Graz University of Technology

Katrin Patrizia Schupfer, BSc

Efficient p-FEM simulations in structural mechanics with high performance computers

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieurin

Master's degree programme: Civil Engineering Sciences and Structural Engineering

submitted to

Graz University of Technology

Supervisor

Univ.-Prof. Dr.-Ing. habil. Thomas-Peter Fries

Institute of Structural Analysis

Graz, August 2021

AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

Graz, August 2021

Katrin Schupfer

Contents

Notation

a	Scalar
$\boldsymbol{v}$	Vector
$\mathbf{A}, \mathbf{n}$	Matrix or tensor
$\|\boldsymbol{v}\|$	Euclidean norm of $\boldsymbol{v}$
$\Delta\boldsymbol{v}$	Laplace operator of a tensor / matrix / vector or scalar field
$\nabla\boldsymbol{v}$	Gradient of a tensor / matrix / vector or scalar field
Ω	Domain
Γ_{D}	Dirichlet boundary
Γ_{N}	Neumann boundary

List of Abbreviations

BC	boundary condition
BFS	breadth-first search
BVP	boundary value problem
CG	conjugate gradient
DOF	degree of freedom
FE	finite element
FEM	finite element method
HPC	high performance computer
K	thousand
M	million
MIMD	multiple instructions, multiple data
MLRB	multilevel recursive bisection
MPI	message passing interface
PDE	partial differential equation
SPMD	single program, multiple data
SDDA	shortest distance decomposition algorithm

Preface

During my studies at TU Graz I developed a special interest in the topic of numerical simulations, especially using the finite element method. Furthermore, I discovered a big interest in computer science and programming and also attended different courses throughout the computer science bachelor. Therefore, the topic of this thesis was a perfect fit for me. In this thesis, I combined my interest in numerical simulations and computational engineering with my ever-growing interest in computer science.

Above all, I want to thank professor Dr.-Ing.habil. T.-P. Fries for the outstanding supervision of my thesis. Prof. Fries always provided me with effective feedback and recommendations during our weekly meetings. I really enjoyed discussing the current state of this project and talking about further decisions.

Furthermore, I want to thank the ZID (Zentraler Informatikdienst) at TU Graz, for enabling me to perform my simulations on a state of the art high-performance computer. Especially I want to thank Mario Lang and Dipl.-Ing. Simon Kainz, who always provided very friendly support.

I also want to thank my whole family, especially my parents Heidrun and Herbert, who enabled me to study and always encouraged me to go my own way. I want to thank my cousin Lisa who took the time to proofread this thesis and my boyfriend David who always had an open ear for me during my studies and was open to discuss also topics outside his field of expertise. Furthermore I want to thank all my fellow students, especially David, Baumi, Berni and my colleagues from the SBZ, who turned my study years into a very special time in my life.

Katrin Schupfer

Abstract

The aim of this master thesis is to implement an efficient parallelization of a given finite element (FE) code and perform p-FEM simulations of applications in structural mechanics with the use of high performance computers. The overall gain is measured in terms of the speedup, indicating the time reduction with respect to increasing the number of cores ($\sim$ processors) employed in the simulation.

In the first part of this thesis, two methods for reducing the degrees of freedom of the underlying system of equations on an element level are described. In the first method, the Dirichlet nodes are removed from the element matrices. In the second method, static condensation of the inner element nodes in higher-order elements is performed. It is shown that this can reduce the size of the system of equations significantly.

In the next part, the decomposition of the domain among the processes is discussed. Therefore, two different methods – the "shortest distance decomposition algorithm" and the "multilevel recursive bisection" implemented in the open source software library METIS – are compared in order to find an efficient and high-quality algorithm for the decomposition.

In the third part of the thesis, the core of the simulation – the parallel solution of the systems of equations resulting from the FEM – is described. The primary focus lies on the parallel conjugate gradient solver alongside a description of the implemented communication scheme.

In the last part, different simulations in structural mechanics are performed on a high performance computer. In addition to the number of processors also the number and order of the elements are varied to obtain a good overview of the achieved speedup. It is shown that with the use of an efficient parallelization scheme, a remarkable speedup can be achieved, especially for higher order simulations.

Kurzfassung

Das Ziel dieser Masterarbeit ist die effiziente Parallelisierung eines bestehenden Finite Elemente Codes, sowie die Durchführung von p-FEM Simulationen im Bereich der Strukturmechanik auf Hochleistungsrechnern. Des Weiteren werden die durchgeführten Simulationen zeitlich analysiert und der produzierte Speedup berechnet, also die zeitliche Einsparung den eingesetzten Rechenkernen ($\sim$ Prozessoren) gegenübergestellt.

Im ersten Teil dieser Arbeit werden zwei Methoden zur Reduzierung der Freiheitsgrade des zugrundeliegenden Gleichungssystems präsentiert. Die erste Methode stellt die Entfernung der Dirichlet Knoten aus dem Gleichungssystem dar. In der zweiten Methode werden die inneren Elementknoten statisch kondensiert. Es wird gezeigt, dass insbesondere die statische Kondensation die Größe des Gleichungssystems für Elemente höherer Ordnung erheblich reduziert.

Im folgenden Teil wird die Zerlegung und Aufteilung des Gebietes auf die verwendeten Prozesse beschrieben. Dafür werden zwei Gebietszerlegungsalgorithmen – der „Shortest Distance Decomposition Algorithm“ und die „Multilevel Recursive Bisection“, welche in der Softwarebibliothek METIS implementiert ist – näher betrachtet.

Im dritten Teil wird der Kern der Simulation – das parallele Lösen des Gleichungssystems, resultierend aus der Anwendung der FEM – beschrieben. Der Fokus liegt hierbei auf dem parallelen Algorithmus konjugierter Gradienten, sowie dem implementierten Kommunikationsschema.

Im letzten Teil der Arbeit werden verschiedene Simulationen aus dem Bereich der Strukturmechanik auf einem Hochleistungsrechner durchgeführt und analysiert. Zusätzlich zur Anzahl der Prozessoren wird auch die Elementordnung variiert, um einen guten Überblick über den erreichten Speedup zu erlangen. Dabei zeigt sich, dass unter Verwendung eines effizienten Parallelisierungsschemas, vor allem für Elemente höherer Ordnung, ein bemerkenswerter Speedup erzielt werden kann.

1 Introduction

The finite element method (FEM) is a very powerful simulation method that is able to approximate many boundary value problems (BVPs)[1] representing physical models. Therefore, it is used in numerous fields of studies such as physics, structural engineering and many more. In order to achieve a high level of accuracy, a large computational effort is often necessary. To meet the needs of computational resources, high performance computing has become increasingly popular over the past few decades [1]. However, with the constant increase of computational capabilities, also the demands regarding the simulation rise. It is often no longer sufficient to perform a serial simulation, i.e., to use only one core[2].

Very soon after electronic computing began in 1942 with the ENIAC[3], the race for increased computing power started [3]. Especially in scientific research the need for faster computers developed. The idea to decrease the computing time by using several processors and share the work load between them was born. Therefore, multiple processors were connected via a network, creating the first computing clusters. Modern high performance computers feature thousands (even up to millions) of cores. The Fugaku, which is currently the worlds fastest high performance computer is located in Japan and features an incredible number of 7.6 million cores [4]. In an optimal setup, using n cores would reduce the computation time by a factor of $1/n$. However, as the involved algorithms do not perfectly distribute the work load, the gain is often much less then the optimal $1/n$ mentioned before. The true gain needs to be measured and is called the speedup.

In this thesis, an efficient parallelization of simulations in structural mechanics is implemented and investigated. In order to achieve this, two well known BVPs are considered, namely the **Poisson equation** and the **Navier-Cauchy equations**. The former is one of the simplest and most famous partial differential equations (PDEs) that is applicable to several physical problems, such as electrostatic potential or heat flows. The latter describes an arbitrary linear elastic material[4] with forces acting on it. Due to small deformations, linear elastic behavior is relevant in most structural engineering applications. Therefore, the Navier-Cauchy equations can be used to compute the stresses and strains for numerous applications in civil engineering (e.g., buildings, dams, ect.).

In this chapter a short introduction to some important basics for the efficient parallelization of a finite element simulation is provided.

[1]A BVP is a (set of) partial differential equations (PDEs) in combination with additional constraints, known as boundary conditions (BCs).

[2]Each processor (CPU) may contain multiple cores.

[3]The Electronic Numerical Integrator and Computer (ENIAC) was the first programmable general-purpose electronic digital computer [2].

[4]In a linear elastic material behavior the strain is proportional to the stress and no plastic deformations occur.

1.1 Short overview on the Finite Element Method

A basic understanding of the finite element method is expected herein. However, a short introduction to the topics which are most important for this work is given in this section[5].

In order to investigate and predict the behavior of real world phenomena, such as the behavior of static structures or dynamic objects, different physical models exist (or need to be derived). These models are often given in the form of boundary value problems (BVPs). In most cases an exact solution of these BVPs is impossible. The finite element method (FEM) is a numerical simulation method that *approximates* the solution of a considered BVP with a certain accuracy. It is a very powerful method that is used in numerous fields of studies. Fig. 1 visualizes the described context.

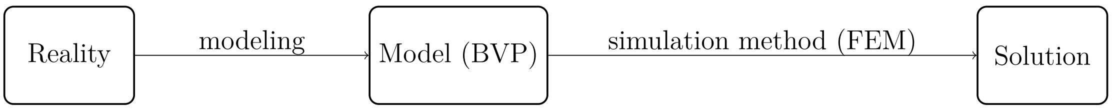


Fig. 1: Reality - Model - Solution.

Every FE simulation can be split into the following steps:

1. Preprocessing
    - Choice of the model including material parameters.
    - Decomposition of the considered domain into a mesh of non overlapping elements.
2. Processing
    - Assembly of the system of equations.
    - Application of the boundary conditions (BCs).
    - Solving the system of equations to obtain the solution at the nodes of a mesh.
3. Postprocessing
    - Obtaining the solution everywhere in the domain by numerical interpolation.
    - Visualization and interpretation of the results.

As stated above, it is necessary to decompose the domain into a mesh which is a discretization of the domain of interest. This mesh is composed by nodes and elements of some uniform shape (e.g., triangular or quadrilateral). Figure 2 shows an example finite element mesh in two dimensions (2D).

[5]This section is based on the lecture notes of the lecture FEM by Prof. Dr.-Ing. T.-P. Fries at TU Graz 2018/19 [5].

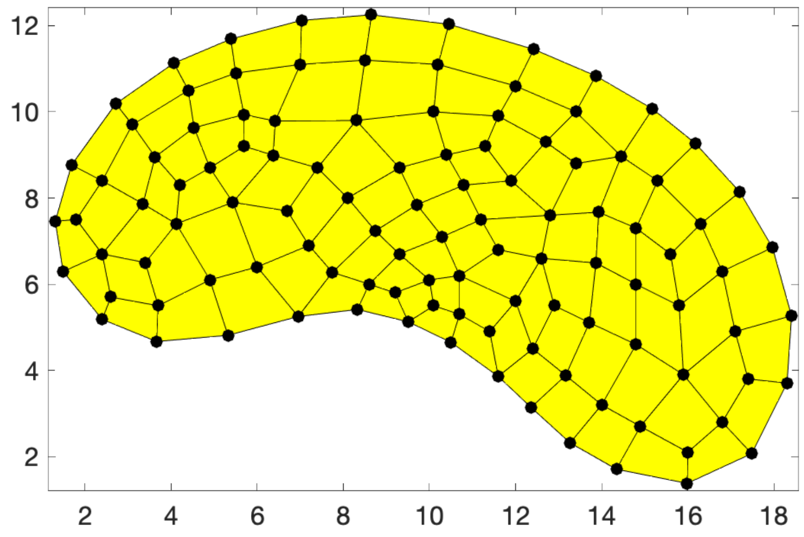


Fig. 2: An example of a mesh with linear, quadrilateral elements.

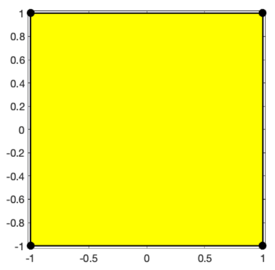
The elements are defined in a given reference space and are projected into the real domain in terms of a mapping

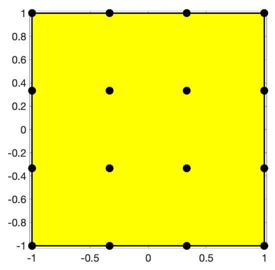
$$\boldsymbol{x}(\boldsymbol{r}) = \sum_{i=1}^{n} N_i(\boldsymbol{r}) \cdot \boldsymbol{x}_i. \tag{1.1}$$

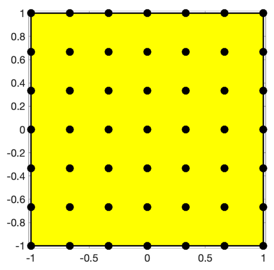
Here $\boldsymbol{r}$ corresponds to the coordinates of a point in the *reference* element, whereas $\boldsymbol{x}$ corresponds to the coordinates of the same point in the real (physical) domain. For every node i of the *reference* element, a continuous (typically) polynomial shape-function $N_i(\boldsymbol{r})$ is defined. The involved terms of these polynomials are related to the order of the element. For a given number of nodes in the element, the order may be given as an intrinsic property of that element. More nodes relate to a higher order. For all simulations considered in this thesis, the shape functions fulfil the Kronecker-delta property:

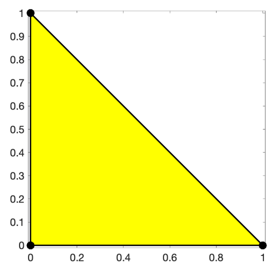
$$N_i(\boldsymbol{r}_j) = \begin{cases} 1 & j = i \\ 0 & j \neq i \end{cases} \qquad \text{with } \boldsymbol{r}_j \text{ being the coordinates of node } j. \tag{1.2}$$

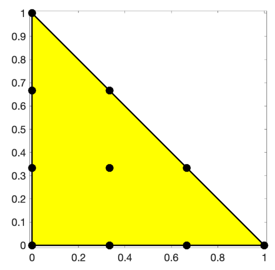
After the processing step, the solution is obtained at all nodes. The higher the number of nodes, the more accurate is the obtained (approximate) solution in the overall domain. In order to achieve a more accurate solution, in general two possibilities exist: On the one hand, the number of elements can be increased, which is commonly known as h-FEM. On the other hand, elements with a higher order can be used, which is referred to as p-FEM. The number of nodes depends on the element type (e.g., quadrilateral or triangular in 2D) and element order p. The following two figures (Fig. 3 and Fig. 4) show some example elements in 2D and 3D, respectively.


(a) quadrilateral - linear


(b) quadrilateral - cubic


(c) quadrilateral - 6th order


(d) triangular - linear


(e) triangular - cubic

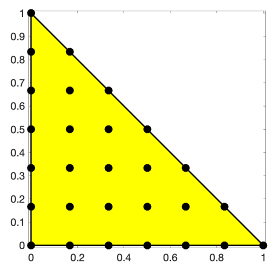
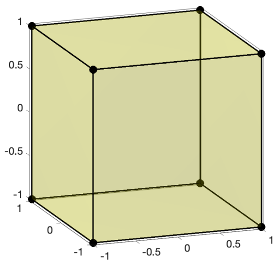
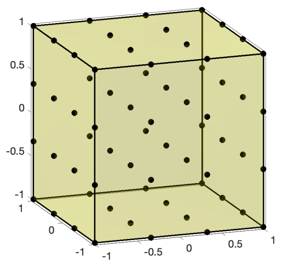
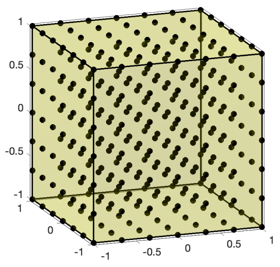
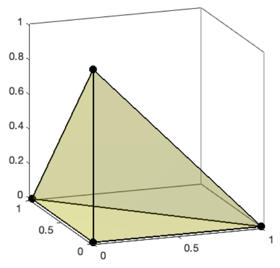

(f) triangular - 6th order

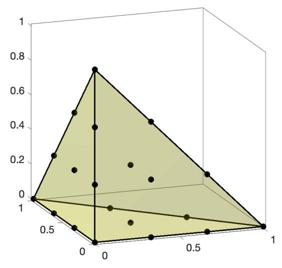
Fig. 3: 2D FE reference elements with different types and orders.


(a) hexahedral - linear


(b) hexahedral - cubic


(c) hexahedral - 6th order


(d) tetrahedral - linear


(e) tetrahedral - cubic

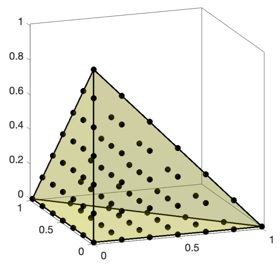

(f) tetrahedral - 6th order

Fig. 4: 3D FE reference elements with different types and orders.

In order to approximate a BVP, the given partial differential equation (PDE) has to be restated in a weak form. This is done by multiplying the PDE with a test function, integrating over the domain and applying the divergence theorem. For illustrative purposes this process is shown for the Poisson BVP.

The strong from of the Poisson BVP sounds: Find $u(\boldsymbol{x})$ such that

$$\begin{aligned} -\Delta u(\boldsymbol{x}) &= q(\boldsymbol{x}) \quad \text{for}\, \boldsymbol{x} \in \Omega, \\ u(\boldsymbol{x}) &= \hat{u}(\boldsymbol{x}) \quad \text{for}\, \boldsymbol{x} \in \Gamma_{\mathrm{D}}, \\ \nabla u(\boldsymbol{x}) \cdot \boldsymbol{n} &= \hat{t}(\boldsymbol{x}) \quad \text{for}\, \boldsymbol{x} \in \Gamma_{\mathrm{N}}, \end{aligned} \tag{1.3}$$

is fulfilled. Herein Γ_{D} is the Dirichlet boundary and Γ_{N} the Neumann boundary.

Applying a test function $w(\boldsymbol{x})$ with suitable continuity and integrating over the domain yields the weak form: Find $u(\boldsymbol{x})$ such that for every $w(\boldsymbol{x})$

$$\int_\Omega -w(\boldsymbol{x}) \cdot \Delta u(\boldsymbol{x})\, d\Omega = \int_\Omega w(\boldsymbol{x}) \cdot q(\boldsymbol{x})\, d\Omega, \tag{1.4}$$

holds. After applying the divergence theorem the equation can be restated as:

$$\int_\Omega \nabla w(\boldsymbol{x}) \cdot \nabla u(\boldsymbol{x})\, d\Omega = \int_\Omega w(\boldsymbol{x}) \cdot q(\boldsymbol{x})\, d\Omega + \int_\Gamma w(\boldsymbol{x})\, \hat{t}(\boldsymbol{x})\, d\Gamma. \tag{1.5}$$

Furthermore, it is necessary to restrict the function space of the solution. Therefore, the Ansatz

$$u^h(\boldsymbol{x}) = \sum N_i(\boldsymbol{x}) \cdot u_i = \boldsymbol{N}^T \boldsymbol{u} \tag{1.6}$$

is made. The weak form is then discretized as

$$\begin{aligned} \int_\Omega \nabla w^h(\boldsymbol{x}) \cdot \nabla u^h(\boldsymbol{x})\, d\Omega &= \int_\Omega w^h(\boldsymbol{x}) \cdot q(\boldsymbol{x})\, d\Omega + \int_\Gamma w^h(\boldsymbol{x})\, \hat{t}(\boldsymbol{x})\, d\Gamma, \\ \underbrace{\int_\Omega \nabla \boldsymbol{N} \cdot \nabla \boldsymbol{N}^T\, d\Omega}_{\mathbf{K}}\; \boldsymbol{u} &= \underbrace{\int_\Omega \boldsymbol{N}\, q(\boldsymbol{x})\, d\Omega + \int_\Gamma \boldsymbol{N}\, \hat{t}(\boldsymbol{x})\, d\Gamma}_{\boldsymbol{f}}, \end{aligned} \tag{1.7}$$

leading to a system of equations $\mathbf{K} \cdot \boldsymbol{u} = \boldsymbol{f}$. Hence, $\boldsymbol{u}$ is a vector of the solution at the nodes. In the context of linear elasticity, $\mathbf{K}$ is called the stiffness matrix.

It should be considered that $\hat{t}(\boldsymbol{x})$ is unknown on the Dirichlet boundary Γ_{D}. However, when applying the Dirichlet BCs (explained in section 2.1) the entries of the right-hand side at the Dirichlet nodes are reduced or overwritten anyway.

The integral over the domain is equal to a sum of integrals over the elements. Therefore, each element contributes to the system of equations in the form of an element matrix. This element matrix is then assembled into the system matrix $\mathbf{K}$. Assuming a very simple 2D domain, the assembly of the system of equations is shown below.

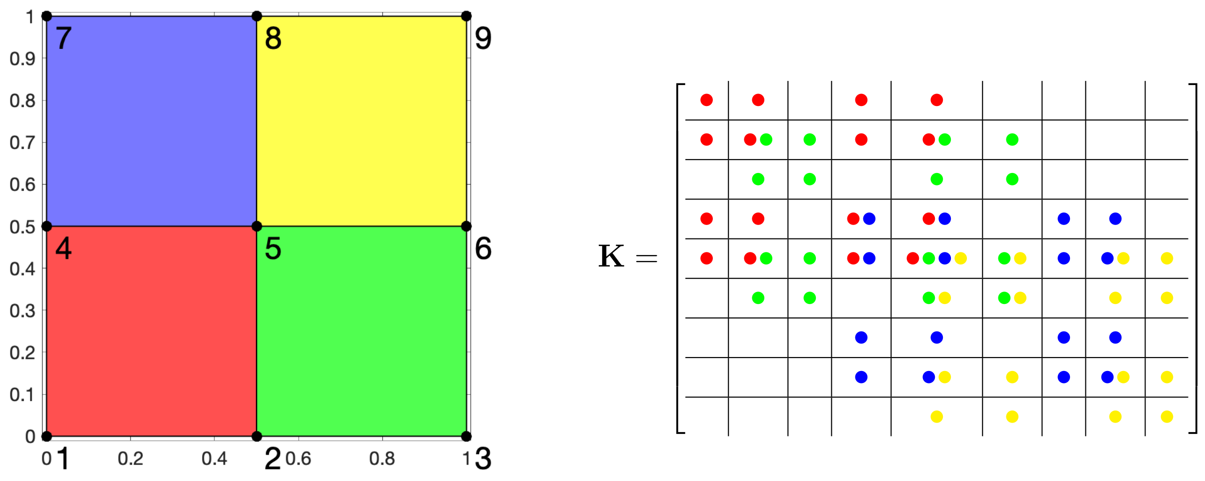


Fig. 5: Assembly of the system of equations of a simple 2D mesh.

Each coloured dot represents the contribution of the element of the same color to the given entry in the matrix. Since the reference element has 4 nodes, every element contributes a 4×4 element matrix to the 9×9 system matrix $\mathbf{K}$.

Considering equation 1.7, the system matrix $\mathbf{K}$ – as well as the element matrices – are symmetric and also positive definite, that is:

$$\boldsymbol{x}^T \cdot \mathbf{K} \cdot \boldsymbol{x} > 0 \qquad \forall \boldsymbol{x} \in \mathbb{R}^n \setminus \mathbf{0}. \tag{1.8}$$

This property is very important in the context of algorithms to solve the system of equations as discussed in section 1.2.

With a growing number of elements, $\mathbf{K}$ becomes increasingly sparse due to the fact that each element is only connected to a small number of other elements. Implementationally, in most finite element codes, the system matrix is stored in the (i,j,v)-format where only the non-zero entries are stored, in order to reduce the storage requirements compared to a full matrix array [6].

In BVPs, the boundary of each domain can be split into two types. The first is the Dirichlet boundary Γ_D where the values of the solution are prescribed (e.g., support of a building). The second type is the Neumann boundary Γ_N (e.g., forces acting on a structure). The Neumann BCs influence the right-hand side of the system of equations and are typically considered when assembling the system of equations. In contrast, the Dirichlet BCs prescribe some of the nodes in the mesh which directly influences (the structure of) the system of equations.

1.2 Overview on iterative solvers

In most FE simulations, the solution of the system of equations is the most time-consuming part of the computation. For large-scale FE simulations a direct solution is unfeasible, since it is too expensive with regard to solution time as well as memory requirements. Therefore, the solution is often obtained in an iterative manner. These iterative methods produce an approximation of the solution after a finite number of steps with user-defined accuracy.

Let us consider different iterative solvers for the linear system of equations

$$\mathbf{K} \cdot \boldsymbol{u} = \boldsymbol{f}. \tag{1.9}$$

The choice of the "best" solver depends on the properties of the matrix $\mathbf{K}$. Furthermore, a preconditioning matrix $\mathbf{M}$ can be used in order to reduce the number of required iterations. Then, instead of solving equation 1.9 the system

$$(\mathbf{M}^{-1} \cdot \mathbf{K}) \cdot \boldsymbol{u} = \mathbf{M}^{-1} \cdot \boldsymbol{f} \tag{1.10}$$

is solved. The closer $\mathbf{M}$ is to the inverse of the system matrix $\mathbf{K}$, the higher the reduction of iterations. However, the computation of the inverse is very expensive and must be avoided. Therefore, different methods to *approximate* $\mathbf{K}^{-1}$ exist. In the following the most popular iterative solvers are discussed [7] and the best method for the FE simulations considered herein is chosen.

1. Conjugate gradient (CG) algorithm
   - $\mathbf{K}$ must be symmetric and positive definite.
   - Is known to be very efficient.
   - Uses a set of $\mathbf{K}$-orthogonal vectors as search directions.
2. Minimum residual (MINRES) algorithm
   - $\mathbf{K}$ must be symmetric.
   - Minimizes the Euclidean norm of the residual.
3. Biconjugate gradient (BICG) algorithm [8]
   - $\mathbf{K}$ must be square but not necessarily symmetric.
   - Generalizes CG to non-symmetric problems.
   - Solves two sets of conjugated gradients.
   - Is computationally cheap but unstable (irregular convergence).
   - Is the basis for many other, more effective iterative methods.
4. Biconjugate gradient stabilized (BICGSTAB) algorithm [9]
   - $\mathbf{K}$ must be square but not necessarily symmetric.
   - Stabilizes the BICG algorithm with the use of GMRES steps to reduce irregular convergence.

5. Generalized minimum residual (GMRES) algorithm
    - $\mathbf{K}$ must be square but not necessarily symmetric.
    - Generalizes MINRES to non-symmetric problems.
    - Computes the solution as sequence of orthonormal vectors.
    - Storage requirement grows linearly with the number of iterations, since GMRES stores all previous vectors. (After a chosen number of steps it restarts with the current solution vector as initial guess to remedy this problem.)

In order to assure that the best algorithm is chosen for the considered problems, some preliminary studies were conducted in Matlab. Therefore, the listed algorithms and the following preconditioners were considered:

1. No preconditioner.
2. Jacobi preconditioner.
3. Incomplete LU-factorization (iLU).
4. Incomplete Cholesky-factorization (iChol).

Fig. 6 and Fig. 7 show the results of the pre-studies on an elastic continuum. The system of equations has approximately 1.8 million DOFs for the simulation with linear, 1.1 million DOFs for the the simulation with cubic and 0.3 million DOFs for the simulation with 6th order elements. The pre-studies on the simulations of the Poisson equation show similar results.

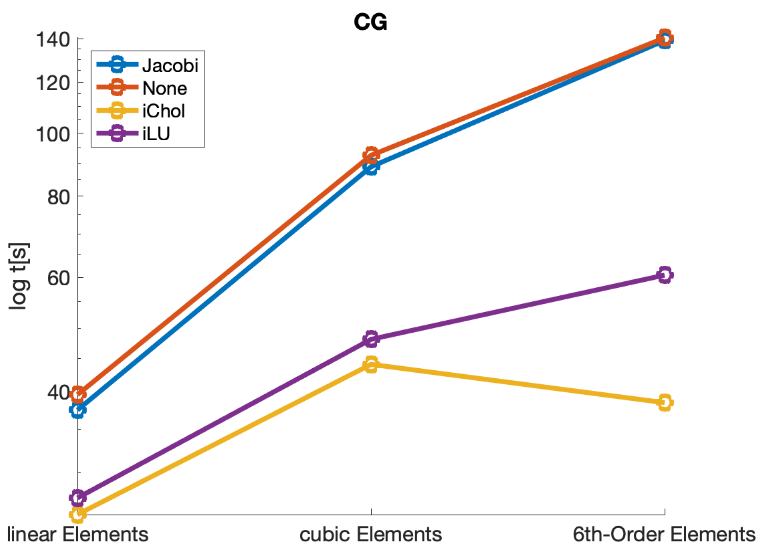
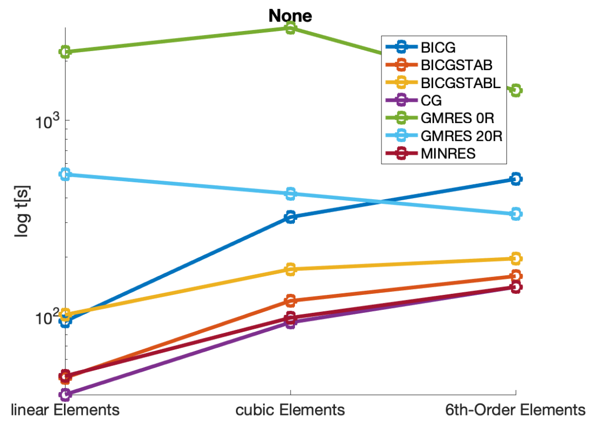
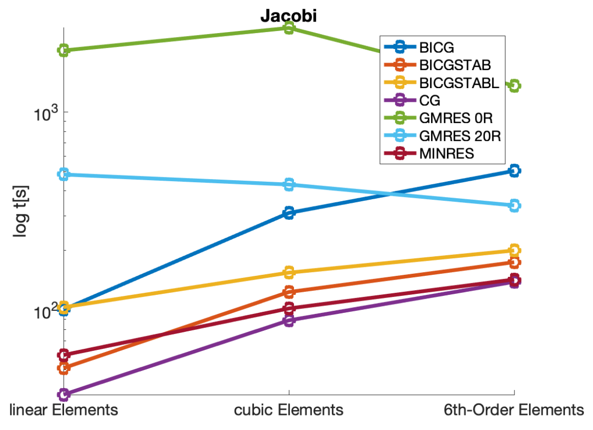


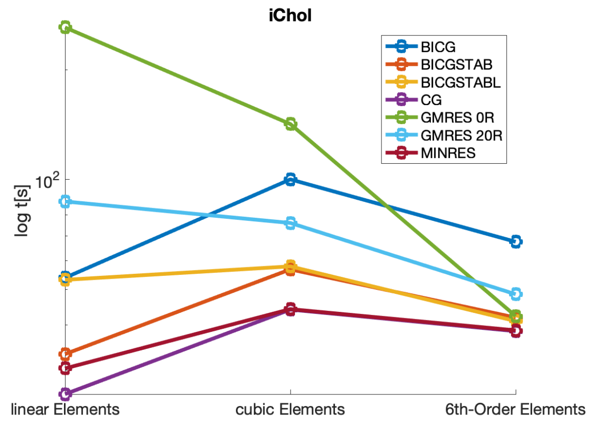
Fig. 6: Comparison of different preconditioners with the CG solver in a FEM simulation of an elastic continuum.



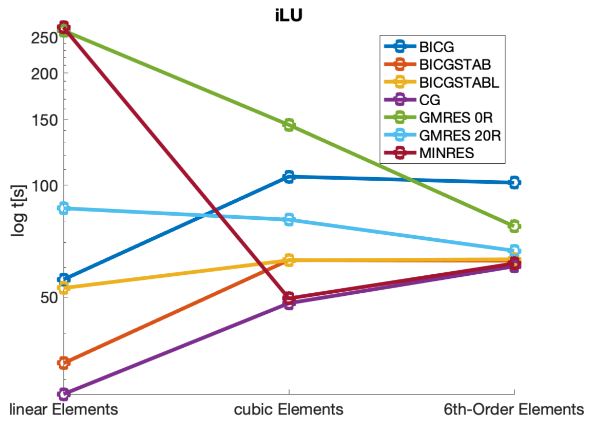
(a) no preconditioner



(b) Jacobi preconditioner



(c) iLU preconditioner



(d) iChol preconditioner

Fig. 7: Comparison of iterative solvers and preconditioners in a FEM simulation of an elastic continuum.

These studies show that the conjugate gradient (CG) algorithm is the most effective solver for the considered simulations, regardless of the chosen preconditioner. Therefore, it is used as solving algorithm for all further computations and is described in some more detail below. The incomplete Cholesky factorization was found to be the best preconditioning method for the applications considered here. Nevertheless, the Jacobi preconditioner was chosen for all further computations, since it is highly parallelizable and finding the best preconditioner in a parallel environment goes beyond the scope of this thesis.

1.2.1 Preconditioned conjugate gradient algorithm

The CG algorithm is a Krylov subspace method. It constructs its search directions from its residuals in a way that each search direction is **K**-orthogonal to all previous search directions and residuals [10]. The preconditioned conjugate gradient (CG) algorithm is considered to be the most effective solver [7]. However, the requirements on the matrix $\mathbf{K}$ are rather high (symmetric and positive definite), but this is fortunately fulfilled for the applications considered here. It has a time complexity of

$$\mathcal{O}(\sqrt{\mathcal{K}(\mathbf{A})}), \tag{1.11}$$

with

$$\begin{aligned} \mathcal{K}(\mathbf{A}) &= \|\mathbf{A}\|\|\mathbf{A}^{-1}\|, \\ \mathbf{A} &= \mathbf{M}^{-1}\mathbf{K}, \end{aligned}$$

$\mathcal{K}(\mathbf{A})$ being the condition number of the matrix $\mathbf{A}$ and $\mathbf{A}$ being the preconditioned system matrix [11].

Given a symmetric and positive definite $n \times n$ matrix, the CG algorithm returns the exact solution after at most n iterations. However, a result with a sufficiently small error is returned with much fewer iterations. As stopping criterion the squared norm of the current residual is compared to the squared norm of the initial residual. Also in terms of space complexity CG is very efficient since it only needs to store a limited number of vectors.

Algorithm 1.1: Preconditioned conjugate gradient algorithm [12, p. 37]

```
Input: system matrix $\mathbf{K}$, system rhs $\boldsymbol{f}$, preconditioning matrix $\mathbf{M}$, accuracy $\varepsilon$
$\boldsymbol{r}^0 = \mathbf{K} \cdot \boldsymbol{u}_0 - \boldsymbol{f}$,  $\boldsymbol{z}^0 = \mathbf{M}^{-1} \cdot \boldsymbol{r}^0$,  $\boldsymbol{p}^0 = \boldsymbol{z}^0$,  $\varrho_0 = (\boldsymbol{z}^0, \boldsymbol{r}^0)$
if $\varrho_0 < \varepsilon^2$ then
    return $\boldsymbol{u}^0$
end
for $i=0,1...,n-2$ do
    $\boldsymbol{s}^k = \mathbf{K} \cdot \boldsymbol{p}^k$,  $\sigma_k = (\boldsymbol{s}^k, \boldsymbol{p}^k)$,  $\alpha_k = \frac{\varrho_k}{\sigma_k}$
    $\boldsymbol{u}^{k+1} = \boldsymbol{u}^k - \alpha_k \boldsymbol{p}^k$
    $\boldsymbol{r}^{k+1} = \boldsymbol{r}^k - \alpha_k \boldsymbol{s}^k$
    $\boldsymbol{z}^{k+1} = \mathbf{M}^{-1} \cdot \boldsymbol{r}^{k+1}$
    $\varrho_{k+1} = (\boldsymbol{z}^{k+1}, \ \boldsymbol{r}^{k+1})$
    if $\varrho_{k+1} < \varepsilon^2 \varrho_0$ then
        break
    end
    $\boldsymbol{p}^{k+1} = \boldsymbol{z}^{k+1} + \beta_k \boldsymbol{p}^k$,  $\beta_k = \frac{\varrho_{k+1}}{\varrho_k}$
end
return $\boldsymbol{u}^{k+1}$
```

1.3 Introduction to high performance computing

Over the past few decades computer architectures have become increasingly parallel. Nowadays all modern computers have multiple cores and sometimes even multiple processors.

Therefore, the use of serial programs does not take advantage of the potential computing power and the need for parallel (simulation) codes is increasing, even more so as the number of cores in modern HPCs is continuously increasing. Over the past two to three decades the performance of HPCs (often measured in floating point operations per second (flops)) has multiplied by 5×10^5 with no end in sight [13]. The performance of todays fastest computers can even be measured in the scale of Exaflops (10^{18} flops). High performance computers (HPCs) are not only used in science but also have become increasingly popular in industry (e.g., artificial intelligence, medical imaging and diagnosis)[6]. The following section gives a brief introduction to parallel and high performance computing.

Generally, there are different ways to classify parallel computers. One of the most famous classifications of parallel computer architectures is Flynn's taxonomy [14]. It distinguishes between single and multiple concurrent instruction[7] and data streams[8]:

- **SISD - single instruction single data**: A serial computer with only one instruction and one data stream (e.g., old computers, minicomputers).
- **SIMD - single instruction multiple data**: A parallel computer where all processing units perform the same task on different data elements (e.g., graphic processing units (GPUs)).
- **MISD - multiple instruction single data**: A parallel computer where all processing units perform different tasks on the same data elements (hardly any example exists).
- **MIMD - multiple instruction multiple data**: A parallel computer where all processing units perform different tasks on different data elements (e.g., most current HPCs).

Furthermore, parallel computers can be distinguished on the basis of their memory architecture [13]:

- **Shared memory**: All processors operate independently from each other but share the same global address space (memory).
- **Distributed memory**: Each processor has its own local memory and therefore can operate fully independent from other processors. The processors are connected via a network where data can be exchanged between processors.
- **Hybrid Distributed-Shared memory**: A combination of distributed and shared memory. The different nodes have their own local memory. Each node however, consists of multiple processors sharing the same memory.

The advantage of a shared memory architecture is that data sharing between processes is easy and fast. However, shared memory computers are not scalable, because the traffic on the shared memory part increases with the number of involved CPUs [13]. In contrast, distributed memory computers have the potential to scale up to a very large numbers of cores.

[6]Nowadays almost 20% of all HPCs are used in industry [13].
[7]An instruction stream is the sequence of instructions sent to the control unit of the CPU.
[8]A data stream is the sequence of data elements that are required for the execution of the instructions.

However, the communication between the processors is still considered to be a bottleneck. Hybrid Distributed-Shared memory combines the strengths of both architectures and is therefore employed by todays fastest supercomputers.

Fig. 8 visualizes the architecture of a HPC (distributed memory). The different nodes are connected via a network for data communication. Every node itself is a parallel computer with multiple cores and sometimes also multiple CPUs. The memory of a single node is shared among all cores and CPUs on that node. Therefore, if a node has more than one CPU, the architecture of the HPC is considered to be hybrid distributed-shared, since each node then employs a shared architecture.

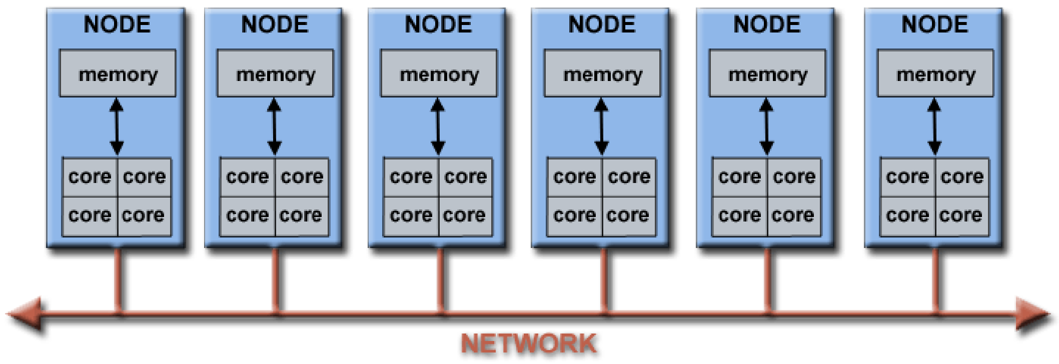


Fig. 8: Architecture of a HPC[13].

In contrast to memory architectures also in the context of programming models it can be distinguished between the shared memory programming model and the distributed memory programming model [13]. The latter is also known as Message Passing model, since messages are passed between distinct tasks, which is called "communication". The most often used programming standard for this model is the message passing interface (MPI), which is further described in section 4.2.1. While it can significantly increase the speed of a program, it is not strictly necessary that the programming model and the architecture of the system are of the same type. In other words, it is well possible to execute MPI programs on shared- as well as on distributed memory computers.

For all simulations in this thesis the aCluster from the TU-Graz was used and the implemented code employs a distributed memory programming model. The technical properties of the aCluster are shown below [15],[16]:

- 32 compute nodes with 64 cores (2304 cores, AMD EPYC 7452)
- 1024GB RAM per node (16GB/Core)
- 14 of the compute nodes have a GPU (NVIDIA Tesla T4 16GB)[9]
- 200GBit/s Infiniband Interconnect

[9]The GPUs were not considered/used in this thesis.

1.4 Parallelization strategy

Finding a good parallelization strategy is of major importance in order to achieve a maximum speedup of the FE simulation. Therefore, one may choose from different alternatives. In the following a few approaches are discussed and the chosen approach is described.

The most basic parallelization approach is to assemble the whole system matrix in a serial manner and only parallelize the *solution* of the system of equations. That is, after the serial assembly, the system matrix and right-hand side are split into several parts, where each process is assigned one of these parts. However, in most simulations setting up the full system of equations in a serial manner should be avoided for two main reasons. First, the system matrix of a large-scale FE simulation can become prohibitively large, especially in the context of three dimensional higher-order elements. This amount of memory is often not available in a single process. Secondly, the assembly of the system matrix is an inherently parallel process, because the element matrices can be built independently from one another. Consequently, the assembly part of the simulation can be parallelized very efficiently or, in other words, trivially.

A different approach is the so-called element-by-element method, where only the element matrices are generated and directly used in the parallel solver. In this method each process is responsible for one element [17]. However, the number of available processors is almost always smaller than the number of elements in the mesh. Therefore, it becomes necessary to assign several processes to one processor. In the given MPI context (Sec. 4.2.1) this is also known as oversubscribing and is often not suggested, since it might lead to a decrease in performance [18].

Therefore, in the chosen strategy each processor is assigned exactly one process. The domain is divided among the processes, such that each process is responsible for a coherent part of the domain. Each process then assembles a subsystem of equations which is used in the parallel solving. The serial parts are performed by all processes in order to avoid unnecessary communication. Fig. 9 visualizes the concept of the chosen strategy.

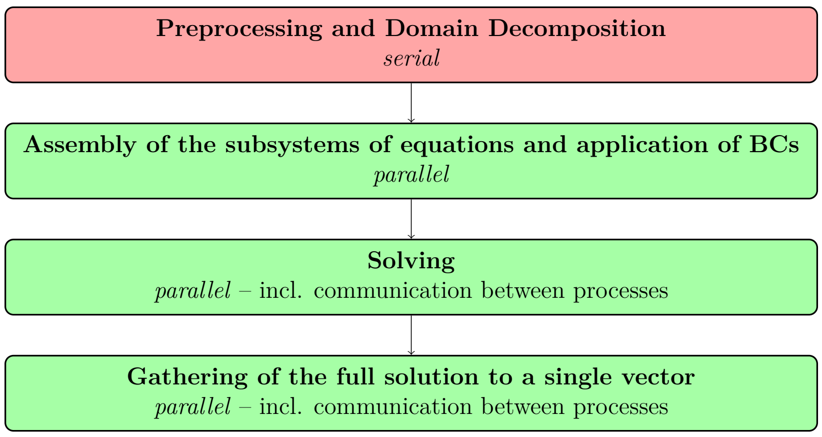

Preprocessing and Domain Decomposition
serial
Assembly of the subsystems of equations and application of BCs
parallel
Solving
parallel – incl. communication between processes
Gathering of the full solution to a single vector
parallel – incl. communication between processes

Fig. 9: Parallelization concept – serial steps are marked in red and parallel steps are marked in green.

Instead of assembling the sub-matrices, the simulation could also be performed fully matrix-free [19]. Since only the matrix vector product $\mathbf{K} \cdot \boldsymbol{p}_k$ is needed by the CG solver, instead of computing and storing the sub-system matrices, the matrix vector products can also be computed directly during the solving process [19]. Therefore, the number of memory accesses can be reduced drastically. In return the amount of computational effort increases, since the assembly needs to be recomputed in every step. Matrix-free methods are very popular for non-linear and time-dependent problems, since these problems need a recomputation of the assembly anyway. Matrix-free methods are considered to be very fast also for linear systems of equations of higher order elements [20]. However, the construction of matrix-free preconditioners tends to be rather complicated. In this thesis, the aim was to find a rather general algorithm that works well for linear as well as higher order simulations of different problems, which is why the simulations were not performed matrix-free.

1.5 Outline of the Thesis

In order to implement an efficient parallelization of the provided FE code, different parts need to be considered. Therefore, this thesis is structured as as follows:

To decrease the total computational effort, the reduction of the DOFs of the underlying system of equations is discussed in chapter 2. Therefore, not only the prescribed Dirichlet nodes are removed, but also the inner element nodes in higher-order elements are reduced. Since the full system matrix is never assembled, these reductions are performed on an element level. Furthermore, in a parallel simulation the domain is split among all participating processes. Therefore, an efficient domain decomposition algorithm is discussed in chapter 3. The "shortest distance decomposition algorithm" is compared to the "multilevel recursive bisection" which is implemented in the open source software library METIS. In chapter 4, the parallel solution of the system of equations is discussed. A parallel version of the CG solver as well as the implemented communication scheme are presented. Finally efficiency studies of the considered simulations are performed on a high performance computer and results are presented in chapter 5. The achieved speedup as well as the amount of necessary communication are evaluated. This thesis ends in chapter 6 with a summary and conclusions.

2 Reduction of the degrees of freedom on an element level

In most finite element simulations, the solution of the resulting system of equations is the major performance bottleneck. In addition to choosing an appropriate solver as described in section 1.2, one of the most effective ways to minimize the solving time is to reduce the degrees of freedom (DOFs) of the system of equations. However, the mathematical properties of the matrix (e.g., symmetry, positive definiteness) should still be maintained. Performing the reduction on the element matrices in a parallel manner is highly efficient. In this chapter, two ways of reducing the degrees of freedom on the basis of the element matrices are discussed.

2.1 Removal of the DOFs on the Dirichlet boundary

As explained in section 1.1, the values of the solution are prescribed on the Dirichlet boundary Γ_D. Fig. 10 shows an example mesh with the Dirichlet boundary Γ_D marked in red; the values at the red nodes are prescribed.

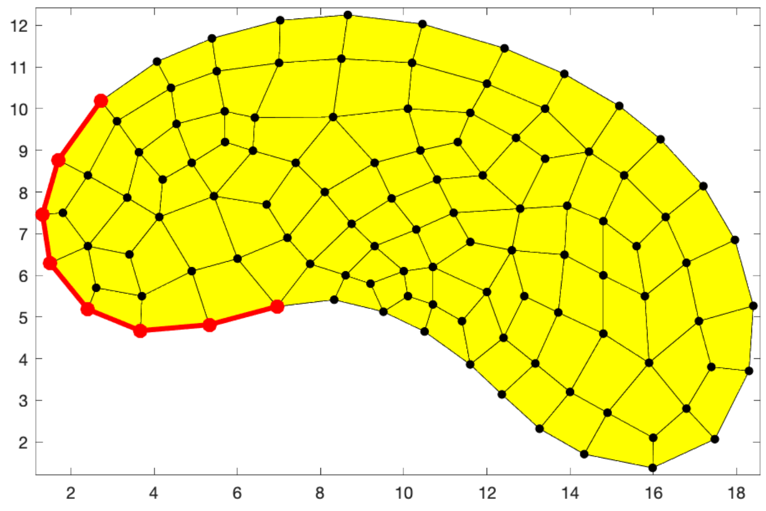


Fig. 10: A mesh of some schematic domain, the Dirichlet boundary is marked in red.

Since the system of equations is solved for $\boldsymbol{u}$, the prescribed values need to be somehow considered in the system of equations. This is most commonly done using one of two different methods. The first is to set the matrix rows of the Dirichlet DOFs to zero except for a one on the main diagonal (assuming shape functions with Kronecker-delta property are used). The second method is to reduce the Dirichlet DOFs as shown below [21].

The system of equations $\mathbf{K} \cdot \boldsymbol{u} = \boldsymbol{f}$ can be split as follows:

$$\begin{bmatrix} \mathbf{K}_{\Omega\Omega} & \mathbf{K}_{\Omega\Gamma_\mathrm{D}} \\ \mathbf{K}_{\Gamma_\mathrm{D}\Omega} & \mathbf{K}_{\Gamma_\mathrm{D}\Gamma_\mathrm{D}} \end{bmatrix} \begin{bmatrix} \boldsymbol{u}_\Omega \\ \boldsymbol{u}_{\Gamma_\mathrm{D}} \end{bmatrix} = \begin{bmatrix} \boldsymbol{f}_\Omega \\ \boldsymbol{f}_{\Gamma_\mathrm{D}} \end{bmatrix}, \tag{2.1}$$

where $\boldsymbol{u}_{\Gamma_\mathrm{D}}$ are the DOFs associated with nodes on the Dirichlet boundary Γ_D and $\boldsymbol{u}_\Omega$ are the remaining DOFs in the domain Ω (including the Neumann boundary Γ_N). Then all Dirichlet rows can be omitted and the remaining system of equations can be rewritten by shifting the Dirichlet columns to the right-hand side,

$$\mathbf{K}_{\Omega\Omega} \cdot \boldsymbol{u}_\Omega = \underbrace{\boldsymbol{f}_\Omega - \mathbf{K}_{\Omega\Gamma_\mathrm{D}} \cdot \boldsymbol{u}_{\Gamma_\mathrm{D}}}_{\boldsymbol{f}_\mathrm{red}}. \tag{2.2}$$

This method is preferred over the first method since it not only reduces the number of DOFs of the system of equations to be solved, but also maintains useful properties of the matrix, such as being symmetric and positive definite.

As shown below, this reduction can also be performed on the basis of the element matrices. Assuming a sparse matrix $\mathbf{K}^{(e)}$ and a vector $\boldsymbol{f}^{(e)}$ which are of the same size as the global system matrix and right-hand side but only have the entries corresponding to the element e, the system of equations can be rewritten as:

$$\left(\sum_{e=1}^{E} \mathbf{K}^{(e)}\right) \cdot \boldsymbol{u} = \sum_{e=1}^{E} \boldsymbol{f}^{(e)}. \tag{2.3}$$

Inserting this into the split system of equations 2.1 yields:

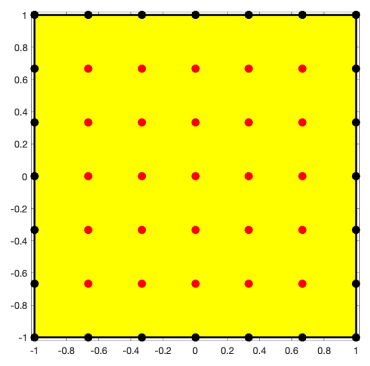
$$\begin{aligned} \left(\sum_{e=1}^{E} \mathbf{K}_{\Omega\Omega}^{(e)}\right) \cdot \boldsymbol{u}_\Omega + \left(\sum_{e=1}^{E} \mathbf{K}_{\Omega\Gamma_\mathrm{D}}^{(e)}\right) \cdot \boldsymbol{u}_{\Gamma_\mathrm{D}} &= \sum_{e=1}^{E} \boldsymbol{f}_\Omega^{(e)}, \\ \Leftrightarrow \left(\sum_{e=1}^{E} \mathbf{K}_{\Omega\Omega}^{(e)}\right) \cdot \boldsymbol{u}_\Omega &= \sum_{e=1}^{E} \boldsymbol{f}_\Omega^{(e)} - \left(\sum_{e=1}^{E} \mathbf{K}_{\Omega\Gamma_\mathrm{D}}^{(e)}\right) \cdot \boldsymbol{u}_{\Gamma_\mathrm{D}}, \\ \Leftrightarrow \left(\sum_{e=1}^{E} \mathbf{K}_{\Omega\Omega}{}^{(e)}\right) \cdot \boldsymbol{u}_\Omega &= \sum_{e=1}^{E} \left(\boldsymbol{f}_\Omega^{(e)} - \mathbf{K}_{\Omega\Gamma_\mathrm{D}}^{(e)} \cdot \boldsymbol{u}_{\Gamma_\mathrm{D}}\right). \end{aligned} \tag{2.4}$$

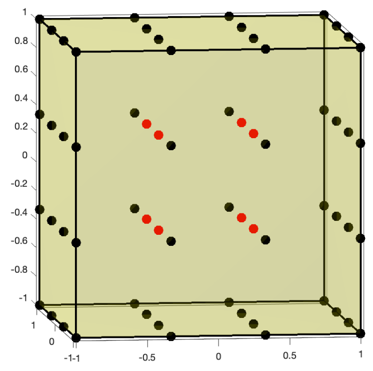
Since there exists a direct map between the local entries in an element matrix or right-hand side and their position in the global system of equations, the reduction can not only be performed on $\mathbf{K}^{(e)}$ and $\boldsymbol{f}^{(e)}$ but also on the element matrix and element right-hand side. It is thus seen that the removal of DOFs related to nodes on the Dirichlet boundary can be highly parallelized.

2.2 Static condensation of inner element nodes

In higher-order FE meshes, each element has one or more inner nodes. These nodes are not connected with any other element and are needed for the definition of the higher-order shape functions in the corresponding element. In most FE simulations, these nodes are not treated differently from nodes belonging to several elements. Therefore, they can significantly contribute to the size of the system of equations. By applying static condensation, the inner nodes of each element can be removed from the global system of equations, while the accuracy of a higher-order solution can still be maintained.

Fig. 11 shows a visualization of the inner nodes (condensed nodes) marked in red. It shows a two-dimensional 6th-order and a three-dimensional 3rd-order reference element, as well as a two-dimensional example mesh with 6th-order elements.


(a) 6th-order 2D reference element


(b) 3rd-order 3D reference element

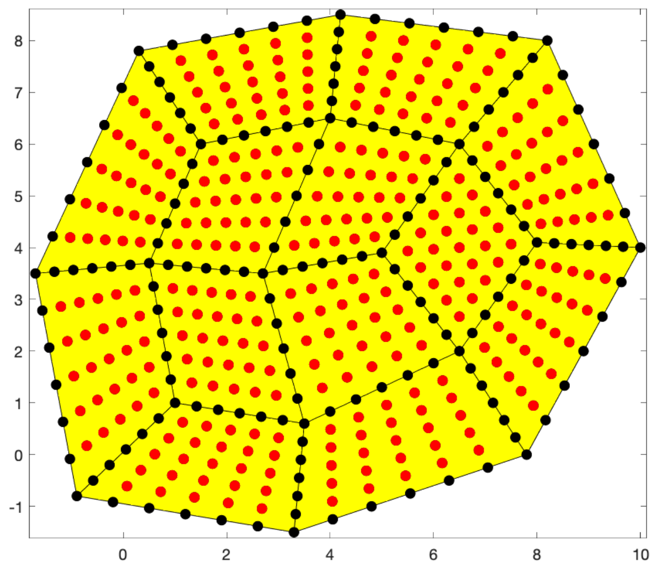

(c) 2D example mesh with 6th-order elements

Fig. 11: Higher-order elements and mesh, inner element nodes are marked in red and are statically condensed.

The principle of static condensation [22] is explained below:

$$\begin{bmatrix} \mathbf{K}_{\mathrm{oo}} & \mathbf{K}_{\mathrm{oi}} \\ \mathbf{K}_{\mathrm{io}} & \mathbf{K}_{\mathrm{ii}} \end{bmatrix} \begin{bmatrix} \boldsymbol{u}_{\mathrm{o}} \\ \boldsymbol{u}_{\mathrm{i}} \end{bmatrix} = \begin{bmatrix} \boldsymbol{f}_{\mathrm{o}} \\ \boldsymbol{f}_{\mathrm{i}} \end{bmatrix} . \tag{2.5}$$

For each element, the DOFs are split into those related to nodes on the element boundary $\boldsymbol{u}_{\mathrm{o}}^{(e)}$ and those related to inner nodes $\boldsymbol{u}_{\mathrm{i}}^{(e)}$ – see Fig. 11a and 11b with $\boldsymbol{u}_{\mathrm{o}}^{(e)}$ marked in black and $\boldsymbol{u}_{\mathrm{i}}^{(e)}$ marked in red. Let $\boldsymbol{u}_{\mathrm{o}}$ be related to *all* nodes on the element boundaries and $\boldsymbol{u}_{\mathrm{i}}$ to *all* inner element nodes. Then we may split the overall system of equations as shown in 2.5. By rearranging the second part of the equation,

$$\begin{aligned} \mathbf{K}_{\mathrm{io}} \cdot \boldsymbol{u}_{\mathrm{o}} + \mathbf{K}_{\mathrm{ii}} \cdot \boldsymbol{u}_{\mathrm{i}} &= \boldsymbol{f}_{\mathrm{i}}, \\ \Leftrightarrow \boldsymbol{u}_{\mathrm{i}} &= \mathbf{K}_{\mathrm{ii}}^{-1} \cdot (\boldsymbol{f}_{\mathrm{i}} - \mathbf{K}_{\mathrm{io}} \cdot \boldsymbol{u}_{\mathrm{o}}) \end{aligned}$$

and substituting $\boldsymbol{u}_{\mathrm{i}}$ in the first equation, $\boldsymbol{u}_{\mathrm{i}}$ can completely be removed from the system of equations:

$$\begin{aligned} \mathbf{K}_{\mathrm{oo}} \cdot \boldsymbol{u}_{\mathrm{o}} + \mathbf{K}_{\mathrm{oi}} \cdot \boldsymbol{u}_{\mathrm{i}} &= \boldsymbol{f}_{\mathrm{o}}, \\ \Leftrightarrow \mathbf{K}_{\mathrm{oo}} \cdot \boldsymbol{u}_{\mathrm{o}} + \mathbf{K}_{\mathrm{oi}} \cdot \mathbf{K}_{\mathrm{ii}}^{-1} \cdot (\boldsymbol{f}_{\mathrm{i}} - \mathbf{K}_{\mathrm{io}} \cdot \boldsymbol{u}_{\mathrm{o}}) &= \boldsymbol{f}_{\mathrm{o}}. \end{aligned}$$

This leads to the following reduced system of equations, with $\mathbf{K}_{\mathrm{red}}$ being the same as the Schur complement of the matrix $\bar{\mathbf{K}} = \begin{bmatrix} \mathbf{K}_{\mathrm{ii}} & \mathbf{K}_{\mathrm{io}} \\ \mathbf{K}_{\mathrm{oi}} & \mathbf{K}_{\mathrm{oo}} \end{bmatrix}$ [23]:

$$\mathbf{K}_{\mathrm{red}} \cdot \boldsymbol{u}_{\mathrm{o}} = \boldsymbol{f}_{\mathrm{red}}, \tag{2.6}$$

with

$$\begin{aligned} \mathbf{K}_{\mathrm{red}} &= \mathbf{K}_{\mathrm{oo}} - \mathbf{K}_{\mathrm{oi}} \cdot (\mathbf{K}_{\mathrm{ii}}^{-1} \cdot \mathbf{K}_{\mathrm{io}}), \\ \boldsymbol{f}_{\mathrm{red}} &= \boldsymbol{f}_{\mathrm{o}} - \mathbf{K}_{\mathrm{oi}} \cdot (\mathbf{K}_{\mathrm{ii}}^{-1} \cdot \boldsymbol{f}_{\mathrm{i}}). \end{aligned}$$

Due to the fact that all entries in the system of equations corresponding to an inner node are only influenced by the element itself, static condensation is here performed on an element level. It is sufficient to sum over the reduced element matrix $\mathbf{K}_{\mathrm{red}}^{(e)}$ and right-hand side $\boldsymbol{f}_{\mathrm{red}}^{(e)}$ to build the global system of equations:

$$\mathbf{K}_{\mathrm{red}} = \sum_{e=1}^{E} \mathbf{K}_{\mathrm{red}}^{(e)} \quad \text{and} \quad \boldsymbol{f}_{\mathrm{red}} = \sum_{e=1}^{E} \boldsymbol{f}_{\mathrm{red}}^{(e)}, \tag{2.7}$$

with

$$\begin{aligned} \mathbf{K}_{\mathrm{red}}^{(e)} &= \mathbf{K}_{\mathrm{oo}}^{(e)} - \mathbf{K}_{\mathrm{oi}}^{(e)} \cdot \left((\mathbf{K}_{\mathrm{ii}}^{(e)})^{-1} \cdot \mathbf{K}_{\mathrm{io}}^{(e)} \right), \\ \boldsymbol{f}_{\mathrm{red}}^{(e)} &= \boldsymbol{f}_{\mathrm{o}}^{(e)} - \mathbf{K}_{\mathrm{oi}}^{(e)} \cdot \left((\mathbf{K}_{\mathrm{ii}}^{(e)})^{-1} \cdot \boldsymbol{f}_{\mathrm{i}}^{(e)} \right). \end{aligned}$$

For the computation of $\mathbf{K}_{\mathrm{red}}^{(e)}$ and $\boldsymbol{f}_{\mathrm{red}}^{(e)}$, the LU-decomposition of the matrix $\mathbf{K}_{\mathrm{ii}}^{(e)}$ is computed, which is a local element-wise operation without any costly communication. For the comparatively small matrix $\mathbf{K}_{\mathrm{ii}}^{(e)}$, the LU-decomposition can be done in a tolerable amount of time, while keeping a maximum accuracy. Furthermore, the decomposition is only performed once, whereas the fast backsubstitution is performed for the computation of $\mathbf{K}_{\mathrm{red}}^{(e)}$ and $\boldsymbol{f}_{\mathrm{red}}^{(e)}$, respectively.

The number of DOFs in $\mathbf{K}_{\text{red}}$ is less by the number of inner nodes times the number of DOFs per node. For example, in a 3rd-order hexahedral element (Fig. 11b) with a total of $4^3 = 64$ nodes, 4 of these nodes can be condensed, leading to a reduction of the DOFs of (only) $\frac{4}{64} \approx 6\%$. In a 6th-order hexahedral element with a total of $7^3 = 343$ nodes, 125 of these nodes can be condensed, leading to a reduction of $\frac{125}{343} \approx 36\%$. In the context of linear elasticity in three dimensions, having 3 DOFs per node, the 1029×1029 element matrix of a 6th-order hexahedron can be reduced to a 654×654 matrix. It is thus seen that the gain is directly related to the element order and significantly improves with higher orders. Fig. 12 shows the percentage of condensed nodes of an hexahedral element over increasing element order.

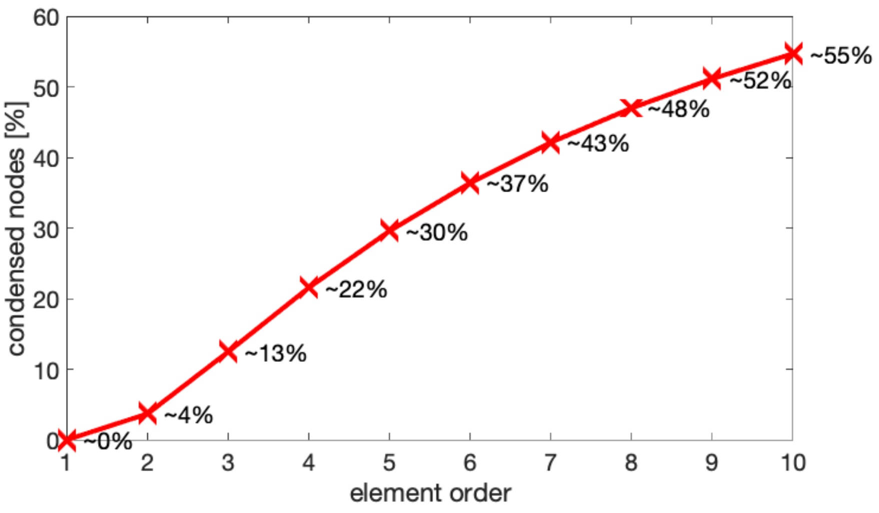


Fig. 12: Percentage of condensed nodes in an hexahedral element plotted over element order.

3 Domain decomposition

In order to obtain a parallel simulation of the problem, it is necessary to divide the domain into smaller parts, such that each process is responsible for the solution of one part. These parts – the so-called subdomains – can then be treated as independent domains for the assembly of the (sub)system of equations. Therefore, the domain is split alongside the element boundaries. In that way, each subdomain gets assigned several connected elements. Fig. 13 shows an idealistic decomposition of a rather basic example of a two-dimensional domain.

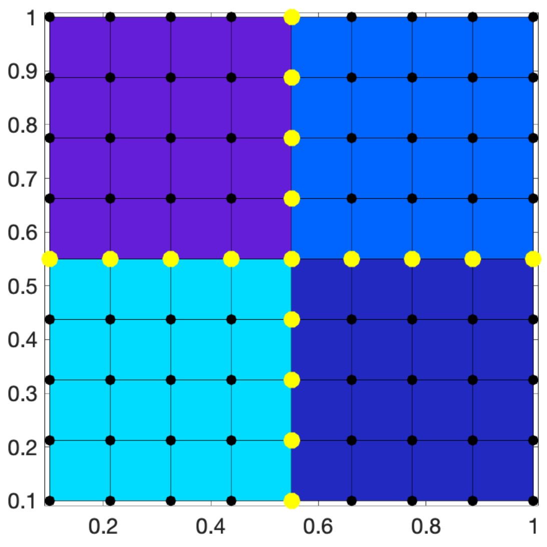


Fig. 13: Example 2D domain decomposition.

It can be seen that the nodes which lie on the boundary between two or more subdomains (marked in yellow) are assigned to more than one subdomain. These nodes are called interface nodes in the remainder of this thesis.

When assembling the (sub)system of equations, each process assembles only the influence of the elements within its own subdomain. Therefore, in order to obtain a correct solution for the global problem, communication of the entries corresponding to the interface nodes is required when solving the system of equations. The solving and communication will be explained in more detail in chapter 4. However, it should be noted here that communication is still considered a bottleneck of most parallel applications and reducing communication as much as possible is essential. Therefore, the quality of a decomposition can be measured on the basis of two criteria:

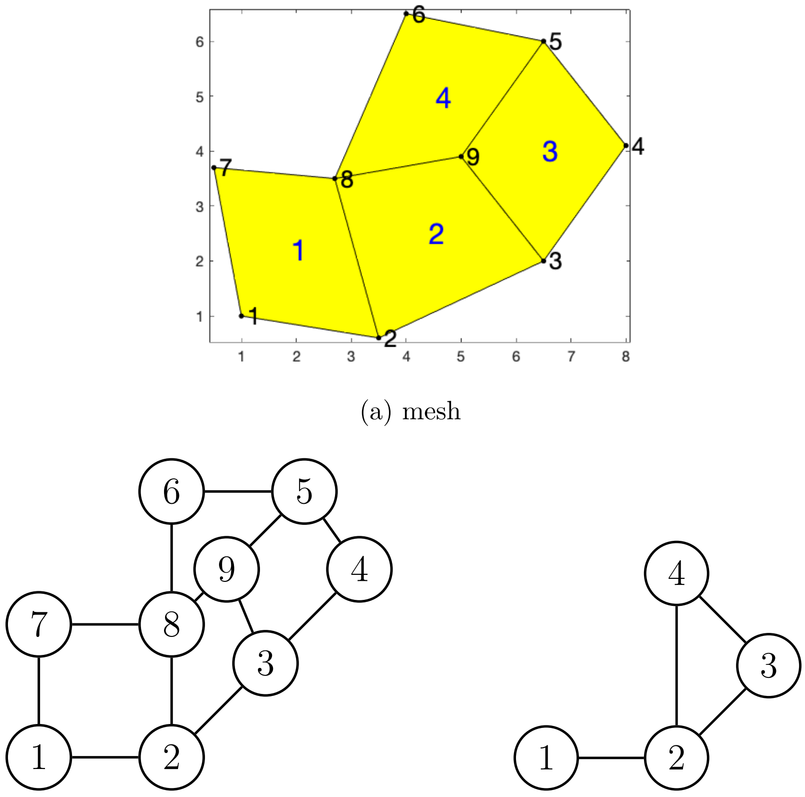
- **Minimizing the number of interface nodes** in order to reduce the necessary communication.
- **Equally distributing the elements to each process** in order to balance the workload of all processors.

In this chapter, two different serial domain decomposition approaches are compared in order to find the best fit for the FE-simulations which are considered in this work. The first approach is the "shortest distance decomposition algorithm" [24]. This algorithm is primarily used in transportation problems but allegedly also very effective for structural engineering simulations. The second approach is the multilevel graph partitioning of the meshes dual-graph. Therefore, the well known software package METIS, which is almost considered as standard for graph partitioning problems, is used [25].

3.1 Shortest distance decomposition algorithm

According to [24], the shortest distance decomposition algorithm (SDDA) is a new and simple, yet very efficient domain decomposition algorithm which was shown to even outperform METIS in more than two thirds of the tested transportation examples. It is investigated in this work how this algorithm performs in the context of general FE simulations. In this section not only the SDDA is explained but also the simplifications implemented in order to create a more efficient decomposition algorithm to also handle large-scale problems.

In structural engineering domains the goal is to assign whole elements to the different processes. Therefore, the first modification is to use the dual graph of the mesh.


(a) mesh

(b) nodal graph

(c) dual graph

Fig. 14: Nodal and dual graph of a 2D example mesh.

In contrast to the nodal graph – where every vertex corresponds to one node – in the dual graph every vertex corresponds to one element of the mesh. If two elements are connected in the mesh, an edge between the two corresponding vertices exists in the dual graph. However, in order to reduce the necessary computing time a graph with fewer edges is desirable. Therefore, in 3D two elements are only considered connected, if they have a common element *face*. Elements that only share edges or a single node are not considered to be connected. Fig. 14 shows a comparison of the nodal graph and the dual graph of a simple two-dimensional example mesh. In this 2D dual graph two elements are considered connected if they share a common *edge*.

The main idea of the SDDA is to first compute the so-called source elements. Each source element is the basis of one subdomain. Every subdomain is then iteratively assigned another connected element until each element is assigned to exactly one subdomain. In order to obtain the best decomposition, the source elements should have the largest possible distance to each other. It should be noted here that in this chapter, distance never refers to the actual metric distance between the elements but to the number of edges between two elements in the dual graph. This is equivalent to the number of elements that lie between the two considered elements. The SDDA as well as its individual steps are described below [24].

Algorithm 3.1: The shortest distance decomposition algorithm SDDA

```
Input: an unweighted dual graph of the mesh, chosen number of subdomains
foreach element do
    compute rank;                              // number of connected elements
foreach subdomain do
    choose next source element $s$ (algorithm 3.3);
    compute distance from $s$ to all other elements with modified BFS (algorithm 3.2);
populate subdomains (algorithm 3.4);
return vector of element assignments to subdomains
```

3.1.1 Shortest path algorithm

In order to obtain the distance of an element to all other elements, an efficient shortest path algorithm should be available. This is a very important topic also in the sense of network topology and other computer science domains. Hence, a lot of different, well studied algorithms for this problem exist, e.g., Dijkstra algorithm, polynomial label correcting algorithm by Bellman-Ford-Moore [26].

Assuming a graph with n vertices (number of elements) and m edges (total number of connections between elements), these algorithms have a runtime of $\mathcal{O}(m \cdot \log n)$, $\mathcal{O}(m \cdot n)$ up to $\mathcal{O}(n^2)$ depending on the chosen algorithm and the used data-structures [26]. In the context of 3D structural engineering domains where the upper bound of connections of on an element is given by a constant C (6 for hexahedral and 4 for tetrahedral elements), an upper bound of the total number of edges in the mesh can be estimated as Cn. Using this information, the runtimes can be expressed as $\mathcal{O}(n \cdot \log n)$ or $\mathcal{O}(n^2)$.

Due to the fact that the considered dual graphs are always unweighted graphs, meaning that each connection between two elements has the same "distance", a more efficient algorithm to obtain the shortest path exits. The so-called breadth-first search (BFS) algorithm is generally used for finding data in a tree data structure. Since it is an optimal and complete search algorithm it always finds the shortest path in a graph [27]. Provided that proper modifications are made, the time complexity of the BFS algorithm is given by $\mathcal{O}(n)$, therefore making it an optimal choice for the implemented shortest path algorithm.

Algorithm 3.2: Modified breadth-first search

```
Input: an unweighted graph $\mathbf{G}$, source element $s$

// Initialize
$\boldsymbol{d}[j] = \infty \qquad \forall j \in \mathbf{G} \setminus s$;
$\boldsymbol{d}[s] = 0$;
add $s$ to the queue named Next;

// calculate distances
while Next is not empty do
    remove the first element $i$ from Next;
    foreach neighbor element $j$ of $i$ do
        if $\boldsymbol{d}[j] = \infty$ then
            $\boldsymbol{d}[j] = \boldsymbol{d}[i] + 1$;
            add $j$ to Next;

return vector $\boldsymbol{d}$ of distances from $s$ to every other element
```

In the beginning, the distances of all elements are initialized with ∞, except for the source element which has a distance of 0. The distances of all neighbors of the source element are then updated with $0 + 1 = 1$. All updated elements are appended to the queue[10] to insure that the distance of their neighbors is updated as well. Due to the fact that the distance between two elements is always 1 (unweighted graph), it is not necessary to consider an element again once it has been updated. This leads to the major runtime advantage over the general shortest path algorithms which also work for weighted graphs.

3.1.2 Choice of the source elements

After every element is assigned a rank, which is equal to the number of elements it is connected to in the dual graph, the source elements of the decomposition are chosen. As a first source node, the node with the lowest rank is chosen. Therefore, corner elements which

[10]A queue is a list of data that is processed in a FIFO (first in, first out) manner. It can be thought of as a queue at a ticket counter.

are very suitable as source elements are easily identified. Often several elements share the same rank. Then, the element with the lowest ID is chosen. Afterwards, all other source elements are computed in an iterative fashion.

In order to find a good decomposition, the source elements should be spread evenly across the domain. Furthermore, their distance should be maximized to achieve maximum flexibility when assigning elements to the different subdomains. This is very important in order to create equally sized subdomains and minimize the interface nodes when populating the subdomains. However, in contrast to the original algorithm as described in [24], the elements were not only evaluated based on their distance, since this method tended to produce an uneven distribution of the source elements. Therefore, in every iteration instead of choosing the element with the largest total distance to all other source elements, a fitness value is computed for every element. The element with the highest fitness is then chosen as the new source element. The fitness function considers the total distance to all other source elements, the rank, the minimum of the distances to all other source elements and the range between minimum and maximum distance. These four quantities are weighted by normalizing weights, such that no quantities are larger than 1. The weights are chosen as follows: $w_1 = \frac{1.0}{D_{\max}}$, $w_2 = \frac{0.5}{k_{\max}}$, $w_3 = \frac{0.5}{D_{\max}}$ and $w_4 = \frac{1.0}{D_{\max}}$ where $D_{\max}$ is the maximum possible distance in the graph and $k_{\max}$ is the maximum rank.

Algorithm 3.3: Choose next source element

```
Input: distances to existing source elements, element ranks $k_e$
if $s$ is the first source element to be chosen then
    choose $s$ as element with lowest rank;
else
    $f_{\max} = -\infty$;
    foreach unassigned element $e$ do
        $d_{\text{tot}}$ = sum over distances from $e$ to all previously chosen source elements;
        $d_{\min}$ = minimum of distances from $e$ to all previously chosen source elements;
        $d_{\max}$ = maximum of distances from $e$ to all previously chosen source elements;
        $r = d_{\max} - d_{\min}$;
        $f_e = w_1 \cdot d_{\text{tot}} - w_2 \cdot k_e - w_3 \cdot r + w_4 \cdot d_{\min}$;        // element fitness
        if $f_e > f_{\max}$ then
            $s = e$;
            $f_{\max} = f_e$;
return next source element $s$
```

Fig. 15 shows the obtained source elements in two domains which are split into 9 subdomains. The source elements are marked in yellow and numbered in the order in which they were obtained.

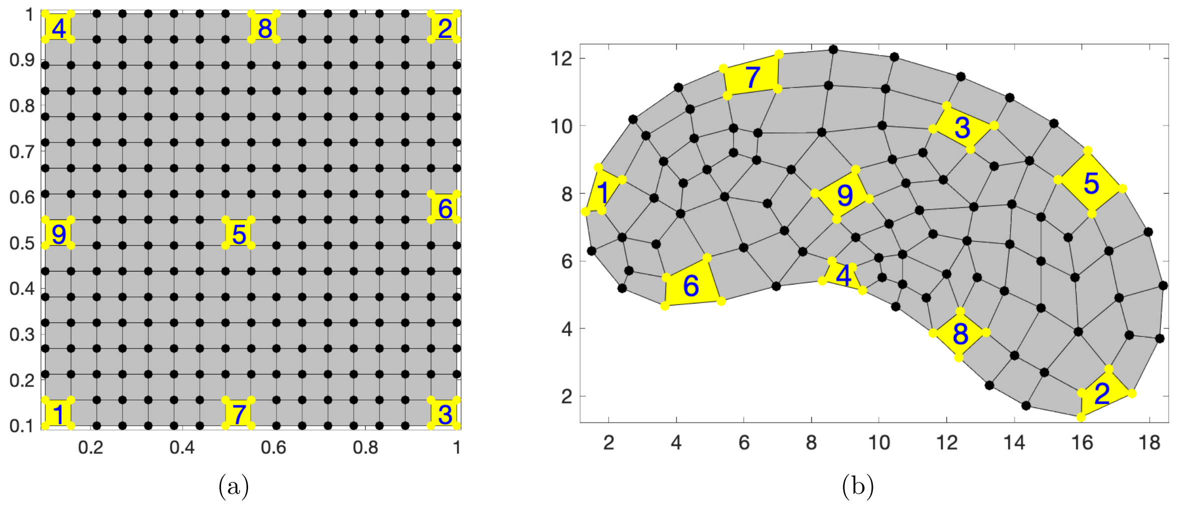


Fig. 15: Choice of the source elements in two example domains in 2D.

3.1.3 Population of the subdomains

After every subdomain got assigned a source element, the subdomains are populated in an iterative manner. In the original algorithm [24] in every iteration each subdomain iterates over all elements, to find the best fit. However, this causes the runtime of the algorithm to be $\mathcal{O}(n^2)$ which is not acceptable for a preprocessing step in a large-scale simulation. As remedy, an element ranking is computed for every subdomain. This ranking is based on the following quantities:

- Distance of the element to the source element of the current subdomain.
- Rank of the element.
- Range of the element.

The elements are ranked in such a way that first, the distance is minimized in order to cluster the subdomain around the source element. If two elements have the same distance, the element with the lower rank achieves a higher score, in order to start clustering from the boundary to the inside. If there is still a tie between two elements, the element with the largest range achieves a higher score, assuming that it is further away from other source nodes. Using these factors, an element ranking is computed for each subdomain.

Thereafter, the elements are assigned. In each iteration every subdomain chooses the next unassigned element based on its ranking until all elements are assigned. However, sometimes a chosen element might not be connected to any element in the current subdomain. This happens especially when most of the elements are already assigned and some subdomains are already surrounded by other subdomains. Therefore, if the chosen element is not connected to the subdomain no element is assigned to this subdomain in that iteration.

Algorithm 3.4: Populate subdomain

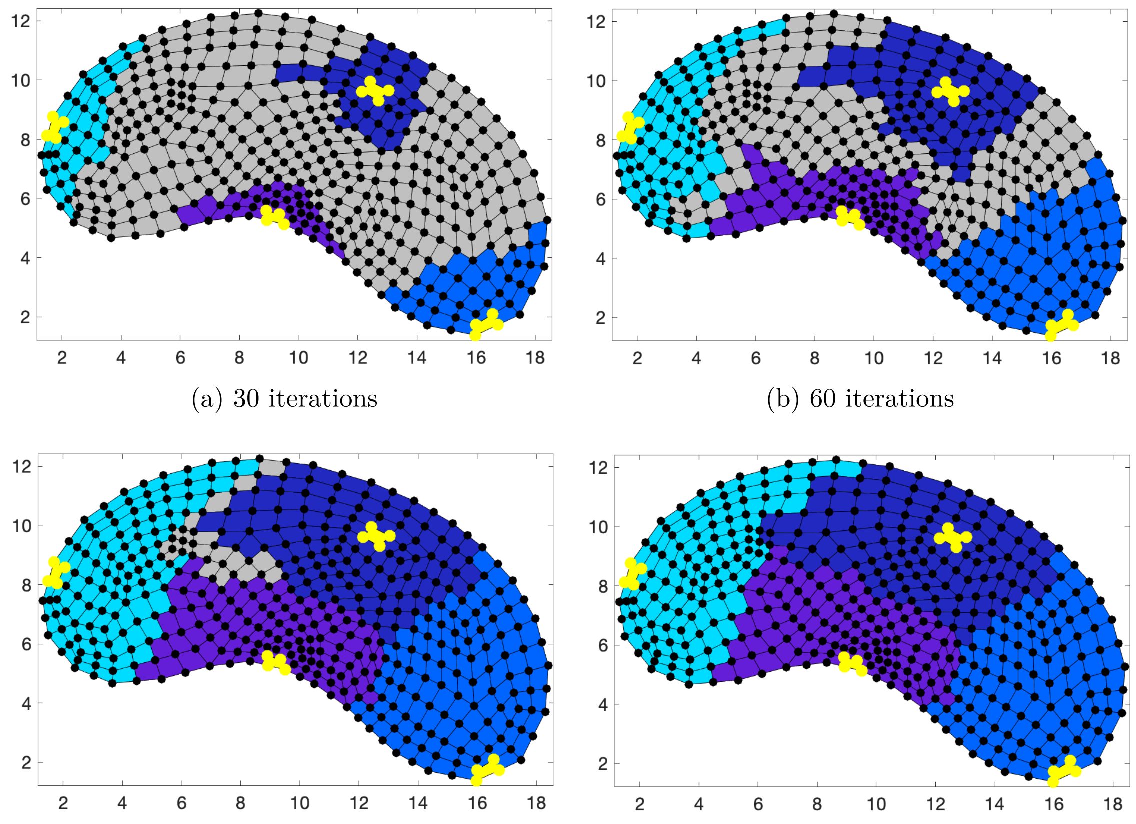
```
Input: chosen source elements, distances, ranks, ranges

foreach subdomain $s$ do
    compute element ranking based on distance, rank and range;

while not every element is assigned do
    foreach subdomain $s$ do
        $e$ = the unassigned element with the highest score;
        if $e$ is connected to $s$ then
            assign $e$ to $s$;

return vector of element assignments to subdomains
```

In rare cases, some elements might be left unassigned (e.g., the elements were not chosen because they were not connected to the subdomain when they appeared in the respective ranking). Therefore, a safety measure was implemented: In the case, that all subdomains considered all ranked elements but not every element was assigned, the remaining elements are assigned to their neighbor domain.



(a) 30 iterations (b) 60 iterations

(c) 90 iterations (d) 113 iterations

Fig. 16: Population of the subdomains in a two-dimensional mesh.

Fig. 16 shows a two-dimensional example domain with 376 elements. The domain is split into 4 subdomains. The assignment of the elements is plotted after 30, 60, 90 and 113 iterations. The source elements are again marked in yellow.

Considering all steps of the algorithm, the total runtime can be expressed as $\mathcal{O}(p \cdot n)$, with n being the number of elements and p the number of subdomains.

3.2 METIS multilevel graph partitioning

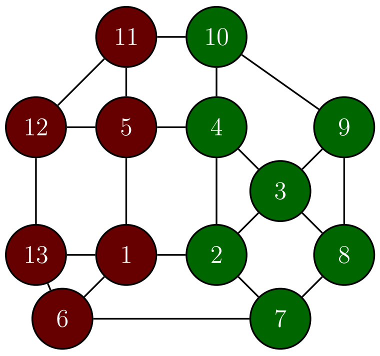
The serial, open source software package METIS is a very popular tool for partitioning large and irregular graphs as those resulting from large meshes. It provides several different routines with numerous user-defined options and parameters to select and specify algorithms. For the decomposition of the considered structural engineering domains, the METIS routine **METIS_PartMeshDual** is the most convenient choice. This function computes a partitioning of the mesh into a given number of parts based on the dual graph. The user may specify how many common nodes two elements in the mesh must have in order for them to be connected in the dual graph [25, pp. 6, 28]. In the code present, two 3D elements are considered connected if they have a common element face. Elements that only share edges or a single node are not connected.

The internal decomposition algorithm which is then used on the dual graph is either multilevel recursive bisection or multilevel k-way bisection. However, in order to obtain a better understanding of the partitioning, the principle of recursive graph bisection, which is the basis of several advanced graph partitioning algorithms, such as multilevel recursive bisection, is explained first.

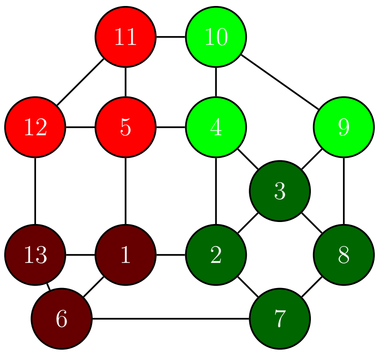
3.2.1 Recursive graph bisection

The recursive graph bisection recursively divides the graph into two roughly equal graphs until the number of desired partitions is reached. Therefore, different algorithms may be used. For example one could use the greedy graph growing partitioning algorithm which is based on the breadth first search (see section 3.1.1). Thereby starting with a chosen source element and iteratively collecting adjacent elements (depending on a given criterion) until half of the elements are chosen. According to the authors of [28] once multilevel graph partitioning methods are used, the choice of the partitioning algorithm itself seems to have little effect on the efficiency of the partitioning. Therefore, the different partitioning algorithms are not described in more detail in this section.

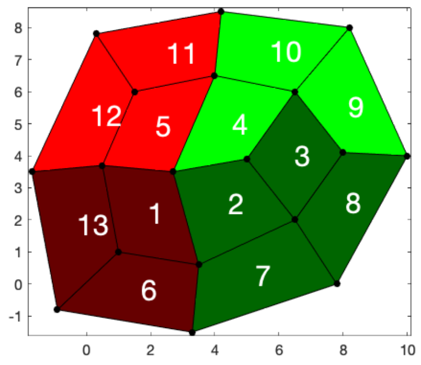
Fig. 17 shows the different steps of the recursive bisection of a simple graph into 4 subdomains. It also shows an example mesh for the given graph.



(a) 1st split - 2 subdomains



(b) 2nd, 3rd split - 4 subdomains



(c) split mesh

Fig. 17: Principle of recursive bisection shown on an example 2D domain.

3.2.2 Multilevel recursive bisection

In contrast to simple recursive bisection as described above, in multilevel recursive bisection (MLRB), the initial partitioning is not performed on the full dual graph, but only on its coarsened version. Therefore, each bisection step can be split into the following three phases, which are visualized in Fig. 18 [28, pp. 363–370].

- **Coarsening phase**: A sequence of smaller graphs G_1 - G_m is constructed by collapsing several vertices of G_i into one vertex in the coarser graph G_{i+1}.
- **Partitioning phase**: A bisection of the coarsest graph G_m is computed.
- **Uncoarsening phase**: The partition is projected back from G_m onto the original graph. This happens by going through the sequence of coarsened graphs, where each vertex of graph G_{i+1} contains a distinct subset of vertices of G_i. Furthermore, after each projection a refinement algorithm is used in order to reduce the edge-cut of the bisection[11].

[11]Reducing the edge-cut of a partitioning is equal to minimizing the number of interface nodes of the split.

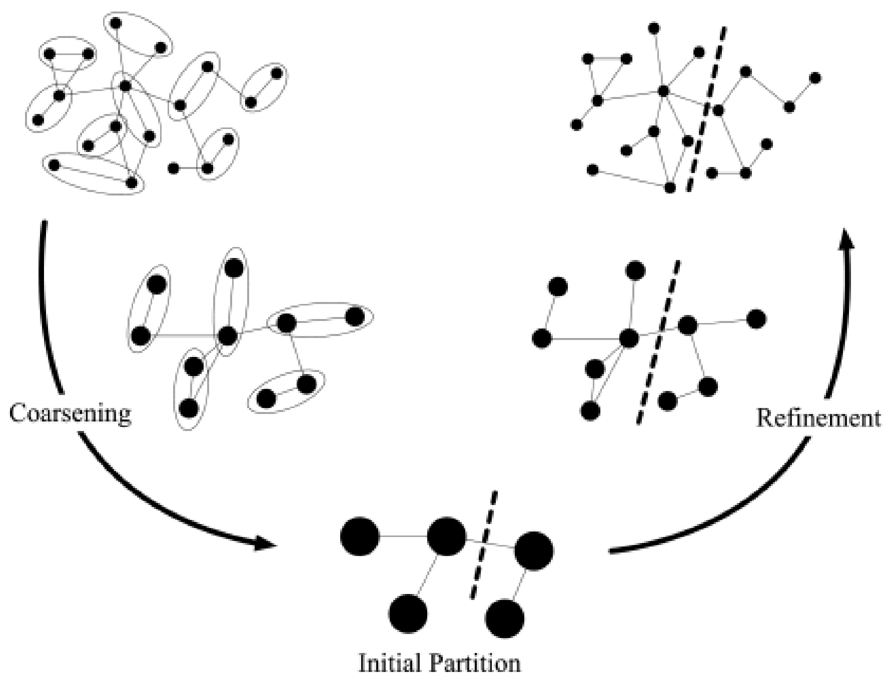


Fig. 18: Multilevel recursive bisection [29].

The MLRB has shown to produce significantly better and faster partitionings then other well known recursive bisection algorithms. Its complexity is given as $\mathcal{O}(m \cdot \log p)$, with m being the number of the edges in the graph and p being the number of subdomains into which the domain is split [30, p. 97]. For the considered structural engineering domains, the number of edges is direct proportional to the number of elements. Therefore, the complexity can also be expressed as $\mathcal{O}(n \cdot \log p)$, with n being the number of elements.

In contrast to the MLRB, the multilevel k-way bisection only coarsens the graph once and then performs a k-way split[12] on the coarsened graph. The METIS user can define whether the multilevel recursive bisection or the multilevel k-way bisection should be performed as internal partitioning algorithm [30, p. 97]. Due to the fact that both algorithms produced similar results, the MLRB, which is the default partitioning algorithm, was used in this thesis.

3.3 Comparison

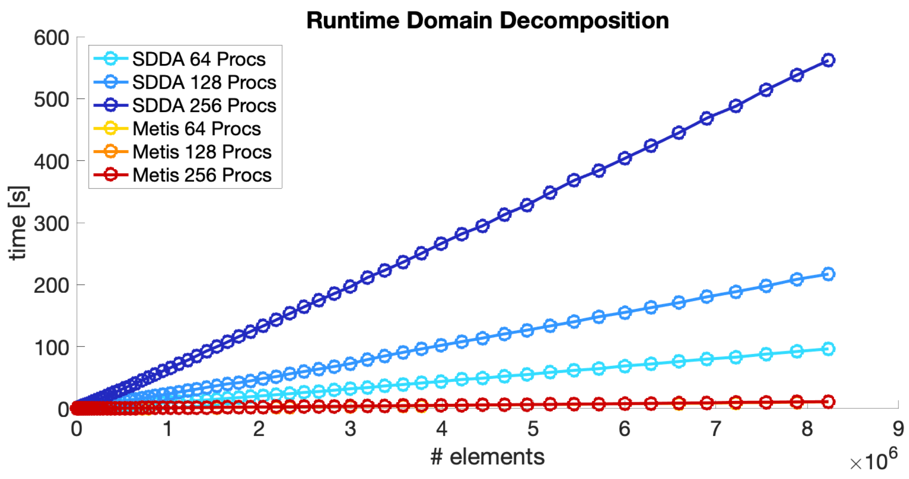
In order to compare the SDDA and METIS version of the MLRB, their performance when used on the same problem was compared using 64, 128 and 256 processes where the number of processes is of course equal to the number of subdomains. In order to achieve a maximum speedup when increasing the number of processes, it is important that the cost of the additional communication and decomposition is minimal compared to the advantage in solution time. Therefore, the following two factors are considered:

[12] A k-way split is a direct split of the graph into k subdomains, rather then recursively performing bisections.

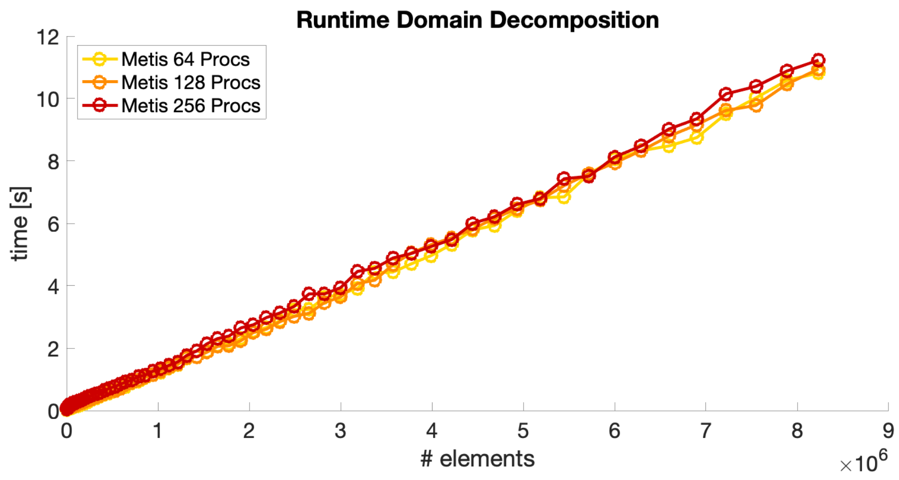
- Quality of the decomposition, measured by the needed solving time of the problem.
- Runtime of the domain decomposition as a serial preprocessing step.

In terms of the quality of the decomposition both algorithms produce satisfying results. However, METIS outperformed the SDDA when comparing the needed solving times. The use of METIS leads to lower computation as well as communication times. This means that the decomposition computed by METIS is better not only in the sense of minimizing the interface nodes but also in balancing the workload.

Comparing the runtime of the decompositions it can be seen that the well-established METIS manages to perform the decomposition in a significantly smaller amount of time. Fig. 19 shows a comparison of the runtimes with increasing number of tetrahedral elements.



(a) SDDA and METIS



(b) METIS only

Fig. 19: Runtime comparison of the SDDA and METIS.

Even though both algorithms have a linear runtime, the runtime of METIS is significantly smaller. Furthermore, using METIS the chosen number of subdomains has less influence on the runtime than with the SDDA. This can be explained by the different approaches of the algorithms. The SDDA computes source elements and then populates each subdomain. By doubling the number of subdomains, the number of computed source elements and therefore also distances as well as the number of element rankings also doubles, resulting in a runtime of $\mathcal{O}(n \cdot p)$. METIS, however, performs a recursive bisection of a coarsened graph. Due to the nature of recursive bisection, doubling the number of subdomains does not double the runtime. METIS performs in a runtime of $\mathcal{O}(n \cdot \log p)$.

The SDDA takes a relatively large amount of time. For simulations with linear elements the time needed for the decomposition is even in the same magnitude as the time needed for the parallel simulation. Therefore, it is not suitable for large-scale problems with a high number of elements.

Comparing the SDDA and METIS, it can be shown that the SDDA might be a very effective algorithm when transportation problems are considered. However, in large-scale structural engineering simulations the well-established METIS still offers superior algorithms for domain decomposition. Therefore, METIS was used in all further simulations.

4 Parallel solution with the Jacobi-preconditioned CG solver

4.1 The parallel Jacobi-preconditioned CG solver

As discussed in section 1.2 for solving symmetric positive definite linear systems of equations, as they are produced in the considered FE simulations, the conjugate gradient (CG) algorithm is an excellent choice. Since most of the time in a FEM simulation is needed for the solving procedure, the proper parallelization of the solver is of critical importance in order to achieve a significant speedup. Considering the CG algorithm 1.1, it can be seen that not every step necessarily has to be performed on the global data. Three communication routines per iteration are sufficient in order to obtain a global search direction and step size. Therefore, accumulated vectors are computed. These accumulated vectors are similar to the local vectors, but also contain the contributions of the adjacent processes. The parallel CG algorithm is shown below.

In order to make the difference between local and accumulated variables more visible, all accumulated variables are of the form $\widetilde{\boldsymbol{z}}$ compared to the local variables $\boldsymbol{z}$. The communication steps are also marked and will be explained in more detail later in this chapter.

The Jacobi method 4.1 was chosen for preconditioning the system of equations in this thesis. Due to its simplicity, this algorithm can be easily parallelized. It is noted here, that as shown in section 1.2 there are more effective preconditioning methods like the incomplete Cholesky factorization. However, systematically finding and implementing a very effective and yet highly parallelizable preconditioning algorithm is beyond the scope of this thesis.

Algorithm 4.1: Parallel Jacobi preconditioner [31]

```
Input: local system matrix $\mathbf{K}$
$\mathbf{M} = diag(k_{11}, k_{22}, ..., k_{nn})$;
Sum up local contributions of adjacent Processes into $\widetilde{\mathbf{M}}$;        // Communication 1
$\widetilde{\mathbf{M}}^{-1} = diag(\frac{1}{\widetilde{\mathbf{M}}_{11}}, \frac{1}{\widetilde{\mathbf{M}}_{22}}, ..., \frac{1}{\widetilde{\mathbf{M}}_{nn}})$;
return $\widetilde{\mathbf{M}}^{-1}$;
```

In order to increase efficiency in the implementation this function also computes the inverse Jacobi matrix $\widetilde{\mathbf{M}}^{-1}$ which is then stored instead. Furthermore, the inverse Jacobi matrix as well as the local system matrix $\mathbf{K}$ are stored in the sparse (i,j,v)-format to provide efficient memory management. In order to maximize data locality[13] in the iterative CG solver the nodes are renumbered with the reverse Cuthill-McKee algorithm at the beginning of the simulation [33]. This algorithm also improves the performance of graph algorithms – as used in the domain decomposition described in chapter 3.

[13]Increasing data locality can significantly decrease the computing time, due to the caching behavior of modern processors [32].

Algorithm 4.2: Parallel preconditioned conjugate gradient algorithm [17, p. 4]

```
Input: local system matrix $\mathbf{K}$, local system rhs $\boldsymbol{f}$, accumulated preconditioning matrix
       $\widetilde{\mathbf{M}}$, accuracy $\varepsilon$
$\widetilde{\boldsymbol{u}}^0 = \mathbf{0}, \quad \boldsymbol{r}^0 = \boldsymbol{f}$;
$\boldsymbol{z}^0 = \widetilde{\mathbf{M}}^{-1} \cdot \boldsymbol{r}^0$;
Sum up local contributions of adjacent Processes into $\widetilde{\boldsymbol{z}}^0$;          // Communication 2
$\varrho_0 = (\widetilde{\boldsymbol{z}}^0, \boldsymbol{r}^0)$;
Sum up local contributions of adjacent Processes into $\widetilde{\varrho}_0$;          // Communication 3
$\widetilde{\boldsymbol{p}}^0 = \widetilde{\boldsymbol{z}}^0$;

for $i=0,1,\ldots,n-2$ do
    $\boldsymbol{s}^k = \mathbf{K} \cdot \widetilde{\boldsymbol{p}}^k, \quad \sigma_k = (\boldsymbol{s}^k, \widetilde{\boldsymbol{p}}^k), \ \frac{1}{\alpha_k} = \frac{\sigma_k}{\widetilde{\varrho}_k}$;
    Sum up local contributions of adjacent Processes into $\frac{1}{\widetilde{\alpha}_k}$;      // Communication 4
    $\widetilde{\boldsymbol{u}}^{k+1} = \widetilde{\boldsymbol{u}}^k + \widetilde{\alpha}_k \widetilde{\boldsymbol{p}}^k$;
    $\boldsymbol{r}^{k+1} = \boldsymbol{r}^k - \widetilde{\alpha}_k \boldsymbol{s}^k$;
    $\boldsymbol{z}^{k+1} = \widetilde{\mathbf{M}}^{-1} \cdot \boldsymbol{r}^{k+1}$;
    Sum up local contributions of adjacent Processes into $\widetilde{\boldsymbol{z}}^{k+1}$;   // Communication 5
    $\varrho_{k+1} = (\widetilde{\boldsymbol{z}}^{k+1}, \boldsymbol{r}^{k+1})$;
    Sum up local contributions of adjacent Processes into $\widetilde{\varrho}_{k+1}$;   // Communication 6
    if $\widetilde{\varrho}_{k+1} < \varepsilon^2 \widetilde{\varrho}_0$ then
        break;
    end
    $\widetilde{\boldsymbol{p}}^{k+1} = \widetilde{z}_{k+1} + \widetilde{\beta}_k \widetilde{\boldsymbol{p}}^k, \quad \widetilde{\beta}_k = \frac{\widetilde{\varrho}_{k+1}}{\widetilde{\varrho}_k}$;
end
return $\widetilde{\boldsymbol{u}}^{k+1}$;
```

4.2 Communication

Even though the performance of computers and communication devices is continuously improving, communication is still considered a bottleneck of most parallel applications. The authors in [34] found in their studies that with a growing amount of processors up to half of the time needed by a parallel CG algorithm is communication time. There are two determining factors for achieving a maximum speedup. The first and most important one is to reduce the amount of communication needed. As discussed in chapter 3, the best way to achieve a minimum need of communication is to produce a domain decomposition with very few interface nodes. The second one is a good concept for the still necessary communication, which can also lead to a significant decrease in the required communication time.

4.2.1 Message Passing Interface

The message passing interface (MPI) is the most commonly used communication standard for programming high performance applications. Most of its implementations such as Open MPI or MPICH have very optimized algorithms to achieve fast communication. MPI offers several communication routines for different scenarios. The most basic ones and the ones used in the underlying code are outlined briefly in this section [35].

Firstly, it should be noted that programs using MPI incorporate the so-called single program, multiple data (SPMD) parallel computing model, meaning that instances of a single program operate on distinct data at the same time. However, the instances of the program are not synchronized at each individual operation level. Therefore, SPMD is equal to the MIMD (see section 1.3) computing model [36]. All processes created by MPI run exactly the same code from the beginning and no processes are spawned during runtime. However, most programs also have serial parts (e.g., saving the result of a simulation) which do not need to be conducted by all processes. Therefore, each process has an individual rank, which can be queried in order to branch to the designated destinations in the code. In most applications the process with the ID 0 is chosen as root process, which performs these serial tasks. Furthermore, communication routines use these ranks for addressing the sending and receiving processes.

Generally, one can distinguish between blocking and non-blocking communication routines. Blocking routines stall the current process until the requested communication has been completed. For receiving data this means that the data has been received completely. For sending data, however, the data has either been received by the receiving process or has been buffered by MPI. Whether data is buffered depends on different factors. One of the most defining factors is the size of the message. Nevertheless, one should never rely on buffering in order to avoid potential deadlocks[14] [37]. In more difficult communication scenarios with several involved processes or when the received data is not needed immediately, it may also be useful to use non-blocking calls. These non-blocking calls immediately continue. However, when the transferred data is needed or the memory of the sent data should be reused for other purposes, another function has to be called in order to assure that the communication has finished.

The most basic one-to-one[15] communication routines provided by MPI are [38]:

- **MPI_Send** blocking send.
- **MPI_Recv** blocking receive.
- **MPI_Isend** non-blocking send.
- **MPI_Irecv** non-blocking receive.
- **MPI_Wait** blocking - stalls the process until the provided MPI call has finished.

[14]A deadlock is a situation where several processes cannot continue, because they wait on each other to finish.

[15]One-to-one communication is defined as direct communication between two processes.

- **MPI_Waitall** blocking - stalls the process until a list of provided MPI calls have finished.
- **MPI_Waitany** blocking - stalls the process until one MPI call out of a list of provided MPI calls has finished.

In collective communication (one-to-many[16], many-to-one[17] or many-to-many[18] communication) some of the most important, blocking routines are [35]:

- **MPI_Bcast** broadcasts the information from the chosen root process to several other processes.
- **MPI_Reduce** performs a mathematical operation (e.g., sum) on the values of several processes (including the root) and provides the result to the chosen root process.
- **MPI_Allreduce** same as **MPI_Reduce** but the result is provided to all processes.
- **MPI_Gather** gathers the data from each process (including the root) into a coherent array of the chosen root process (the data is ordered by process number).

The visualization of **MPI_Bcast** shows the advantage of MPIs collective routines. Compared to calling **MPI_Send** in a loop for every process, **MPI_Bcast** can distribute the information much faster.

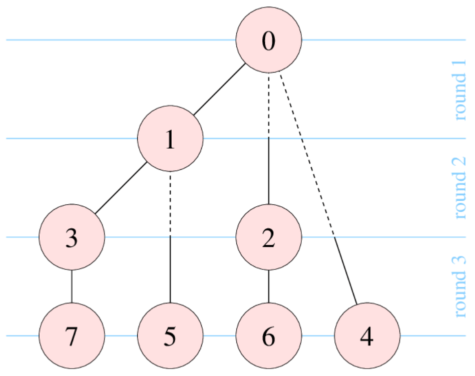


Fig. 20: **MPI_Bcast** scheme [39].

Figure 20 shows that in the first round process 0 sends the information to process 1, in the second round process 1 sends the information to process 3 while process 0 sends it to process 2 and so on. For distributing information to 7 other processes **MPI_Bcast** needs only 3 rounds, whereas repeatedly calling **MPI_Send** would require 7 rounds. Therefore, whenever several processes are involved in one communication, the use of collective communication routines should be preferred.

[16]One-to-many communication is defined as communication where one process sends information to multiple other processes.
[17]Many-to-one communication is defined as communication where multiple processes send information to one process.
[18]Many-to-many communication is defined as communication between several processes where multiple processes send their message to multiple other processes.

4.2.2 Implemented communication scheme

In the parallel CG algorithm 4.2, three communication procedures are needed for each iteration (Communication 4-6). Additionally two communication procedures are needed for the initialization (Communication 2-3) as well as one for generating the preconditioning matrix (Communication 1) in algorithm 4.1. However, all these communications can be divided into two different procedures.

1. **Global sum over all processes - Communication 3, 4 and 6**
   A single value per process is to be summed up to one global sum. The sum then should be distributed to each process. Therefore, a simple **MPI_Allreduce** operation is sufficient.
2. **Sum over targeted interface processes - Communication 1, 2 and 5**
   A more complex communication scheme where each process needs to communicate some entries of a vector with an individual number of interface processes. Since the preconditioning matrix is a diagonal matrix – stored in the sparse (i,j,v)-format – it can also be treated as a vector for communication purposes.

For the second communication procedure, different possibilities exist. One idea, as proposed in [17, pp. 6–7] would be that each process sends the values of its interface nodes to the root process which then sums up the values of all interface nodes in a designated vector and distributes the results to the individual processes. However, this procedure has very high requirements on the root processor which needs to perform additional calculations for building the sum of the interface nodes. In the setting of a high performance cluster with equally powerful processors, this is not a good choice. In contrast, each process could generate an array of the size of all global interface nodes. Therefore, a **MPI_Allreduce** operation would be sufficient. Yet this would lead to a major increase of the message sizes, since every process would need to send messages in the size of the total number of interface nodes instead of the number of its own interface nodes. Therefore, a new communication concept was created so that each process only sends its interface nodes to the respective processes and performs the sum for its local interface nodes itself. This concept is described below.

In algorithm 4.3, each process determines which entries it needs to share with which processes. Therefore, the following input data is provided:

- **Influencing processes per node**: A list of vectors (one per node) that store all processes influencing that node.
- **Local interface nodes**: A globally sorted vector of all interface nodes of the current subdomain. It is important that this vector is sorted on a global basis, so that all processes construct their communication data in the same order.
- **Local to global index transformer**: A vector that maps each local node-ID to the global ID of that node.

- **Reduced Dirichlet index transformer**: A vector that maps each DOF in the unreduced subdomain to the corresponding DOF in the subdomain after the reduction of the Dirichlet DOFs (as described in section 2.1) was performed.

Iterating over all interface nodes, each process then creates a data structure to store its interface DOFs consisting of two vectors. The first vector includes the IDs of all adjacent processes in ascending order. Every entry in the second vector corresponds to the process at the same position in the first vector and points to another vector that stores the common DOFs with the corresponding process. This method is called once before the CG algorithm and Jacobi preconditioning takes place.

Algorithm 4.3: Process interface DOFs

```
Input: influencing processes per node, local interface nodes, local to global index
       transformer, reduced Dirichlet index transformer
foreach local interface node $n$ do
    calculate global node-ID $\tilde{n}$;
    foreach influencing process $p$ of $\tilde{n}$ do
        save $p$ into a vector of local interface processes;
        for 1, ..., DOF do
            calculate local DOF-ID $d_{\text{red}}$ after Dirichlet DOFs removal;
            if $d_{\text{red}}$ exists then
                save $d_{\text{red}}$ in the corresponding vector of interface DOFs;
return vector of local interface processes, vectors of corresponding interface DOFs
```

Using the provided interface data, the accumulated vectors can easily be obtained by three loops over the individual interface processes as shown below.

Algorithm 4.4: Communication of local interface entries

```
Input: local vector $\boldsymbol{z}$, interface information as returned by 4.3
foreach local interface process do
    start receiving the data in a corresponding receive buffer using the non-blocking
      MPI_Irecv;
foreach local interface process do
    copy the corresponding data into a send buffer;
    start sending the data in the corresponding send buffer using the non-blocking
      MPI_Isend;
for 1, ..., number of local interface processes do
    wait until the receiving of any process has finished using the blocking
      MPI_Waitany;
    add the data from the corresponding receive buffer to the local vector $\boldsymbol{z}$;
wait until all calls to MPI_Isend have finished using the blocking MPI_Waitall;
return accumulated vector $\tilde{\boldsymbol{z}}$ with global contributions
```

Non-blocking send and receive methods are used for the communication in order to provide a highly parallel method. By using blocking send and receive calls instead, some ordering of the communication would be necessary in order to prevent deadlocks (e.g., each process receives first from the processes with a lower rank, then sends the data and afterwards receives from all processes with a higher rank). This, however, would cause the communication to become serial, since process 2 would need to wait until it has received the data from process 1 before it can send to process 3, and so on. Of course in a three-dimensional environment, where every process has several different interface nodes, this would not necessarily cause the whole communication to be serial. However, the amount of unnecessary waiting times would certainly be high.

During each iteration, the data which shall be exchanged is copied into a designated send and from a designated receive buffer in order to provide MPI with coherent data structures and to assure that no data is changed before it is sent. It should be noted here, that MPI also provides so-called "derived types" where the user can define a data structure which is not coherent in memory that can then be used for arbitrary MPI calls. While these data types are very efficient once they are set up, their creation is very expensive [40]. In order to provide out-of-order receiving, it is still necessary to use buffers. For the sending procedure, such data types could be used instead of copying the data into a buffer. However, it would be necessary to finish sending before the received data can be added to the given vector. Results have shown that both concepts need almost the same amount of time. Generally the use of derived data types is even slightly slower. Therefore, the communication concept as shown in algorithm 4.4 was used for all further computations.

4.3 Assembly of the full solution

In order to obtain the full solution in the root process, communication is necessary one last time. Therefore, the blocking routines **MPI_Gather** and **MPI_Gatherv** are used. In contrast to **MPI_Gather**, **MPI_Gatherv** gathers the received data into specified memory locations for each process. Thereby each process can send a different amount of memory [40].

In algorithm 4.5, first the number of nodes is collected from each process. Therefore, the total number of entries in a concatenated index transformer as well as a concatenated subsolution vector is obtained. This information is necessary for the use of **MPI_Gatherv** to specify the memory locations into which the received data should be stored. After all index transformers and subsolutions were received, the root process iterates over each entry in the global index transformer and saves the value of the concatenated subsolution vector into the corresponding entry of the global solution vector. Since the results produced by the CG algorithm (4.2) are global, no adding up of the values is necessary and multiple returned results of the same DOF can just be ignored. In the case where static condensation has been performed according to section 2.2, the root process does not know which nodes were condensed. Hence, it first assembles all received values into a full solution vector and afterwards removes all nodes for which it did not receive a value.

However, only the root process needs to obtain the full solution. Therefore, all other processes only perform algorithm 4.6, shown below.

Algorithm 4.5: Assembly of the full solution vector - root process

```
Input: global solution of the subdomain $\widetilde{\boldsymbol{u}_p}$, index transformer from local to global
       node-ID
Collect the number of nodes from each process using MPI_Gather;
Create a vector of memory offsets as needed by MPI_Gatherv;
Concatenate the index transformers of each process using MPI_Gatherv;
Update the vector of memory offsets to fit the subsolutions;
Concatenate the subsolutions of each process using MPI_Gatherv;
foreach entry in the concatenated index transformer do
    calculate the global node index;
    foreach DOF do
        save the result into the correct position of $\tilde{\boldsymbol{u}}$;
if static condensation was performed then
    foreach global node do
        if no result for this node has been received then
            remove node from $\tilde{\boldsymbol{u}}$;
return $\tilde{\boldsymbol{u}}$
```

Algorithm 4.6: Assembly of the full solution vector - other processes

```
Input: global solution of the subdomain $\widetilde{\boldsymbol{u}_p}$, index transformer from local to global
       node-ID
Send the number of nodes in the subdomain using MPI_Gather;
Send the index transformer of the subdomain using MPI_Gatherv;
Send the subsolution of the subdomain using MPI_Gatherv;
```

Alternatively the root process could also iterate over each process and directly assemble the subsolution into the global solution vector as shown in algorithm 4.7. As collective MPI calls such as **MPI_Gather** are faster than iteratively calling **MPI_Send**, the first method should be preferred and was therefore used for all further computations. However, since this communication is only performed once for each simulation, it is not a major performance issue and both algorithms produce satisfying results.

Algorithm 4.7: Alternative assembly of the full solution vector - root process

```
Input: global solution of the subdomain $\widetilde{\boldsymbol{u}_p}$, index transformer from local to global
       node-ID

foreach process $p$ do
    Receive $\widetilde{\boldsymbol{u}_p}$ and the index transformer of $p$ using the blocking MPI_Recv;
    assemble $\widetilde{\boldsymbol{u}_p}$ into the global solution vector $\widetilde{\boldsymbol{u}}$;

if static condensation was performed then
    foreach global node do
        if no result for this node has been received then
            remove node from $\widetilde{\boldsymbol{u}}$;

return $\widetilde{\boldsymbol{u}}$
```

5 Timings and efficiency

According to Amdahl's law [13], the maximum possible speedup s of a program depends on the fraction of parallel code f_P and is given as

$$s = \frac{1}{1 - f_\mathrm{P}}. \tag{5.1}$$

Therefore, if 100% of the code can be run in parallel, the theoretical speedup grows up to infinity. The speedup for a given number of processors can be approximated by

$$s = \frac{1}{\frac{f_\mathrm{P}}{p} + f_\mathrm{S}}, \tag{5.2}$$

with f_P being the parallel and f_S the serial fraction of the code. p is the number of used processors. Fig. 21 shows the theoretical speedup over the number of processors for different amounts of parallel code.

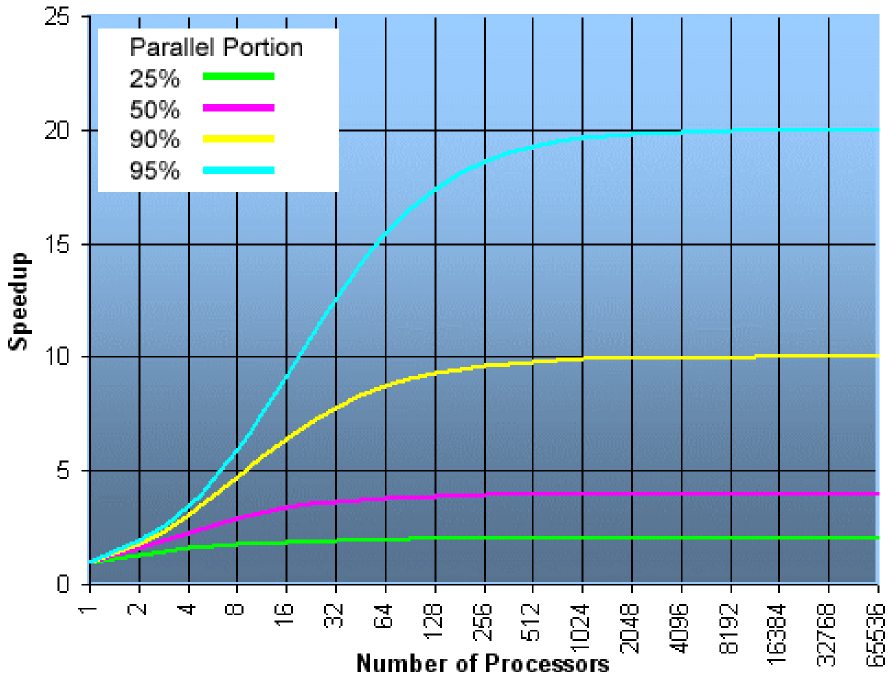


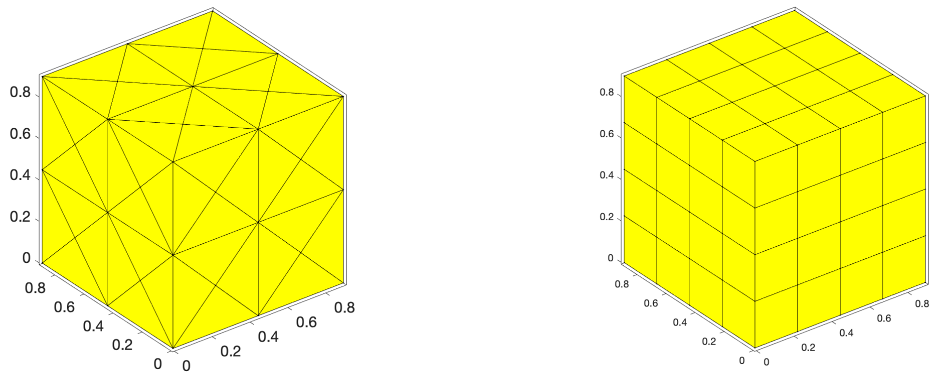
Fig. 21: Amdahl's law - limits of potential program speedup [13].

In this chapter, efficiency studies for the implemented parallelization of the considered simulations – elastic continua and heat flows – are presented. The speedup of individual parts of the simulations as well as the total simulation times are discussed. Furthermore, the amounts of necessary communication compared to the computation are evaluated.

One distinguishes weak and strong scalability [13]. When investigating *strong* scalability, a concrete problem is specified and kept fixed (in particular, the mesh remains unchanged) while increasing the number of processors. The behavior is then expected to follow Amdahl's law mentioned before. For *weak* scalability, one increases the problem size (in particular, the meshes are refined) proportionally to the number of processors and expects a *constant* execution time in the optimal situation. In this work, the focus is on strong scalability.

5.1 Efficiency studies on a unit cube

The initial efficiency studies were performed on a unit-cube with an equal number of elements in each coordinate direction. Fig. 22 shows two coarse example meshes of the given domain. However, the meshes used in the efficiency studies were of course much finer.



(a) tetrahedral elements (b) hexahedral elements

Fig. 22: Two example meshes of the considered domain for the initial efficiency studies.

Simulations for linear, cubic and 6th-order elements were performed with different numbers of elements in order to study the achieved efficiency. In this chapter, only some of the results are provided. However, the other results show a similar behavior.

5.1.1 Simulations of linear-elastic continua

The following figures show the timings for simulations of an elastic continuum. Tab. 1 provides additional information on the simulations underlying the following plots. The reason to pick these concrete problem specifications is seen in the fact that the resulting simulations shall still be computable on a single processor in reasonable time, say at most in one day. It is also noted that static condensation only applies to some of the higher-order elements, as linear elements do not have inner nodes to be condensed.

		normal		static condensation	
type	**order**	**DOFs**	**CG iterations**	**DOFs**	**CG iterations**
tetra	1	1.7 M	759		
tetra	3	1.4 M	1380		
tetra	6	1.4 M	3927	1.0 M	2591
hexa	1	3.2 M	798		
hexa	3	1.8 M	1320	1.3 M	812
hexa	6	0.35 M	5578	0.15 M	1312

Tab. 1: DOFs and number of CG iterations of the performed simulations of an elastic continuum.

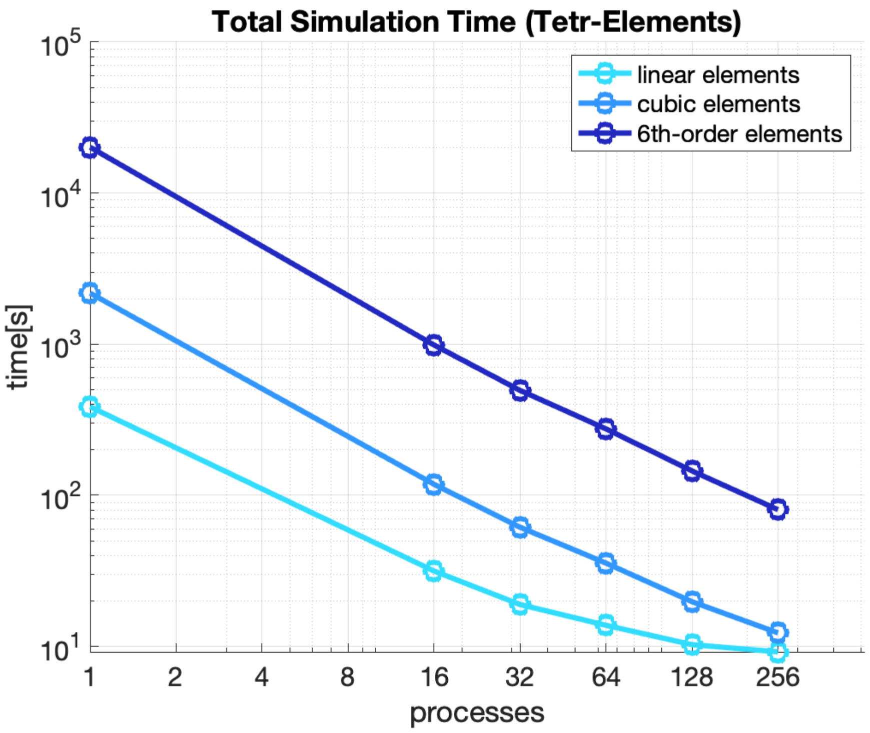
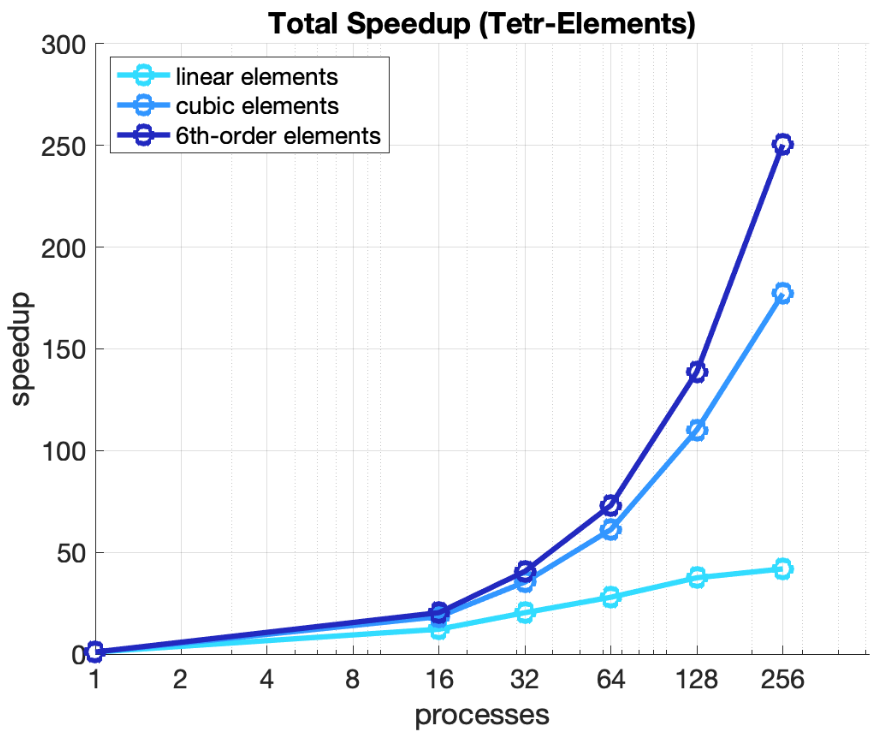
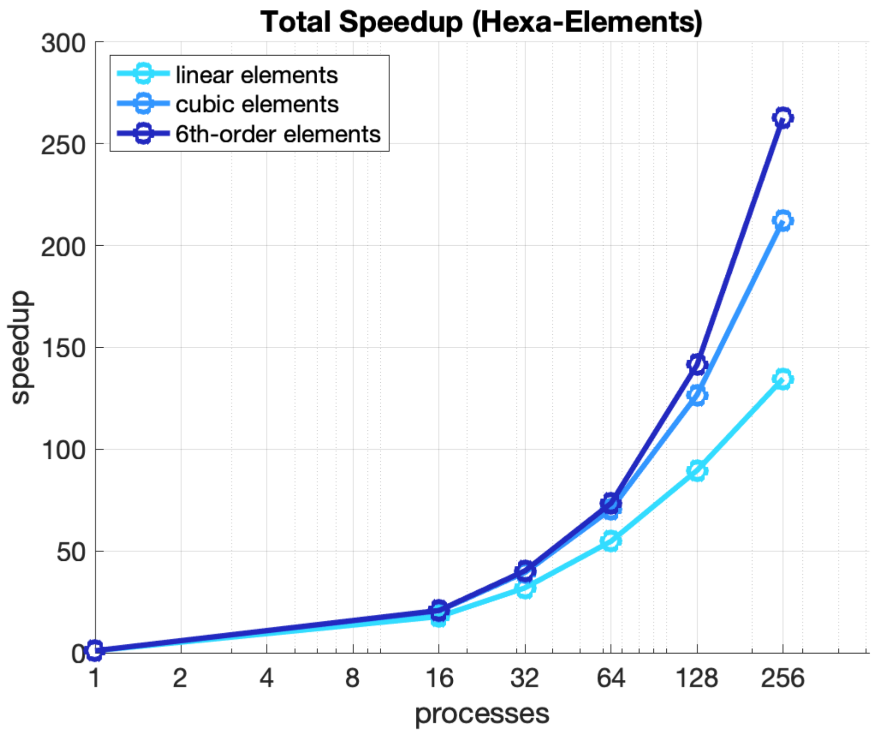


Fig. 23: Total time of simulations of an elastic continuum with linear, cubic and 6th-order tetrahedral elements.



(a) tetrahedral elements



(b) hexahedral elements

Fig. 24: Total speedup of simulations of an elastic continuum with linear, cubic and 6th-order elements.

Fig. 24 shows that very high speedups were achieved. For example, for 6th-order elements, the speedup for 256 cores was larger than 250 for tetrahedral and hexahedral elements, which is remarkable. Concerning the simulation with linear tetrahedral elements, for more than 128 processes, the speedup begins to stagnate. However, at that point, the preprocessing takes 16 seconds, whereas the solving time was reduced from the initial 1041 seconds to 10 seconds. Apart from the fact that a simulation time of 26 seconds is more than sufficient, the remaining computational effort per process is too small to be decreased effectively (remember that the problem size does not change during the studies, i.e., strong scalability is investigated herein). Studies in section 5.2 show that also for linear elements, higher speedups can be produced. However, such simulations are mostly too big to be computed in serial (to obtain some reference time for comparison), wherefore they were not considered here.

Tab. 2 shows the produced speedups for 256 processes. Furthermore, it provides the approximate fraction of parallel code according to the rearranged relation of Amdahl:

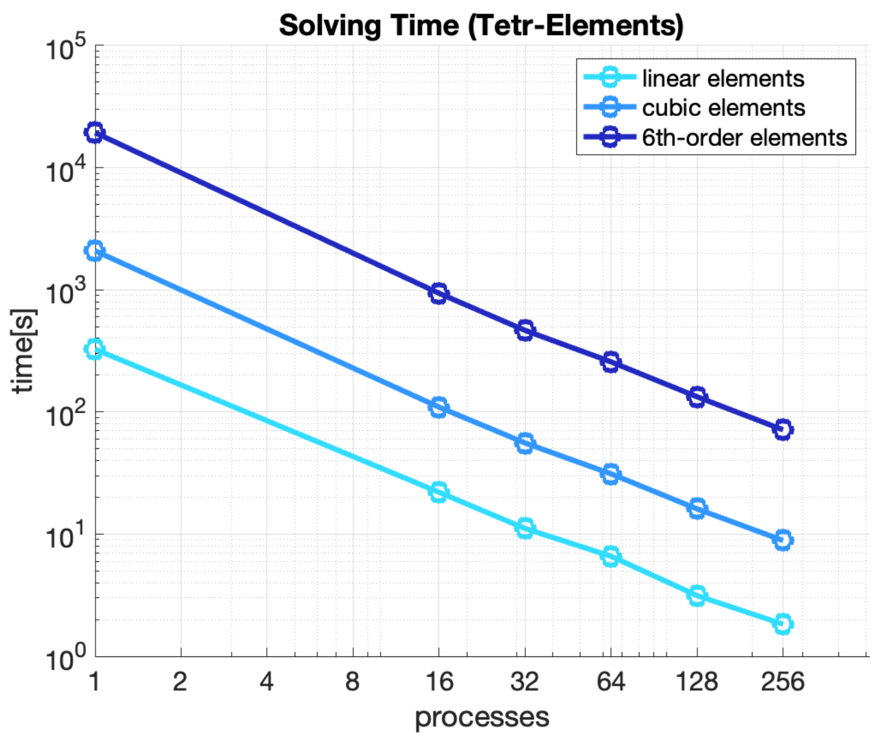
$$f_\mathrm{P} = \frac{p - s \cdot p}{s \cdot (1-p)} \qquad \forall p > 1. \tag{5.3}$$

This table demonstrates that the fraction of parallel code according to this formula is only an approximate value and varies slightly. It depends on the percentage of the computation that is used for the parallel assembly and solving of the system of equations in contrast to the serial preprocessing (creation of the mesh, domain decomposition). This relation, however, depends on different factors, e.g., type of the simulation, size of the problem, order of the elements, connectivity of the elements and type of the elements (hexahedral or tetrahedral). Tab. 2 clearly confirms the successful parallelization achieved as all values for $\boldsymbol{f_\mathrm{P}}$ are close to 100%.

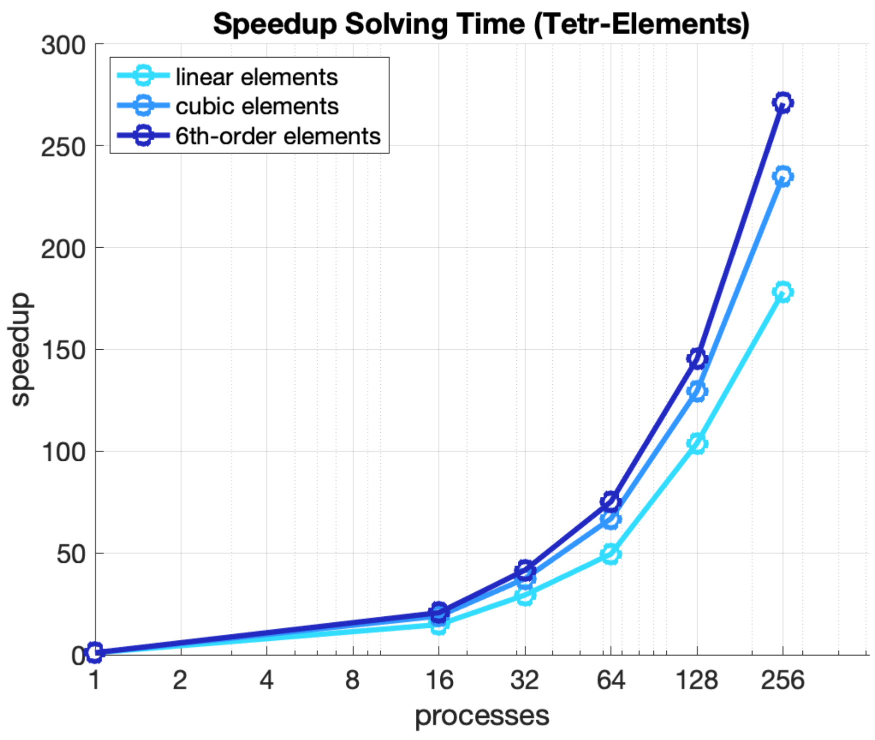
type	**order**	**speedup 256 procs.**	$\boldsymbol{f_\mathrm{P}}$
tetra	1	~ 42	98.00%
tetra	3	~ 177	99.82%
tetra	6	~ 250	99.99%
hexa	1	~ 134	99.64%
hexa	3	~ 212	99.92%
hexa	6	~ 262	100.00%

Tab. 2: Approximated fraction of parallel code according to Amdahl's law.

Let us now study some timings of *individual* tasks in the overall simulation rather than the total time. Fig. 25 shows the *solving time* as well as the speedup of the solving time when using tetrahedral elements. It can be seen that the speedup grows linear with the amount of processes and does yet not begin to flatten. Again, depending on the computational effort of the solution, the slope of the speedups is slightly different for linear, cubic and 6$^{\text{th}}$-order elements.

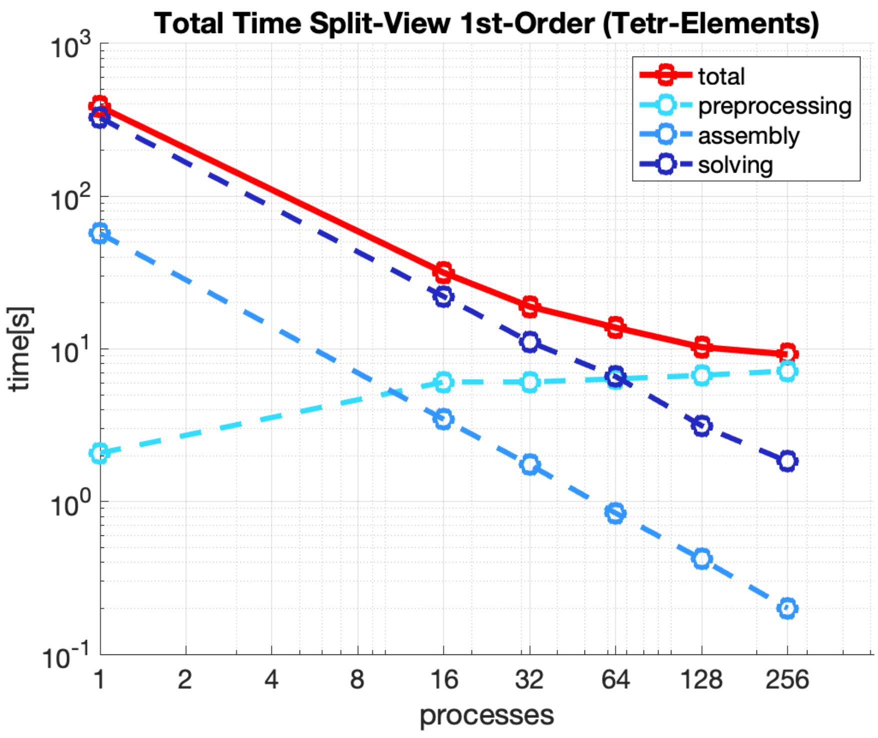


(a) solving time

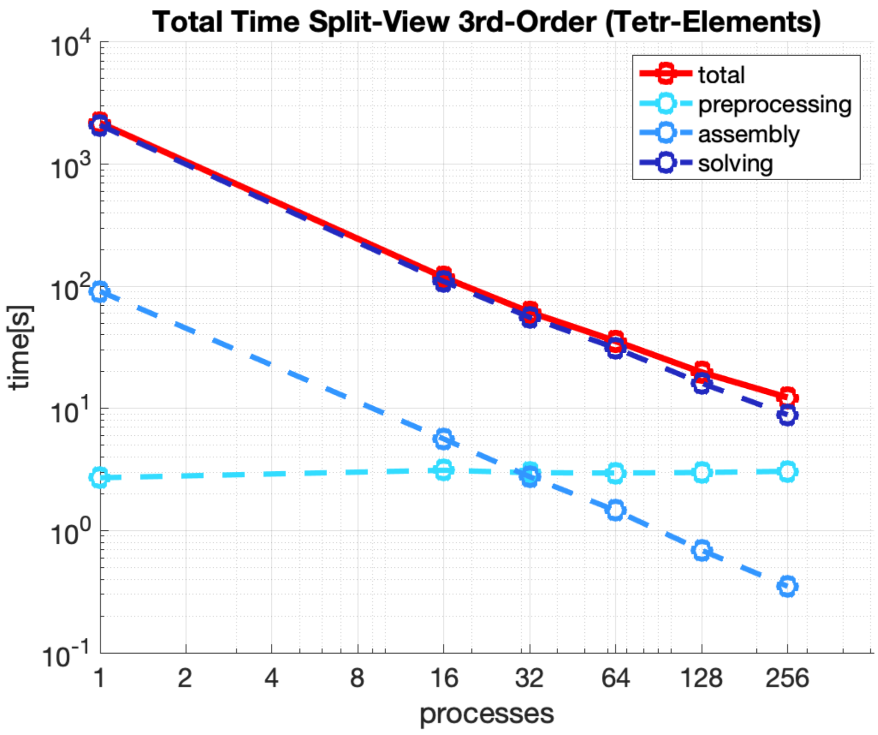


(b) solving speedup

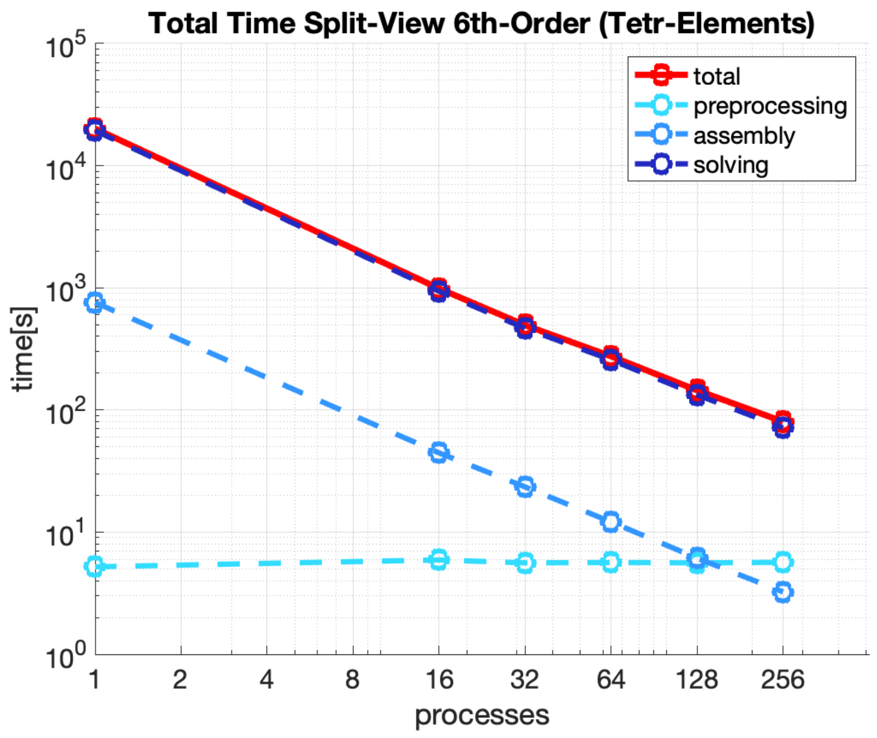
Fig. 25: Solving time and speedup of simulations of an elastic continuum with linear, cubic and 6th-order tetrahedral elements.



(a) 1st-order elements



(b) 3rd-order elements

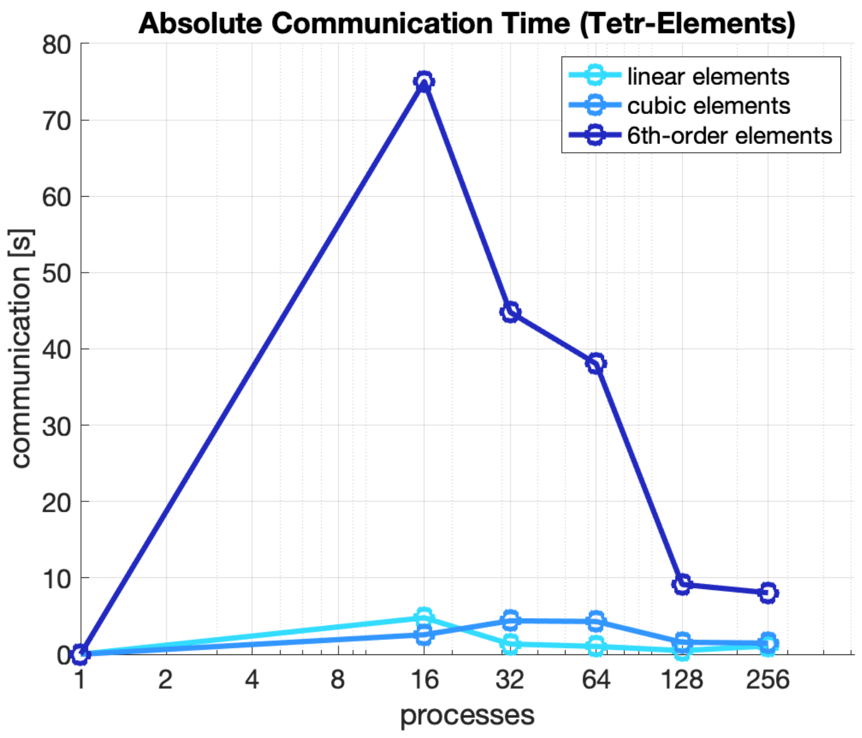


(c) 6th-order elements

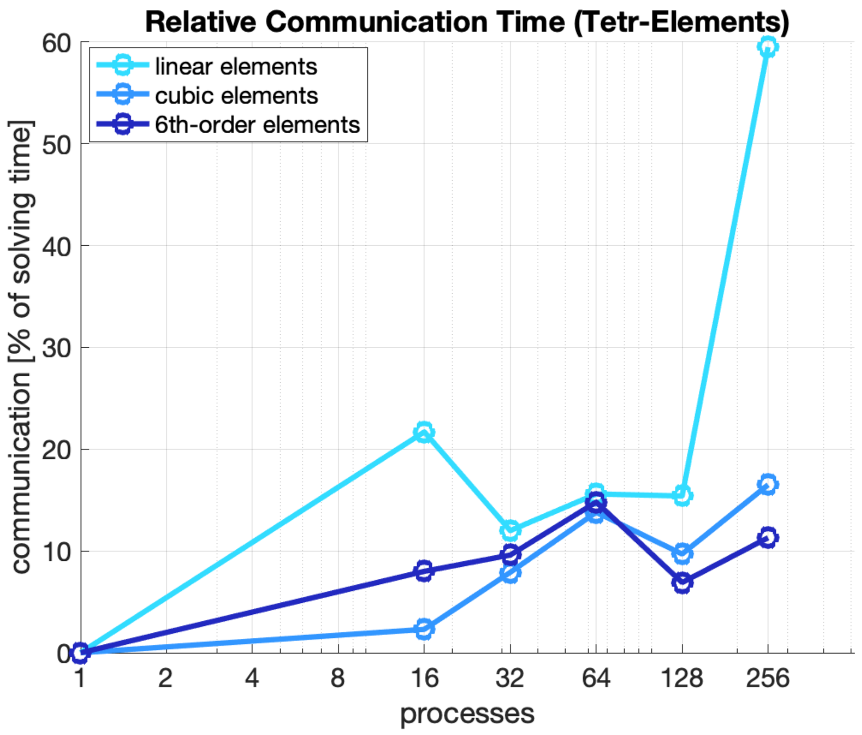
Fig. 26: Split view of the total time of simulations of an elastic continuum with linear, cubic and 6th-order tetrahedral elements.

Fig. 26 shows the composition of the total simulation time composed into the preprocessing, assembly and solving time for linear, cubic and 6th-order elements. For the parallel simulations, the assembly time corresponds to the time needed for assembling the sub-system matrices and sub right-hand sides. It clearly shows the contributions of these individual tasks as well as the overall reduction of time and therefore the total speedup. For 6th-order elements the vast majority of the total simulation time is caused by the solving. Since the solving procedure is parallel and shows a very high speedup as shown before, also the speedup of the overall simulation time is almost perfect. In contrast, for linear elements the preprocessing time starts to dominate for more than about 64 processes.

Next, the focus is on the communication time. It was found that approximately 10% − 25% of the solving time is needed for communication, see Fig. 27. In general, with an increasing number of processors a slight growth of the relative amount of communication is observed. The amount of required communication mostly depends on the quality of the domain decomposition. Fig. 27 shows the absolute as well as relative communication time in the solving procedure. Again, for linear elements the increase of the relative amount of communication can be quite high, but this is mostly due to the fact that the absolute solving time for 256 processes is rather low (~ 7 seconds).

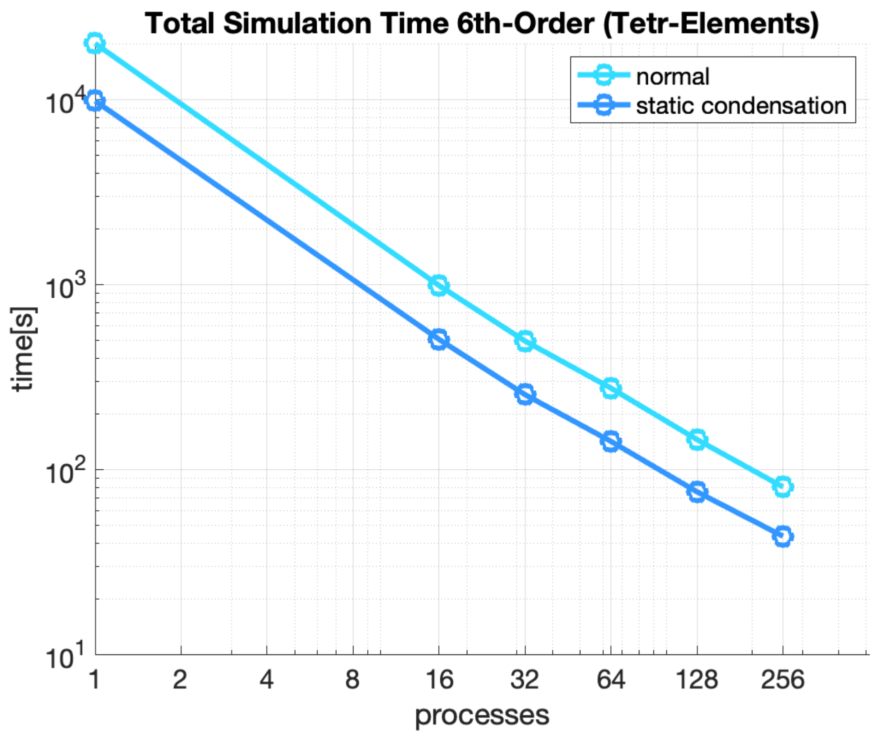


(a) absolute communication

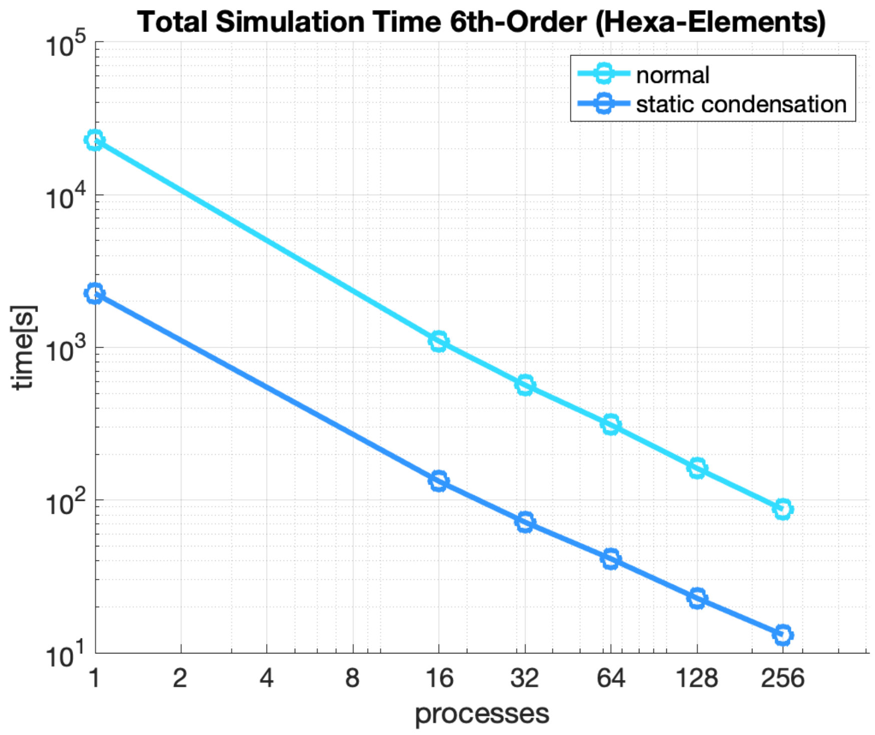


(b) relative communication

Fig. 27: Amount of required communication in the solving procedure of simulations of an elastic continuum with linear, cubic and 6th-order tetrahedral elements.



(a) tetrahedral elements



(b) hexahedral elements

Fig. 28: Total time of simulations of an elastic continuum with 6$^{\text{th}}$-order elements with and without static condensation. Note that for 6th-order elements, the simulation time for static condensation has decreased by a factor of about 2 for tetrahedral and about 10 for hexahedral elements.

Finally, the effect of static condensation is analysed. For higher-order elements, static condensation can significantly reduce the size of the system of equations and hence the number of required iterations and the computation time. For the simulations with 6th-order elements the number of iterations was reduced by $\sim 34\%$ for tetrahedral elements and $\sim 76\%$ for hexahedral elements (Tab. 1, p. 43).

Fig. 28 shows the total time of the simulation with 6th-order elements with and without static condensation. It can be observed that both simulations show a similar behavior in the context of reduction of simulation time with an increasing number of processes. However, the total simulation time for the simulations with static condensation is lower by a factor k. For the simulation with 6th-order *tetrahedral* elements, $k \approx 2$. For the simulation with 6th-order *hexahedral* elements, $k \approx 10$. Thus, it can be seen that static condensation is more effective for hexahedral elements which is obviously due to the smaller factor of the number of outer nodes versus the total number of element nodes in hexahedra compared to tetrahedra.

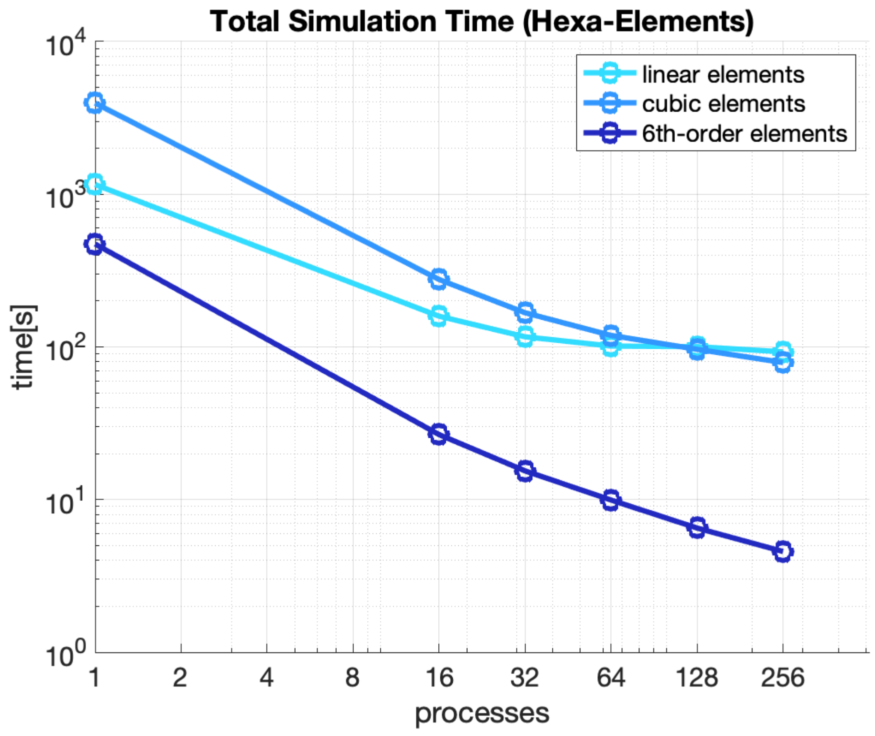
5.1.2 Simulations of heat flows using the Poisson equation

Next, we consider heat flow based on the Poisson equation and present timings and speedups similar to the previous section; the domain is still the unit cube. In the following table (Tab. 3), some information on the simulations is provided.

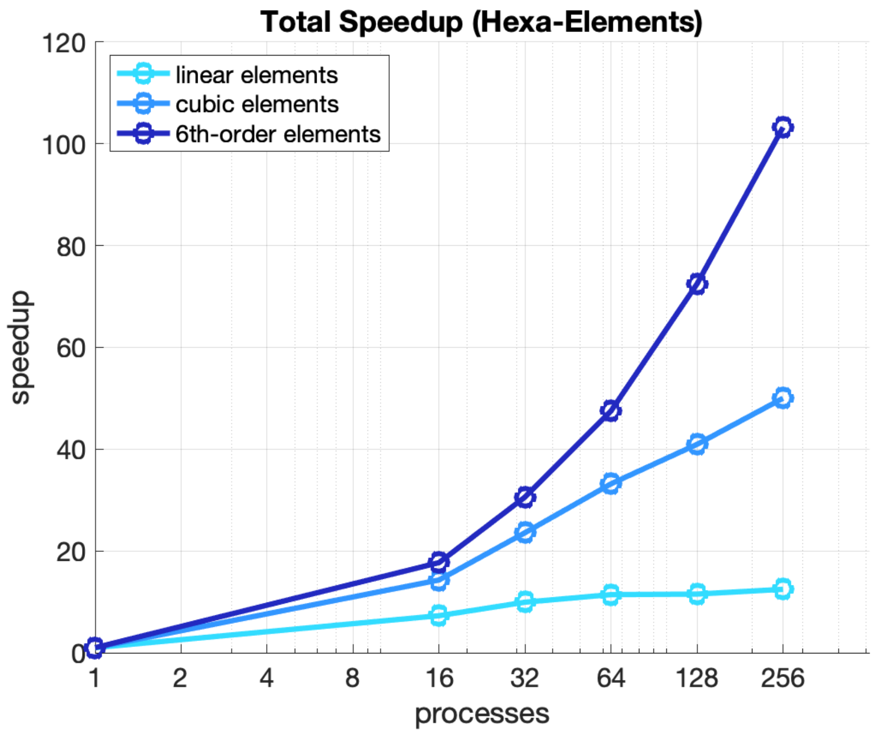
		normal		**static condensation**	
type	**order**	**DOFs**	**CG iterations**	**DOFs**	**CG iterations**
tetra	1	13.6 M	558		
tetra	3	12.1 M	901		
tetra	6	5.0 M	1791	3.6 M	1292
hexa	1	16.6 M	358		
hexa	3	7.0 M	590	4.9 M	374
hexa	6	0.17 M	2977	0.07 M	801

Tab. 3: DOFs and number of CG iterations of the performed simulations of a heat flow.

In Fig. 29 a similar behavior to section 5.1.1 is observed. However the achieved speedups for the simulations of the elastic continuum are higher than for the simulations of the heat flow. Even for a very large simulation (16 million DOFs) with linear elements using more than 64 processes does not provide a significant speedup. However, with 64 processes the total simulation time is already only ~ 100 seconds, with a solving time of only ~ 25 seconds.

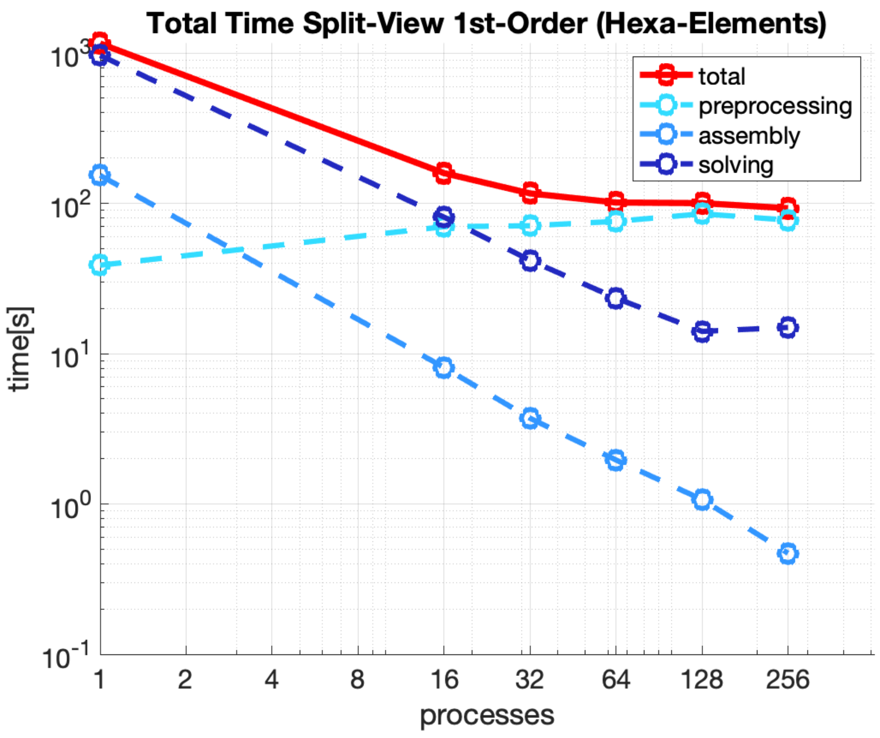


(a) simulation time

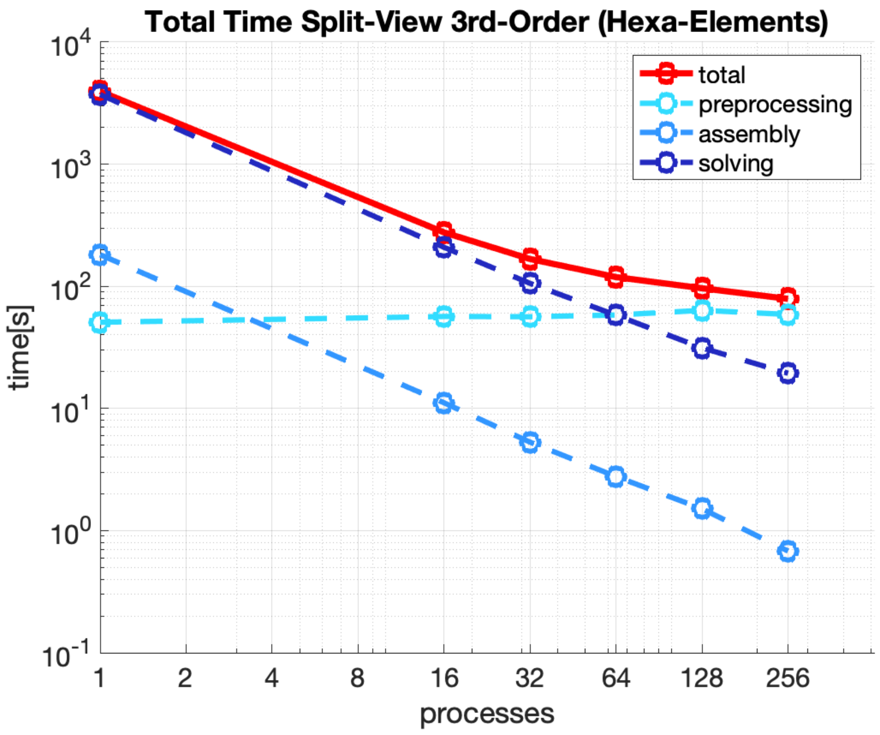


(b) speedup

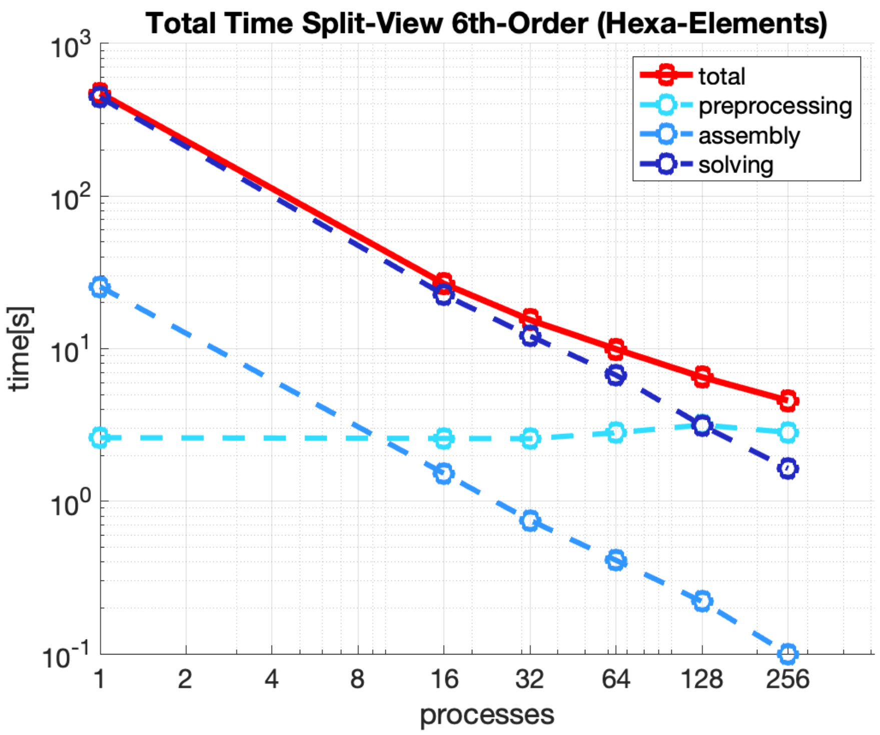
Fig. 29: Total time and speedup of simulations of a heat flow with linear, cubic and 6$^{\text{th}}$-order hexahedral elements.



(a) 1st-order elements



(b) 3rd-order elements



(c) 6th-order elements

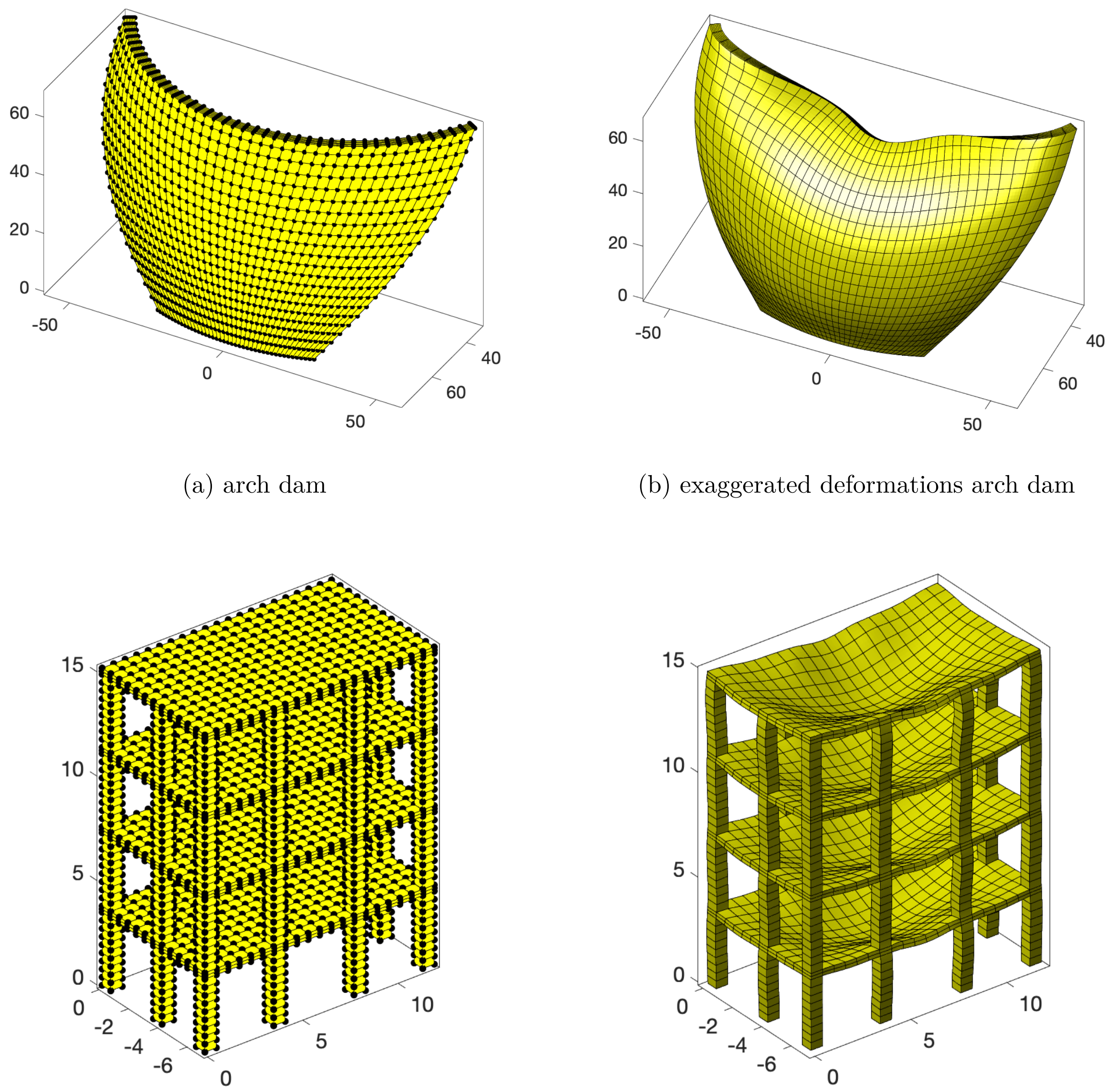
Fig. 30: Split view of the total time of simulations of a heat flow with linear, cubic and 6th-order hexahedral elements.

Fig. 30 shows that for linear elements, the preprocessing time already dominates the solving time at a number of only 32 processes. When comparing Fig. 29 and Fig. 30 it can be observed that the inflection point of the speedup lies approximately at the point, where the preprocessing starts to dominate the solving time of the simulation.

Summarizing the above results, the following behavior can be observed. The produced speedups for the elastic continuum are better than for the heat flow. This is due to the fact that for the elastic continuum, the system of equations is larger so more computational effort is shifted to the solver which performs close to optimal in the parallel setup. This is because the considered number of DOFs per node for the heat flow is only one whereas for the elastic continuum in three dimensions, three DOFs per node are considered. Both BVPs show that the speedup of simulations with higher-order elements is better than with linear elements.

5.2 Efficiency studies in general domains

In addition to the provided studies, efficiency studies for elastic continua are also conducted in more realistic domains. The first domain is an arch dam under dead- and water loads. The second domain is a building under dead- and payloads. Fig. 31 shows the two domains, discretized by a coarse mesh composed by linear elements. In order to better understand the computed results, the deformed configurations are shown next to the meshes.



(a) arch dam

(b) exaggerated deformations arch dam

(c) building

(d) exaggerated deformations building

Fig. 31: Example meshes for the considered real life domains. The undeformed domains are displayed on the left, the deformed configurations on the right.

5.2.1 Arch dam

The following table provides additional information on the simulations of the dam. The systems of equations of the conducted simulations have a very high number of DOFs and non-zero entries. Therefore, no *serial* (i.e., single core) simulations could be performed due to the immense memory requirements. As a result, the numbers of DOFs for the systems of equations after static condensation are not available, since the full system matrices are never assembled in the parallel simulations and no communication routine was used to determine the resulting reduced number of DOFs.

		normal		**static condensation**
type	**order**	**DOFs**	**CG iterations**	**CG iterations**
tetra	1	1.2 M	4.1 K	
tetra	3	3.6 M	9.9 K	
tetra	6	3.5 M	27.8 K	19.1 K
hexa	1	9.7 M	8.7 K	
hexa	3	4.1 M	12.3 K	6.6 K
hexa	6	0.41 M	92.4 K	15.8 K

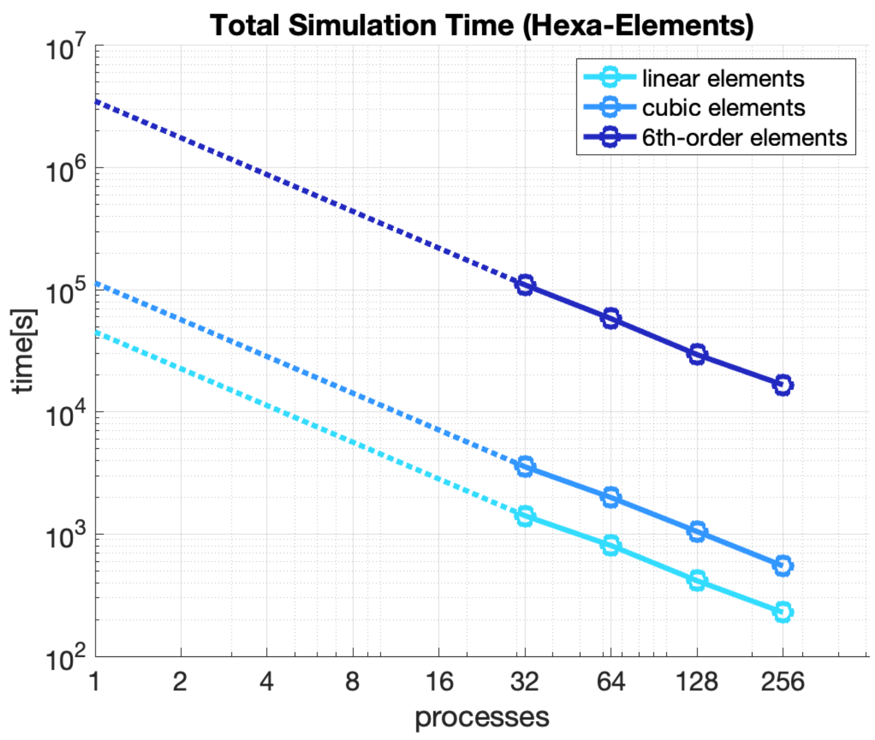
Tab. 4: DOFs and number of CG iterations of the performed simulations of the dam.

In order to be able to compute a reasonable speedup even though no serial simulations were performed, the times for the serial simulations were approximated. Therefore, the presented speedups may vary slightly from the actual speedups. Let t_{32} be the time for the simulation with 32 processes and t_1 the approximated time for the serial simulation. Then t_1 is simply approximated as follows:

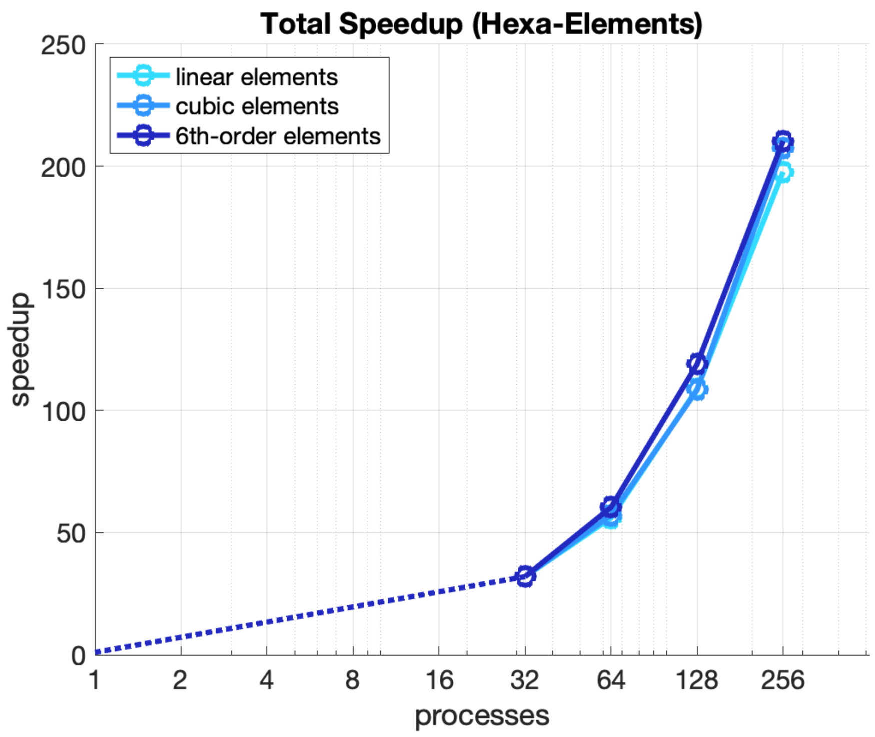
$$t_1 = 32 \cdot t_{32}. \tag{5.4}$$

This is equal to assuming an optimal speedup from the serial simulation time to the time with 32 processes. This is justified by the studies conducted above indicating (close to) optimal speedups for 32 processes. This approximation of t_1 is also confirmed by the fact that the inclinations of the extrapolated lines are equal to the inclinations of the actual simulation data. The extrapolated values are represented by dotted lines and are thus clearly distinguished from the actually measured data. It is seen that for the 6th-order hexa elements, $t_1 \approx 3.5 \cdot 10^6$s, indicating a simulation taking about 40 days on a single core!

From the following figures (Fig. 32, Fig. 33 and Fig. 34) it can be seen, that the simulations resemble the behavior described in section 5.1. However, the speedup for the simulation with linear elements is seen to be much higher, due to the larger simulation. Fig. 33 shows that in contrast to the linear simulations in section 5.1, the vast majority of the simulation time is now caused by the solving. Since the speedup of the solving time grows linearly with the number of processors, also the speedup of the total simulation time does.



(a) simulation time



(b) speedup

Fig. 32: Total time and speedup of simulations of the dam with linear, cubic and 6$^{\text{th}}$-order hexahedral elements.

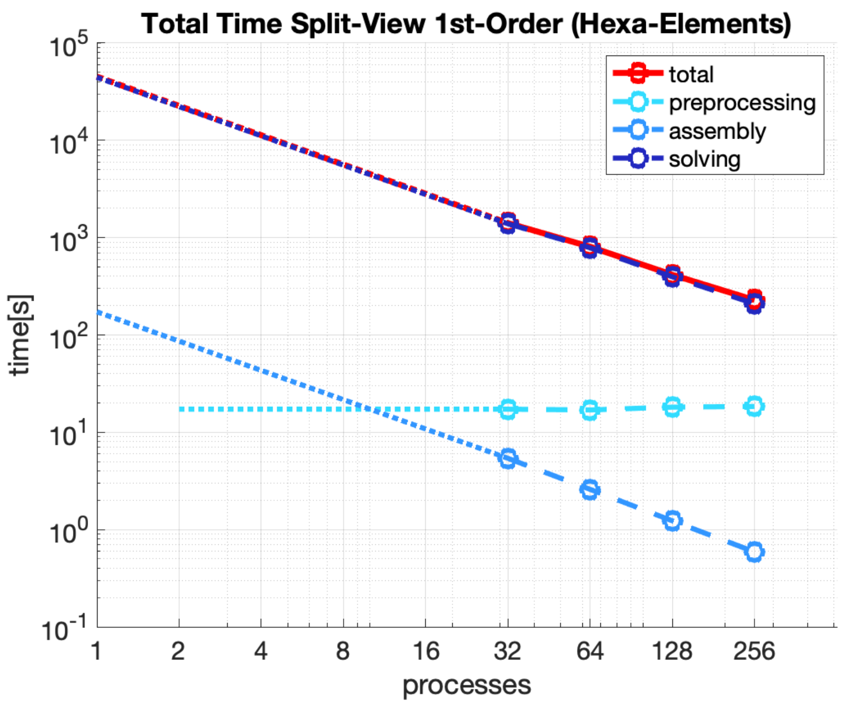


Fig. 33: Split view of the total time of simulations of the dam with linear hexahedral elements.

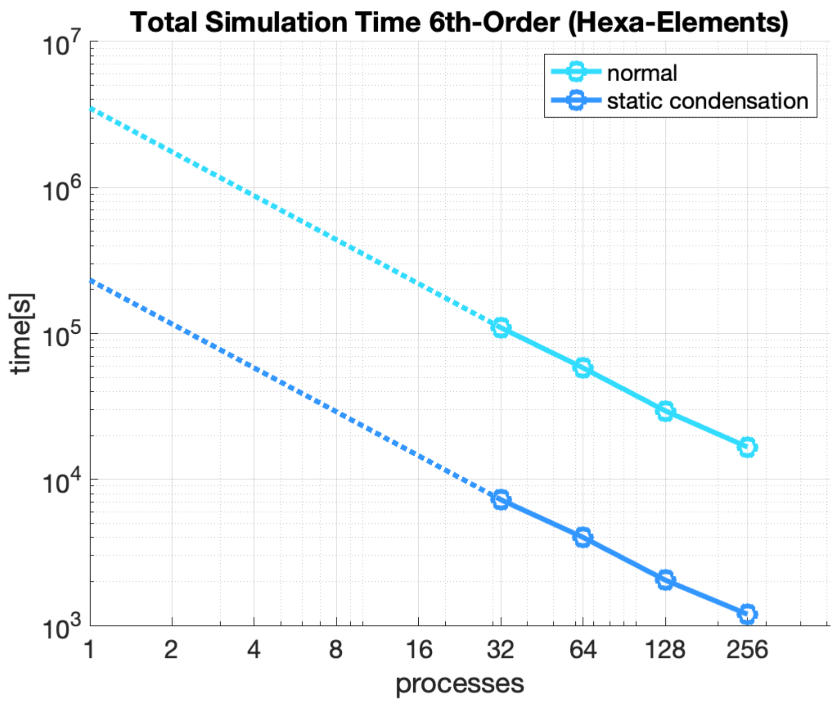


Fig. 34: Total time of simulations of the dam with 6$^{\text{th}}$-order hexahedral elements with and without static condensation.

Furthermore, the simulation time *per DOF* was investigated and is shown in Fig. 35. In an optimal simulation the inclination of the $\frac{\text{time}}{\text{DOF}}$ plotted over the number of processes should be -1 (log-log scale). In other words, when doubling the number of processes the $\frac{\text{time}}{\text{DOF}}$ should be halved. This optimal behavior is shown by the black dashed reference lines in Fig. 35. A close to optimal behavior can be observed. It can also be seen that the amount of required time per DOF is larger for higher-order elements. This corresponds to the number of required iterations provided in Tab. 4. The higher effort in solving is caused by the denser system of equations with a higher number of non-zero entries. It is noted that the results in Fig. 35 are only based on the measured data and do not involve the extrapolated value of t_1.

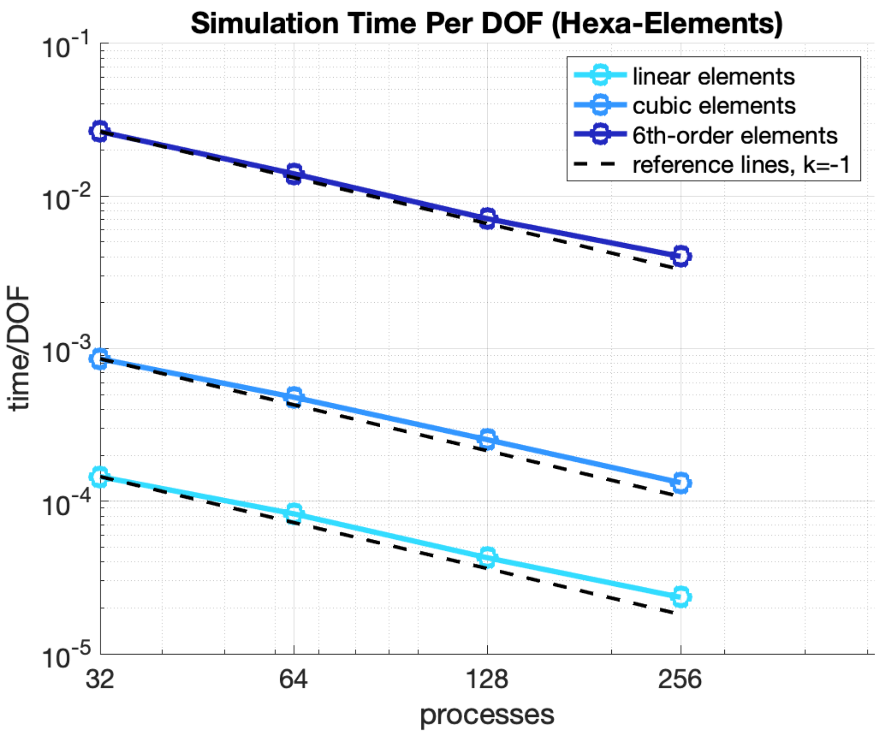


Fig. 35: Total time per DOF of simulations of the dam with linear, cubic and 6th-order hexahedral elements.

5.2.2 Multi-story building

In order to analyse the speedup of large simulations even better, also a run with 512 processes was performed. Again, due to the huge systems of equations no serial simulations were conducted. Tab. 5 provides additional information on the simulations of the building.

		normal		static condensation
type	**order**	**DOFs**	**CG iterations**	**CG iterations**
tetra	1	2.0 M	29.0 K	
tetra	3	6.4 M	67.5 K	
tetra	6	6.4 M	180.0 K	116.0 K
hexa	1	29.5 M	58.2 K	
hexa	3	1.7 M	41.3 K	22.0 K
hexa	6	0.17 M	312.1 K	52.7 K

Tab. 5: DOFs and number of CG iterations of the performed simulations of the building.

Once more, a close to optimal behavior can be observed, when analyzing the simulation time per DOF (Fig. 36). Even for a total of 512 processes, the speedup does not yet flatten (Fig. 37) and the communication time (Fig. 38) can still be kept relatively low ($< 16\%$). The slight kink in the simulation time at 256 processes is also seen in the solving time, but not in the communication time. Therefore, it might result from a domain decomposition with a slightly worse load-distribution.

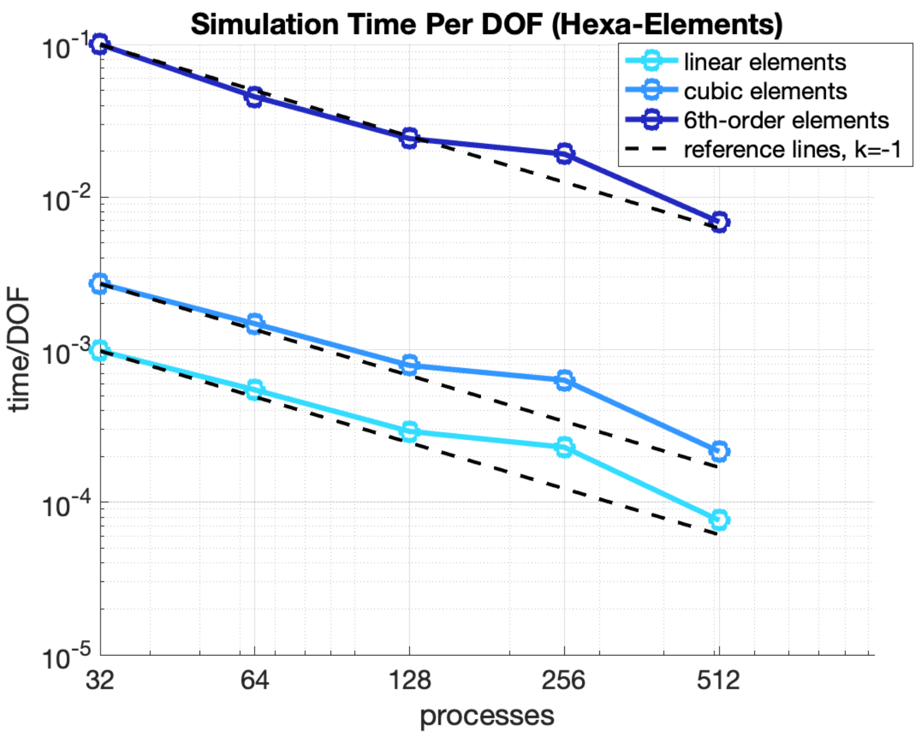
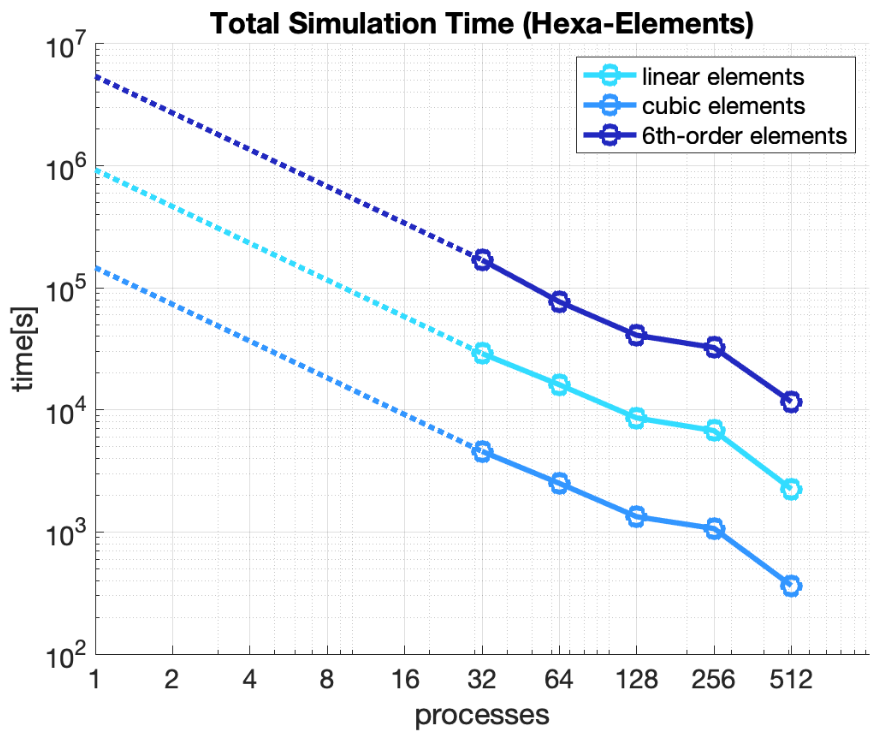
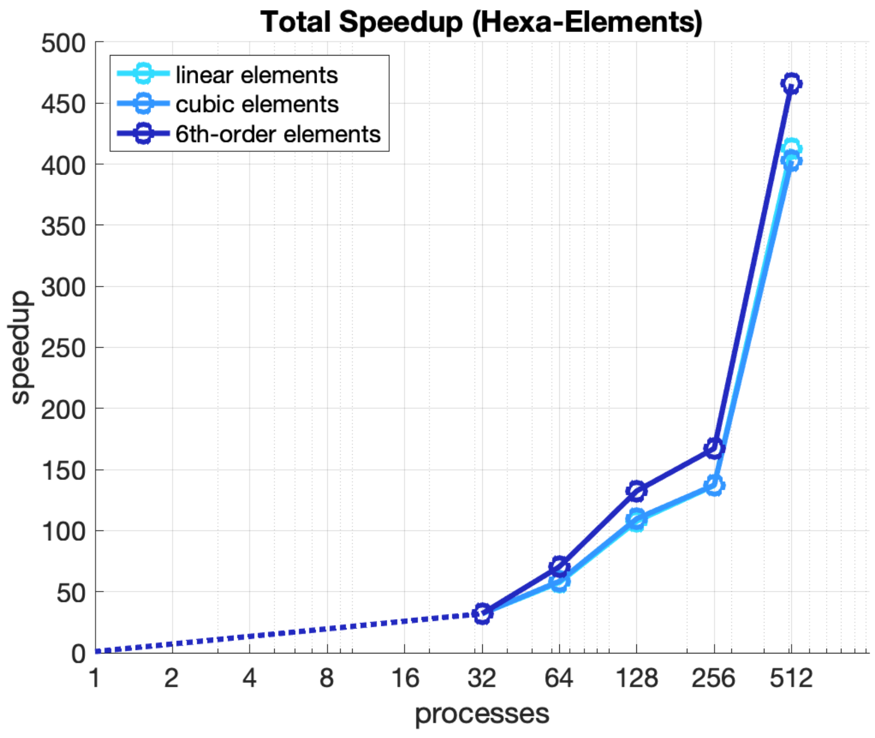


Fig. 36: Total time per DOF of simulations of the building with linear, cubic and 6th-order hexahedral elements.

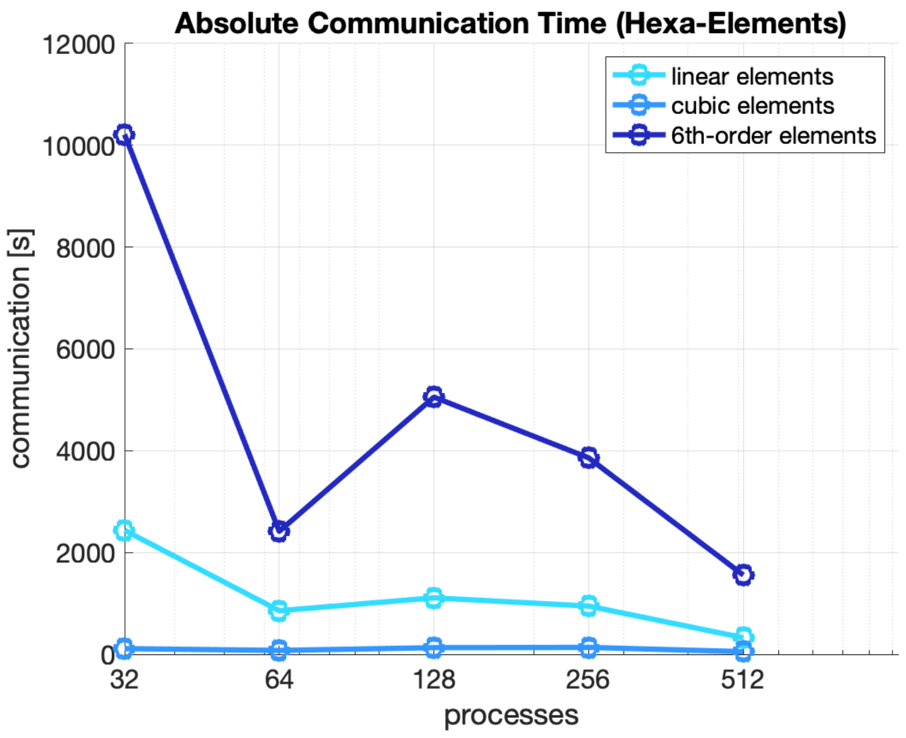


(a) simulation time

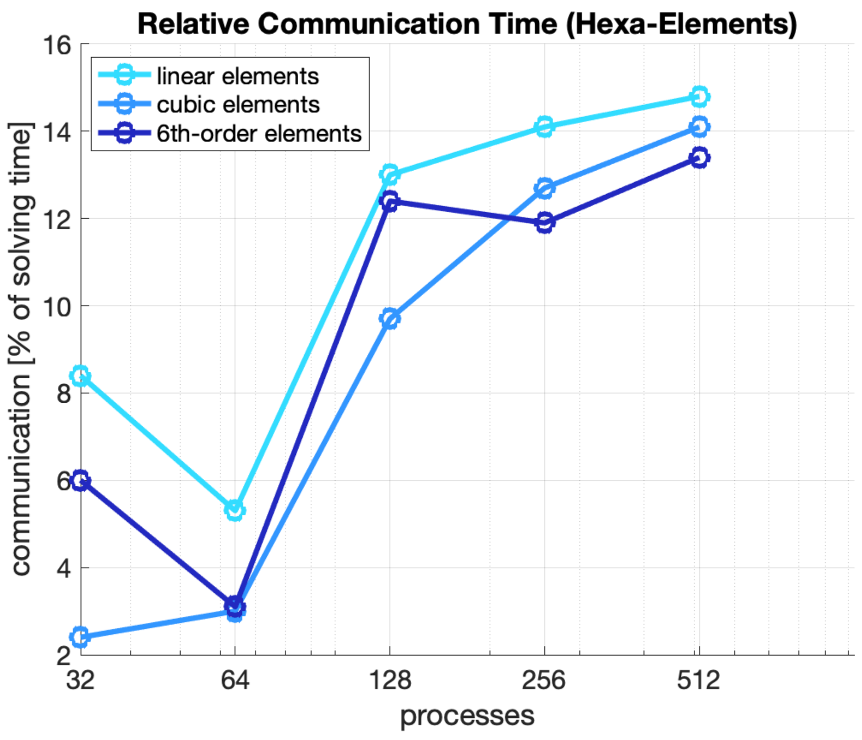


(b) speedup

Fig. 37: Total time and speedup of simulations of the building with linear, cubic and 6th-order hexahedral elements.



(a) absolute communication



(b) relative communication

Fig. 38: Amount of required communication in the solving procedure of simulations of the building with linear, cubic and 6th-order hexahedral elements.

6 Conclusion

Summarizing the results presented in this thesis it is seen that a very effective parallelization scheme was developed. High speedups could be obtained especially in the context of higher-order FEM. It was observed that reducing the DOFs with static condensation is a very useful tool in reducing the computation time of a higher-order simulation. Since the number of reduced DOFs depends on the number of inner element nodes, static condensation has more impact on simulations with hexahedral elements than on simulations with tetrahedral elements.

Furthermore, it was found that using the well established open-source software METIS is recommended to achieve a good domain decomposition for large-scale FE simulations in structural mechanics. METIS is not only comparatively fast but also creates very good domain decompositions regarding load-balancing and reduction of the interface nodes.

The communication scheme presented in this work has also proven to be very effective. By directly communicating the values of the interface nodes with the corresponding interface processes, no processor has to transfer unnecessary data. By using non-blocking MPI routines, the waiting times in communication can also be kept low. In the conducted studies, the amount of communication hardly increased above 25% of the total solution time, even for a number of 512 processes.

The conducted efficiency studies show that the produced speedup grows with the size and density of the given system of equations. This comes from the fact that the assembly and solving are fully parallelized. The higher the percentage of assembly and solving in contrast to the preprocessing and optional postprocessing, the higher is the fraction of parallel code. Therefore, the highest speedups are obtained for higher-order simulations. In this thesis two BVPs were considered, relevant for heat flows and linear elasticity. However, the implemented parallelization techniques, especially the communication procedure, are independent on the application and should therefore also produce significant speedups for other simulations under the condition that the produced system matrix is symmetric and positive definite (being the requirements for the CG algorithm).

For further work it will be interesting to consider the use of a parallel domain decomposition, especially for simulations with linear elements. Furthermore, finding and implementing a more effective yet highly parallelizable preconditioner would certainly lead to a further decrease in the simulation time.

Appendix

A How to execute code on the aCluster

A.1 Setup

1. **Acquire login data for the aCluster from the ZID-TU Graz.**
2. **Setup and start a VPN to the TU-Graz network.**
3. **Setup the rsa key for ssh login without a password.**[19]
   - Create a rsa key on your local machine.
     Type *ssh-keygen* and do not enter a password.
   - Copy the public key to the aCluster using ssh-copy-id.
     Type *ssh-copy-id -i /.ssh/id_rsa.pub <username>@acluster.tugraz.at.*
4. **Setup a runscript for syncing your files with rsync.**
   An example script is provided below[20]:

```
#!/bin/sh
REMOTEHOST=<username>@acluster.tugraz.at
SYNCP=/home/<username>/myCode/
LOCALP=~/Documents/myLocalCode/
PTOSYNCH="$1"

rsync -r $REMOTEHOST:$SYNCP$PTOSYNCH/ $LOCALP$PTOSYNCH
exit 0
```

5. **Setup a runscript for your code.**
   An example script for a run with MPI and 128 processes is provided below:

```
#!/bin/bash
#SBATCH --nodes=2
#SBATCH --ntasks=128

mpirun ./myProg
```

[19]A more detailed description is provided on https://www.thegeekstuff.com/2008/11/3-steps-to-perform-ssh-login-without-password-using-ssh-keygen-ssh-copy-id/.

[20]This script was provided by the Institute of Structural Analysis - TU Graz.

A.2 Execute the code on the aCluster

1. **Assure VPN is active.**
2. **[optional] Snyc your files to/from the aCluster using rsync.**
   Type *./mySyncFile*.
3. **Connect to the aCluster via ssh.**
   Type *ssh <username>@acluster.tugraz.at*. You can now navigate through your folders and files.
4. **Compile the code.**
   Compile the code as usual (common compilers like gcc or clang are installed).
5. **Queue your program.**
   The aCluster uses the queueing system SLURM to determine which resources are free and which jobs should be run next. You should therefore **NEVER** directly run your program with *./myProg*. You can queue your program with *sbatch myRunScript.sh*. You can use *squeue* to see the list of currently running and enqueued jobs. You can use *scancel <jobID>* to cancel a queued or running job. The output of your program will be written to an out-file with the name slurm-<jobID>.out. With *sinfo* you can see how many nodes are currently occupied by the cluster and how many are on idle.

For further information you can read the man page of the aCluster, by typing *man aCluster* on the cluster. Furthermore it is also possible to contact the ZID with any open questions.

A concrete example of the commands needed for the execution of a program with 128 processes (assuming an active vpn) is provided below:

```
./mySyncFile                         #sync files to the cluster

ssh <username>@acluster.tugraz.at    #conect to the cluster

cd myCode/                           #cd and compile code
./MakeProg MPI2 clean

sbatch myRunScript128.sh             #queue the program (128 procs)

exit 0                               #close existing ssh connection
```

List of Figures

List of Tables

Further acknowledgements
I want to thank the ZID (Zentraler Informatiksienst) for enabling me to work on a high performance cluster of the TU-Graz and DI Dr.techn. Daniel Schöllhammer, BSc for providing me with the LATEX-template.

References

[1] *History and overview of high performance computing*, http://www.math-cs.gordon.edu/courses/cps343/presentations/History_and_Overview_of_HPC.pdf, [2021-07-20], Gordon College, 2020.

[2] P. Freiberger, *Eniac*, https://www.britannica.com/technology/ENIAC, [2021-08-14].

[3] J. Sale, *A brief history of high-performance computing (hpc)*, https://confluence.xsede.org/pages/viewpage.action?pageId=1677620, [2021-08-14].

[4] Top500, *Fugaku holds top spot, exascale remains elusive*, https://www.top500.org/news/fugaku-holds-top-spot-exascale-remains-elusive/, [2021-08-14].

[5] T.-P. Fries, *Lecture notes Finite Element Method*, Technische Universität Graz, 2018.

[6] B. Rajinikanth, *Data structures - sparse matrix*, http://www.btechsmartclass.com/data_structures/sparse-matrix.html, [2021-08-14].

[7] *Iterative methods for linear systems*, https://www.mathworks.com/help/matlab/math/iterative-methods-for-linear-systems.html, [2021-07-10], The MathWorks Inc.

[8] *Matlab tutorial on bicg*, https://de.mathworks.com/help/matlab/ref/bicg.html, [2021-07-10], The MathWorks Inc.

[9] *Matlab tutorial on bicgstab*, https://de.mathworks.com/help/matlab/ref/bicgstab.html, [2021-07-10], The MathWorks Inc.

[10] J. R. Shewchuk, "An introduction to the conjugate gradient method without the agonizing pain", 1994.

[11] P. Gangl, *Lecture notes Technische Numerik 1*, Unpublished, Technische Universität Graz, 2018.

[12] O. Steinbach, *Technische Numerik, Berichte aus dem Institut für Mathematik Numerik und Partielle Differentialgleichungen*, Technische Universität Graz, 2005.

[13] *Introduction to parallel computing tutorial*, https://hpc.llnl.gov/training/tutorials/introduction-parallel-computing-tutorial, [2021-07-20], Livermore Computing Center.

[14] *Flynn's classical taxonomy*, https://hpc.llnl.gov/tutorials/introduction-parallel-computing/flynns-classical-taxonomy, [2021-07-20], Livermore Computing Center.

[15] S. Kainz and M. Lang, *Manual page - acluster*, Technische Universität Graz, 2021.

[16] *Technische details - hochleistungsrechnen*, https://tu4u.tugraz.at/bedienstete/it-services/forschung/hochleistungsrechnen/technische-details/, [2021-07-20], Technische Universität Graz.

[17] Y. Liu, W. Zhou, and Q. Yang, “A distributed memory parallel element-by-element scheme based on jacobi-conditioned conjugate gradient for 3d finite element analysis”, *Finite Elements in Analysis and Design*, vol. 43, pp. 494–503, 2007.

[18] Oracle, *Running programs with the mpirun command*, `https://docs.oracle.com/cd/E19708-01/821-1319-10/ExecutingPrograms.html`, [2021-07-05], 2010.

[19] Karl Ljungkvist, “Matrix-free finite-element operator application on graphics processing units”, 2014, pp. 450–461, ISBN: 978-3-319-14312-5.

[20] K. Williams and A. Zhiliakov, *Matrix-free finite element method*, University of Houston, 2019.

[21] G. Of, *Lecture notes Technische Numerik 2*, Unpublished, Technische Universität Graz, 2019.

[22] P. Höfer, *Statische kondensation*, `https://me-lrt.de/statische-kondensation`, [2021-05-20], 2011.

[23] Netgen/NGSolve team, *Static condensation*, `https://ngsolve.org/docu/nightly/i-tutorials/unit-1.4-staticcond/staticcond.html`, [2021-07-05], 2017.

[24] P. W. Johnson, “Efficient domain decomposition algorithms and applications in transportation and structural engineering”, PhD thesis, Old Dominion University, 2016.

[25] G. Karypis, *Metis a software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices version 5.1.0*, University of Minnesota, Minneapolis, MN, 2013.

[26] M. Hribar, V. Taylor, and D. Boyce, “Choosing a shortest path algorithm”, 2001.

[27] A. Felfernig, *Grundlagen der artificial intelligence und logik, Search*, Technische Universität Graz, 2021.

[28] G. Karypis and V. Kumar, “A fast and high quality multilevel scheme for partitioning irregular graphs”, *SIAM Journal on Scientific Computing*, vol. 20, pp. 359–392, 1998.

[29] T. Davis, W. Hager, S. Kolodziej, and S. Yeralan, *Algorithm 1003: Mongoose, a graph coarsening and partitioning library*, 2020.

[30] G. Karypis and V. Kumar, “Multilevel k-way partitioning scheme for irregular graphs”, *Journal of parallel and distributed Computing*, vol. 48, pp. 96–129, 1998.

[31] J. Dongarra, *Jacobi preconditioning*, `http://www.netlib.org/linalg/html_templates/node55.html`, [2021-06-05], 1995.

[32] S. Mangard, *Computer organization and networks, Chapter 12: Building faster processors*, Technische Universität Graz, 2020.

[33] A. Azad, M. Jacquelin, A. Buluç, and E Ng, *The reverse cuthill-mckee algorithm in distributed-memory*, Lawrence Berkeley National Laboratory, 2017.

[34] D. G. Chester, S. A. Wright, and S. A. Jarvis, “Understanding communication patterns in hpcg”, *Electronic Notes in Theoretical Computer Science*, vol. 340, pp. 55–65, 2018.

[35] *Mpi tutorial*, `https://mpitutorial.com/tutorials/`, [2021-05-30], 2021.

[36] University of Florida Research Computing Help and Documentation site, *Parallel computing*, `https://help.rc.ufl.edu/doc/Parallel_Computing`, [2021-07-05], 2017.

[37] *Message passing interface forum, Communication modes*, `https://www.mpi-forum.org/docs/mpi-2.2/mpi22-report/node53.htm`, [2021-05-30], 2009.

[38] *Introduction to parallel programming with mpi, Non-blocking communication*, `https://pdc-support.github.io/introduction-to-mpi/06-nonblocking/index.html`, [2021-05-30], CC-BY-SA 4.0.

[39] T. Hoefler, C. Siebert, and W. Rehm, "A practically constant-time mpi broadcast algorithm for large-scale infiniband clusters with multicast", 2007, pp. 1–8.

[40] D. McCaughan, *Mpi: Beyond the basic*, `https://www.csd.uwo.ca/~mmorenom/CS433-CS9624/Resources/MPI_Beyond_Basics.pdf`, [2021-05-31], 2008.