bmatrix} u_I^i \\ u_\Delta^i \\ u_\Pi^i \end{bmatrix} = \begin{bmatrix} u_B^i \\ u_\Pi^i \end{bmatrix} \tag{3.8}$$

Continuity of the solution on primal nodes is enforced by global subassembly of the subdomain matrices A^i. To ensure continuity along dual nodes, we introduce Lagrange multipliers, cf. Figure 3.9. The FETI-DP system is then defined to iterate on these Lagrange multipliers. Consequently, continuity of the solution on primal nodes is satisfied in each iteration, continuity of the dual nodes only at convergence. The way how to choose the interior boundary nodes to be either dual or primal is crucial and will be described later.

As interior and dual variables are purely local on each subdomain, we collect them to the vector of local variables u_B^i. To be consistent with Definition 2, we denote the set of all primal and dual interior boundary nodes with $\mathcal{I}_\Pi$ 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.

To enforce continuity on the dual variables, we introduce a discrete jump operator J, such that the solution on the dual 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 $x_n \in \mathcal{I}_{i,\Delta}$, $x_m \in \mathcal{I}_{j,\Delta}$, $i \neq j$ and $\mathcal{R}_j^i(x_n) = x_m$.

If a dual node b corresponds to more than two subdomains, thus $degree(b) \geq 3$, there is some choice in the number of constraints for vertex b. We can either 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. Otherwise, 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 described later, but makes formal analysis of the FETI-DP method more complicated. A detailed discussion on this topic can be found in [91, Chapter 4.1].

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

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

and the total number of dual variables is given by

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

We denote by $\mathcal{C}(b,i,j) \mapsto [1,\ldots,\mathcal{C}_n]$, with $i,j \in \mathcal{W}(b)$ and $i < j$, the global index of the constraint associated to the dual node b on subdomains Ω_i and Ω_j. Then, the jump

operator matrix J is of size $\mathcal{C}_n \times \Delta_n$ and defined by

$$J_{k,l} = \begin{cases} 1 & \text{, if } \mathcal{C}(b,i,j) = k \text{ and } \widetilde{b}_i = l \\ -1 & \text{, if } \mathcal{C}(b,i,j) = k \text{ and } \widetilde{b}_j = l \\ 0 & \text{, otherwise} \end{cases} \tag{3.9}$$

where $\widetilde{b}_i = l$ denotes that there exists some component number c sucht that the local DOF index d of the dual node b in subdomain i maps to the global DOF index l, i.e. $matIndex_i(d,c) = l$, cf. 3.2. According to the splitting of unknown variables into primal, dual and local variables, we partition the local matrices A^i as follows

$$A^i = \begin{bmatrix} A^i_{BB} & A^i_{B\Pi} \\ A^i_{\Pi B} & A^i_{\Pi\Pi} \end{bmatrix}, \; A^i_{BB} = \begin{bmatrix} A^i_{II} & A^i_{I\Delta} \\ A^i_{\Delta I} & A^i_{\Delta\Delta} \end{bmatrix}, \; A^i_{B\Pi} = \begin{bmatrix} A^i_{I\Pi} \\ A^i_{\Delta\Pi} \end{bmatrix}, \; A^i_{\Pi B} = \begin{bmatrix} A^i_{\Pi I} & A^i_{\Pi\Delta} \end{bmatrix} \tag{3.10}$$

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 restriction matrices R^i_Π, which restrict the global vector of primal nodes to to primal nodes contained in partition i

$$\widetilde{A}_{\Pi\Pi} = \sum_{i=1}^{p} {R^i_\Pi}^T A^i_{\Pi\Pi} R^i_\Pi \tag{3.11}$$

and

$$\widetilde{A}^i_{B\Pi} = A^i_{B\Pi} R^i_\Pi, \; \widetilde{A}^i_{\Pi B} = {R^i_\Pi}^T A^i_{\Pi B} \tag{3.12}$$

$$A_{BB} = \operatorname{diag}^p_{i=1}(A^i_{BB}), \; \widetilde{A}_{B\Pi} = \begin{bmatrix} \widetilde{A}^1_{B\Pi} \\ \vdots \\ \widetilde{A}^p_{B\Pi} \end{bmatrix}, \; \widetilde{A}_{\Pi B} = \begin{bmatrix} \widetilde{A}^1_{\Pi B} & \dots & \widetilde{A}^p_{\Pi B} \end{bmatrix} \tag{3.13}$$

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

$$\widetilde{A} = \begin{bmatrix} A_{BB} & \widetilde{A}_{B\Pi} \\ \widetilde{A}_{\Pi B} & \widetilde{A}_{\Pi\Pi} \end{bmatrix} \text{ and } \widetilde{f} = \begin{bmatrix} f_B \\ \widetilde{f}_\Pi \end{bmatrix} \tag{3.14}$$

and finally add the discrete jump operator J to ensure solution continuity across interior boundaries

$$\begin{bmatrix} A_{BB} & \widetilde{A}_{B\Pi} & J^T \\ \widetilde{A}_{\Pi B} & \widetilde{A}_{\Pi\Pi} & 0 \\ J & 0 & 0 \end{bmatrix} \begin{bmatrix} u_B \\ \widetilde{u}_\Pi \\ \lambda \end{bmatrix} = \begin{bmatrix} f_B \\ \widetilde{f}_\Pi \\ 0 \end{bmatrix} \tag{3.15}$$

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

$$\begin{bmatrix} J & 0 \end{bmatrix} \begin{bmatrix} A_{BB} & \widetilde{A}_{B\Pi} \\ \widetilde{A}_{\Pi B} & \widetilde{A}_{\Pi\Pi} \end{bmatrix}^{-1} \begin{bmatrix} J^T \\ 0 \end{bmatrix} \lambda = \begin{bmatrix} J & 0 \end{bmatrix} \begin{bmatrix} A_{BB} & \widetilde{A}_{B\Pi} \\ \widetilde{A}_{\Pi B} & \widetilde{A}_{\Pi\Pi} \end{bmatrix}^{-1} \begin{bmatrix} f_B \\ \widetilde{f}_\Pi \end{bmatrix} \tag{3.16}$$

$$\begin{aligned}\begin{bmatrix} A_{BB} & \widetilde{A}_{B\Pi} \\ \widetilde{A}_{\Pi B} & \widetilde{A}_{\Pi\Pi} \end{bmatrix}^{-1} &= \begin{bmatrix} I & -A_{BB}^{-1}\widetilde{A}_{B\Pi} \\ 0 & I \end{bmatrix} \begin{bmatrix} A_{BB}^{-1} & 0 \\ 0 & \widetilde{S}_{\Pi\Pi}^{-1} \end{bmatrix} \begin{bmatrix} I & 0 \\ -\widetilde{A}_{\Pi B}A_{BB}^{-1} & I \end{bmatrix} \\ &= \begin{bmatrix} A_{BB}^{-1} + A_{BB}^{-1}\widetilde{A}_{B\Pi}\widetilde{S}_{\Pi\Pi}^{-1}\widetilde{A}_{\Pi B}A_{BB}^{-1} & -A_{BB}^{-1}\widetilde{A}_{B\Pi}\widetilde{S}_{\Pi\Pi}^{-1} \\ -\widetilde{S}_{\Pi\Pi}^{-1}\widetilde{A}_{\Pi B}A_{BB}^{-1} & \widetilde{S}_{\Pi\Pi}^{-1} \end{bmatrix}\end{aligned} \tag{3.17}$$

with the primal Schur complement

$$\widetilde{S}_{\Pi\Pi} = \widetilde{A}_{\Pi\Pi} - \widetilde{A}_{\Pi B}A_{BB}^{-1}\widetilde{A}_{B\Pi} \tag{3.18}$$

In short notation, the system can be written as

$$F\lambda = d \tag{3.19}$$

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

$$\begin{aligned} F &= JA_{BB}^{-1}(I + \widetilde{A}_{B\Pi}\widetilde{S}_{\Pi\Pi}^{-1}\widetilde{A}_{\Pi B}A_{BB}^{-1})J^T \\ d &= JA_{BB}^{-1}(f_B - \widetilde{A}_{B\Pi}\widetilde{S}_{\Pi\Pi}^{-1}(\widetilde{f}_\Pi - \widetilde{A}_{\Pi B}A_{BB}^{-1}f_B)) \end{aligned} \tag{3.20}$$

Note that the matrix F is never build 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.

To precondition the FETI-DP system, two different preconditioners are commonly used: the Dirichlet preconditioner P_D^{-1} and the lumped preconditioner P_L^{-1} [44, 75]. We additionally define the following block matrices

$$\begin{aligned} A_{II} &= \mathrm{diag}_{i=1}^{p}(A_{II}^i) \\ A_{I\Delta} &= \mathrm{diag}_{i=1}^{p}(A_{I\Delta}^i) \\ A_{\Delta I} &= \mathrm{diag}_{i=1}^{p}(A_{\Delta I}^i) \\ A_{\Delta\Delta} &= \mathrm{diag}_{i=1}^{p}(A_{\Delta\Delta}^i) \end{aligned} \tag{3.21}$$

The Dirichlet preconditioner is then defined by

$$P_D^{-1} = J_S R_\Delta^{B^T}(A_{\Delta\Delta} - A_{\Delta I}A_{II}^{-1}A_{I\Delta})R_\Delta^B J_S^T \tag{3.22}$$

The lumped preconditioner is obtained by simplifying the Dirichlet preconditioner to its leading term $A_{\Delta\Delta}$

$$P_L^{-1} = J_S R_\Delta^{B^T} A_{\Delta\Delta} R_\Delta^B J_S^T \tag{3.23}$$

Here, R_Δ^B are restriction matrices which restrict the non primal indices on each subdomain to the dual variables. J_S is a scaled variant of the jump operator J, where the contribution from and to each dual node is scaled by the inverse of the multiplicity of the node. The lumped preconditioner reduces computational complexity, but optimality w.r.t. parallel scaling of the FETI-DP method can be proven only when using the Dirichlet preconditioner. In this case, the condition number of the preconditioned system grows asymptotically as [75]

$$\mathrm{O}\left(1 + \log^2\left(\frac{H}{h}\right)\right) \tag{3.24}$$

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 fix 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 the number of processor is increased. Here, H/h is constant and thus also the condition number of the FETI-DP system remains constant.

3.4.1 Implementation issues

We now discuss the efficient implementation of the FETI-DP method based on algorithms and data structures described in Section 3.2. This includes:

1. creating all nodes on interior boundaries and deciding them to be either primal or dual nodes,
2. creating matrices A_{BB}, $\widetilde{A}_{B\Pi}$, $\widetilde{A}_{\Pi B}$, $\widetilde{A}_{\Pi\Pi}$ and J,
3. creating 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 (3.19).

Creating parallel DOF mappings

The partitioning of the unknowns, and thus also the corresponding vectors and matrices, into sets of interior, dual and primal is a key concept of the FETI-DP method. Our implementation of the algorithms and data structures described in Section 3.2 allows for an efficient implementation with a small amount of source code. For the first, we consider creating the set of primal nodes in 2D. Thus we have to define all cross points of interior domains to be primal. For simplicity, we assume that the PDE to be solved consists of only one component. In our FETI-DP implementation, the source code for creating a parallel DOF mapping of primals reads

```
1  ParallelDofMapping primals(COMPONENT_WISE);
2  primals.init(...);
3  DofContainerSet& vertices =
4     meshDistributor->getBoundaryDofInfo().geoDofs[VERTEX];
5
6  for (DofContainerSet::iterator it = vertices.begin();
7   it != vertices.end(); ++it) {
8     if (meshDistributor->isRankDof(*it))
9        primals.insertRankDof(*it);
10    else
11       primals.insertNonRankDof(*it);
12 }
13 primals.update();
```

The first line creates a new parallel DOF mapping object, which may define different mappings for different component. Another choice is to fix the mapping for all components sharing the same finite element space. The second line of the code is used to initialize the

parallel DOF mapping. During initialization, the FETI-DP solver object registers on the global mesh distributor that the solver requires geometrical information of the interior boundary nodes and that it must access these nodes by their geometrical entity. This information is created on initialization of the mesh distributed, stored in the boundary DOF info object (cf. Section 3.2), and will be updated after each mesh repartitioning. The third line of the code queries for this information. Then the code loops over all vertex cross points of all interior boundaries and inserts them to the parallel DOF mapping of primal DOFs. It uses the mesh distributer to decide whether the particular primal DOFs are owned by the rank or not. The final call to the `update()` function eventually creates the global DOF mapping of primal DOFs. In the general case, when a PDE contains more than one variable, the code above is extended by an outer loop iterating over the different components. In this way, it is straightforward to specify different selection algorithms for primal DOFs of different components.

Creating a parallel DOF mapping for the dual DOFs is as simple as it is for primal ones. Within our FETI-DP implementation, the source code reads

```
1  ParallelDofMapping duals(COMPONENT_WISE);
2  duals.init(...);
3  DofContainer allBoundaryDofs;
4  meshDistributor->getAllBoundaryDofs(allBoundaryDofs);
5
6  for (DofContainer::iterator it = allBoundaryDofs.begin();
7   it != allBoundaryDofs.end(); ++it)
8     if (primals.isSet(**it) == false)
9        duals.insertRankDof(**it);
10 duals.update();
```

The first two lines create and initialize a new parallel DOF mapping object. Then, the mesh distributer is asked to create a set of all boundary DOFs. For this, the function `getAllBoundaryDofs()` collects all DOFs from all DOF communicator objects on each subdomain. From this set of DOFs, all non primal DOFs are added to the parallel DOF mapping of dual DOFs. Note that we do not need to distinguish here between rank owned or not rank owned DOFs, as dual nodes are always local in each subdomain.

Creating matrices and vectors

Finally, the FETI-DP method must create matrices and vectors based on the parallel DOF mappings defined before. For these data structure, we make use of PETSc [14, 13], which also provides a rich set of operations on these distributed data structures, e.g. matrix-vector multiplications and a large set of iterative Krylov subspace solver methods.

One of the key points in our flexible FETI-DP implementation is to allow the subdomain and the subdomain solver to be an arbitrary PETSc based solver class, see also Section 3.7 for more details on the solver class structure. After computing the parallel DOF mappings for local and primal DOFs, the subdomain data structures can be initialized with only four lines of code

```
1  subDomain->setDofMapping(&locals);
2  subDomain->setCoarseSpaceDofMapping(&primals);
3  subDomain->fillPetscMatrix(mat);
4  subDomain->fillPetscRhs(vec);
```

The first two lines inform the subdomain solver to use the local DOF mapping to define the local subdomain, and to use the primal DOF mapping to specify a coarse space matrix. Thereafter, the locally created matrices and vectors are forwarded to the subdomain which creates the corresponding PETSc data structures, which may be either distributed or also of pure local type. To work on them, the variable `subDomain` provides access to the assemble matrices via `getMatInterior()` to get A_{BB}, `getMatCoarse()` for the coarse space matrix $\widetilde{A}_{\Pi\Pi}$, `getMatInteriorCoarse()` and `getMatCoarseInterior()` for the coupling matrices $\widetilde{A}_{B\Pi}$ and $\widetilde{A}_{\Pi B}$. Similar functions to access the corresponding vectors are also available.

Within the FETI-DP algorithm, it is often the case to create temporary vectors for intermediate results. This functionality is directly provided within the implementation of parallel DOF mappings and can be done with only two lines of code

```
1 Vec petscVecPrimals;
2 primals.createVec(petscVecPrimals);
```

We further have to create the matrix J. Each row of this matrix specifies the jump between two nodes on interior boundaries. Therefore, each row has exactly two entries, 1 and -1, connecting two dual nodes in two different subdomains, see Algorithm 6. It requires one DOF communicator for the interior boundaries, which is used to get all subdomain indices of each dual node, thus to compute $\mathcal{W}$, cf. Definition 4. Furthermore, the algorithm makes use of two parallel DOF mappings: one for the 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.

Solving for the Schur complement system $\widetilde{S}_{\Pi\Pi}$

There are two different ways to implement the solution of system with the operator $\widetilde{S}_{\Pi\Pi}$, see 3.18. Either its action on a vector is implemented and an iterative matrix-free 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 A_{BB} and one vector-vector addition. As A_{BB} is a block matrix, and for standard exact FETI-DP one block corresponds to one local subdomain, solving with A_{BB} can be done independently of each other and without communication. When a direct solver is used for the local solution with A^i_{BB}, the LU or Cholesky factorization of the local matrices must be computed only once. Then, in each FETI-DP iteration solving with A_{BB} and some right-hand side vector reduces two only one backward and one forward solver with triangular matrices, for which the computational costs are mostly negligible compared to matrix factorization.

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 a sparsity structure that is similar to those of the original system. It highly depends on the number of FETI-DP iterations required to solve a linear system and to the number of processors used in the computation, whether a directly assembled and factorized Schur complement matrix, or an iterative procedure to solve the Schur complement results in a faster overall solution procedure. Algorithm 7 provides the information how to explicitly

Algorithm 6: Computation of the matrix J

```
input  : Set of local dual node indices I_{i,Δ}, DOF communicator object dofComm,
         parallel DOF mapping constraints for global indices of Lagrangian constraints,
         parallel DOF mapping locals for the local DOFs
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 = constraints.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 = locals.matIndex(x)
                if W(i) == mpiRank then
                    value = 1
                else
                    value = −1
                J[matRowIndex][matColIndex] = value
            end
            matRowIndex++
        end
    end
end
```

compute the primal Schur complement matrix. We have implemented both methods and will compare their efficiency in Section 3.4.2.

Solving the FETI-DP system

The algorithm to apply the FETI-DP operator, cf. (3.20), on a vector is described in Algorithm 8. 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. If not other stated, the solver is stopped if the absolute residual is reduced to less than 10^{-8}. The same solver and stopping criteria are 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 [30, 31].

3.4.2 Numerical results

Weak scaling for phase field crystal

First of all, solving the Phase Field Crystal (PFC) equation in 2D is used to examine weak scaling of the FETI-DP method. The equation was introduced in [37] as a model

Algorithm 7: 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 8: 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 $A_{BB}t_1 = t_0$
$v = Bt_1$

for elasticity on atomic scales. It can be derived as an approximation to classical density functional theory [38, 118]. The simplest dimensionless form of the free energy is

$$\mathcal{F} = \int \frac{1}{2}\psi\left(-\epsilon + (\Delta + 1)^2\right)\psi + \frac{1}{4}\psi^4 \, d\mathbf{x} \tag{3.25}$$

which leads to the following conserved evolution law

$$\partial_t \psi = \Delta\left\{(-\epsilon + (1 + \Delta)^2)\psi + \psi^3\right\} \tag{3.26}$$

where ψ is a rescaled density field of the underlying particles. The density field minimizing the energy (3.25) is peaked at the atomic positions in the crystalline state and homogeneous in the liquid state. In order to solve the 6^{th} order PDE, we rewrite (3.26) as a system of three second order equations

$$\begin{aligned} v &= \Delta\psi, \\ \partial_t\psi &= \Delta u, \\ u &= (1-\epsilon)\psi + 2\Delta\psi + \Delta v + \psi^3 \end{aligned} \tag{3.27}$$

The derivative of the potential is linearized with $(\psi^{n+1})^3 \approx 3(\psi^n)^2\psi^{n+1} - 2(\psi^n)^3$. A brief review of the finite element discretization of the PFC equation, that uses a semi-implicit time discretization, is stated in [7].

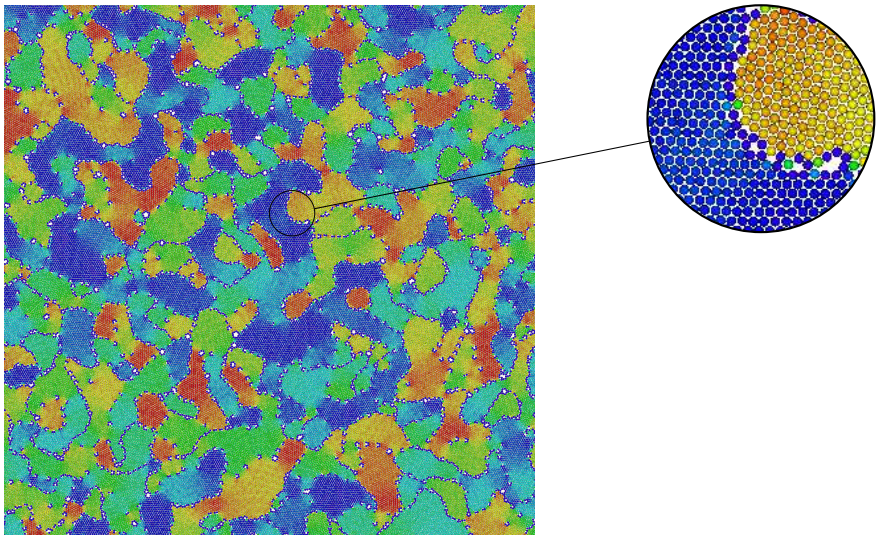


Figure 3.10: Snapshot of the solution of the PFC equation using 4,096 cores. Each maximum of the density functional is shown here with a circle and colored with its local orientation. Zoom shows two grains having different orientations and which are separated from each other by a grain boundary. Postprocessing and visualization of the data is done with OVITO [111, 112].

For the benchmarks calculations, the coarse mesh is always setup up such that each processor contains a rectangular subdomain of dimension 100×100, consisting of two coarse elements. 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. These large scale simulations are used by the author of this thesis to analyze grain coarsening [8]. See Figure 3.10 for an illustration.

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 also be formulated in a symmetric, but still indefinite way, but loses diagonal dominance. This 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 3.11 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 performs better for small size computations, but shows bad scaling for larger number of processors. When using 4,096 processors, the direct solver was not able to compute the very first timestep within 30 minutes. The main reason for this behavior is the structure of the coarse grid. All processors contribute to the coarse

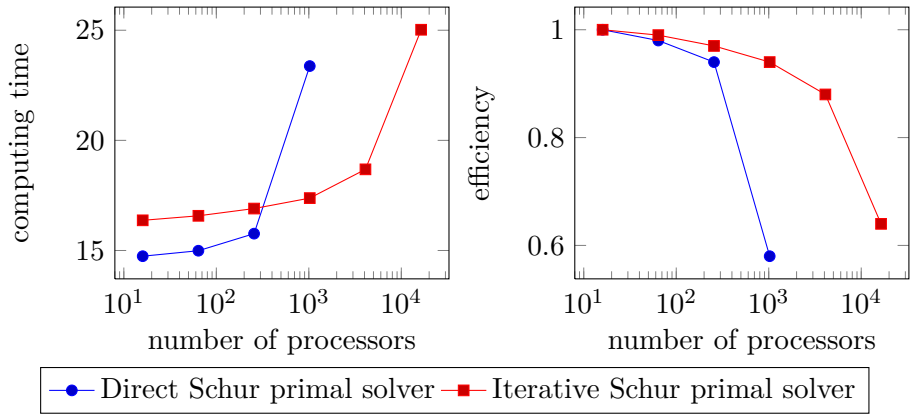


Figure 3.11: Computing time for one time timestep of a 2D PFC computation using FETI-DP with two different solvers for the Schur primal system.

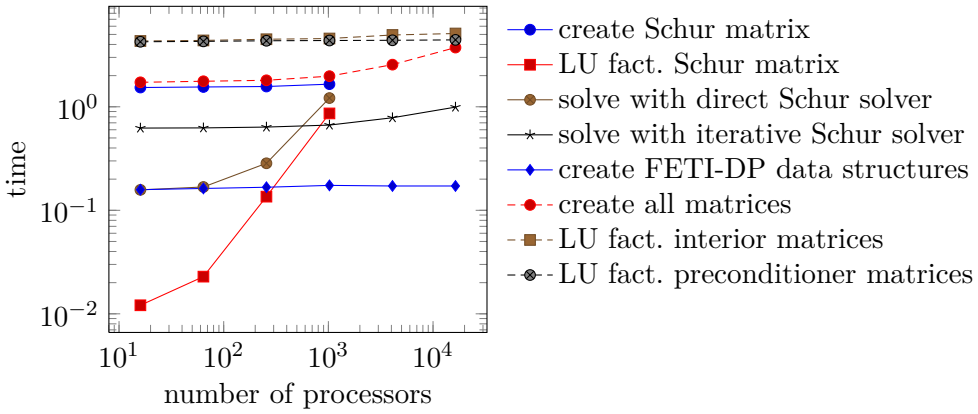


Figure 3.12: Weak scaling of the FETI-DP method

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 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. If this number increases, the direct Schur primal solver becomes even more efficient for small domain sizes. The performance of the FETI-DP implementation is analyzed in more detail in Figure 3.12. 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. This is mainly due to the very sparse coarse mesh and is mostly responsible for observed decreasing

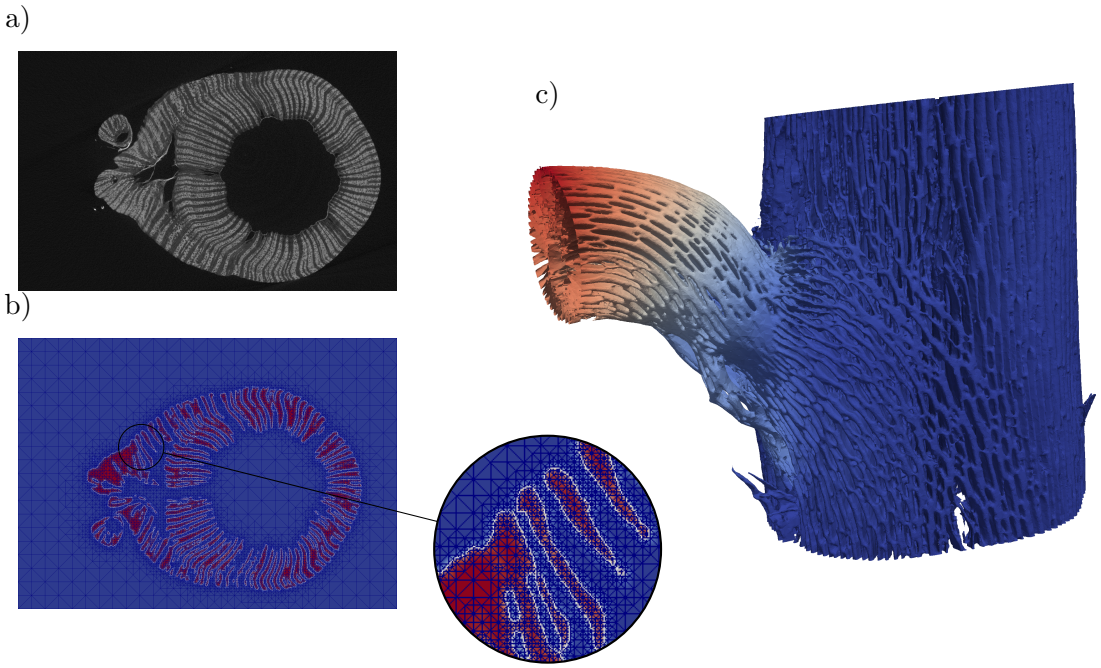


Figure 3.13: Computing linear elasticity in a cactus. a) One μCT image of the cactus b) Corresponding slice of the mesh colored with the phase field. The white colored 0.5 contour of the phase field represents the geometry. c) Visualization of the results on the implicit defined cactus geometry colored with the magnitude of the displacement field. Preprocessing of the geometry and calculations are done by Michael Wenzlaff.

computational efficiency of the FETI-DP implementation for large processor numbers. FETI-DP's setup phase, i.e. creating primal, dual and interior node information and the corresponding parallel DOF mappings, needs a constant time on a very low level. Creating all required matrices shows a small increase in time for larger number of processors. This is also due to the globally distributed and very sparse coarse space matrices $\widetilde{A}_{\Pi\Pi}$, $\widetilde{A}_{\Pi B}$ and $\widetilde{A}_{B\Pi}$. Still, if the number of outer FETI-DP iterations is large enough, this does not play an important role the for overall efficiency.

Strong scaling for linear elasticity

We consider a diffuse domain approximation [82] of a linear elasticity problem in biomechanics. The lamellar structure of a columnar cactus is analyzed using the FETI-DP method, see [128] for more information about these calculations. The geometry results from preprocessing a stack of μCT images of the cactus. Meshconv [110] is used to create an implicitly defined geometry of the cactus and an appropriately refined mesh. Figure 3.13 shows an example of a μCT image of the cactus, a slice through the mesh at the corresponding position and the result of the computation colored with the magnitude of the displacement field. The mesh has more than 55 million elements and around 11 million vertices.

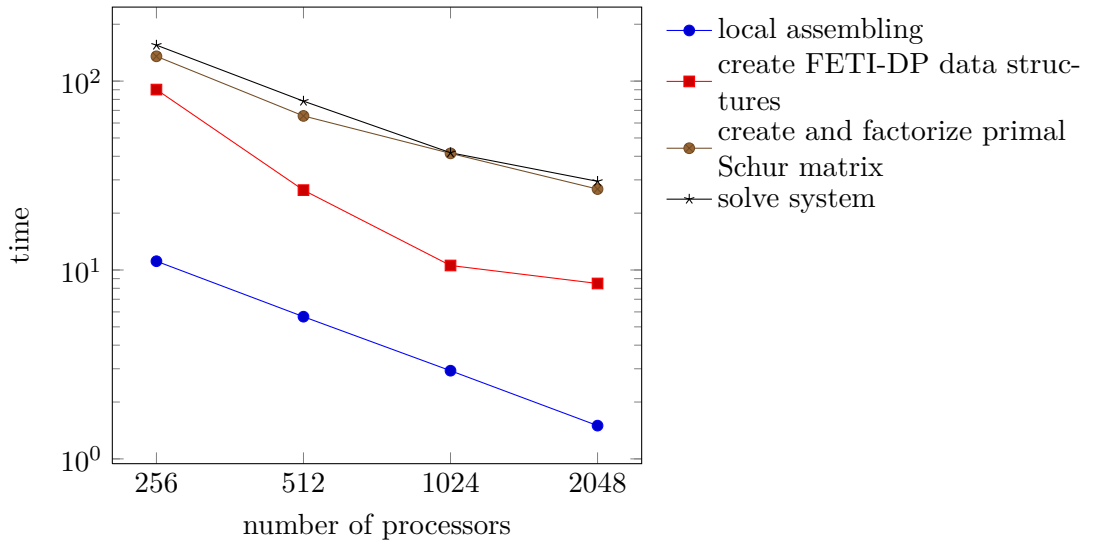


Figure 3.14: Strong scaling of the FETI-DP method for computing linear elasticity in a 3D diffuse domain

cores	**avrg. unknowns**	**unbalancing**	**runtime** [s]	**efficiency**
256	40,934	6.7%	380.46	100%
512	20,934	10.96%	170.17	111.7%
1,024	10,740	18.46%	85.7	110.9%
2,048	5,532	25.63%	64.75	73.4%

Table 3.2: Data for benchmarking the FETI-DP solver of linear elasticity in a 3D diffuse domain configuration. The load unbalancing is defined w.r.t. the unknowns of the linear system. Efficiency is computed w.r.t the calculation with 256 processors.

To examine strong scaling of the FETI-DP method, we performed this computation with a varying number of processors ranging from 256 to 2,048. Scaling results are shown in Figure 3.14. Therein, the time for creating and factorizing the local matrices is included in creating FETI-DP data structures. Computing the LU factorization is responsible for more than 95% of this time. Overall runtime, load unbalancing information and overall efficiency of the solver method are presented in Table 3.2. The super scalar speedup for computations with 512 and 1,024 processors is related to scaling of the multifrontal direct solver. The computational complexity of factorizing an $n \times n$ matrix, resulting from discretization of a PDE with the finite element method in 3D, is in $\mathrm{O}\left(n^2\right)$. Thus, when the size of the subdomain is halved, time for local factorization is quartered. In opposite to this, the size of the coarse space problem, i.e. the Schur primal matrix, is increased.

3.5 Extensions of the standard FETI-DP

Though the standard FETI-DP method is quasi optimal from a mathematical point of view, there are two drawbacks: the use of direct solvers for the factorization of local matrices and the Schur primal matrix, and a very sparse and globally distributed coarse grid. Using direct solvers for 2D problems is very efficient, but the computational scaling is totally different for 3D problems. With an optimal ordering strategy, a direct solver requires $\mathrm{O}\left(n^{3/2}\right)$ floating point operations to factorize an $n \times n$ matrix resulting from the discretization of a PDE in 2D. The complexity increases to $\mathrm{O}\left(n^2\right)$ floating point operations, if the PDE is discretized in a 3D domain. Furthermore, memory requirements scales $\mathrm{O}\left(n \log n\right)$ in 2D, but $\mathrm{O}\left(n^{4/3}\right)$ in 3D. This prevents the use of FETI-DP for very large 3D simulations. To circumvent the problem, the exact local solver must be replaced by some approximate solutions. This idea leads to *inexact FETI-DP* methods, which are considered in Section 3.5.1.

The primal Schur matrix is a distributed matrix with a very low ratio of rows per processor. In the 2D benchmarks presented in Section 3.4.2, each processor contains at most nine rows of the primal Schur matrix. For using 4,096 processors, the matrix is of size 12663×12663 and is very sparse. Factorizing this matrix locally on one processor can be done with a nearly negligible amount of computing time and memory. But as the matrix is stored in a distributed way, the amount of communication is extremely high in contrast to the computation that can be done locally. In our benchmarks, the multifrontal direct solver MUMPS failed to compute the factorization of this matrix. The same difficultly is observed and discussed in [73, 74]. One possibility to circumvent this problem is to define a multilevel method, which introduces some hierarchical decomposition of the coarse space. For FETI and FETI-DP methods, this is considered in [73, 74, 76]. Similar approaches are considered for the BDDC method in [90, 125]. In Section 3.5.2, we present a novel *multilevel* FETI-DP approach. Our multilevel FETI-DP method differs in several points from the existing approaches. In contrast to the work [73, 74], our method is based on the FETI-DP method only, and does not require the computation of pseudoinverse matrices, which are needed in the FETI method. Though only presented for a two level decomposition of the coarse grid, our multilevel FETI-DP approach directly generalizes to the general case of arbitrary number of levels. The main advantage of our approach is the independency of the number of iterations with respect to the decomposition of the coarse grid. Though this is not proven in this work, it is highly indicated by the presented benchmarks.

3.5.1 Inexact FETI-DP

In inexact FETI-DP methods, the factorization of local subdomain matrices and factorization of the primal Schur matrix can be replaced by some inexact solution method. Inexact FETI-DP methods are first considered in [71, 70]. For a brief introduction into these methods, we consider only symmetric matrices, thus we can write the FETI-DP saddle point problem as

$$\begin{bmatrix} A_{BB} & \widetilde{A}_{\Pi B}^T & J^T \\ \widetilde{A}_{\Pi B} & A_{\Pi\Pi} & 0 \\ J & 0 & 0 \end{bmatrix} \begin{bmatrix} u_B \\ \widetilde{u}_\Pi \\ \lambda \end{bmatrix} = \begin{bmatrix} f_B \\ f_\Pi \\ 0 \end{bmatrix} \tag{3.28}$$

or in short form as

$$\begin{bmatrix} \widetilde{A} & B^T \\ B & 0 \end{bmatrix} \begin{bmatrix} \widetilde{u} \\ \lambda \end{bmatrix} = \begin{bmatrix} \widetilde{f} \\ 0 \end{bmatrix} \tag{3.29}$$

or simply as

$$\mathcal{A}x = \mathcal{F} \tag{3.30}$$

In [71, 70], two different inexact FETI-DP methods are proposed. The *iFETI-DP* method for the full saddle point system, and the so called *irFETI-DP* method defined for a reduced system. The first one is defined by specifying the following block triangular preconditioner for the system (3.29)

$$\widehat{\mathcal{B}}_L^{-1} = \begin{bmatrix} \widehat{A}^{-1} & 0 \\ M^{-1}B\widehat{A}^{-1} & -M^{-1} \end{bmatrix} \tag{3.31}$$

where the block $\widehat{A}^{-1}$ is a spectrally equivalent preconditioner for $\widetilde{A}$, with bounds independent of the discretization parameters h and H, and M^{-1} is a good preconditioner for the FETI-DP system (3.20), i.e. it can be chosen to be the Dirichlet preconditioner (3.22) or the lumped preconditioner (3.23), respectively. The first inexact method is then given by using an appropriate Krylov subspace method, e.g. GMRES, for the non-symmetric preconditioner system

$$\widehat{\mathcal{B}}_L^{-1}\mathcal{A}x = \widehat{\mathcal{B}}_L^{-1}\mathcal{F} \tag{3.32}$$

The standard FETI-DP method is derived from (3.28) by eliminating u_B and $\widetilde{u}_\Pi$. Instead, to define the second inexact FETI-DP method (irFETI-DP), we eliminate only u_B by one step of block Gaussian elimination and obtain the reduced system

$$\begin{bmatrix} \widetilde{S}_{\Pi\Pi} & -\widetilde{A}_{\Pi B}A_{BB}^{-1}J^T \\ -JA_{BB}^{-1}\widetilde{A}_{\Pi B}^T & -JA_{BB}^{-1}J^T \end{bmatrix} \begin{bmatrix} \widetilde{u}_\Pi \\ \lambda \end{bmatrix} = \begin{bmatrix} \widetilde{f}_\Pi - \widetilde{A}^{\Pi B}A_{BB}^{-1}f_B \\ -JA_{BB}^{-1}f_B \end{bmatrix} \tag{3.33}$$

where $\widetilde{S}_{\Pi\Pi} = \widetilde{A}_{\Pi\Pi} - \widetilde{A}_{\Pi B}A_{BB}^{-1}\widetilde{A}_{\Pi B}^T$. For this system, we also use the short notation

$$\mathcal{A}_r x_r = \mathcal{F}_r \tag{3.34}$$

For this saddle point problem, the following block preconditioner is proposed in [71]

$$\mathcal{B}_{r,L}^{-1} = \begin{bmatrix} \widehat{S}_{\Pi\Pi}^{-1} & 0 \\ -M^{-1}JA_{BB}^{-1}\widetilde{A}_{\Pi B}^T\widehat{S}_{\Pi\Pi}^{-1} & -M^{-1} \end{bmatrix} \tag{3.35}$$

Here, $\widehat{S}_{\Pi\Pi}^{-1}$ is assumed to be a spectrally equivalent preconditioner for $\widetilde{S}_{\Pi\Pi}$, independent of the discretization parameters h and H, and M^{-1} is a good preconditioner for the FETI-DP system.

Both inexact FETI-DP methods allow to replace the exact solver for the interior matrices and for the Schur primal matrix by some approximation methods. In [71], it was proven that the condition number of both methods is asymptotically of the same quality as for the standard, exact FETI-DP methods. In [71, 70, 74], detailed benchmarks are presented for both inexact FETI-DP methods. They show that inexact FETI-DP method can dramatically reduce computational time for solving large systems, for some simulations even more than one order of magnitude. The convergence rates of the inexact methods are comparable to those of the standard FETI-DP method if good approximative solvers are used as preconditioning blocks.

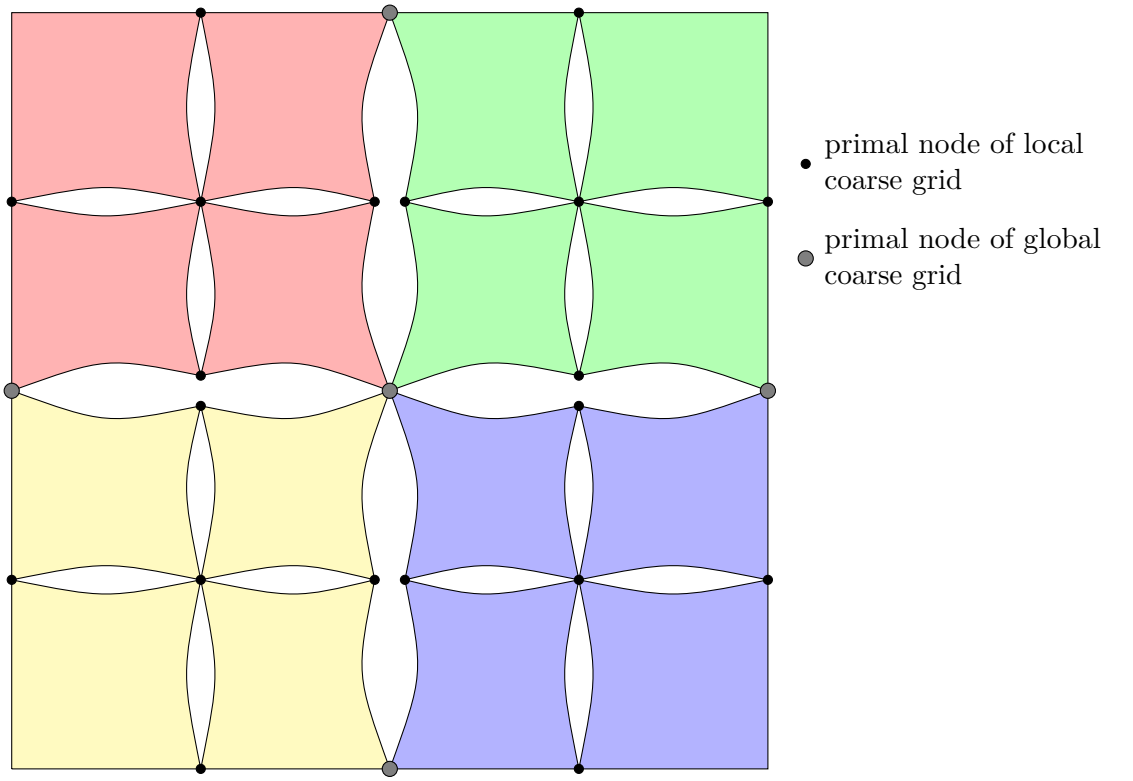


Figure 3.15: Example of multilevel FETI-DP composition of 16 subdomains. Four subdomains colored with the same color share a local coarse space. A global coarse space couples all subdomains. Lagrange constraints to ensure continuity of the solution along dual and primal nodes of the local coarse spaces are not shown in this figure
.

3.5.2 Multilevel FETI-DP

The coarse space problem of the standard FETI-DP method leads to a globally distributed matrix, of which every processor contains only a few rows. In our numerical experiments for showing scalability of the FETI-DP method on solving the PFC equation, cf. Section 3.4.2, each processor contains at most nine rows of the coarse space matrix. This distribution leads to a breakdown in the computational scalability of the FETI-DP method for large number of processors, as the ratio of local computations to communication is far away from being optimal. For relatively small number of processors, one can try to circumvent the problem by computing the coarse space matrix and its factorization on only one processor, and distributing the factorized matrix to all other processors. This is done, e.g. in [75]. For large number of processors, this is also not a scaling procedure.

We introduce a novel method to deal with this problem, which we call multilevel FETI-DP. Our approach creates multiple coarse space matrices, all of which are only weakly coupled. The coarse space matrices are restricted to small number of processors, which leads to a more localized communication and ensures also computational scalability of the

FETI-DP method.

Though we call our method a *multilevel FETI-DP*, its presentation in this work is limited to the case of two coarse grid levels. From the presentation it should be clear that the method directly generalized to an arbitrary number of levels. The formal generalization and its implementation is ongoing research of the author.

In the standard FETI-DP approach, all subdomains are coupled by one global coarse grid. For the multilevel approach, an arbitrary but fixed number of subdomains is used to create a *cluster* of subdomains. Each cluster contains a *local coarse grid*, which ensures continuity of the solution on the primal nodes within one cluster. A *global coarse grid* is then used to connect the clusters among each other. This is illustrated in Figure 3.15 for four clusters each consisting of four subdomains. In what follows, a cluster is defined as a set of subdomain indices $\mathcal{C}_i$ and the number of clusters is defined by n_c. Each subdomain is enclosed in exactly one cluster. In contrast to the standard FETI-DP method, we have now to distinguish between primal nodes local to a cluster and global primal nodes. We denote unknowns related to primal nodes local to the cluster by u_{Π_c}, and unknowns related to global primal nodes by u_{Π_g}. According to the splitting of unknowns, the local matrix of subdomain $i \in \mathcal{C}_j$ is partitioned as follows

$$A^i = \begin{bmatrix} A^i_{BB} & A^i_{B\Pi_c} & A^i_{B\Pi_g} \\ A^i_{\Pi_c B} & A^i_{\Pi_c\Pi_c} & A^i_{\Pi_c\Pi_g} \\ A^i_{\Pi_g B} & A^i_{\Pi_g\Pi_c} & A^i_{\Pi_g\Pi_g} \end{bmatrix} \tag{3.36}$$

We make the assumption that cluster-local primal nodes and global primal nodes are chosen such that the supports of basis functions related to cluster primal nodes and global primal nodes do not overlap. In this case, the corresponding coupling matrices are zero and A^i can be simplified to

$$A^i = \begin{bmatrix} A^i_{BB} & A^i_{B\Pi_c} & A^i_{B\Pi_g} \\ A^i_{\Pi_c B} & A^i_{\Pi_c\Pi_c} & 0 \\ A^i_{\Pi_g B} & 0 & A^i_{\Pi_g\Pi_g} \end{bmatrix} \tag{3.37}$$

with A^i_{BB} defined as by (3.10) and

$$A^i_{B\Pi_c} = \begin{bmatrix} A^i_{I\Pi_c} \\ A^i_{\Delta\Pi_c} \end{bmatrix}, \; A^i_{B\Pi_g} = \begin{bmatrix} A^i_{I\Pi_g} \\ A^i_{\Delta\Pi_g} \end{bmatrix}, \; A^i_{\Pi_c B} = \begin{bmatrix} A^i_{\Pi_c I} & A^i_{\Pi_c\Delta} \end{bmatrix}, \; A^i_{\Pi_g B} = \begin{bmatrix} A^i_{\Pi_g I} & A^i_{\Pi_g\Delta} \end{bmatrix} \tag{3.38}$$

For each cluster i, we define the subassembled matrices for the cluster-local coarse grid by

$$\widetilde{A}^i_{\Pi_c\Pi_c} = \sum_{j\in\mathcal{C}_i} {R^j_{\Pi_c}}^T A^j_{\Pi_c\Pi_c} R^j_{\Pi_c} \tag{3.39}$$

Then, $\widetilde{A}_{\Pi_c\Pi_c}$ is the block matrix of all cluster-local coarse grid matrices

$$\widetilde{A}_{\Pi_c\Pi_c} = diag_{i=1}^{n_c} \widetilde{A}^i_{\Pi_c\Pi_c} \tag{3.40}$$

The global coarse grid matrix is subassembled from the local matrices in the same way with

$$\widetilde{A}_{\Pi_g\Pi_g} = \sum_{i=1}^{p} {R^i_{\Pi_g}}^T A^i_{\Pi_g\Pi_g} R^i_{\Pi_g} \tag{3.41}$$

The subassembled coupling matrices $\widetilde{A}_{B\Pi_c}$ and $\widetilde{A}_{\Pi_c B}$, and the globally subassembled coupling matrices $\widetilde{A}_{B\Pi_g}$ and $\widetilde{A}_{\Pi_g B}$ are defined correspondingly. The complete FETI-DP systems reads

$$\begin{bmatrix} A_{BB} & \widetilde{A}_{B\Pi_c} & \widetilde{A}_{B\Pi_g} \\ \widetilde{A}_{\Pi_c B} & \widetilde{A}_{\Pi_c\Pi_c} & 0 \\ \widetilde{A}_{\Pi_g B} & 0 & \widetilde{A}_{\Pi_g\Pi_g} \end{bmatrix} \begin{bmatrix} u_B \\ \widetilde{u}_{\Pi_c} \\ \widetilde{u}_{\Pi_g} \end{bmatrix} = \begin{bmatrix} f_B \\ \widetilde{f}_{\Pi_c} \\ \widetilde{f}_{\Pi_g} \end{bmatrix} \tag{3.42}$$

To ensure continuity of the solution across interior boundaries, we have to define Lagrange constraints not only on the dual nodes, but also on cluster primal nodes which correspond to global nodes shared by more than one cluster, cf. also Figure 3.15. We denote by λ_Δ the Lagrange multipliers defined for dual nodes, and by λ_{Π_c} the Lagrange multipliers defined for a subset of the cluster primal nodes. The corresponding discrete jump operators are denoted by J_Δ and J_{Π_c}. By introducing them to ensure continuity of the solution of (3.42), the following system is obtained

$$\begin{bmatrix} A_{BB} & \widetilde{A}_{B\Pi_c} & \widetilde{A}_{B\Pi_g} & J_\Delta^T & 0 \\ \widetilde{A}_{\Pi_c B} & \widetilde{A}_{\Pi_c\Pi_c} & 0 & 0 & J_{\Pi_c}^T \\ \widetilde{A}_{\Pi_g B} & 0 & \widetilde{A}_{\Pi_g\Pi_g} & 0 & 0 \\ J_\Delta & 0 & 0 & 0 & 0 \\ 0 & J_{\Pi_c} & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} u_B \\ \widetilde{u}_{\Pi_c} \\ \widetilde{u}_{\Pi_g} \\ \lambda_\Delta \\ \lambda_{\Pi_c} \end{bmatrix} = \begin{bmatrix} f_B \\ \widetilde{f}_{\Pi_c} \\ \widetilde{f}_{\Pi_g} \\ 0 \\ 0 \end{bmatrix} \tag{3.43}$$

By block Gaussian elimination of the unknowns u_B, $\widetilde{u}_{\Pi_c}$ and $\widetilde{u}_{\Pi_g}$, we obtain a reduced linear system defined only for the Lagrange multipliers

$$\begin{bmatrix} J_\Delta & 0 & 0 \\ 0 & J_{\Pi_c} & 0 \end{bmatrix} \mathcal{S}_{ml}^{-1} \begin{bmatrix} J_\Delta^T & 0 \\ 0 & J_{\Pi_c}^T \\ 0 & 0 \end{bmatrix} \begin{bmatrix} \lambda_\Delta \\ \lambda_{\Pi_c} \end{bmatrix} = \begin{bmatrix} J_\Delta & 0 & 0 \\ 0 & J_{\Pi_c} & 0 \end{bmatrix} \mathcal{S}_{ml}^{-1} \begin{bmatrix} f_B \\ \widetilde{f}_{\Pi_c} \\ \widetilde{f}_{\Pi_g} \end{bmatrix} \tag{3.44}$$

with

$$\mathcal{S}_{ml} = \begin{bmatrix} A_{BB} & \widetilde{A}_{B\Pi_c} & \widetilde{A}_{B\Pi_g} \\ \widetilde{A}_{\Pi_c B} & \widetilde{A}_{\Pi_c\Pi_c} & 0 \\ \widetilde{A}_{\Pi_g B} & 0 & \widetilde{A}_{\Pi_g\Pi_g} \end{bmatrix} = \begin{bmatrix} \widetilde{A}_c & \widetilde{A}_{cg} \\ \widetilde{A}_{gc} & \widetilde{A}_{\Pi_g\Pi_g} \end{bmatrix} \tag{3.45}$$

and

$$\widetilde{A}_c = \begin{bmatrix} A_{BB} & \widetilde{A}_{B\Pi_c} \\ \widetilde{A}_{\Pi_c B} & \widetilde{A}_{\Pi_c\Pi_c} \end{bmatrix}, \widetilde{A}_{cg} = \begin{bmatrix} \widetilde{A}_{B\Pi_g} \\ 0 \end{bmatrix}, \widetilde{A}_{gc} = \begin{bmatrix} \widetilde{A}_{\Pi_g B} & 0 \end{bmatrix} \tag{3.46}$$

Based on block factorization, the inverse of $\mathcal{S}_{ml}$ can be explicitly written as

$$\begin{aligned} \mathcal{S}_{ml}^{-1} &= \begin{bmatrix} I & -\widetilde{A}_c^{-1}\widetilde{A}_{cg} \\ 0 & I \end{bmatrix} \begin{bmatrix} -\widetilde{A}_c^{-1} & 0 \\ 0 & S_g \end{bmatrix} \begin{bmatrix} I & 0 \\ -\widetilde{A}_{gc}\widetilde{A}_c^{-1} & I \end{bmatrix} \\ &= \begin{bmatrix} \widetilde{A}_c^{-1} + \widetilde{A}_c^{-1}\widetilde{A}_{cg}S_g^{-1}\widetilde{A}_{gc}\widetilde{A}_c^{-1} & -\widetilde{A}_c^{-1}\widetilde{A}_{cg}S_g^{-1} \\ -S_g^{-1}\widetilde{A}_{gc}\widetilde{A}_c^{-1} & S_g^{-1} \end{bmatrix} \end{aligned} \tag{3.47}$$

with the Schur complement on the global primal nodes

$$S_g = \widetilde{A}_{\Pi_g\Pi_g} - \widetilde{A}_{gc}\widetilde{A}_c^{-1}\widetilde{A}_{cg} \tag{3.48}$$

The inverse of matrix $\widetilde{A}_c$ can be explicitly written in the same way

$$\begin{aligned}\widetilde{A}_c^{-1} = \begin{bmatrix} A_{BB} & \widetilde{A}_{B\Pi_c} \\ \widetilde{A}_{\Pi_c B} & \widetilde{A}_{\Pi_c\Pi_c} \end{bmatrix}^{-1} &= \begin{bmatrix} I & -A_{BB}^{-1}\widetilde{A}_{B\Pi_c} \\ 0 & I \end{bmatrix} \begin{bmatrix} -A_{BB}^{-1} & 0 \\ 0 & S_c \end{bmatrix} \begin{bmatrix} I & 0 \\ -\widetilde{A}_{\Pi_c B}A_{BB}^{-1} & I \end{bmatrix} \\ &= \begin{bmatrix} F_c & -A_{BB}^{-1}\widetilde{A}_{B\Pi_c}S_c^{-1} \\ -S_c^{-1}\widetilde{A}_{\Pi_c B}A_{BB}^{-1} & S_c^{-1} \end{bmatrix}\end{aligned} \tag{3.49}$$

with F_c the cluster-local FETI-DP operator defined by

$$F_c = A_{BB}^{-1} + A_{BB}^{-1}\widetilde{A}_{B\Pi_c}S_c^{-1}\widetilde{A}_{\Pi_c B}A_{BB}^{-1} \tag{3.50}$$

and the Schur complement on the cluster primal nodes, i.e. the cluster-local coarse grid matrix, defined by

$$S_c = \widetilde{A}_{\Pi_c\Pi_c} - \widetilde{A}_{\Pi_c B}A_{BB}^{-1}\widetilde{A}_{B\Pi_c} \tag{3.51}$$

Then (3.48) can be simplified to

$$S_g = \widetilde{A}_{\Pi_g\Pi_g} - \widetilde{A}_{\Pi_g B}F_c^{-1}\widetilde{A}_{B\Pi_g} \tag{3.52}$$

This shows a direct correlation between the cluster Schur complement S_c and the global Schur complement S_g. The first one includes the inverse of local matrices while in the latter the corresponding inverse is replaced by the solution of the FETI-DP operator local to each cluster.

We now combine (3.44) and (3.47) to get the global FETI-DP system:

$$F_g \begin{bmatrix} \lambda_\Delta \\ \lambda_{\Pi_c} \end{bmatrix} = d_g \tag{3.53}$$

with the global FETI-DP operator

$$F_g = \begin{bmatrix} J_\Delta & 0 \\ 0 & J_{\Pi_c} \end{bmatrix} \left(\widetilde{A}_c^{-1} + \widetilde{A}_c^{-1}\widetilde{A}_{cg}S_g^{-1}\widetilde{A}_{gc}\widetilde{A}_c^{-1} \right) \begin{bmatrix} J_\Delta^T & 0 \\ 0 & J_{\Pi_c}^T \end{bmatrix} \tag{3.54}$$

and the reduced right-hand-side vector of the global FETI-DP system

$$d_{ml} = \begin{bmatrix} J_\Delta & 0 \\ 0 & J_{\Pi_c} \end{bmatrix} \left(\left(\widetilde{A}_c^{-1} + \widetilde{A}_c^{-1}\widetilde{A}_{cg}S_g^{-1}\widetilde{A}_{gc}\widetilde{A}_c^{-1} \right) \begin{bmatrix} f_B \\ \widetilde{f}_{\Pi_c} \end{bmatrix} - \widetilde{A}_c^{-1}\widetilde{A}_{cg}S_g^{-1}\widetilde{f}_{\Pi_g} \right) \tag{3.55}$$

Once the solution for the Lagrange multipliers λ_Δ and λ_{Π_c} is computed, the solution of the other variables of system (3.43) can be obtained with

$$\begin{bmatrix} u_B \\ \widetilde{u}_{\Pi_c} \\ \widetilde{u}_{\Pi_g} \end{bmatrix} = \mathcal{S}_{ml}^{-1} \left(\begin{bmatrix} f_B \\ \widetilde{f}_{\Pi_c} \\ \widetilde{f}_{\Pi_g} \end{bmatrix} - \begin{bmatrix} J_\Delta^T & 0 \\ 0 & J_{\Pi_c}^T \\ 0 & 0 \end{bmatrix} \begin{bmatrix} \lambda_\Delta \\ \lambda_{\Pi_c} \end{bmatrix} \right) \tag{3.56}$$

Implementation issues

We now discuss the efficient implementation of the multilevel FETI-DP method. Creating the required parallel DOF mappings, vectors and matrices is identical as it is done in the standard FETI-DP method and is thus covered by the discussion in Section 3.4.1. Here, we discuss the implementation of the FETI-DP operators F_c and F_g, and the explicit computation of the Schur primal matrices S_c and S_g.

The Schur primal matrix S_c is local to a cluster and can be computed exactly in the same way as it is done for the global Schur primal matrix $\widetilde{S}_{\Pi\Pi}$ in the standard FETI-DP method. This is described by Algorithm 7. The solution of the cluster FETI-DP operator F_c is required only inside of the computation of $\widetilde{A}_c^{-1}$, cf. (3.49), which is required in two places: for explicit computation the global Schur matrix S_g and when the global multilevel FETI-DP operator F_g is applied to some vector. Computing the action of $\widetilde{A}_c^{-1}$ is implemented by Algorithm 9. One application of this operator requires three solves with local interior matrices A_{BB} and two solves with the cluster Schur primal matrices S_c.

Algorithm 9: Application of $\widetilde{A}_c^{-1}$, cf. (3.49)

input : Matrices defined within $\widetilde{A}_c$, vectors b_0 and b_1
output: $\begin{bmatrix} x_0 \\ x_1 \end{bmatrix} = \widetilde{A}_c^{-1} \begin{bmatrix} b_0 \\ b_1 \end{bmatrix}$
solve for $A_{BB} t_0 = b_0$
compute $t_1 = \widetilde{A}_{\Pi_c B} t_0$
solve for $S_c t_2 = t_1$
compute $t_1 = \widetilde{A}_{B\Pi_c} t_2$
solve for $A_{BB} x_0 = t_1$
compute $x_0 = t_0 + x_0$ // $x_0 = F_c b_0$
solve for $S_c t_0 = b_1$
compute $x_1 = t_0 - t_2$ // $x_1 = -S_c^{-1} \widetilde{A}_{\Pi_c B} A_{BB}^{-1} b_0 + S_c^{-1} b_1$
compute $t_1 = \widetilde{A}_{B\Pi_c} t_0$
solve $A_{BB} t_2 = t_1$
compute $x_0 = x_0 - t_2$ // $x_0 = F_c b_0 - A_{BB}^{-1} \widetilde{A}_{B\Pi_c} S_c^{-1} b_1$

Computing the global Schur primal matrix S_g, cf. (3.52), can be done equivalently to the explicit computation of the global Schur primal matrix in the standard FETI-DP method. The only major change is to replace the solve with the local interior matrices A_{BB} with the solve of the cluster-local FETI-DP operator F_c. Finally, we have to specify an algorithm for the application of the global multilevel FETI-DP operator F_g, cf. (3.54). For an efficient algorithm, we first reformulated the inner part of the operator as following

$$\widetilde{A}_c^{-1} + \widetilde{A}_c^{-1} \widetilde{A}_{cg} S_g^{-1} \widetilde{A}_{gc} \widetilde{A}_c^{-1} \equiv \left(I + \widetilde{A}_c^{-1} \begin{bmatrix} \widetilde{A}_{B\Pi_g} S_g^{-1} \widetilde{A}_{\Pi_g B} & 0 \\ 0 & 0 \end{bmatrix} \right) \widetilde{A}_c^{-1} \tag{3.57}$$

This directly leads to Algorithm 10, which specifies an efficient procedure to implement the action of the global FETI-DP operator.

Algorithm 10: Application of the multilevel FETI-DP operator

input : Matrices defined within the multilevel FETI-DP operator, global Schur complement operator S_g, some vectors λ_Δ and λ_{Π_c}

output: $\begin{bmatrix} v_0 \\ v_1 \end{bmatrix} = F_g \begin{bmatrix} \lambda_\Delta \\ \lambda_{\Pi_c} \end{bmatrix}$

compute $t_0 = J_\Delta^T \lambda_\Delta$ and $t_1 = J_{\Pi_c}^T \lambda_{\Pi_c}$

solve for $\widetilde{A}_c \begin{bmatrix} t_2 \\ t_3 \end{bmatrix} = \begin{bmatrix} t_0 \\ t_1 \end{bmatrix}$

compute $t_4 = \widetilde{A}_{B\Pi_g} S_g^{-1} \widetilde{A}_{\Pi_g B} t_2$

solve for $\widetilde{A}_c \begin{bmatrix} t_5 \\ t_6 \end{bmatrix} = \begin{bmatrix} t_4 \\ 0 \end{bmatrix}$

compute $t_2 = t_2 + t_5$ and $t_3 = t_3 + t_6$

compute $v_0 = J_\Delta t_2$ and $v_1 = J_{\Pi_c} t_3$

For estimating the computational costs of applying the FETI-DP operator in the standard and in the multilevel FETI-DP methods, we focus on the major blocks: solving with the local interior matrices and solving with the distributed primal Schur matrices. Time and memory for the other operations, e.g. matrix vector multiplications or computing vector products, can be neglected. For one application of the standard FETI-DP operator (3.20), it is required to solve twice a system with the local interior matrices and to solve once with the global Schur primal matrix. For the application of the multilevel FETI-DP operator, we need to compute twice the solution with $\widetilde{A}_c$ and one solution with the global Schur primal matrix. Each solution with $\widetilde{A}_c$ requires three solves with the interior matrices and two solves with the local Schur primal matrices. Note that in one of the both solves with $\widetilde{A}_c$, the second vector of the right hand side is always zero. Careful analysis of Algorithm 9 directly shows that in this case the last solve with the interior matrices and local Schur primal matrices can be omitted, as the solution is always the corresponding zero vector. Altogether, one application of the multilevel Schur primal matrix requires five solves with the interior matrices, three solves with the local Schur primal matrices and one solve with the global Schur primal matrix. As the interior matrices are the same in both FETI-DP methods, also the computing time for solving with these matrices is the same. Comparing the time complexity of solving the different Schur primal matrices in both FETI-DP methods is not straightforward. Nevertheless, our numerical experiments, which are presented in the next section, show that splitting the globally distributed Schur primal matrix arising in the standard FETI-DP method into a level structured set of more locally defined Schur primal matrices results in a faster and more scalable solution method.

Numerical results

To compare the proposed multilevel FETI-DP method with the standard FETI-DP method, we employ the same 2D PFC simulation as described in Section 3.4.2. As we have not defined a preconditioner for the multilevel FETI-DP method, we do not use any preconditioner also for the standard FETI-DP method in this section. Therefore, the following results cannot

cores/s.p.c.	1 (FETI-DP)	4	16	64	256	1024
16	0.0072	**0.014**	-	-	-	-
64	0.015	0.020	**0.015**	-	-	-
256	0.060	0.043	**0.029**	0.03227	-	-
1024	1.8	0.26	**0.068**	1.0	22.4	
4096	-	8.8	3.9	**1.1**	24.1	401.8

Table 3.3: Comparing time required for factorization of the Schur primal matrix for standard FETI-DP solver with time required for factorization of cluster and global Schur primal matrices in multilevel FETI-DP solver with varying number of subdomains per cluster (s.p.c.). Red colored time numbers denote the fastest multilevel FETI-DP configuration for a fixed number of cores.

cores/s.p.c.	1 (FETI-DP)	4	16	64	256	1024
16	23.5	**23.9**	-	-	-	-
64	25.5	**24.2**	24.6	-	-	-
256	34.5	25.5	**25.4**	28.1	-	-
1024	127.7	27.9	**26.0**	29.7	52.2	-
4096	-	84.1	**35.0**	35.7	67.0	311.6

Table 3.4: Comparing average solution time for one timestep of a 2D PFC simulation with using either standard FETI-DP or the multilevel FETI-DP solver with varing number of subdomains per cluster (s.p.c.). The time does not contain the setup time of the solver. Red colored time numbers denote the fastest multilevel FETI-DP configuration for a fixed number of cores.

be directly compared with those in Section 3.4.2. Nevertheless, as the proposed FETI-DP preconditioners are not crucial for the parallel computational scaling, the following results will immediately carry over to the general case with using either the Dirichlet or the lumped preconditioner.

We consider weak scaling from 16 to 4,096 cores. Furthermore, we restrict this comparison to the direct solution of the Schur primal matrices, as in general this leads to a more efficient and robust solution method. We make use of MUMPS to compute the LU factorizations of distributed Schur primal matrices in both FETI-DP methods.

Table 3.3 compares time required for factorizing the Schur primal matrix in the standard FETI-DP method with the time required to compute the factorization of the cluster and the global Schur primal matrices in the multilevel FETI-DP method with varying cluster size. When using more than 64 cores, factorization of the Schur primal matrices in the multilevel FETI-DP can be computed faster than in the standard FETI-DP method. MUMPS fails to compute the LU factorization in the standard FETI-DP method when using 4,096 cores, whereas it requires only 1.1 seconds in the multilevel FETI-DP method, when each cluster contains 64 subdomains.

The number of iterations of the FETI-DP solver is always the same, independently whether the standard or the multilevel version is used. Furthermore, the iteration count of

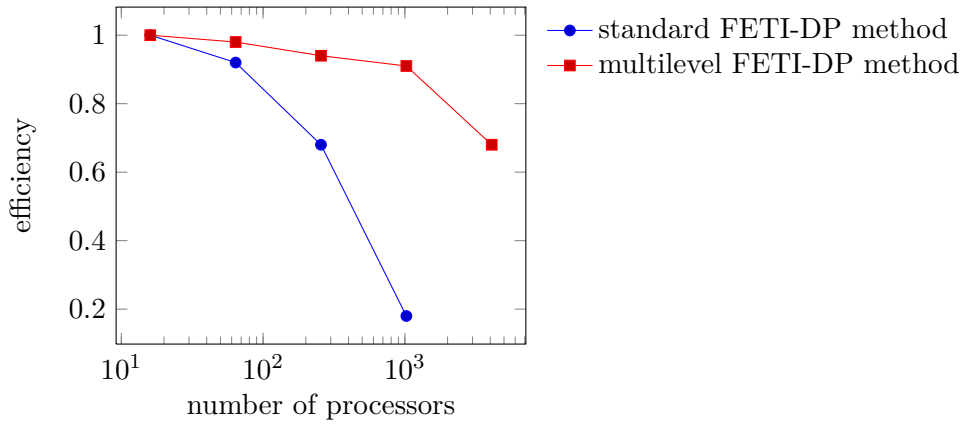


Figure 3.16: Comparing efficiency of standard FETI-DP and multilevel FETI-DP for a 2D PFC computation.

the presented multilevel FETI-DP method is independent of the cluster size. Therefore, no comparison of iterations numbers is done in the following. Table 3.4 compares the solution time of the standard and the multilevel FETI-DP method with varying cluster size. It shows that the multilevel FETI-DP method is superior to the standard FETI-DP method. It has a nearly constant solution time up to 1,024 cores. When using 4,096 cores the solution time slightly increases, where the standard FETI-DP cannot be applied as MUMPS fails to compute the LU factorization of the Schur primal matrix.

3.6 A Navier-Stokes solver

The opposite of general black box solvers are highly specialized solution methods that work only for one specific PDE. Sometimes they are designed even only for a limited parameter range of a specific PDE. Because of their limited use, they can be designed in a very efficient way taking into account not only knowledge of algebraic or geometric discretization structures, but also of the physics behind the PDE. For many widely used PDEs highly specialized methods to solve the resulting systems of linear equations are known, such as for the Maxwell's equation or the Stokes and Navier-Stokes equations. In this section, we present the implementation of a linear solver for systems of equation arising from discretization of the instationary, linearized Navier-Stokes equation in 2D and 3D. This solver was proposed in [103], without considering its parallelization. Its parallel implementation exemplifies that our concepts for implementing parallel solver enables a fast and simple implementation of problem specific solver methods.

First, we give a short introduction to the Navier-Stokes solver proposed in [103], where parallelization issues are not considered. The Navier-Stokes equation for modeling incompressible flow reads as follows

$$\begin{aligned} \frac{\partial u}{\partial t} + u \cdot \nabla u - \nu \nabla^2 u + \nabla p &= f \\ \nabla \cdot u &= 0 \end{aligned} \tag{3.58}$$

together with appropriate boundary and initial conditions. Here, u is the fluid velocity, p is the pressure, ν is the viscosity parameter and f is some force applied to the system. For time discretization, we restrict ourselves here to the backward Euler scheme

$$\begin{aligned} \frac{u^{n+1}-u^n}{\tau} + u^* \cdot \nabla u^{n+1} - \nu \nabla^2 u^{n+1} + \nabla p^{n+1} &= f \\ \nabla \cdot u^{n+1} &= 0 \end{aligned} \tag{3.59}$$

and for linearization we choose $u^* = u^n$. Thus, at each timestep a generalized Oseen problem arises: given a divergence-free vector field w, we search for u and p that are solutions of

$$\begin{aligned} \frac{1}{\tau} u + w \cdot \nabla u - \nu \nabla^2 u + \nabla p &= g \\ \nabla \cdot u &= 0 \end{aligned} \tag{3.60}$$

Discretization of this PDE with the finite element method leads to a system of linear equations of the form

$$\begin{bmatrix} F & B^T \\ B & 0 \end{bmatrix} \begin{bmatrix} u \\ p \end{bmatrix} = \begin{bmatrix} g \\ 0 \end{bmatrix} \tag{3.61}$$

where F is the discrete convection diffusion operator and B is the discrete pressure divergence matrix. This system has the saddle point property, and many methods are known for building efficient preconditioners and solvers for this type of systems [20]. The basic approach for defining a preconditioner of a block matrix, is the block factorization of the according system

$$\begin{bmatrix} F & B^T \\ B & 0 \end{bmatrix} = \begin{bmatrix} I & 0 \\ BF^{-1} & I \end{bmatrix} \begin{bmatrix} F & B^T \\ 0 & S \end{bmatrix} \tag{3.62}$$

where $S = -BF^{-1}B^T$ is the Schur complement of F. This directly motivates the use of block triangular preconditioners of the from

$$\mathcal{P} = \begin{bmatrix} F & B^T \\ 0 & S \end{bmatrix} \tag{3.63}$$

Such preconditioners were first considered in [25], see also [20] and references therein for more information on this topic. The inverse of the block triangular preconditioner $\mathcal{P}$ reads in factorized form as follows

$$\mathcal{P}^{-1} = \begin{bmatrix} F^{-1} & 0 \\ 0 & I \end{bmatrix} \begin{bmatrix} I & B^T \\ 0 & -I \end{bmatrix} \begin{bmatrix} I & 0 \\ 0 & -S^{-1} \end{bmatrix} \tag{3.64}$$

It can be shown, that if F^{-1} and S^{-1} are solved exactly, then GMRES will solve system (3.61) precondition by (3.64) in at most two iterations [20]. For an efficient solver method, we have to replace both F^{-1} and S^{-1} by some approximations $\hat{F}^{-1}$ and $\hat{S}^{-1}$. In [103], the following approximations are proposed for the case the system of equations results from discretization of the Oseen problem (3.60): $\hat{F}^{-1}$ is defined by one multigrid V-cycle and $\hat{S}^{-1}$ is defined by the following expression, which is sometimes called the F_p approximation

$$\hat{S}^{-1} = \hat{Q}^{-1} F_p \hat{H}^{-1} \tag{3.65}$$

where $\hat{H}^{-1}$ is the approximate inverse of the pressure Laplace matrix, $\hat{Q}^{-1}$ is the approximate inverse of the pressure mass matrix, and F_p is the convection-diffusion operator of the Oseen problem discretized in pressure space. We choose $\hat{Q}^{-1}$ to approximate with two CG iterations preconditioned with diagonal scaling, and $\hat{H}^{-1}$ is approximated with one multigrid V-cycle. In [103], it was shown that this choice leads to a solver whose iteration count is nearly independent of mesh size and timestep. Furthermore, the solver shows only a mild influence on the viscosity parameter.

3.6.1 Implementation issues

Our implementation of the Navier-Stokes solver is based on the global matrix solver implemented by the class `PetscSolverGlobalMatrix`, cf. Section 3.7, and PETSc's concept of the `PCFIELDSPLIT` preconditioner [26], which implements a general interface for writing problem specific preconditioners based on blocks matrices. Creating the solver splits into the following parts:

1. Create matrix blocks for the velocity and pressure unknowns: To extract blocks from the globally distributed matrix, PETSc provides the concept of `IS` (index set), which is similar in its functionality to our parallel DOF mappings. Thus, the Navier-Stokes solver must create two parallel DOF mappings for the velocity and the pressure unknowns, convert them to PETSc's `IS` data structure and use them to initialize the `PCFIELDSPLIT` preconditioner. This procedure is already implemented in the function `createFieldSplit` in class `PetscSolverGlobalMatrix`, which must be called twice from the Navier-Stokes solver.

2. Create the auxiliary matrices Q, H and F_p: To show how simple it is to create such kind of matrices for problem specific preconditioners, we present the source code from our Navier Stokes solver implementation to create the mass matrix Q:

```
1 DOFMatrix massMatrix(pressureFeSpace, pressureFeSpace);
2 Operator massOp(pressureFeSpace, pressureFeSpace);
3 Simple_ZOT massTerm;
4 massOp.addTerm(&massTerm);
5 massMatrix.assembleOperator(massOp);
6 massMatrixSolver = createSubSolver(pressureComponent, "mass_");
7 massMatrixSolver->fillPetscMatrix(&massMatrix);
```

   In the first five lines, on each processor a local matrix is created and assembled with the finite element identify operator. The sixth line creates a new sub solver from the `PetscSolverGlobalMatrix` solver restricted to the pressure space. The last line copies the local matrix to a globally distributed PETSc matrix. Altogether only seven lines of code are required to create a distributed matrix, fill it with a discrete FEM operator and initialize a parallel solver method for this matrix.

3. Initialize the inner and outer solvers: Once all the inner solvers and matrices are created, one has to inform PETSc about the specific solver type and solver parameter, e.g. stopping criteria.

In summary, our concept for parallel solver and its efficient implementation allow to implement the described Navier-Stokes solver in around 120 lines of code.

	$h = 1/16$	$h = 1/32$	$h = 1/64$	$h = 1/128$
$\nu = 1/100$				
$\tau = 0.1$	12	12	11	10
$\tau = 1$	16	15	14	13
$\tau = 10$	19	16	15	15
$\nu = 1/200$				
$\tau = 0.1$	13	13	12	11
$\tau = 1$	19	17	15	14
$\tau = 10$	23	21	18	17
$\nu = 1/500$				
$\tau = 0.1$	15	15	14	12
$\tau = 1$	72	62	17	15
$\tau = 10$	-	322	27	23

Table 3.5: Number of iterations for the Navier-Stokes solver to solve a driven cavity flow in 2D with varying discretization parameters. The solver did not converge within 1,000 iterations for $Re = 500$, $\tau = 10$ and $h = 1/16$.

3.6.2 Numerical results

First, we show that the proposed Navier-Stokes solver is optimal in the sense that it is mostly independent on the discretization parameters, i.e. the mesh size, the timestep and viscosity. For this, we consider a standard driven cavity flow problem in 2D [127, 47]. The boundary condition is given by

$$u(x, y) = \begin{cases} (1, 0) & y = 1 \\ (0, 0) & \text{otherwise} \end{cases} \qquad \text{on } \partial\Omega \tag{3.66}$$

Table 3.5 shows the iteration count of the Navier-Stokes solver for varying mesh sizes, timesteps and viscosity parameters. All of these computations are performed using eight processors. The iteration count is nearly constant with increasing mesh refinement and even tends to decrease as the mesh is much finer than the solution singularities that must be resolved on it. For a fixed mesh size, the iteration count tends to an asymptotic maximum value. This is also observed in [103] for the Navier-Stokes driven cavity flow. The increasing iteration count for decreasing viscosity is in agreement with observations for the steady-state Navier-Stokes equation in [39, 65].

To verify parallel scalability of the Navier-Stokes solver, the same Navier-Stokes driven cavity problem in 2D with $\nu = 1/1000$ and $\tau = 0.1$ is computed for 10 timesteps. The number of processors is varied between 8 and 2,048. Figures 3.17 and 3.18 show scaling for initializing all parallel data structures and the Navier-Stokes solver, and solving the systems of linear equations, respectively. In Figure 3.18, timing results are scaled by the number of iterations, as the iteration count decreases for very fine meshes. Weak and strong scaling of initializing the solver are nearly perfect up to 2,048 processors, as long as the

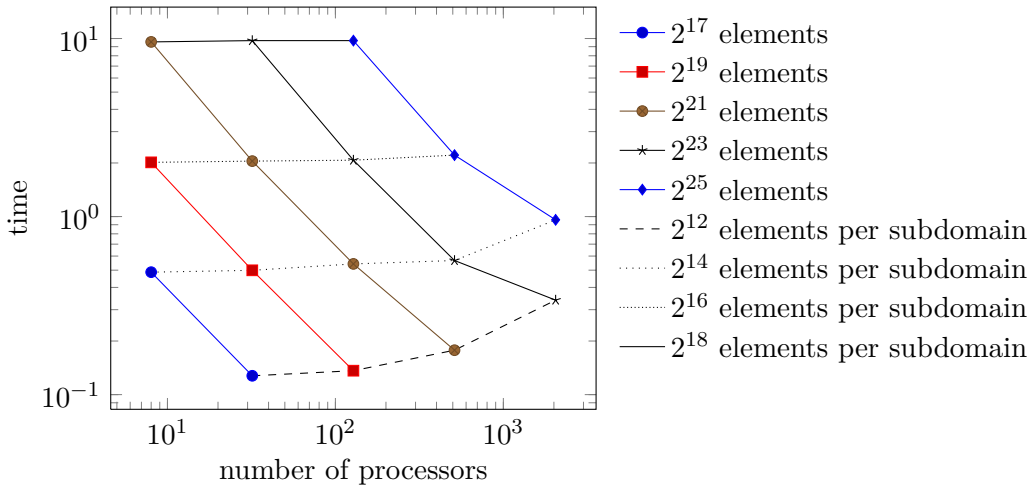


Figure 3.17: Weak and strong scaling of create distributed matrices and initializing the parallel Navier-Stokes preconditioner using 8 to 2,048 processors.

local problem size becomes not too small. The breakdown of scaling for solving the systems of linear equations is mostly related to parallel scaling of the used algebraic multigrid method. In our benchmarks, we use BoomerAMG included in hypre (High Performance Preconditioners) [41, 42]. Hereby we see that its setup phase does not scale very well for the used problem size and number or processors. If the number of iterations becomes too small, the non-scaling setup phase dominates and explains the deterioration of parallel scaling efficiency in our benchmarks.

3.6.3 Diffuse domain approach

The proposed Navier-Stokes solver was extended in [66] 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 optimality results for this solver, which are provided in [103], are not directly applicable anymore in this situation. Nevertheless, the solver still provides an efficient method for the incompressible Navier-Stokes equation in complicated geometries.

We consider a flow channel with 10 spherical particles. They are implicitly described by a phase-field variable, which in the current situation is fixed in time. The initial coarse mesh is of size $8 \times 2 \times 2$ and consists of 12,288 tetrahedrons. All tetrahedrons are further bisectioned 3 times to create a sufficiently fine computational mesh, and an adaptive refinement procedure is used to resolve the phase transition. The final mesh consist of $2,746,954$ elements. We use Taylor-Hood elements for an inf-sup stable discretization. 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. We run the computation with a timestep $\tau = 10^{-1}$ and viscosity $\nu = 1/100$. Figure 3.19 shows the result at $t = 1.0$. The system to be solved in each timestep consists of $1.15 \cdot 10^7$ unknowns. We solve this configuration with 32 to 512 processors. The initial mesh is created in a sequential

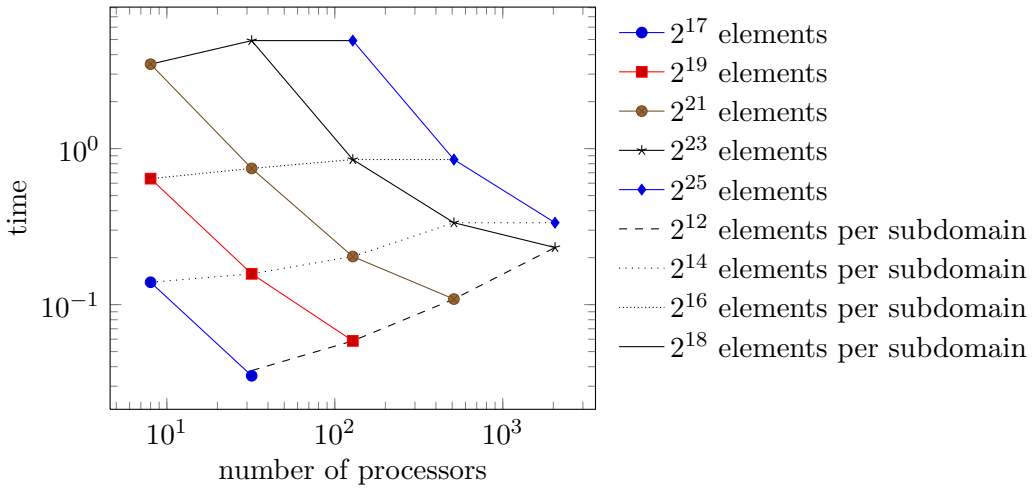


Figure 3.18: Weak and strong scaling of solving driven cavity flow in 2D using 8 to 2,048 processors.

cores	**avrg. elements**	**avrg. unknowns**	**load unbalancing**	**efficiency**
32	85,842	359,144	5.3%	100%
64	42,921	179,572	13.3%	98.4%
128	21,460	89,786	14.2%	103.3%
256	10,730	44,893	14.1%	105.0%
512	5,365	22,446	40.9%	82.6%

Table 3.6: Data for benchmarking the Navier-Stokes solver for a diffuse domain configuration in 3D. The load unbalancing is defined w.r.t. the unknowns of the linear system. Efficiency is computed w.r.t the calculation with 32 processors.

preprocessing step, as it is the same in all runs. Also the initial partitioning is sequentially computed in a preprocessing step using METIS [99]. In our experiments, this leads to a partitioning with a much better load balancing when compared with ParMETIS results in a distributed environment. The runtimes of the individual sub-algorithms are shown in Figure 3.20. The overall efficiency w.r.t. using 32 processors is around 100% for all runs up to 256 processors and goes down to 82% when using 512 processors, where the sub-problems become quite small (around 5,000 elements per subdomain) and load balancing becomes worse (around 40% load unbalancing) in comparison to the other runs, see data in Table 3.6.

3.7 Software concepts

In this section, we provide implementation details for the concepts presented in this chapter. The class structure of the implementation in the finite element toolbox AMDiS is explained. The reader, who is not familiar with AMDiS can find more information about its data

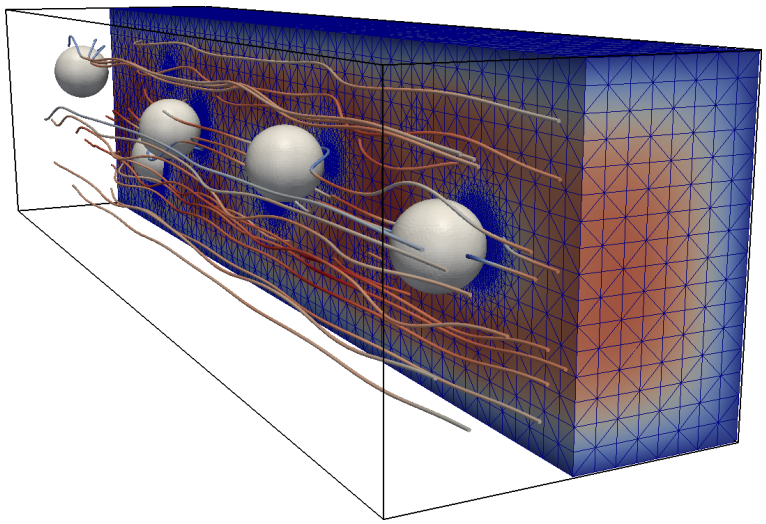


Figure 3.19: Test configuration at $t = 1.0$. The spheres are holes within the domain, stream lines and mesh is colored with the magnitude of the velocity field.

structures and algorithm in [123, 124]. Figure 3.21 (only the most important classes, functions and data members are shown here) shows the class structure diagram of all classes which are related to the `MeshDistributor` class, which is the central point of our implementation of parallel distributed meshes. The most important class functions, which can be used by parallel solver methods, are the following ones:

- `registerDofMap`: A parallel solver can create an arbitrary parallel DOF mapping, see Section 3.2.3. This must be registered to the mesh distributor object such that it can be updated after mesh change.
- `synchVector`: DOFs on interior boundaries are duplicated to all participating subdomains. But only one subdomain is the owner of a particular DOF can is thus responsible for its value. This function synchronize distributed DOF vectors such that the value of DOFs on interior boundaries is the same on all subdomains. For this, a `DofComm` object is used to send the values from owner of these DOFs to all other subdomains.
- `setBoundDofRequirement`: A parallel solver must call this function if geometrical information of DOFs along interior boundaries are required.
- `getBoundDofInfo`: When a solver has requested geometrical information of DOFs along interior boundaries, this information can be queried with this function.
- `checkMeshChange`: Before a solver starts its assembling procedure, it must always call this functions which checks whether the mesh has been changed due to some local refinement procedure. If this is the case, first the mesh is adapted such that no hanging nodes on interior boundaries occur, see Section 3.2.1. If repartitioning is allowed,

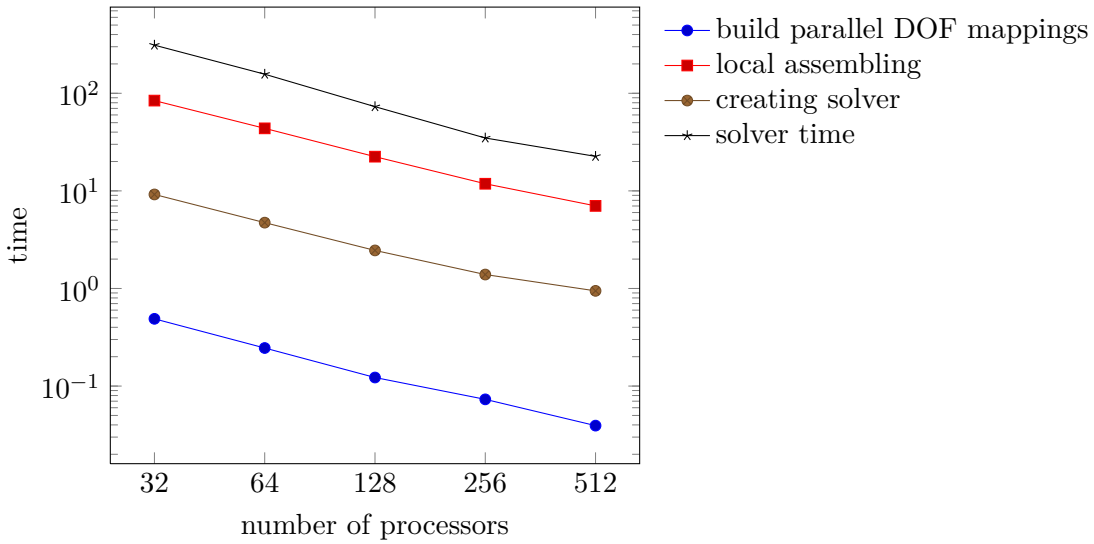


Figure 3.20: Scaling of the Navier-Stokes solver for a diffuse domain configuration in 3D using 32 to 512 processors.

the mesh distributor executes the function `repartitionMesh`. It checks, whether mesh repartitioning is required due to load unbalancing and possibly computes a new mesh partitioning. Finally, `updateDofMapping` is responsible to update all registered parallel DOF mappings to the new mesh.

The element object database, see class `ElObjDB`, is created by the mesh distributor during its initialization and it updated after each mesh repartitioning. It makes use of information about ownership of coarse mesh elements, which is computed by a mesh partitioner object. The element object database is mainly used to create the interior boundary handler, see class `InteriorBoundary`. Once this is established, it can be asked to return either all interior boundaries of the processor's subdomain, only boundaries which are owned by the processor, only boundaries which are owned by another processor, or to return the periodic boundaries of the subdomain. The usage of DOF communicators, see class `DofComm`, is very similar. First, they must be initialized with an interior boundary handler object. Then they can be asked for all DOFs on processor owned boundaries, other interior boundaries or periodic boundaries.

Parallel DOF mappings, cf. Section 3.2.3, are implemented by class `ParallelDofMapping` (also abbreviated with `PDM` in Figures 3.21 and 3.22). When a parallel DOF mapping object is created by some solver class, it must already be fixed to one of the two available *modes*:

- *finite element space mode*: The mapping of local to global DOFs is the same for all components, i.e. unknowns of the PDE, of a particular finite element space.
- *component mode*: The mapping may vary for all components, even if the they are defined on the same finite element space.

For "simple" solvers, such as the global matrix solver, the first mode is required as all components of one finite element space should be handled in the same way. For more complicated solver methods it might be required to differ the DOF mapping for different components. One example is the FETI-DP solver, where the coarse space is defined only for a subset of all components. In general, the component mode can always be used but it leads to higher memory usage as the index mapping is stored for each component independently. In both cases, the parallel DOF mapping must be initialized with a set of the finite element spaces of all components. Afterwards, the solver can specify individual DOFs to be part of the mapping. Calling the function `update` finalizes the DOF insertion procedure and creates the appropriate DOF mapping.

Figure 3.22 (also here, only the most important classes, functions and data members are shown here) shows the class structure diagram related to parallel solvers. This diagram is linked to what is described before for the class diagram in Figure 3.21 via `ParallelDofMapping`, which is contained in both diagrams. All these classes are linked to the sequential part of our finite element code AMDiS via class `ProblemStat`, see also [123, 124], which specifies a stationary PDE. In parallel computations, the class `PetscProblemStat` is derived from `ProblemStat` and overrides the functions for initialization and solving the linear system of equations. This concept allows to create PDE solvers independently of whether AMDiS is compiled to run sequentially or in parallel. In both cases, the user's code is the same and most parallelization details are hidden from the user.

The basis class of all parallel solvers is `ParallelCoarseSpaceSolver`. It contains the functionality to assemble distributed PETSc matrices and to compute the required non zero structures, which are necessary of efficient memory allocation of sparse matrices. Handling multiple coarse spaces, e.g. different coarse spaces for different components, is directly supported by all functions and data structures of this class. The class `PetscSolver` derives from `ParallelCoarseSpaceSolver` and is an abstract class that contains some virtual functions to establish a general interface for PETSc bases solver methods. These must be specified by a derived class. In our implementation, there exists, e.g., the global matrix solver, a FETI-DP solver (see Section 3.4) and a solver based on "simple" iterative substructuring for 2D domains (`SchurSolver`, see also [91, Chapter 3.5]). The Navier-Stokes solver (see Section 3.6) shows that specialized solver methods can also be derived from already existing solver classes.

One of the key concepts for a flexible FETI-DP implementation is the definition of the variable `subDomain`, which specifies the solver type for the subdomains of the FETI-DP approach. It is also responsible for assembling both, the local and the coarse space matrices. As this variable may be an arbitrary solver of type `PetscSolver`, we allow for most possible flexibility in defining subdomains within a FETI-DP solver. The default setting is to use a `GlobalMatrixSolver` and to restrict each FETI-DP subdomain to one rank's subdomain. But in general, a FETI-DP subdomain can be spanned by arbitrary many local subdomains. Then, the "local" FETI-DP matrices are again distributed PETSc matrices, each limited to a subset of all ranks. As the FETI-DP solver just calls the function `fillPetscMatrix()`, `fillPetscRhs()` and `solve()` of the `subDomain` variable, it has no knowledge about the underlying matrix and vectors structures, and the solver method which is used for solving the local systems. This definition of FETI-DP subdomains and the corresponding solver method also allows to directly replace the standard exact local solvers by inexact ones.

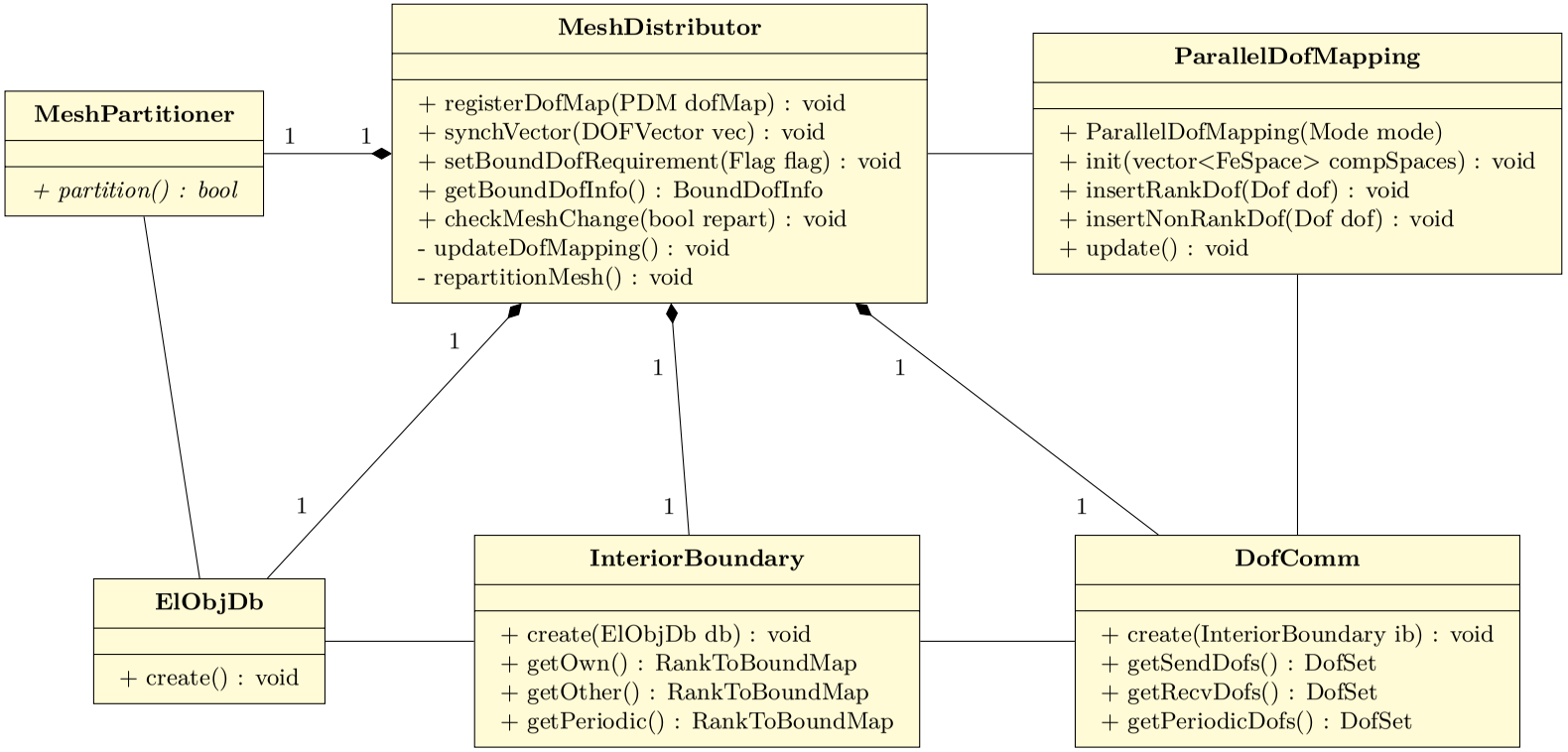


Figure 3.21: Class structure for our concept to provide mesh dependent data of distributed meshes to parallel solvers. This class diagram is connected via `ParallelDofMapping` (`PDM`) with the class diagram in Figure 3.22.

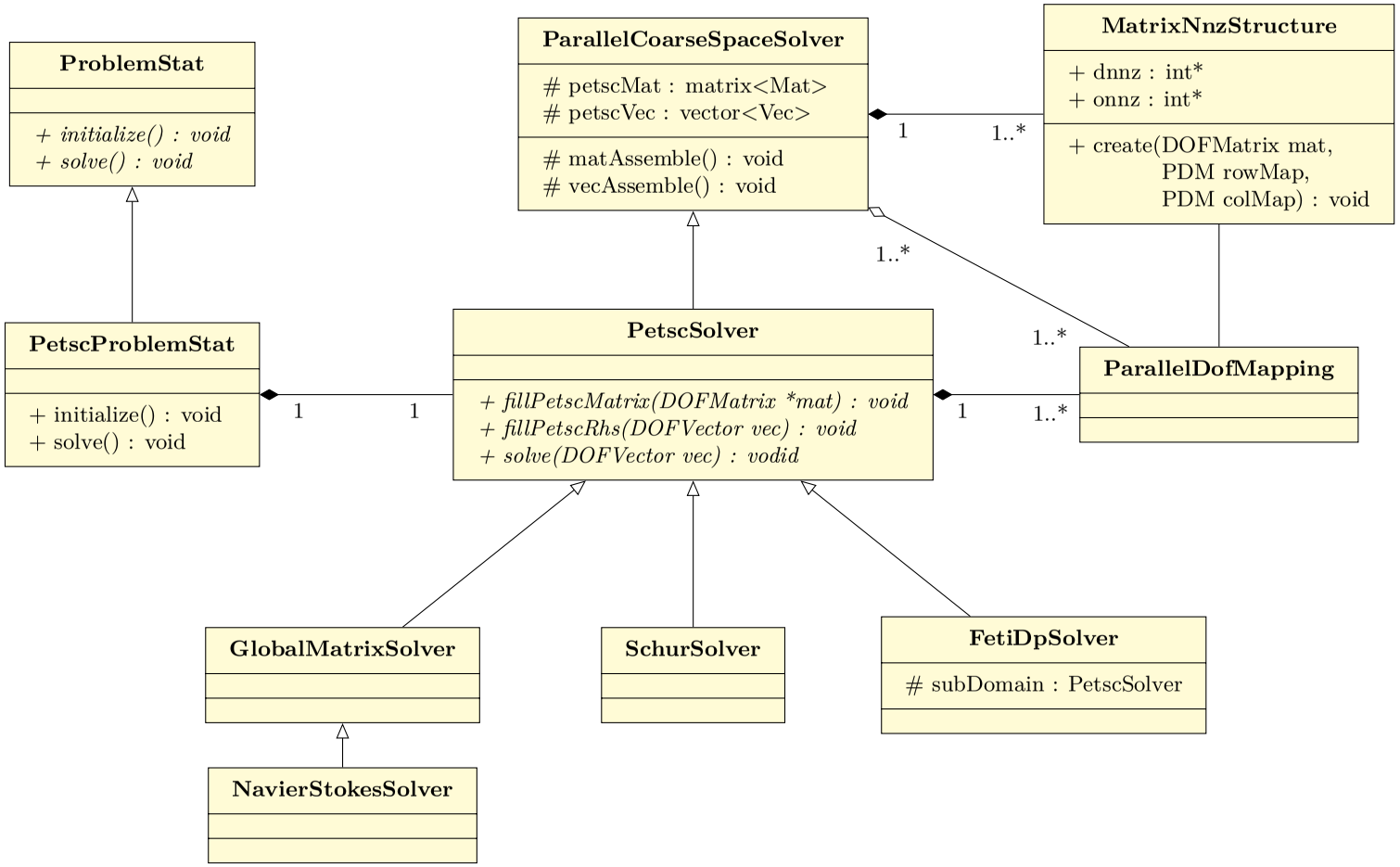


Figure 3.22: Class structure for parallel solver methods. Only some important class members and functions are mentioned. This class diagram is connected via `ParallelDofMapping` (`PDM`) with the class diagram in Figure 3.21.

4 Multi-mesh method for Lagrange finite elements

Modeling multiphysics problems very often results in systems of PDEs. When these are solved with the finite element method, the mesh has to be adapted to the solution's behavior of all components. If these behaviors are different, the use of a single mesh, even if it is adaptively refined, may lead to a not optimal numerical method. In this chapter, we propose a multi-mesh finite element method that makes it possible to resolve the local nature of different components independently of each other. This method works for Lagrange elements of arbitrary degree in any dimension. Furthermore, the method works "on top" of standard adaptive finite element methods. This makes it possible to easily implement the proposed multi-mesh here in existing finite element codes. The multi-mesh method can then directly make use of optimized single-mesh methods for calculating local element matrices, as for example precalculated integral tables or fast quadrature rules. We have implemented the multi-mesh method in the finite element software AMDiS for Lagrange finite elements up to fourth degree for 1D, 2D and 3D.

The use of multiple, independently refined meshes for the discretization of systems of PDEs is not new. To our best knowledge, [101] was the first work which considers a multi-mesh method in the context of adaptive finite elements. In [81, 35, 56], a very similar technique was used to simulate dendritic growth. An *hp*-FEM multi-mesh method is considered in [108, 109]. It is implemented in the finite element toolbox Hermes [107]. Although introducing a multi-mesh technique, in none of these publications the method is formally derived. Furthermore, implementation issues are not discussed and detailed runtime results, which compare the overall runtime between the single-mesh and the multi-mesh method are missing. In contrast, in this work we formally show how multiple meshes are used for assembling matrices and vectors in the finite element method. Furthermore, we compare the runtimes of using the single-mesh and the multi-mesh method for different simulations, and show that the multi-mesh method is superior to the single-mesh method, when one component of a system of PDEs can locally be resolved on a coarser mesh.

An alternative method commonly used to deal with different meshes for different components of coupled systems, especially in the case of multi-physics applications where independent simulation codes must be coupled, is MpCCI (mesh-based parallel code coupling interface) [60]. In this approach an interpolation between the different solutions from one mesh to the other is performed which for different resolutions of the involved meshes will lead to a loss in information and is thus not the method of choice for the problems to be discussed in this work.

Section 4.1 introduces the so called *virtual mesh assembling*, which is the basis of our multi-mesh method. It is shown, how to build coupling meshes in a virtual way and how the corresponding coupling operators are assembled on them. In Section 4.2, we present

several numerical experiments in 2D and 3D that show the advantages of the multi-mesh method.

4.1 Virtual mesh assembling

The basis of our multi-mesh method is the so called *virtual mesh assembling* approach. It allows to create union of two meshes in a virtual and thus memory efficient way. A virtual union of two meshes can be used for the discretization of terms which couples two PDEs within a system. First, this section provides a short introduction to systems of PDEs and presents the difficulties in using multiple meshes for their discretization. Section 4.1.2 introduces the *dual mesh traverse*. This algorithm simultaneously traverses two meshes and provides in this way a virtual union of theses meshes. Section 4.1.3 derives an assembling procedure to discretize coupling terms on virtual meshes. The last section presents details for an efficient implementation of the multi-mesh method in existing finite element codes.

4.1.1 Coupling terms in systems of PDEs

For an illustration, we consider the homogeneous biharmonic equation as a simple example for a system of PDEs. This equation reads

$$\Delta^2 u = 0 \text{ in } \Omega \quad \text{and} \quad u = \frac{\partial u}{\partial n} = 0 \text{ on } \partial\Omega \tag{4.1}$$

with $u \in C^4(\Omega) \cap C^1(\bar{\Omega})$. Using operator splitting, the biharmonic equation can be rewritten as a system of two second order PDEs

$$\begin{aligned} -\Delta u + v &= 0 \\ \Delta v &= 0 \end{aligned} \tag{4.2}$$

The standard mixed variational formulation of this system is [61]: find $(u, v) \in H_0^1(\Omega) \times H^1(\Omega)$ such that

$$\begin{aligned} \int_\Omega \nabla u \nabla \phi dx + \int_\Omega v\phi dx &= 0 \qquad \forall \phi \in H^1(\Omega) \\ \int_\Omega \nabla v \nabla \psi dx &= 0 \qquad \forall \psi \in H_0^1(\Omega) \end{aligned} \tag{4.3}$$

To discretize these equations, we assume that $\mathcal{T}_h^0$ and $\mathcal{T}_h^1$ are two different partitions of the domain Ω into simplices. Then, $V_h^0 = \{v_h \in H_0^1 : v_h|_T \in P^n \ \forall T \in \mathcal{T}_h^1\}$ and $V_h^1 = \{v_h \in H^1 : v_h|_T \in P^n \ \forall T \in \mathcal{T}_h^0\}$ are finite element spaces of globally continuous, piecewise polynomial functions of an arbitrary but fixed degree n. We thus obtain: find $(u_h, v_h) \in V_h^0 \times V_h^1$ such that

$$\begin{aligned} \int_\Omega \nabla u_h \nabla \phi dx + \int_\Omega v_h\phi dx &= 0 \qquad \forall \phi \in V_h^0(\Omega) \\ \int_\Omega \nabla v_h \nabla \psi dx &= 0 \qquad \forall \psi \in V_h^1(\Omega) \end{aligned} \tag{4.4}$$

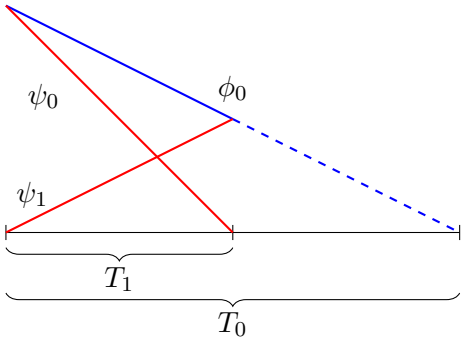


Figure 4.1: Linear combination of basis function ϕ_0 of the element T_0, restricted to the element T_1 as the linear combination $\psi_0 + \frac{1}{2}\psi_1$ of local basis functions on element T_1.

Defining $\{\phi_i \mid 1 \leq i \leq n\}$ and $\{\psi_i \mid 1 \leq i \leq m\}$ to be the nodal basis of V_h^0 and V_h^1, respectively, u_h and v_h can be written as the linear combinations $u_h = \sum_{i=1}^{n} u_i \phi_i$ and $v_h = \sum_{i=1}^{m} v_i \psi_i$, with u_i and v_i unknown real coefficients. Using these relations and replacing integrals over Ω by integrals over its partitions, (4.4) is rewritten to

$$\begin{aligned} \sum_{j=1}^{n} u_j \left(\sum_{T \in \mathcal{T}_h^0} \int_T \nabla\phi_j \cdot \nabla\phi_i \right) + \sum_{j=1}^{m} v_j \left(\sum_{T \in \mathcal{T}_h^0 \cup \mathcal{T}_h^1} \int_T \psi_j \phi_i \right) = 0 \qquad i = 1 \ldots n \\ \sum_{j=1}^{m} v_j \left(\sum_{T \in \mathcal{T}_h^1} \int_T \nabla\psi_j \cdot \nabla\psi_i \right) = 0 \qquad i = 1 \ldots m \end{aligned} \tag{4.5}$$

To compute the coupling term $\int_T \psi_j \phi_i$, we have to define the union of two different partitions, $T_h^0 \cup T_h^1$. For this, we make a restriction on the partitions: any element $T^0 \in \mathcal{T}_h^0$ is either a subelement of an element $T^1 \in \mathcal{T}_h^1$, or vice versa. This restriction is not very strict. It is always fulfilled for the standard refinement algorithms, e.g., bisectioning or red-green refinement (cf. Section 2.1), if the initial coarse mesh is the same for both partitions. Under these conditions, $\mathcal{T}_h^0 \cup \mathcal{T}_h^1$ is the union of the locally finest simplices.

The most common way to compute the integrals in (4.5) is to define local basis functions. We define $\phi_{i,j}$ to be the j-th local basis function on an element $T_i \in \mathcal{T}_h^0$. $\psi_{i,j}$ is defined in the same way for elements in partition $\mathcal{T}_h^1$.

Because the global basis functions ϕ_i and ψ_j are defined on different partitions of the same domain, it is not straightforward to calculate the coupling term $\int_\Omega \psi_j \phi_i$ in an efficient manner. For evaluating this integral, two different cases may occur: either the integral has to be evaluated on an element from the partition $\mathcal{T}_h^0$ or on an element from $\mathcal{T}_h^1$. For the first case, we must evaluate the integral

$$\int_{T_i \in \mathcal{T}_h^0} \psi_{k,l} \phi_{i,j} \tag{4.6}$$

for some j and l, and there exists an element $T_k \in \mathcal{T}_h^1$, with $T_i \subset T_k$. As we consider a multi-mesh method that should work on top of existing single-mesh methods, the integral is

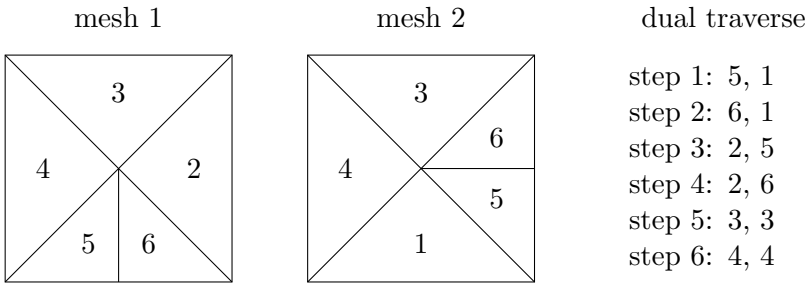


Figure 4.2: Dual mesh traverse on two independently refined meshes, which are defined on the same coarse mesh.

not in an appropriate form as $\psi_{k,l}$ is not a local basis function of element T_i. To overcome this problem we define the basis functions $\psi_{k,l}$ by a linear combination of local basis functions of T_i

$$\int_{T_i \in \mathcal{T}_h^0} \psi_{k,l}\phi_{i,j} = \int_{T_i \in \mathcal{T}_h^0} \sum_m (c_{k,m}\phi_{i,m})\phi_{i,j} \tag{4.7}$$

with some real coefficients $c_{k,m}$. Figure 4.1 illustrates this situation in 1D. For the other case, i.e., the integral in the coupling term is evaluated on an element $T_i \in \mathcal{T}_h^1$, we have

$$\int_{T_i \in \mathcal{T}_h^1} \psi_{k,l}\phi_{i,j} = \int_{T_i \in \mathcal{T}_h^1} \psi_{k,l} \sum_m (c_{i,m}\psi_{k,m}) \tag{4.8}$$

In summary, to evaluate the coupling terms, two different techniques have to be defined and implemented: first, the method requires that a union of two meshes is build. The resulting algorithm, which we name *dual mesh traverse*, is discussed in the next section. Once the union is obtained, we need to calculate the coefficients $c_{i,j}$, and to incorporate them into the finite element assemblage procedure such that the overall change of the standard method is as small as possible.

4.1.2 Creating a virtual mesh

The simplest way to obtain the union of two meshes is to employ the data structure they are stored in. In our case, we could explicitly build the union by joining the binary trees of both meshes into a set of new binary trees. However, such a procedure is not only too time-consuming, especially when we consider meshes that change in time, but also requires additional memory to store the joined mesh. We therefore do not work directly on the mesh data but instead use the mesh traverse algorithm, that creates the requested element data on demand. Using this method, we define the *dual mesh traverse* that traverses two meshes in parallel and creates the union of both meshes in a virtual way.

For the dual mesh traverse the only requirement is that both meshes must share the same coarse mesh structure, but they can be refined independently of each other. When using bisectioning, it then can be ensured: if the intersection of two elements of two different meshes is non empty, then either both elements are equal or one element is a real subelement of the other. To retrieve the leaf level of the virtual mesh, the dual mesh

traverse simultaneously traverses two binary trees, each corresponding to the same coarse element in both coarse meshes. The algorithm then calls a user defined function, e.g., the element assembling function or an element error estimator, that works on pairs of elements, with both, the larger and the smaller element, of the current traverse. The larger of both elements is fixed as long as all smaller subelements in the other mesh are traversed. Figure 4.2 shows a simple example for a coarse mesh consisting of four coarse elements and the order of traversed elements for the dual traverse algorithm. In the first mesh, coarse element 1, and in the second mesh coarse element 2, are refined once.

4.1.3 Assembling element matrices

Even though the integrals (4.7) and (4.8) are defined for basis functions on the local element, the form is not optimal for an efficient implementation in standard finite element single-mesh codes. Therefore, we redefine this transformation and distinguish for this two cases: the smaller of both elements defines either the space of test functions, or it defines the space of trial functions. For the first case, we consider the coupling term $\int_\Omega \psi_i \phi_j$ in (4.5), with some local basis functions ψ_i and ϕ_j, that must be assembled on a virtual mesh. Then, for some elements T and T', with $T' \subset T$, the element matrix $M_{T'}$ is given by

$$\begin{aligned} M_{T'} &= \begin{bmatrix} \int_{T'} \psi_0\phi_0 & \dots & \int_{T'} \psi_0\phi_n \\ \vdots & & \vdots \\ \int_{T'} \psi_n\phi_0 & \dots & \int_{T'} \psi_n\phi_n \end{bmatrix} \\ &= \begin{bmatrix} \int_{T'} \sum_i (c_{0i}\phi_i)\phi_0 & \dots & \int_{T'} \sum_i (c_{0i}\phi_i)\phi_n \\ \vdots & & \vdots \\ \int_{T'} \sum_i (c_{ni}\phi_i)\phi_0 & \dots & \int_{T'} \sum_i (c_{ni}\phi_i)\phi_n \end{bmatrix} \\ &= \begin{bmatrix} \sum_i c_{0i} \int_{T'} \phi_i\phi_0 & \dots & \sum_i c_{0i} \int_{T'} \phi_i\phi_n \\ \vdots & & \vdots \\ \sum_i c_{ni} \int_{T'} \phi_i\phi_0 & \dots & \sum_i c_{ni} \int_{T'} \phi_i\phi_n \end{bmatrix} \\ &= \mathcal{C} \cdot \begin{bmatrix} \int_{T'} \phi_0\phi_0 & \dots & \int_{T'} \phi_0\phi_n \\ \vdots & & \vdots \\ \int_{T'} \phi_n\phi_0 & \dots & \int_{T'} \phi_n\phi_n \end{bmatrix} \end{aligned} \tag{4.9}$$

where ϕ_i are the local basis functions defined on T', ψ_i are the local basis functions defined on T, and $\mathcal{C}$ is the transformation matrix for the local basis function from T to T'. This shows, that to assemble the element matrix on a virtual element, there is no need for large changes within the assembly procedure. The finite element code needs only to assemble the element matrix of the smaller element T' and multiply the result with the transformation matrix. Hence, if the transformation matrices can be computed easily, the overhead for virtual element assembling can be eliminated.

We now consider the second case, where the smaller element defines the space of trial

functions. Similar calculations as above show that the following holds

$$M_{T'} = \begin{bmatrix} \int_{T'} \phi_0\phi_0 & \dots & \int_{T'} \phi_0\phi_n \\ \vdots & & \vdots \\ \int_{T'} \phi_n\phi_0 & \dots & \int_{T'} \phi_n\phi_n \end{bmatrix} \cdot \mathcal{C}^T \tag{4.10}$$

The above calculations can be immediately generalized for general zero order terms of the form $\int_\Omega \psi_i c \phi_j$ with $c \in L^\infty(\Omega)$, as the transformation matrices $\mathcal{C}$ are independent of c. In a similar way we can also reformulate the element matrices for general first and second order terms. For a general second order term of the form $\int_\Omega \nabla\psi_i \cdot \mathbf{A}\nabla\phi_j$, with $\mathbf{A} : \Omega \mapsto \mathbb{R}^{d\times d}$, the element matrix $M_{T'}$ can be rewritten in the same way as we have done in (4.9) for zero order terms

$$\begin{aligned} M_{T'} &= \begin{bmatrix} \int_{T'} \nabla\psi_0 \cdot \mathbf{A}\nabla\phi_0 & \dots & \int_{T'} \nabla\psi_0 \cdot \mathbf{A}\nabla\phi_n \\ \vdots & & \vdots \\ \int_{T'} \nabla\psi_n \cdot \mathbf{A}\nabla\phi_0 & \dots & \int_{T'} \nabla\psi_n \cdot \mathbf{A}\nabla\phi_n \end{bmatrix} \\ &= \mathcal{C}_\nabla \cdot \begin{bmatrix} \int_{T'} \nabla\phi_0 \cdot \mathbf{A}\nabla\phi_0 & \dots & \int_{T'} \nabla\phi_0 \cdot \mathbf{A}\nabla\phi_n \\ \vdots & & \vdots \\ \int_{T'} \nabla\phi_n \cdot \mathbf{A}\nabla\phi_0 & \dots & \int_{T'} \nabla\phi_n \cdot \mathbf{A}\nabla\phi_n \end{bmatrix} \end{aligned} \tag{4.11}$$

where the coefficients of the matrix $\mathcal{C}_\nabla$ are defined such that

$$\nabla\psi_i = \sum_{j=0}^{n} c_{ij} \nabla\phi_i \tag{4.12}$$

If basis functions on the smaller element define the space of trial functions, we can establish the same relation as in (4.10). For general first order terms of the form $\int_\Omega \psi_i \mathbf{b} \cdot \nabla\phi_j$, with $\mathbf{b} \in [L^\infty(\Omega)]^d$, it is straightforward to verify that in the case the smaller element defines the test space, the element matrix $M_{T'}$ can be calculated on the smaller element and multiplied with $\mathcal{C}$ from the left. If the smaller element defines the space of trial functions, the element matrix calculated on the smaller element must be multiplied with $\mathcal{C}_\nabla^T$ from the right.

4.1.4 Calculation of transformation matrices

Once the transformation matrix is calculated for a given tuple of a small and a large element, the additional costs for virtual mesh assembling are small. We now consider computing these transformation matrices efficiently. For this, we assume that $\mathcal{C}$ and $\mathcal{C}_\nabla$ are calculated once for both children of the reference element. The transformation matrix of a specific pair of elements is then recursively defined using these reference transformation matrices. We formally define a *virtual element pair* of a small and a large element, which correspond to an element in the virtual mesh, as the tuple

$$(T, \{\alpha_0, \dots, \alpha_n\}) = (T, \alpha) \text{ with } \alpha_i \in \{\mathrm{L}, \mathrm{R}\} \tag{4.13}$$

where T is the larger element of the pair and α is an ordered set that is interpreted as the refinement sequence for the smaller element starting in element T. This directly correspond

to a mesh structure code, cf. Section 2.3. Here, L denotes the "left" children, and R denotes the "right" children of an element. Furthermore, we define a function TRA that uniquely maps a virtual element pair to the smaller element. It is defined recursively by

$$\begin{aligned} TRA(T, \emptyset) &= T \\ TRA(T, \{\alpha_0, \alpha_1, \ldots, \alpha_n\}) &= \begin{cases} TRA(T_{\mathrm{L}}, \{\alpha_1, \ldots, \alpha_n\}) & \text{if } \alpha_0 = \mathrm{L} \\ TRA(T_{\mathrm{R}}, \{\alpha_1, \ldots, \alpha_n\}) & \text{if } \alpha_0 = \mathrm{R} \end{cases} \end{aligned}$$

where T_{L} is the left child of element T, and T_{R} the right child of element T, respectively. In the same way we define transformation matrices as functions on refinement sequence

$$\mathcal{C}(\emptyset) = \mathrm{I} \tag{4.14}$$

$$\mathcal{C}(\{\alpha_0, \alpha_1, \ldots, \alpha_n\}) = \begin{cases} \mathcal{C}_{\mathrm{L}} \cdot \mathcal{C}(\{\alpha_1, \ldots, \alpha_n\}) & \text{if } \alpha_0 = \mathrm{L} \\ \mathcal{C}_{\mathrm{R}} \cdot \mathcal{C}(\{\alpha_1, \ldots, \alpha_n\}) & \text{if } \alpha_0 = \mathrm{R} \end{cases} \tag{4.15}$$

where $\mathcal{C}_L$ and $\mathcal{C}_R$ are the transformation matrices for the left child and the right child, respectively, of the reference element.

4.1.5 Implementation issues

Although computing one transformation matrix can be done very fast, it may considerably increase the time for assembling the linear system if there are many transformation matrices to be computed. This is especially the case, if one mesh is much coarser in some regions than the other mesh. To circumvent this problem, we have implemented a cache, that stores previously computed transformation matrices in a hash table [50]. In the mesh traverse routine, a 64 bit integer data type stores the refinement sequence between the large and the small element bit-wise, as it is defined by (4.13). If the bit on the i-th position is set, the finer element is a right-refinement of its parent element, otherwise it is a left-refinement of it. Using a fixed size of 64 bits to store the refinement sequence limits the level gap between the coarser and the finer element level to be less or equal to 64. But we have not found any practical simulations where this value is higher than 30. In favor of, this value can directly be used as a key in a hash table, and thus allows to efficiently check whether an element matrix was computed before for the same virtual mesh element. If this is not the case, the transformation matrix must be computed as stated in (4.14) and will be stored in the hash table. In general the hash table should be restricted to a maximum number of matrices to prevent uncontrolled memory usage if the number of entries increases. But in all of our simulations, the number of transformation matrices that should be stored in the cache never exceeded 100,000. Even for 3D simulations with linear elements, the overall memory usage is around 2 MByte, and can thus be neglected. Thus it has not yet been necessary to implement an upper limit for the cache. More information about the number and memory usage of the transformation matrices is given in the next section when presenting the numerical results.

All of the algorithms described here can easily be adjusted if data structure other than binary trees are used to represent the mesh. This may be the case if, e.g., a red-green refinement strategy is used, or if the mesh consists of quadrilaterals or cubes. In both

cases, quadtrees or octrees are employed to represent the adaptive mesh structure. For these data structures, the transformation matrices can be calculated in the same way as we have done it in this section for binary trees. Of course, the refinement sequences, as defined in Section 4.1.4, cannot be stored in this way. Here, either at least two bits for quadtrees or at least three bits for octrees are required to store the information about the children on the next refinement level.

4.2 Numerical results

We present several examples, where the multi-mesh approach is superior in contrast to the standard single-mesh finite element method. The examples considered here are phase-field equations applied to the study of solid-liquid and solid-solid phase transitions. For a recent review we refer to [94]. These equations involve at least one variable, the phase-field variable, which is almost constant in most parts of the domain and therefore can be discretized within these parts using a coarse mesh. At the interface region a high resolution is required to resolve the smooth transition between the different phases. A second variable in such systems is typically a diffusion field which in most cases varies smoothly across the whole domain, requiring a finer mesh outside of the interface and a coarser mesh within the interface. Such problems are therefore well suited for a multi-mesh approach, as has already be demonstrated in [101, 81, 35]. No detailed runtime comparisons are provided in these works to show the advantages of the multi-mesh method. To do this for dendritic growth, we will consider solidification and coarsening phenomena in binary alloys.

Other examples for which large computational savings due to the use of the multi-mesh approach are expected are diffuse interface and diffuse domain approximations for PDEs to be solved on surfaces that are within complicated domains. The approaches introduced in [96, 82], use a phase-field function to implicitly describe the domain the PDE has to be solved on. For the same reason as in phase-transition problems the distinct solution behavior of the different variables will lead to large savings if the multi-mesh approach is applied. The approach has already been used for applications such as chaotic mixing in microfluidics [1], tip splitting of droplets with soluble surfactants [114], and chemotaxis of stem cells in 3D scaffolds [77].

As a further example we demonstrate that the multi-mesh approach can also be used to easily fulfill the inf-sup condition for Stokes like problems if both components are discretized with linear Lagrange elements. We demonstrate this numerically for the incompressible Navier-Stokes equation with piecewise linear elements for velocity and pressure, but a finer mesh used for the velocity. Such an approach might be superior to mixed finite elements of higher order or stabilized schemes in terms of computational efficiency and implementation efforts. For a review on efficient finite element methods for the incompressible Navier-Stokes equation we refer to [117]. We demonstrate the applicability of the multi-mesh approach on the classical driven cavity problem.

4.2.1 Dendritic growth

We first consider dendritic growth using a phase-field model, as already introduced in Section 3.2.4. To compare the multi-mesh method with a standard adaptive finite element

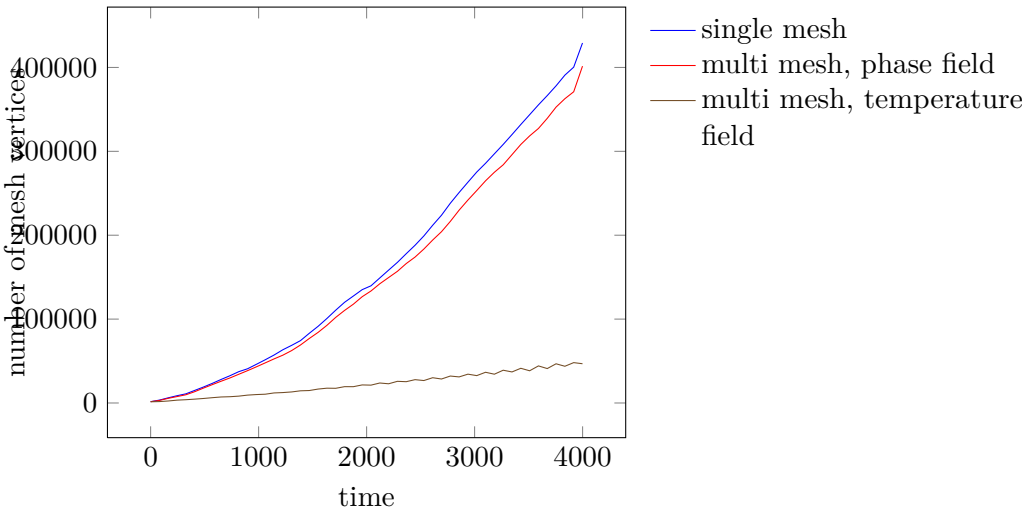


Figure 4.3: Evolution of degrees of freedom in time for the phase field and the temperature field using single-mesh and multi-mesh method.

approach, we have computed a dendrite using linear finite elements with the following parameters: $\xi = 0.65, D = 1.0, \epsilon = 0.05$. We use a constant timestep $\tau = 1.0$ up to time 4000. To speedup the computation we have employed the symmetry of the solution and limited the computation to the upper right quadrant with a domain size of 800 in each direction. The adaptive mesh refinement relies on the residuum based a posteriori error estimate, cf. (2.1) and (2.2. By setting C_0 to 0 and C_1 to 1, we restrict the estimator to the jump residuum only. We have set the tolerance to be $tol_\phi = 0.5$, and $tol_u = 0.25$. For adaptivity, the equidistribution strategy with parameters $\theta_R = 0.8$, and $\theta_C = 0.2$ was used. Thus, the interface thickness is resolved by approximately 20 grid points.

The result of both computations coincides at the final timestep. As a quantitative comparison we use the tip velocity of the dendrite. As reported in [63], for this parameter set analytical calculations lead to a steady-state tip velocity $V_{tip} = 0.0469$. In both of our calculations, the tip velocity varies by around 1% of this value. Using the single-mesh method, by the final timestep the mesh consist of 429,148 DOFs for each component. When our multi-mesh method is used, the same solution can be obtained with a mesh for the phase-field with 401,544 vertices, and 46,791 vertices for the temperature field. The gap between the number of vertices required to resolve the phase field and the temperature field increases in time, see Figure 4.3 that shows the development of the mesh size for both methods. Figure 4.4 qualitatively compares the meshes of the phase-field variable and the temperature field, which shows the expected finer resolution of the phase-field mesh within the interface, and its coarser resolution within the solid and liquid region.

The computational time for both methods is compared in Table 4.1. The assemblage procedure in the multi-mesh method is around 6% faster, although computing the element matrices is slower due to the extra matrix-matrix multiplication. This is easily explained by the fact that we have much less element matrices to compute and the overall matrix data structure is around 50% smaller with respect to the number of non-zero entries, see

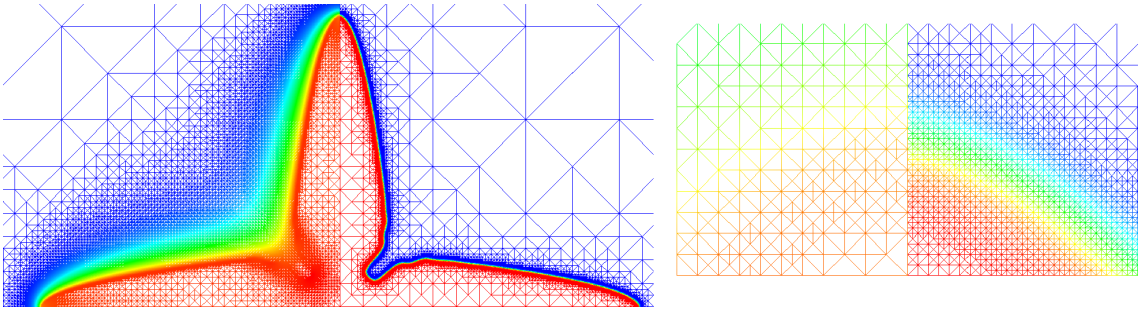


Figure 4.4: a) 2D dendrite computed at $t = 4000$ using the multi-mesh method with the parameters $\xi = 0.65$, $D = 1.0$, $\epsilon = 0.05$, and a timestep $\tau = 1.0$. Left shows the mesh of the temperature field and right shows the mesh of the phase field. b) Zoom into the upper tip showing the different resolution of both meshes.

also the more detailed data in Table 4.2. This is also reflected in the solution time for the linear system. We have run all computation twice, with using either UMFPACK, a multifrontal sparse direct solver [30], or the BiCGStab(ℓ) with the parameter $\ell = 2$ and diagonal preconditioning, that is part of the Matrix Template Library (MTL4) [52]. When using the first one within the multi-mesh method, the solution time can be reduced by 40% and also the memory usage, which is often the most critical limitation in the use of direct solvers, is reduced in this magnitude. An even more drastic reduction of the computation time can be achieved when using an iterative solver. Here, the number of iterations is around 30% less with the multi-mesh method and each iteration is faster due to the smaller matrix. This is explained in more detail in Table 4.2, that shows all relevant matrix data for the very last timestep of the simulation. Not only the overall matrix size is smaller, also the condition number of the matrices decreases when using the multi-mesh method. Hence, the iterative solver needs less iteration to solve the system of linear equations. The memory overhead for the multi-mesh method is quite small. At the very last timestep, only 3,215 transformation matrices are stored, requiring 0.35 MByte of memory. Figure 4.5 shows the evolution of the number of stored transformation matrices, and the average and maximum length of the refinement sequences which are used as keys to find the precomputed transformation matrices in the hash table.

The time for error estimation is halved, as expected, since it scales linearly with the number of elements in the mesh. Altogether, the time reduction is significant in all parts of the finite element method. In addition the approach also leads to a drastic reduction in memory usage.

The results are even more significant in 3D. Fig. 4.6 shows the result of computing a dendrite with the multi-mesh method and the parameters $\xi = 0.55$, $D = 1$, and $\epsilon = 0.05$. We have run the simulation with a constant timestep $\tau = 1.0$ up to time 2500. The evolution of degrees of freedom over time is quite similar to the 2D example. When using the multi-mesh method, the time for solving the linear system, again using the BiCGStab(ℓ) solver with diagonal preconditioning, can be reduced by around 60%. The time for error estimation is around half the time needed by the single-mesh method. Because the time

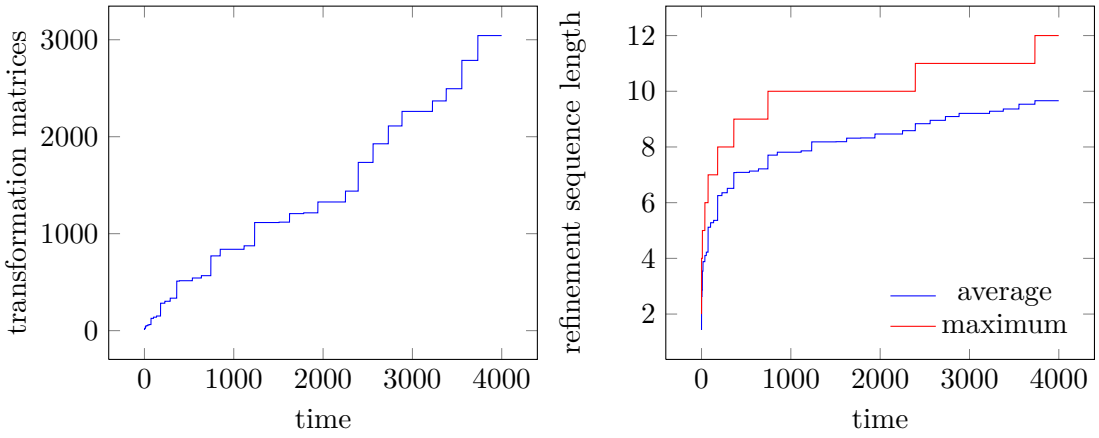


Figure 4.5: Evolution of the number of transformation matrices (left) and the average and maximum refinement sequence length (right) for a 2D dendritic growth computation.

	single-mesh	**multi-mesh**	**speedup**
assembler	5.98s	5.62s	6.0%
solver: UMFPACK	6.69s	4.12s	38.4%
solver: BiCGStab(ℓ)	14.26s	4.28s	69.9%
estimator	3.40s	1.71s	49.7%
overall with UMFPACK	16.07s	11.45s	28.7%
overall with BiCGStab(ℓ)	23.64s	11.61s	50.9%

Table 4.1: Comparison of runtime when using single-mesh and multi-mesh method for a 2D dendritic growth simulation. The values are the average of 4000 timesteps.

for assembling the linear system is more significant in 3D, the overall time reduction with the multi-mesh method is around 64.4%.

4.2.2 Coarsening

As a second example we consider coarsening of a binary structure using a Cahn-Hilliard equation. We here concentrate only on the phenomenological behavior of the solution and thus consider the simplest isotropic model, which reads

$$\begin{aligned} \partial_t \phi &= \Delta \mu \\ \mu &= -\epsilon \Delta \phi + \frac{1}{\epsilon} G'(\phi) \end{aligned} \tag{4.16}$$

for a phase-field function ϕ, and a chemical potential μ. The parameter ϵ defines a length scale over which the interface is smeared out, and $G(\phi) = 18\phi^2(1-\phi)^2$ defines a double-well

	single-mesh	**multi-mesh**
number of unknowns	858,140	448,335 (401,544 + 46,791)
matrix non zero values	11,811,959	6,057,416
matrix size	138.4 MByte	71.0 MByte
transformation matrices size	0 MByte	0.35 MByte
BiCGStab(ℓ) iterations	72	51
condition number estimate	$4.56 \cdot 10^7$	$8.87 \cdot 10^6$

Table 4.2: Matrix related data for the system of linear equations at the final timepoint (t = 4,000) in a 2D dendritic growth simulation.

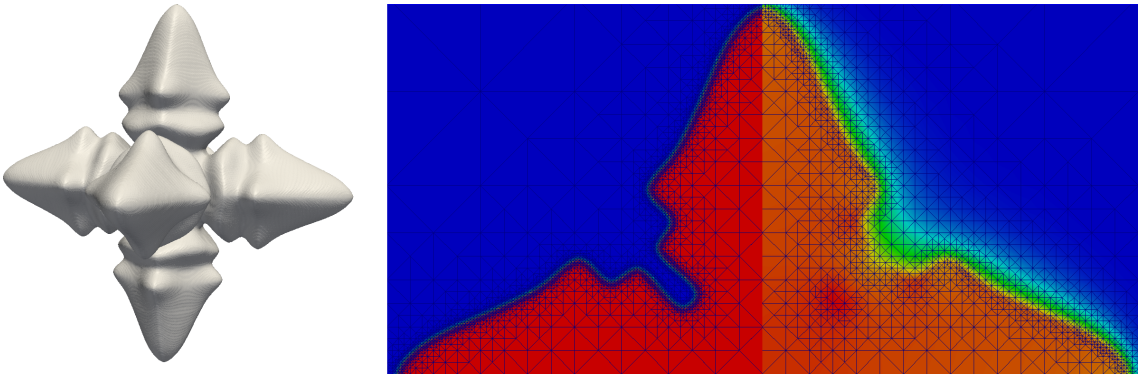


Figure 4.6: a) 3D dendrites at t = 2500 using the multi-mesh method with the parameters $\xi = 0.55$, $D = 1$, $\epsilon = 0.05$ and a timestep $\tau = 1.0$. b) A slice through the dendrites showing the mesh of the phase field on the left and the mesh of the temperature field on the right hand side.

potential. To discretize in time we use a semi-implicit Euler scheme

$$\begin{aligned} \frac{1}{\tau}\phi_{n+1} - \Delta\mu_{n+1} &= \frac{1}{\tau}\phi_n \\ \mu_{n+1} + \epsilon\Delta\phi_{n+1} - \frac{1}{\epsilon}G'(\phi_{n+1}) &= 0 \end{aligned} \tag{4.17}$$

in which we linearize $G'(\phi_{n+1}) \approx G''(\phi_n)\phi_{n+1} + G'(\phi_n) - G''(\phi_n)\phi_n$.

To compare our multi-mesh method with a standard adaptive finite element approach, we have computed the spinodal decomposition and coarsening process in 3D using Lagrange finite elements of fourth order. We use $\epsilon = 5 \cdot 10^{-4}$. The adaptive mesh refinement relies on the residuum based a posteriori error estimate. As we have done it in the dendritic growth simulation, also here only the jump residuum is considered, i.e., the constants C_0 and C_1 are set to 0 and 1, respectively. For both methods, the error tolerance are set to $tol_\phi = 2.5 \cdot 10^{-4}$ and $tol_\mu = 5 \cdot 10^{-2}$. For adaptivity, the equidistribution strategy with parameters $\theta_R = 0.8$ and $\theta_C = 0.2$ was used. Using these parameters, the interface thickness is resolved by around 10 grid points.

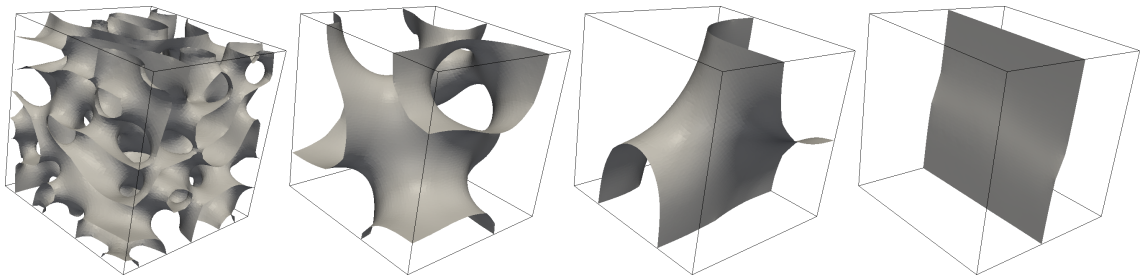


Figure 4.7: Solution of the Cahn-Hilliard equation for $t = 0.02$, $t = 1.0$, $t = 5.0$ and $t = 12.50$

The simulation was started from noise. The first mesh was globally refined with 196,608 elements. The constant timestep was chosen to be $\tau = 10^{-3}$. We have disabled the adaptivity for the first 10 timesteps, until a coarsening in the domain was achieved. Then the simulation was executed up to $t = 13.0$, where both phases are nearly separated. Figure 4.7 shows the phase field, i.e., the 0.5 contour of the first solution variable, for four different timesteps. The number of elements and degrees of freedom is linear to the area of the interface that must be resolved on the domain. Indeed, the chemical potential can be resolved on a much coarser grid, since it is independent of the resolution of the phase field. In the final state, the chemical potential is constant on the whole domain, and the coarse mesh (which consists of 6 tetrahedrons in this simulation) is enough to resolve it. The evolution of the number of elements for both variables over time is plotted in Figure 4.8. As expected, the number of elements for the phase field monotonically decreases as its area shrinks due to the coarsening process. The number of elements for the chemical potential rapidly decreases at the very first beginning, as the initial mesh is over refined to resolve this variable. For most of the simulation, the number of elements of the chemical potential is three orders of magnitude smaller than the number of elements for the phase field variable. This gap is also reflected in the computation time for the single-mesh and the multi-mesh method, which are compared in Table 4.3. The assembling procedure of the multi-mesh method is now 4.7% slower in comparison to the single-mesh method. The main reason therefore is that the transformation matrices are here large due to the 4^{th} order finite element in 3D. They are of size 35×35, and thus slow down the assembling procedure more than in the 2D example before, where the transformation matrices are of size 3×3 for linear finite elements. This small surcharged is payed off when comparing solver and error estimator run times. For solving the linear system of equations we again make use of BiCGStab(ℓ) solver with parameter $\ell = 2$ and diagonal preconditioning. To solve the systems arising in the multi-mesh methods takes less than 25% of time required for solving the systems when using the single-mesh approach. Here we see again both effects as already described in the numerical example before: each iteration of the solver is faster due to smaller matrices and vectors, and the system of equations are better conditioned in the case of the multi-mesh method leading to a smaller number of overall iterations. The overall solution time reduces from around 1,064 minutes when using the single-mesh method to 565 minutes when using the multi-mesh approach.

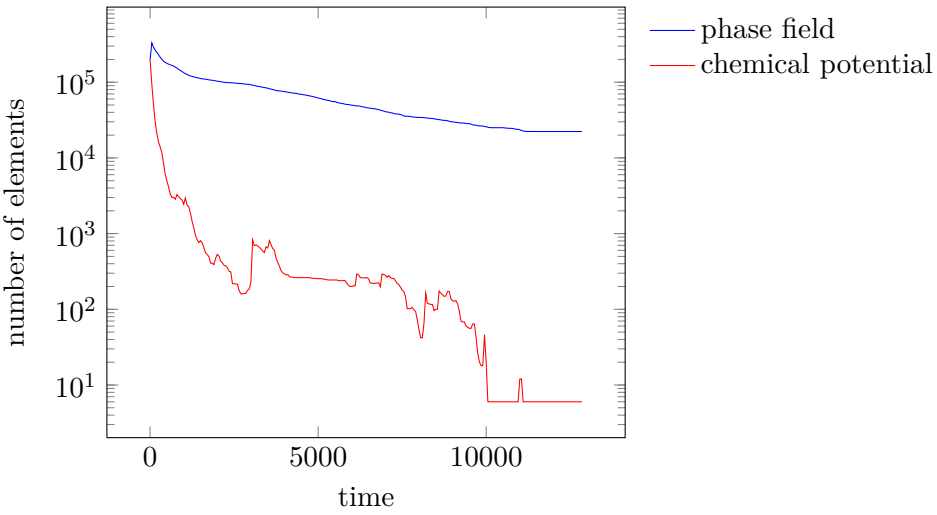


Figure 4.8: Evolution of number of elements for both variables of the Cahn-Hilliard equation.

	single-mesh	**multi-mesh**	**speedup**
assembler	19,718s	20,649s	-4.7%
solver: BiCGStab(ℓ)	26,178s	6,312s	75.8%
estimator	18,016s	6,967s	61.3%
overall	63,912s	33,928s	46.9%

Table 4.3: Comparison of runtime when using single-mesh and multi-mesh method for a 3D coarsening simulation using 4^{th} order Lagrange finite elements.

4.2.3 Fluid dynamics

For the last example, we use the multi-mesh method to solve problems in fluid dynamics using standard linear finite elements. The inf-sub stability is established by using two different meshes. In 2D, the mesh for the velocity components is refined twice more than the mesh for pressure. In the 3D case, the velocity mesh has to be refined three times to get the corresponding refinement structure. This discretization was introduced in [21], and was analyzed and proven to be stable in [120]. Although this is not the most efficient technique to ensure the inf-sub stability condition, it is a very simple way if multiple, independently refined meshes are supported in the finite element software used. We consider the standard instationary Navier-Stokes equation, cf. (3.58)-(3.60) in Section 3.6. The model problem is the "driven cavity" flow, as described and analyzed in [127, 47]. In a unit square, the boundary conditions for the velocity are set to be zero on the left, right and lower part of the domain. On the top, the velocity in the x direction is set to be one and in the y direction to be zero. In the upper corners, both velocities are set to be zero, which models the so called non-leaky boundary conditions. The computation was done for several Reynold

numbers varying between 50 and 1000. First, we have used the single-mesh method with a standard Taylor-Hood element, i.e., second order Lagrange finite elements for the velocity components and linear Lagrange finite elements for the pressure. We have compared these results with the multi-mesh method, where for both components linear finite elements were used and the mesh for velocity was refine twice more than the pressure. All computations were done with a fixed timestep $\tau = 0.01$, and aborted, when the relative change in velocity and pressure was less than 10^{-6}. Figure 4.2.3 shows the results for $Re = 1000$. In Table 4.4, we give the position of all eddies and compare our results with reference values from [47, 127].

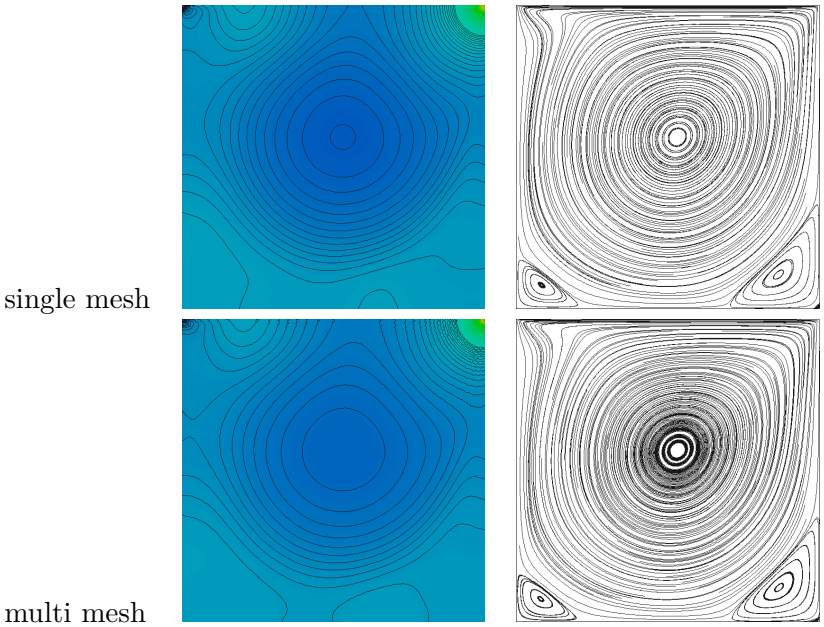


Figure 4.9: Results for $Re = 1000$. The left column shows 30 pressure isolines in the range $-0.12, 0.12$. The right column shows the streamlines contours of the velocity.

	Eddy 1	Eddy 2	Eddy 3	Eddy 4
single-mesh	0.5310, 0.5658	0.8633, 0.1116	0.0838, 0.0775	0.9937, 0.0062
multi-mesh	0.5305, 0.5671	0.8669, 0.1125	0.0813, 0.0750	0.9953, 0.0062
Wall	0.5308, 0.5660	0.8643, 0.1115	0.0832, 0.0775	0.9941, 0.0066
Ghia et al.	0.5313, 0.5625	0.8594, 0.1094	0.0859, 0.0781	0.9922, 0.0078

Table 4.4: Comparison of eddy position for $Re = 1000$ in the driven cavity model.

In both, the single-mesh method and the multi-mesh method, all finite element spaces have the same number of unknowns. For this reason, the multi-mesh method does not provide an improvement over the single-mesh method in terms of computing time. The time for assembling the linear system increases from 4.13 seconds to 5.79 seconds, which is mainly caused by the multiplication of the element matrices with the transformation matrices. Conversely, the average solution time with a BiCGStab(ℓ) solver and ILU preconditioning decreases from 10.18 seconds to 8.88 seconds. Although the linear systems have the same number of unknowns, the linear systems resulting from the single-mesh method are denser due to the usage of second order finite elements. The number of non-zero entries decreases by around 20% when linear elements are used on both meshes.

5 Conclusion and outlook

In the first part of this thesis we have presented software concepts, data structures and algorithms which allow to implement an efficient and scalable parallel finite element method. The central points of our parallelization approach are an implementation of an adaptively refined and distributed mesh data structure based on mesh structure codes and the including mesh-related information in the linear solver. These techniques make it possible to implement a large class of both, problem specific and general purpose linear solvers. In several benchmarks for weak and strong scaling we have shown very good parallel scalability for up to 16,384 cores. Nevertheless, the benchmarks show that some further work has to be done in order to go up to several hundred thousands of cores. The main question here is, if it is possible to get rid of the requirement that each coarse mesh element is contained in exactly one subdomain, but still make use of the very efficient mesh structure codes.

Many solver methods are known for systems of linear equations resulting from the finite element method. Some of them can be proven to have perfect, or nearly perfect, parallel scaling properties, e.g. the FETI-DP method. We have shown an implementation of this up-to-date solver method based on our general software concepts. Though the FETI-DP method is quasi optimal from a mathematical point of view, its very sparse and globally distributed coarse grid matrix leads to a break-down of parallel scalability for more than 1,000 cores. This is not due to a specific implementation as it was also observed by others [73, 74]. To circumvent this problem of using a sparse and globally distributed coarse grid matrix, several multilevel FETI-DP and BDDC methods were developed lately [73, 90, 125]. Though all of them scale much better than the corresponding implementation of the standard FETI-DP method, they have at least one drawback: either they are fixed to an a-priori defined number of coarse grid levels and cannot easily be generalized to a real multilevel method with an arbitrary number of coarse grid levels, or the condition number grows with increasing number of coarse grid levels. We have derived a new multilevel FETI-DP method, which has none of these disadvantages and thus is a good candidate for a general purpose linear solver which is scalable up to a large number of cores. In this work, the method is derived for two coarse levels only. But from the presentation it is clear that it can be directly generalized in a recursive way to an arbitrary number of levels. Furthermore, all benchmarks indicate that the condition number of the preconditioned system is independent of the number and of the size of the coarse grid. Besides a formal analysis of the presented multilevel FETI-DP method, a corresponding preconditioner must be developed to make the method suited for practical use.

To further improve efficiency of adaptive finite element simulations we considered the usage of different adaptively refined meshes for different variables in systems of nonlinear, time-dependent PDEs. The different variables can have very distinct solution behavior. To resolve this the meshes can be independently adapted for each variable. The multi-mesh

method, as defined in this thesis, can make use of Lagrange finite elements of arbitrary degree and is independent of the spatial dimension. The approach is well defined, and can be implemented in existing adaptive finite element codes with minimal effort. The additional computational effort for assembling matrices on virtual meshes is very small and can be neglected in most computations. Only small transformation matrices have to be multiplied with the matrices assembled on mesh elements. These matrices can be stored in an appropriate data structure to avoid unnecessary recalculations. We have demonstrated for various examples that the resulting linear systems are usually much smaller, when compared to the usage of a single mesh, and the overall computational runtime can be more than halved in various cases.

This work is the very first rigorous derivation of a multi-mesh method. In contrast to existing work [101, 108, 109], we have shown that using virtual meshes results in the same matrices when combining the two meshes physically to a union of both. Our approach is very simple to implement in existing finite element codes, as it does not rely on any special data structure. Furthermore, it is not restricted to linear finite elements, but generalizes to Lagrange finite elements of arbitrary degree. Though presented here for triangles/tetrahedrons and bisectioning, it can easily be modified to other elements and other refinement strategies. Finding a general strategy for error estimation and mesh adaption in the multi-mesh finite element method is still an open problem.

Bibliography

[1] S. Aland, J. Lowengrub, and A. Voigt. Two-phase flow in complex geometries - a diffuse domain approach. *Comput. Meth. Eng. Sci.*, 57(1):77 – 108, 2010.

[2] P. Amestoy, I. Duff, J. L'Excellent, and J. Koster. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM Journal on Matrix Analysis and Applications*, 23(1):15–41, 2001.

[3] P. R. Amestoy, A. Guermouche, J.-Y. L'Excellent, and S. Pralet. Hybrid scheduling for the parallel solution of linear systems. *Parallel Computing*, 32(2):136 – 156, 2006.

[4] I. Babuška and W. C. Rheinboldt. A-posteriori error estimates for the finite element method. *International Journal for Numerical Methods in Engineering*, 12(10):1597–1615, 1978.

[5] I. Babuška and M. Suri. The p and h-p versions of the finite element method, basic principles and properties. *SIAM Review*, 36(4):578–632, 1994.

[6] R. Backofen, M. Gräf, D. Potts, S. Praetorius, A. Voigt, and T. Witkowski. A continuous approach to discrete ordering on $\mathbb{S}^2$. *Multiscale Modeling and Simulation*, 9(1):314–334, 2011.

[7] R. Backofen, A. Rätz, and A. Voigt. Nucleation and growth by a phase field crystal (PFC) model. *Philosophical Magazine Letters*, 87(11):813–820, 2007.

[8] R. Backofen, A. Voigt, and T.Witkowski. In preparation. 2013.

[9] R. Backofen, A. Voigt, and T. Witkowski. Particles on curved surfaces: A dynamic approach by a phase-field-crystal model. *Phys. Rev. E*, 81:025701, Feb 2010.

[10] A. Baker, R. Falgout, T. Gamblin, T. Kolev, M. Schulz, and U. Yang. Scaling algebraic multigrid solvers: On the road to exascale. In C. Bischof, H.-G. Hegering, W. E. Nagel, and G. Wittum, editors, *Competence in High Performance Computing 2010*, pages 215–226. Springer Berlin Heidelberg, 2012.

[11] C. Baker and M. Heroux. Tpetra, and the use of generic programming in scientific computing. *Scientific Programming*, 20(2):115–128, 2012.

[12] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, S. Kumar, E. Lusk, R. Thakur, and J. L. Träff. MPI on a million processors. In *Proceedings of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 20–30, Berlin, Heidelberg, 2009. Springer-Verlag.

[13] S. Balay, J. Brown, K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 3.2, Argonne National Laboratory, 2011.

[14] S. Balay, J. Brown, K. Buschelman, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang. PETSc web page. `http://www.mcs.anl.gov/petsc`.

[15] W. Bangerth, C. Burstedde, T. Heister, and M. Kronbichler. Algorithms and data structures for massively parallel generic adaptive finite element codes. *ACM Trans. Math. Softw.*, 38(2):14:1–14:28, Jan. 2012.

[16] W. Bangerth, R. Hartmann, and G. Kanschat. deal.II – a general purpose object oriented finite element library. *ACM Trans. Math. Softw.*, 33(4):24/1–24/27, 2007.

[17] W. Bangerth, T. Heister, and G. Kanschat. deal.II web page. `http://www.dealii.org`.

[18] P. Bastian, M. Blatt, A. Dedner, C. Engwer, J. Fahlke, C. Gräser, R. Klöfkorn, M. Nolte, M. Ohlberger, and O. Sander. DUNE web page. `http://www.dune-project.org`.

[19] T. Belytschko, R. Gracie, and G. Ventura. A review of extended/generalized finite element methods for material modeling. *Modelling and Simulation in Materials Science and Engineering*, 17(4):043001, 2009.

[20] M. Benzi, G. H. Golub, and J. Liesen. Numerical solution of saddle point problems. *Acta Numerica*, 14:1–137, 2005.

[21] M. Bercovier and O. Pironneau. Error estimates for finite element method solution of the Stokes problem in the primitive variables. *Numer. Math.*, 33(2):211–224, 1979.

[22] W. J. Boettinger, J. A. Warren, C. Beckermann, and A. Karma. Phase-field simulation of solidification. *Annual Review of Materials Research*, 32(1):163–194, 2002.

[23] E. Boman, K. Devine, L. A. Fisk, R. Heaphy, B. Hendrickson, V. Leung, C. Vaughan, U. Catalyurek, D. Bozdag, and W. Mitchell. Zoltan web page. `http://www.cs.sandia.gov/Zoltan`.

[24] P. Boyanova, M. Do-Quang, and M. Neytcheva. Efficient preconditioners for large scale binary Cahn-Hilliard models, 2012.

[25] J. H. Bramble and J. E. Pasciak. A preconditioning technique for indefinite systems resulting from mixed approximations of elliptic problems. *Mathematics of Computation*, 50(181):1–17, 1988.

[26] J. Brown, M. G. Knepley, D. A. May, L. C. McInnes, and B. F. Smith. Composable linear solvers for multiphysics. *Preprint ANL/MCS-P2017-0112, Argonne National Laboratory, submitted to the 11th International Symposium on Parallel and Distributed Computing (ISPDC 2012)*, 2012.

[27] C. Burstedde, O. Ghattas, M. Gurnis, T. Isaac, G. Stadler, T. Warburton, and L. C. Wilcox. Extreme-scale AMR. In *SC10: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. ACM/IEEE, 2010.

[28] C. Burstedde, L. C. Wilcox, and O. Ghattas. `p4est`: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees. *SIAM Journal on Scientific Computing*, 33(3):1103–1133, 2011.

[29] U. Catalyurek, E. G. Boman, K. D. Devine, D. Bozdag, R. Heaphy, and L. A. Riesen. A repartitioning hypergraph model for dynamic load balancing. Sandia National Laboratories Tech. Report SAND2008-2304J, Sandia National Laboratories, Albuquerque, NM, 2008. Submitted to J. Par. Dist. Comp.

[30] T. A. Davis. Algorithm 832: UMFPACK v4.3—an unsymmetric-pattern multifrontal method. *ACM Trans. Math. Softw.*, 30(2):196–199, June 2004.

[31] T. A. Davis. A column pre-ordering strategy for the unsymmetric-pattern multifrontal method. *ACM Trans. Math. Softw.*, 30(2):165–195, June 2004.

[32] J. P. De S. R. Gago, D. W. Kelly, O. C. Zienkiewicz, and I. Babuska. A posteriori error analysis and adaptive processes in the finite element method: Part II - adaptive mesh refinement. *International Journal for Numerical Methods in Engineering*, 19(11):1621–1656, 1983.

[33] A. Dedner, R. Klöfkorn, M. Nolte, and M. Ohlberger. A generic interface for parallel and adaptive discretization schemes: abstraction principles and the dune-fem module. *Computing*, 90:165–196, 2010. 10.1007/s00607-010-0110-3.

[34] K. Devine, E. Boman, R. Heaphy, B. Hendrickson, and C. Vaughan. Zoltan data management services for parallel dynamic applications. *Computing in Science and Engineering*, 4(2):90–97, 2002.

[35] Y. Di and R. Li. Computation of dendritic growth with level set model using a multi-mesh adaptive finite element method. *J. Sci. Comput.*, 39(3):441–453, 2009.

[36] M. Dryja, B. Smith, and O. Widlund. Schwarz analysis of iterative substructuring algorithms for elliptic problems in three dimensions. *SIAM Journal on Numerical Analysis*, 31(6):1662–1694, 1994.

[37] K. R. Elder, M. Katakowski, M. Haataja, and M. Grant. Modeling elasticity in crystal growth. *Phys. Rev. Lett.*, 88:245701, Jun 2002.

[38] K. R. Elder, N. Provatas, J. Berry, P. Stefanovic, and M. Grant. Phase-field crystal modeling and classical density functional theory of freezing. *Physical Review B*, 75(6):1–14, Feb. 2007.

[39] H. Elman, D. Silvester, and A. Wathen. Preformance and anlysis of saddle point preconditioners for the discrete steady-state Navier-Stokes equations. Technical report UMIACS-TR-2000-54, 2000.

[40] K. Eriksson and C. Johnson. Adaptive finite element methods for parabolic problems i: A linear model problem. *SIAM Journal on Numerical Analysis*, 28(1):43–77, 1991.

[41] R. Falgout, A. Cleary, J. Jones, E. Chow, V. Henson, C. Baldwin, P. Brown, P. Vassilevski, and U. M. Yang. hypre web page. `http://acts.nersc.gov/hypre/`.

[42] R. Falgout, J. Jones, and U. M. Yang. The design and implementation of hypre, a library of parallel high performance preconditioners. In A. Bruaset and A. Tveito, editors, *Numerical Solution of Partial Differential Equations on Parallel Computers*, volume 51 of *Lecture Notes in Computational Science and Engineering*, pages 267–294. Springer Berlin Heidelberg, 2006.

[43] C. Farhat, M. Lesoinne, P. LeTallec, K. Pierson, and D. Rixen. FETI-DP: a dual-primal unified FETI method-part I: A faster alternative to the two-level FETI method. *International Journal for Numerical Methods in Engineering*, 50(7):1523–1544, 2001.

[44] C. Farhat, M. Lesoinne, and K. Pierson. A scalable dual-primal domain decomposition method. *Numerical Linear Algebra with Applications*, 7(7-8):687–714, 2000.

[45] C. Farhat, J. Li, and P. Avery. A FETI-DP method for the parallel iterative solution of indefinite and complex-valued solid and shell vibration problems. *Internat. J. Numer. Methods Engrg.*, 63(3):398–427, 2005.

[46] S. Filippone and M. Colajanni. PSBLAS: a library for parallel linear algebra computation on sparse matrices. *ACM Trans. Math. Softw.*, 26(4):527–550, Dec. 2000.

[47] U. Ghia, K. Ghia, and C. Shin. High-re solutions for incompressible flow using the Navier-Stokes equations and a multigrid method. *Journal of Computational Physics*, 48(3):387 – 411, 1982.

[48] B. Gmeiner, T. Gradl, H. Kostler, and U. Rüde. Highly parallel geometric multigrid algorithm for hierarchical hybrid grids. In J. Grotendorst, G. Sutmann, G. Gompper, and D. Marx, editors, *Hierarchical Methods for Dynamics in Complex Molecular Systems*, pages 323–330. Forschungszentrum Jülich, 2012.

[49] G. H. Golub and C. F. van Van Loan. *Matrix Computations*. The Johns Hopkins University Press, 1996.

[50] M. T. Goodrich, R. Tamassia, and D. M. Mount. *Data Structures and Algorithms in C++*. Wiley and Sons, 2011.

[51] P. Gottschling and T. Hoefler. Productive parallel linear algebra programming with unstructured topology adaption. In *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*, pages 9 –16, 2012.

[52] P. Gottschling, D. Wise, and M. Adams. Representation-transparent matrix algorithms with scalable performance. In *ICS '07: Proceedings of the 21st annual international conference on Supercomputing*, pages 116–125, New York, NY, USA, 2007. ACM.

[53] T. Hoefler, T. Schneider, and A. Lumsdaine. Characterizing the influence of system noise on large-scale applications by simulation. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[54] J. Hoffman, J. Jansson, C. Degirmenci, N. Jansson, and M. Nazarov. *Unicorn: a Unified Continuum Mechanics Solver*, chapter 18. Springer, 2012.

[55] J. Hoffman, J. Jansson, M. Nazarov, and N. Jansson. Unicorn web page. `https://launchpad.net/unicorn`.

[56] X. Hu, R. Li, and T. Tang. A multi-mesh adaptive finite element approximation to phase field models. *Commun. Comput. Phys.*, 5:1012–1029, 2009.

[57] K. Iglberger, G. Hager, J. Treibig, and U. Rüde. Expression templates revisited: A performance analysis of current methodologies. *SIAM Journal on Scientific Computing*, 34(2):C42–C69, 2012.

[58] I. C. F. Ipsen and C. Meyer. The idea behind Krylov methods. *The American Mathematical Monthly*, 105(10):889–899.

[59] N. Jansson, J. Hoffman, and J. Jansson. Framework for massively parallel adaptive finite element computational fluid dynamics on tetrahedral meshes. *SIAM Journal on Scientific Computing*, 34(1):C24–C41, 2012.

[60] W. Joppich and M. Kürschner. MpCCI-a tool for the simulation of coupled applications. *Conc. Comput. Prac. Exp.*, 10:183–192, 2005.

[61] M. Jung. Fast parallel solvers for fourth-order boundary value problems. In *Proceedings of the 10th ParCo Conference*, pages 267–274, 2003.

[62] A. Karma and W.-J. Rappel. Phase-field method for computationally efficient modeling of solidification with arbitrary interface kinetics. *Phys. Rev. E*, 53:R3017–R3020, Apr 1996.

[63] A. Karma and W.-J. Rappel. Quantitative phase-field modeling of dendritic growth in two and three dimensions. *Phys. Rev. E*, 57:4323–4349, Apr 1998.

[64] D. Kaushik, M. Smith, A. Wollaber, B. Smith, A. Siegel, and W. S. Yang. Enabling high fidelity neutron transport simulations on petascale architectures, 2009. SC'09 Gordon Bell Prize Finalist.

[65] D. Kay and D. Loghin. A Green's function preconditioner for the steady-state Navier-Stokes equations. Computing Laboratory Report 99/06, Oxford University, 1999.

[66] D. Kay and R. Welford. Efficient numerical solution of Cahn-Hilliard-Navier-Stokes fluids in 2D. *SIAM Journal on Scientific Computing*, 29(6):2241–2257, 2007.

[67] D. W. Kelly, J. P. De S. R. Gago, O. C. Zienkiewicz, and I. Babuska. A posteriori error analysis and adaptive processes in the finite element method: Part I - error analysis. *International Journal for Numerical Methods in Engineering*, 19(11):1593–1619, 1983.

[68] H. Kim, C. Lee, and E. Park. A FETI-DP formulation for the Stokes problem without primal pressure components. *SIAM Journal on Numerical Analysis*, 47(6):4142–4162, 2010.

[69] B. S. Kirk, J. W. Peterson, R. H. Stogner, and G. F. Carey. `libMesh`: A C++ Library for Parallel Adaptive Mesh Refinement/Coarsening Simulations. *Engineering with Computers*, 22(3–4):237–254, 2006.

[70] A. Klawonn, L. F. Pavarino, and O. Rheinbach. Spectral element FETI-DP and BDDC preconditioners with multi-element subdomains. *Computer Methods in Applied Mechanics and Engineering*, 198(3-4):511 – 523, 2008.

[71] A. Klawonn and O. Rheinbach. Inexact FETI-DP methods. *International Journal for Numerical Methods in Engineering*, 69(2):284–307, 2007.

[72] A. Klawonn and O. Rheinbach. Robust FETI-DP methods for heterogeneous three dimensional elasticity problems. *Computer Methods in Applied Mechanics and Engineering*, 196(8):1400 – 1414, 2007.

[73] A. Klawonn and O. Rheinbach. A hybrid approach to 3-level FETI. *PAMM*, 8(1):10841–10843, 2008.

[74] A. Klawonn and O. Rheinbach. Highly scalable parallel domain decomposition methods with an application to biomechanics. *ZAMM - Journal of Applied Mathematics and Mechanics / Zeitschrift für Angewandte Mathematik und Mechanik*, 90(1):5–32, 2010.

[75] A. Klawonn and O. B. Widlund. Dual-primal FETI methods for linear elasticity. *Communications on Pure and Applied Mathematics*, 59(11):1523–1572, 2006.

[76] T. Kozubek, M. Jarošlavá, M. Menšík, and A. Markopoulos. Hybrid total FETI method. *PRACE whitepaper*, 2012.

[77] C. Landsberg, F. Stenger, A. Deutsch, M. Gelinsky, A. Rösen-Wolff, and A. Voigt. Chemotaxis of mesenchymal stem cells within 3D biomimetic scaffolds-a modeling approach. *Journal of Biomechanics*, 44(2):359 – 364, 2011.

[78] K. H. Law. A parallel finite element solution method. *Computers and Structures*, 23(6):845 – 858, 1986.

[79] J. Li. *Dual-Primal FETI Methods for Stationary Stokes and Navier-Stokes Equations*. PhD thesis, New York University, 2002.

[80] J. Li and O. B. Widlund. FETI-DP, BDDC, and block cholesky methods. *International Journal for Numerical Methods in Engineering*, 66(2):250–271, 2006.

[81] R. Li. On multi-mesh h-adaptive methods. *J. Sci. Comput.*, 24(3):321–341, 2005.

[82] X. Li, J. Lowengrub, A. Rätz, and A. Voigt. Solving PDEs in complex geometries. *Comm. Math. Sci.*, 7:81–107, 2009.

[83] A. Logg, K.-A. Mardal, G. N. Wells, and et al. FEniCS web page. `http://fenicsproject.org`.

[84] A. Logg, K.-A. Mardal, G. N. Wells, et al. *Automated Solution of Differential Equations by the Finite Element Method*. Springer, 2012.

[85] A. Logg, G. N. Wells, and J. Hake. *DOLFIN: a C++/Python Finite Element Library*. Springer, 2012.

[86] J. Mandel. Iterative solvers by substructuring for the p-version finite element method. *Computer Methods in Applied Mechanics and Engineering*, 80(1-3):117 – 128, 1990.

[87] J. Mandel. On block diagonal and Schur complement preconditioning. *Numerische Mathematik*, 58:79–93, 1990. 10.1007/BF01385611.

[88] J. Mandel. Balancing domain decomposition. *Communications in Numerical Methods in Engineering*, 9(3):233–241, 1993.

[89] J. Mandel and B. Sousedík. BDDC and FETI-DP under minimalist assumptions. *Computing*, 81:269–280, 2007.

[90] J. Mandel, B. Sousedík, and C. Dohrmann. Multispace and multilevel BDDC. *Computing*, 83:55–85, 2008. 10.1007/s00607-008-0014-7.

[91] T. P. Matthew. *Domain Decomposition Methods for the Numerical Solution of Partial Differential Equations*. Springer, 2008.

[92] R. T. Mills, V. Sripathi, G. Mahinthakumar, G. Hammond, P. C. Lichtner, and B. F. Smith. Engineering PFLOTRAN for scalable performance on Cray XT and IBM BlueGene architectures. In *Proceedings of SciDAC 2010 Annual Meeting*, 2010.

[93] J. M. Ortega and R. G. Voigt. Solution of partial differential equations on vector and parallel computers. *SIAM Review*, 27(2):149–240, 1985.

[94] N. Provatas and K. Elder. *Phase-field methods in materials science and engineering*. Wiley, 2010.

[95] N. Provatas, N. Goldenfeld, and J. Dantzig. Efficient computation of dendritic microstructures using adaptive mesh refinement. *Phys. Rev. Lett.*, 80(15):3308–3311, Apr 1998.

[96] A. Rätz and A. Voigt. PDEs on surfaces - a diffuse interface approach. *Comm. Math. Sci.*, 4:575–590, 2006.

[97] A. Ribalta, C. Stoecker, S. Vey, and A. Voigt. 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*, pages 615–621. Springer Berlin Heidelberg, 2008.

[98] Y. Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, 2003.

[99] K. Schloegel, G. Karypis, and V. Kumar. ParMETIS web page. `http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview`.

[100] K. Schloegel, G. Karypis, and V. Kumar. Parallel static and dynamic multi-constraint graph partitioning. *Concurrency and Computation: Practice and Experience*, 14(3):219–240, 2002.

[101] A. Schmidt. A multi-mesh finite element method for phase field simulations. *Lecture Notes in Computational Science and Engineering*, 32:208–217, 2003.

[102] A. Schmidt and K. G. Siebert. *Design of Adaptive Finite Element Software: The Finite Element Toolbox ALBERTA*. Springer, 2005.

[103] D. Silvester, H. Elman, D. Kay, and A. Wathen. Efficient preconditioning of the linearized Navier-Stokes equations for incompressible flow. *Journal of Computational and Applied Mathematics*, 128(1-2):261–279, 2001.

[104] V. Simoncini and D. B. Szyld. Recent computational developments in Krylov subspace methods for linear systems. *Numerical Linear Algebra with Applications*, 14(1):1–59, 2007.

[105] M. A. Smith, C. Rabiti, D. Kaushik, B. Smith, W. S. Yang, and G. Palmiotti. Fast reactor core simulations using the UNIC code. In *Proceedings of the International Conference on the Physics of Reactors, Nuclear Power: A Sustainable Resource*, 2008.

[106] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI - The complete reference: volume 1, the MPI core*. The MIT press, 1998.

[107] P. Solin. Hermes - web pabe. `http://hpfem.org/hermes/`.

[108] P. Solin, J. Cerveny, and L. Dubcova. Adaptive multi-mesh hp-fem for linear thermoelasticity. Research Report No. 2007-08, The University of Texas at El Paso, 2007.

[109] P. Solin, L. Dubcova, and J. Kruis. Adaptive hp-fem with dynamical meshes for transient heat and moisture transfer problems. *J. Comput. Appl. Math.*, 233(12):3103–3112, 2010.

[110] F. Stenger. meshconv tutorial.

[111] A. Stukowski. OVITO: The Open Visualization Tool - web page. `http://www.ovito.org/`.

[112] A. Stukowski. Visualization and analysis of atomistic simulation data with OVITO-the open visualization tool. *Modelling and Simulation in Materials Science and Engineering*, 18(1):015012, 2010.

[113] H. Sundar, G. Biros, C. Burstedde, J. Rudi, O. Ghattas, and G. Stadler. Parallel geometric-algebraic multigrid on unstructured forests of octrees. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 43:1–43:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[114] K. Teigen, A. Rätz, and A. Voigt. A diffuse-interface method for two-phase flows with soluble surfactants. *Journal of Computational Physics*, 230(2):375 – 393, 2011.

[115] C. Traxler. An algorithm for adaptive mesh refinement inn dimensions. *Computing*, 59:115–137, 1997.

[116] U. Trottenberg, C. W. Oosterlee, and A. Schüller. *Multigrid: Basics, Parallelism and Adaptivity.* Academic Press, 2000.

[117] S. Turek. *Efficient solvers for incompressible flow problems: an algorithmic and computational approach.* Springer, 1999.

[118] S. van Teeffelen, R. Backofen, A. Voigt, and H. Löwen. Derivation of the phase-field-crystal model for colloidal solidification. *Phys. Rev. E*, 79:051404, May 2009.

[119] T. Vejchodský, P. Šolín, and M. Zítka. Modular hp-FEM system HERMES and its application to Maxwell's equations. *Mathematics and Computers in Simulation*, 76(1-3):223 – 228, 2007.

[120] R. Verfürth. Error estimates for a mixed finite element approximation of the Stokes equations. *RAIRO Anal. Numer.*, 18:175–182, 1984.

[121] R. Verfürth. A posteriori error estimates for nonlinear problems. Finite element discretizations of elliptic equations. *Math. Comp.*, 62(206):445–475, 1994.

[122] R. Verfürth. A posteriori error estimation and adaptive mesh-refinement techniques. In *ICCAM'92: Proceedings of the fifth international conference on Computational and applied mathematics*, pages 67–83, Amsterdam, The Netherlands, The Netherlands, 1994. Elsevier Science Publishers B. V.

[123] S. Vey. *Adaptive finite elements for systems of PDEs, software concepts, multi-level techniques and parallelization.* PhD thesis, TU Dresden, 2008.

[124] S. Vey and A. Voigt. AMDiS: adaptive multidimensional simulations. *Computing and Visualization in Science*, 10:57–67, 2007.

[125] J. Šístek, J. Mandal, B. Sousedík, and P. Burda. Parallel implementation of multilevel BDDC. *submitted*, 2012.

[126] J. Šístek, B. Sousedík, P. Burda, J. Mandel, and J. Novotný. Application of the parallel BDDC preconditioner to the Stokes flow. *Computers and Fluids*, 46(1):429 – 435, 2011.

[127] W. Wall. *Fluid-Struktur-Interaktion mit stabilisierten Finiten Elementen*. PhD thesis, Institut für Baustatistik der Universität Stuttgart, 1999.

[128] M. Wenzlaff. Numerical analysis of the fibrous structure of coloum cactouses. Belegarbeit, Intitut für Leichtbau und Kunststofftechnik der TU Dresden, 2012.

[129] T. Witkowski, R. Backofen, and A. Voigt. The influence of membrane bound proteins on phase separation and coarsening in cell membranes. *Phys. Chem. Chem. Phys.*, 14:14509–14515, 2012.

[130] M. Xue and J. Jin. Nonconformal FETI-DP methods for large-scale electromagnetic simulation. *Antennas and Propagation, IEEE Transactions on*, PP(99):1, 2012.