Faculty of Civil, Geo and Environmental Engineering
Chair for Computational Modeling and Simulation
Prof. Dr.-Ing. André Borrmann

Parallel Phase-Field Simulations with the Finite Cell Method and Adaptive Refinement

Federico Fernández Erbes

Master's thesis
for the Master of Science program Computational Mechanics

Author:	Federico Fernández Erbes
Matriculation number:	03728352
Supervisor:	PD Dr.-Ing. habil. Stefan Kollmannsberger
	Lisa Hug, M.Sc.
Date of issue:	01. May 2021
Date of submission:	01. November 2021

Involved Organisations

Chair for Computational Modeling and Simulation
Faculty of Civil, Geo and Environmental Engineering
Technische Universität München
Arcisstraße 21
D-80333 München

Declaration

With this statement I declare, that I have independently completed this Master's thesis. The thoughts taken directly or indirectly from external sources are properly marked as such. This thesis was not previously submitted to another academic institution and has also not yet been published.

München, October 29, 2021

Federico Fernández Erbes

Federico Fernández Erbes
Trauchbergstraße 6
D-81539 München
e-Mail: federico.fernandez-erbes@tum.de

Acknowledgements

First off, I want to thank PD Dr.-Ing. habil. Stefan Kollmannsberger for his advice and for the great opportunity to do my Master's thesis at the Chair of Computational Modeling and Simulation. Special thanks go to my supervisor Lisa Hug for her guidance, her patience, and for sharing her knowledge and wisdom with me.

I want to thank all professors, tutors, and supervisors I had during the COME program, who provided me with a superb learning experience despite the difficult times we all went through.

I am particularly grateful to Julieta, for encouraging me to start this program and for supporting me every step of the way and in all the possible means. Completing this program would have not been possible without her love and sacrifice.

Last but not least, I want to thank my family and friends for their support and for being there with me every time I needed them.

Federico Fernández Erbes

München, October 29, 2021

Abstract

The phase-field method has emerged as a promising computational tool to predict and understand fracture failures. The finite cell method (FCM) provides an excellent numerical framework for phase-field analyses. When combined with multi-level *hp*-refinement, the FCM allows an accurate prediction of fracture processes by adaptively refining the mesh in the fracture zone.

To solve large, complex computational problems efficiently, it is common to rely on parallel computing. However, the implementation of the phase-field method with the FCM in parallel is not straightforward for three main reasons. First, additional ghost elements are needed to ensure that preconditioners used to solve the resulting system of equations are correctly computed. Secondly, refinement based on the phase-field solution cannot be applied directly, as not all degrees of freedom in a distributed mesh have correct values. Finally, adaptive refinement unbalances the workload among processes, increasing the execution time.

This thesis presents the algorithms needed to integrate the phase-field method within a framework for large-scale parallel finite cell computations. It introduces the necessary changes to the mesh generation procedures to have additional ghost elements, a modified refinement strategy that is suitable for parallel simulations, and a novel algorithm to rebalance the workload after refinement. The results show good parallel scalability for several processes and demonstrate the benefits of the developed load balancing strategy. A practical example illustrates the potential of the parallel framework to solve problems of engineering relevance.

Contents

1 Introduction 1
  1.1 Background 1
  1.2 Aim and Outline 3

2 Fracture Mechanics 5
  2.1 Fundamentals 5
  2.2 The phase-field method for fracture 6

3 Numerical Description 9
  3.1 The Finite Cell Method 9
  3.2 Multi-level *hp*-FEM 10
  3.3 Solution of the coupled problem 12

4 Parallel phase-field computations 15
  4.1 Parallel computing fundamentals 15
    4.1.1 Serial vs. parallel computing 15
    4.1.2 Hardware and software models for parallel computing 16
    4.1.3 Parallel performance 17
  4.2 Parallel mesh handling 18
    4.2.1 Terminology 18
    4.2.2 Mesh generation 20
    4.2.3 Mesh compatibility 23
    4.2.4 Adaptive mesh refinement 25
    4.2.5 Mesh repartitioning 29
  4.3 Efficient parallel solvers 35
    4.3.1 Parallel linear solvers 35
    4.3.2 Preconditioning FCM analyses 36

5 Results 39
  5.1 Square plate in tension with pre-existing crack 39
  5.2 Square plate in tension with rotated elliptical hole 47
  5.3 Cubic body in tension with spherical inclusions 50
  5.4 Compression test of a core sample 53

6 Conclusion and Outlook 57
  6.1 Conclusion 57
  6.2 Outlook 58

A Determination of surrounding cells 59

Chapter 1

Introduction

The phase-field method is a growing field in computational fracture mechanics that involves the approximation of a crack with the introduction of a scalar field that serves as an indicator of damaged material. By avoiding the complex task of tracking the surface of a propagating crack, the phase-field method simplifies the numerical treatment of fracture problems. Several numerical implementations of the method have been developed during the last years. [Nagaraja et al., 2018] integrated a phase-field model with the Finite Cell Method (FCM) [Parvizian et al., 2007] and multi-level *hp*-refinement [Zander et al., 2016], enabling the solution of problems with high levels of accuracy using non-boundary-conforming meshes. The potential of this development to solve large, resource-demanding problems is, however, limited due to the serial nature of its implementation. This thesis aims to integrate the work developed by [Nagaraja et al., 2018] with the framework for large-scale parallel finite cell computations presented in [Jomo, 2021] to enable the parallel solution of fracture mechanics problems with the phase-field method. The present chapter provides the background and context to introduce the reader to the topics, followed by the challenges posed by the parallel solution of phase-field problems and the goals of the present work.

1.1 Background

The propagation of cracks in solid materials is a phenomenon that has been studied since the last century. One of the pioneering works in this field was performed by [Griffith, 1921]. He formulated a fracture theory based on the first principle of thermodynamics and stated that a flaw in a solid becomes unstable when the change in strain energy due to an increment in the flaw size equals the surface energy of the material due to the creation of new fracture surfaces. It was not until the end of World War II that fracture was extensively studied after the catastrophic failure of several Liberty ships of the US Navy. [Irwin, 1960] extended the Griffith approach to metals including the energy dissipated by plasticity. In the Post-War era and up to the 1980s, several advancements in fracture mechanics enabled its application in fields such as aerospace, energy, and the nuclear power industry. The significant improvement of computer technology towards the end of the 20th century and beginning of the 21st century allowed the development of sophisticated models for the computation of fracture mechanics problems with the Finite Element Method (FEM), such as the (numerical) J-integral [Carpenter et al., 1986], the Extended Finite Element Method (XFEM) [Moes et al., 1999], and Cohesive Zone Models [Elices et al., 2002], among others. The aforementioned models can be easily applied to stationary cracks, but they become cumbersome when dealing with growing cracks, as special meshing strategies and methods to track the crack surfaces must be applied. During the last decades, the phase-field method has gained significant momentum in fracture mechanics research. This method approximates the crack surface with a scalar field that is used to degrade the stiffness of the material in the crack zone, effectively eliminating the necessity of tracking the crack surface and thus reducing the computational effort.

Even though the FEM has been widely used to carry out phase-field fracture mechanics analyses, there is an increasing demand for the solution of more sophisticated problems that has turned the attention to other alternatives such as the FCM. Examples of such problems are the solution of three-dimensional material interface problems without conforming mesh generation [Elhaddad et al., 2017], or the characterization of the influence of process-induced defects on the homogenized behavior of metal lattices [Korshunova et al., 2021]. The FCM uses a non-boundary-conforming mesh that avoids the difficult and time-consuming process of mesh generation in intricate geometries. It does so by embedding the physical domain in a fictitious domain of a simpler shape that can be easily meshed. The original problem is then recovered by the introduction of an indicator function that modifies the weak form and defines whether a point belongs to the physical or fictitious domain. When combined with the multi-level *hp*-refinement, the FCM provides the high flexibility of immersed methods with the good approximation qualities of high order finite elements [Zander et al., 2015; Elhaddad et al., 2017]. Moreover, the *refinement by superposition* approach of the multi-level *hp*-refinement is particularly useful in phase-field analyses, as it facilitates the adaptive refinement of the mesh to resolve the high gradients present in the phase-field solution of a growing crack [Nagaraja et al., 2018].

Although the FCM provides many advantages, its extension to allow for efficient, parallel solutions to large-scale problems comes with difficulties. The FCM suffers from conditioning problems related to small *cut elements* in the finite cell mesh [de Prenter et al., 2017], that prevent the efficient solution of large linear systems with iterative solvers such as the Conjugate Gradient (CG) method. In addition, the use of hardware resources in computing clusters requires well-designed parallel codes for the generation and handling of *distributed meshes*, i.e., meshes that are distributed among several processes. It was not until recently that these difficulties were successfully overcome. In [Jomo, 2021], an approach is presented where light-weight data structures based on adaptive Cartesian grids are used for parallel mesh creation and refinement. Furthermore, a novel preconditioning technique tailored for FCM problems and based on the additive Schwarz lemma is presented, alleviating the problem of the small cut cells [Jomo et al., 2019]. To allow for good scalability, distributed meshes used in this approach contain two layers of *ghost elements*. These are elements in a given mesh that are duplicated from meshes of neighboring processes and permit the computation of stiffness and preconditioning matrices without the exchange of data between processes.

The techniques presented in [Jomo, 2021], however, cannot be directly applied for phase-field analyses. Here, the two layers of ghost elements are not sufficient due to the nonlinearity of the problem. In this case, the computation of the exact preconditioner requires that the value of the solution of all degrees of freedom up to the second layer of ghost elements are valid, which necessitates the presence of a third layer of ghost elements. Moreover, the adaptive refinement used in a phase-field analysis, which is driven by the value of the phase-field solution [Nagaraja et al., 2018], cannot be performed on the outer (third) layer of ghost elements, as they have an invalid solution. Last but not least, the adaptive refinement in a parallel setting inherently unbalances the workload among the intervening processes, requiring the repartitioning of the meshes to recover a well-balanced state and avoid detrimental effects in the parallel efficiency [Hendrickson and Devine, 2000].

1.2 Aim and Outline

The goal of this thesis is to integrate the work of [Nagaraja et al., 2018] and [Jomo, 2021] to provide an efficient framework for the parallel solution of phase-field fracture mechanics problems. The motivation resides not only in the speed-up of computations but also in the ability to solve larger problems with the use of distributed data structures. To achieve this goal, the original algorithms and data structures of [Jomo, 2021] are extended to include a third layer of ghost elements. Furthermore, an adaptive refinement strategy tailored for phase-field analyses with parallel data structures is developed. To deal with the load imbalance created by the adaptive refinement, novel algorithms for repartitioning of the meshes are presented.

This thesis is divided into six chapters: Chapter 2 presents fundamental concepts of fracture mechanics and provides the necessary background on the phase-field model used in this work. Chapter 3 delves into the numerical framework for the solution of quasi-static fracture mechanics problems, covering the basics of the FCM and the Multi-level *hp*-FEM. Chapter 4 provides a brief introduction to parallel computing and explains the algorithms developed in this thesis for the parallel solution of phase-field problems with adaptive refinement and repartitioning. Chapter 5 shows numerical examples that test the implementation, investigate the performance of the parallel framework, and demonstrate the application in a problem of practical relevance. Finally, chapter 6 gives the conclusion and an outlook for further work.

Chapter 2

Fracture Mechanics

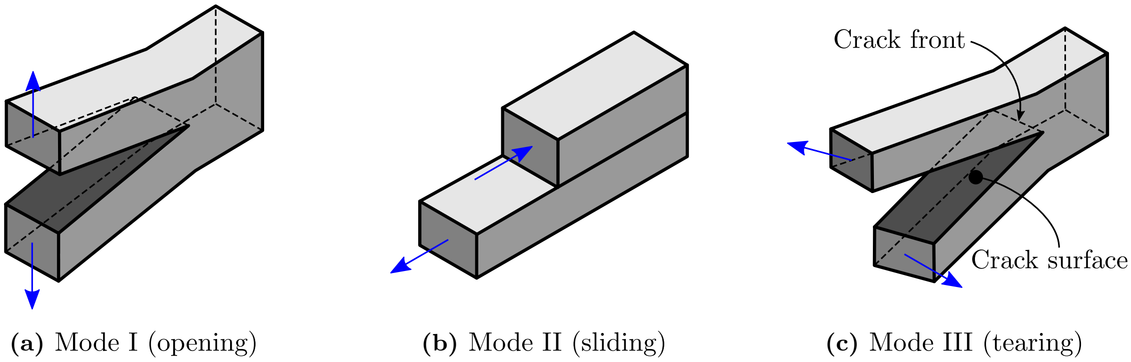
The present chapter deals with the definition of the phase-field fracture problem to solve in parallel, and is organized as follows. First, some fundamental concepts in fracture mechanics including the classification of fracture phenomena and the modes of fracture are given. It is followed by a brief introduction to Griffith's theory of brittle fracture as it is the precursor for the definition of the phase-field problem, which is given next. Finally, the governing equations to model numerically are given, with a short mention of a particular treatment of the stress tensor used in this work.

2.1 Fundamentals

Fracture mechanics studies the phenomena of crack nucleation, growth, and propagation in a solid. Depending on the nature of the material, the rate of loading, and other factors such as temperature, fractures can be classified as *brittle* or *ductile*. Brittle fracture is a sudden, rapid cracking of a solid under stress where the material shows little to no plastic deformation before the crack propagates. In contrast, in ductile fracture, the crack grows slowly with the material undergoing large plastic deformations. Fracture can be further classified into *quasi-static* or *dynamic*. In quasi-static fracture, the fracture process can be described by a sequence of static equilibrium states. In dynamic fracture, the fracture process is accompanied by rapid changes in loading or structure geometry, and inertial effects are considered. In this thesis, we deal with quasi-static brittle fracture problems.

One of the fundamental concepts in fracture mechanics is that of *fracture modes*. Depending on the direction of the loading with respect to the crack surface and crack front, there are three basic modes of fracture:

- **Mode I (*opening*)**: occurs when the load is perpendicular to both the crack surface and the crack front, Figure 2.1a.
- **Mode II (*sliding*)**: occurs when the load is parallel to the crack surface but perpendicular to the crack front, constituting a shearing load, Figure 2.1b.
- **Mode III (*tearing*)**: occurs when the load is parallel to both the crack surface and the crack front, Figure 2.1c.



(a) Mode I (opening) **(b)** Mode II (sliding) **(c)** Mode III (tearing)

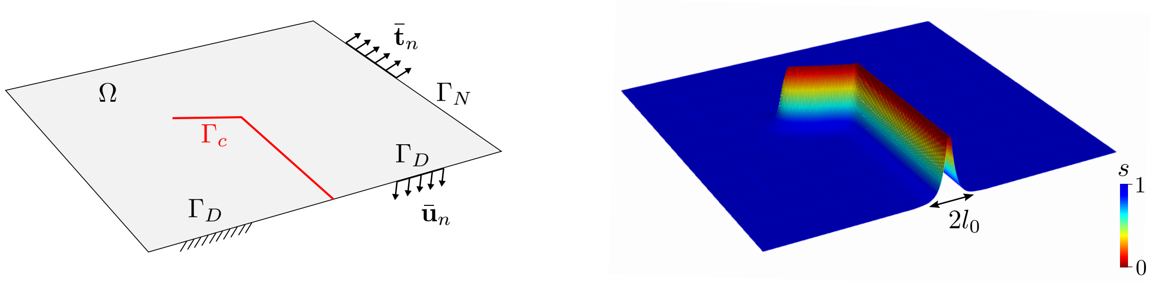
Figure 2.1: Modes of fracture.

Fracture modes can be used to model more complex cases with inclined crack fronts, by using a superposition of the three basic modes and analyzing the effect of the load on each mode separately.

2.2 The phase-field method for fracture

Griffith's theory of brittle fracture

Consider an arbitrary domain $\Omega \subset \mathbb{R}^d$ with $d \in \{1, 2, 3\}$ and a set of discrete cracks Γ_c as shown in Figure 2.2a. The domain has an external boundary $\partial\Omega$ subdivided into non-overlapping parts Γ_D and Γ_N where Dirichlet and Neumann boundary conditions are applied, respectively.



(a) Sharp crack **(b)** Phase-field

Figure 2.2: Sharp crack topology and phase-field regularized crack surface, adopted from [Hug et al., 2020].

The displacement of a material point $\mathbf{x} \in \Omega$ is denoted by $\mathbf{u}(\mathbf{x}) \subset \mathbb{R}^d$. Assuming small deformations, we define the infinitesimal strain tensor $\boldsymbol{\varepsilon}(\mathbf{u}) = \frac{1}{2}\left(\nabla\mathbf{u} + \nabla\mathbf{u}^T\right)$ as a deformation measure. Furthermore, we assume isotropic linear elasticity with an elastic strain energy density given by $\Psi(\boldsymbol{\varepsilon}) = \frac{1}{2}\lambda\,\mathrm{tr}\left(\boldsymbol{\varepsilon}\right)^2 + \mu\,\mathrm{tr}\left(\boldsymbol{\varepsilon}^2\right)$, where λ and μ are the Lamé constants.

According to Griffith's theory of brittle fracture [Griffith, 1921], the energy required to create a unit area of fracture surface is equal to the critical fracture energy density G_c also referred to as *fracture toughness*. The total potential energy E of the body is the sum of the elastic energy and the fracture energy

$$E(\boldsymbol{\varepsilon}, \Gamma_c) = \underbrace{\int_\Omega \Psi(\boldsymbol{\varepsilon}) d\mathbf{x}}_{\text{Elastic energy}} + \underbrace{\int_{\Gamma_c} G_c d\mathbf{x}}_{\text{Fracture energy}} \quad . \tag{2.1}$$

Phase-field approximation

The numerical solution of (2.1) is difficult as the evolving crack surface Γ_c has to be tracked, often requiring costly computations. To overcome this issue, the phase-field method approximates the crack surface by a scalar field s called phase-field, which varies from 0 in the crack surface to 1 in intact regions, as shown in Figure 2.2b. Using the phase-field, [Bourdin et al., 2000] proposed a regularized formulation that approximates the surface integral of the fracture energy with a volume integral

$$E_{l_0}(\boldsymbol{\varepsilon}, s) = \int_\Omega \Psi(\boldsymbol{\varepsilon}) d\mathbf{x} + \int_\Omega G_c \left[\frac{(1-s)^2}{4 l_0} + 2 l_0 |\nabla s|^2 \right] d\mathbf{x}, \tag{2.2}$$

where l_0 is a model parameter called *regularization length* that controls the width of the smooth approximation of the crack, as shown in Figure 2.2b. In the limit $l_0 \to 0$, the phase-field approximation converges to the discrete fracture surface and $E_{l_0} \to E$ [Alberti, 2000]. To model the loss of stiffness in the crack zone, the elastic strain energy density is approximated by

$$\Psi(\boldsymbol{\varepsilon}) = g(s)\Psi^+(\boldsymbol{\varepsilon}) + \Psi^-(\boldsymbol{\varepsilon}), \tag{2.3}$$

where $g(s)$ is the degradation function and $\Psi^+(\boldsymbol{\varepsilon})$ and $\Psi^-(\boldsymbol{\varepsilon})$ are the tensile and compressive parts of the elastic strain energy density, respectively. In this thesis, a quadratic degradation function with the form

$$g(s) = (1-\eta)s^2 + \eta \tag{2.4}$$

is used, where $\eta \approx 0$ is a small parameter that ensures numerical stability in the case of a fully degraded material ($s = 0$). In (2.3), the degradation function is applied only to the tensile part of the strain energy density [Miehe et al., 2010]. This has the purpose of preventing crack propagation in compression and other non-physical crack patterns. Considering the split in the elastic strain energy density, the regularized formulation is finally

$$E_{l_0}(\boldsymbol{\varepsilon}, s) = \int_\Omega g(s)\Psi^+(\boldsymbol{\varepsilon}) + \Psi^-(\boldsymbol{\varepsilon}) d\mathbf{x} + \int_\Omega G_c \left[\frac{(1-s)^2}{4 l_0} + 2 l_0 |\nabla s|^2 \right] d\mathbf{x}. \tag{2.5}$$

Governing equations

The minimization of the functional (2.5) leads to the governing equations in strong form

$$\begin{cases} \nabla \boldsymbol{\sigma} + \rho \mathbf{b} = \mathbf{0} & (2.6a) \\ \left[1 + \dfrac{l_0(1-\eta)}{G_c}\mathcal{H}\right] s - 2l_0^2 \Delta s = 1, & (2.6b) \end{cases}$$

with boundary conditions

$$\begin{cases} \mathbf{u} = \bar{\mathbf{u}}_n \quad \text{on} \quad \Gamma_D & (2.7a) \\ \boldsymbol{\sigma} \cdot \mathbf{n} = \bar{\mathbf{t}}_n \quad \text{on} \quad \Gamma_N & (2.7b) \\ \nabla d \cdot \mathbf{n} = 0 \quad \text{on} \quad \Gamma_D \cup \Gamma_N. & (2.7c) \end{cases}$$

In (2.6) and (2.7), $\mathbf{b}$ is the body force vector, $\boldsymbol{\sigma}$ is the stress tensor defined as

$$\boldsymbol{\sigma} = g(s)\frac{\partial \Psi^+(\boldsymbol{\varepsilon})}{\partial \boldsymbol{\varepsilon}} + \frac{\partial \Psi^-(\boldsymbol{\varepsilon})}{\partial \boldsymbol{\varepsilon}}, \tag{2.8}$$

and $\mathcal{H}$ is the so-called *history variable* defined as

$$\mathcal{H}(\mathbf{x}, t) := \max_{t \in [0,T]} \Psi^+(\boldsymbol{\varepsilon}(\mathbf{u}(\mathbf{x}), t)). \tag{2.9}$$

used to enforce the irreversibility condition $\Gamma(t) \subseteq \Gamma(t + \Delta t)$ in the strong form equations.

Hybrid model

The tension-compression split in the elastic strain energy density renders the equilibrium equation (2.6a) nonlinear. As a result, incrementation techniques are employed to arrive at a numerical solution. To avoid such a computationally expensive approach, the hybrid model was introduced [Ambati et al., 2015]. This model avoids the nonlinearity in (2.6a) by degrading the total strain energy to compute the stress tensor

$$\boldsymbol{\sigma} = g(s)\frac{\partial \Psi(\boldsymbol{\varepsilon})}{\partial \boldsymbol{\varepsilon}} \tag{2.10}$$

while keeping the tension-compression split only for the phase-field equation (2.6b) through the history variable. The hybrid model is used for the examples presented in 5, although the implementation is equally applicable to other models.

Chapter 3

Numerical Description

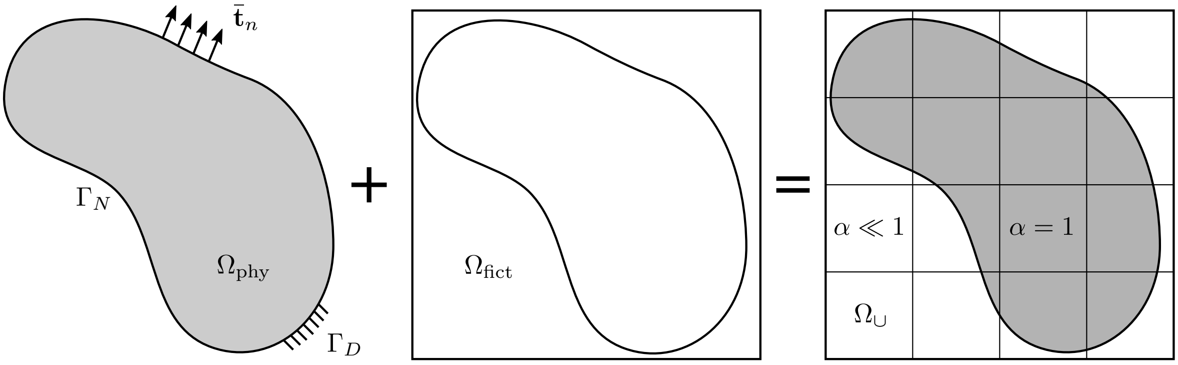
The present chapter introduces the numerical framework used in this thesis for the solution of quasi-static brittle fracture mechanics problems. The framework combines a phase-field model with the finite cell method used in conjunction with multi-level *hp*-refinement, allowing to efficiently resolve the high gradients present in the phase-field solution [Nagaraja et al., 2018], and to work with complex geometries without the need of boundary-conforming meshes. The chapter starts with a brief introduction of the FCM, highlighting its differences with the FEM. Afterward, the multi-level *hp*-refinement is explained. Finally, the solution approach for the coupled elasticity and phase-field problems is given.

3.1 The Finite Cell Method

The traditional FEM uses a boundary-conforming mesh for the approximation of the solution. To ensure adequate numerical accuracy, elements of the boundary-conforming mesh have to fulfill certain shape requirements. Therefore, the process of mesh generation for complex geometries is involved and time-consuming. To overcome this difficulty, non-boundary-conforming methods such as the FCM have emerged [Parvizian et al., 2007]. The FCM combines a fictitious or embedded approach with high-order finite element methods, yielding a simple generation of meshes with high convergence rates [Zander et al., 2015]. In this method, the physical domain Ω_{phy} is embedded in a fictitious domain Ω_{fict} of simpler shape that can be trivially meshed with structured or Cartesian grids as shown in Figure 3.1. The elements of this mesh are called *finite cells* to differentiate them from their boundary-conforming counterparts. To recover the original initial boundary value problem, an indicator function $\alpha(\mathbf{x})$ that associates a given point $\mathbf{x}$ with the physical or fictitious domain is introduced

$$\alpha(\mathbf{x}) = \begin{cases} 1 & \forall \mathbf{x} \in \Omega_{\text{phy}} \\ \alpha_{\text{FCM}} \ll 1 & \forall \mathbf{x} \in \Omega_{\text{fict}}, \end{cases} \tag{3.1}$$

where α_{FCM} is a small constant greater than zero that ensures numerical stability by avoiding ill-conditioning of the stiffness matrices due to elements that belong to the fictitious domain. The weak form of the problem is multiplied by the indicator function to penalize the contributions of the fictitious domain. As a consequence, the effort of a complex mesh generation in the FEM is shifted to the integration stage in the FCM. The indicator function introduces a discontinuity in the weak form at cells that are cut by the boundaries of the physical domain. As the standard Gauss-Legendre quadrature shows poor convergence for non-smooth functions, more suitable methods have been proposed [Düster et al., 2008]. In the present work, the quadtree and octree approach with recursive subdivision of cut cells up to a fixed partitioning depth is used for 2D and 3D respectively.

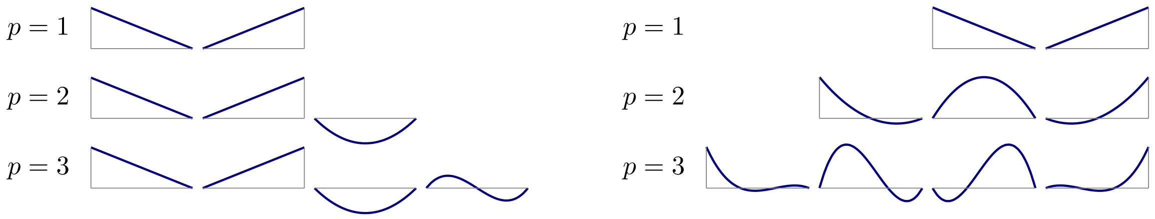


(a) Physical domain **(b)** Fictitious domain **(c)** Structured discretization

Figure 3.1: Visualization of the FCM.

3.2 Multi-level *hp*-FEM

The quality of the approximated solution provided by the FEM depends on the size and the polynomial degrees of the elements. To improve the approximation, several approaches have been developed [Babuska and Guo, 1992]. One of them is the *h*-version of the finite element method (*h*-FEM) where the spatial resolution of the mesh is increased while leaving the polynomial degree unchanged. In contrast, in the *p*-version of the finite element method (*p*-FEM), the spatial resolution of the mesh remains constant while the polynomial degree is increased. Hierarchic shape functions are particularly useful in this case, as they allow to elevate the polynomial degree of the basis with the simple addition of new shape functions to the already existing ones [Szabó and Babuška, 1991], see Figure 3.2a. This is advantageous compared to the commonly-used Lagrange shape functions where all basis functions have to be replaced, see Figure 3.2b.



(a) Integrated Legendre basis **(b)** Lagrange basis

Figure 3.2: Comparison of the linear, quadratic and cubic one-dimensional integrated Legendre and Lagrange basis functions, adopted from [Jomo, 2021].

The combination of both approaches, i.e., a variable spatial resolution with a changing polynomial degree, leads to the (classical) *hp*-version of the finite element method (*hp*-FEM). With this approach, elements are *replaced* with a set of smaller elements in regions of high error contributions, while coarse elements with a high polynomial degree are used in regions where the solution has a high regularity. Although this combination yields exponential convergence [Babuska and Guo, 1992], its applicability is limited due to the high implementational effort of the *refine-by-replacement* approach. The difficulty comes from the fact that shape functions in fine elements do not have a matching counterpart in the coarse, adjacent elements, rendering the mesh irregular. Therefore, additional steps are required to restore inter-element continuity so that compatibility of the shape functions and convergence of the method is ensured.

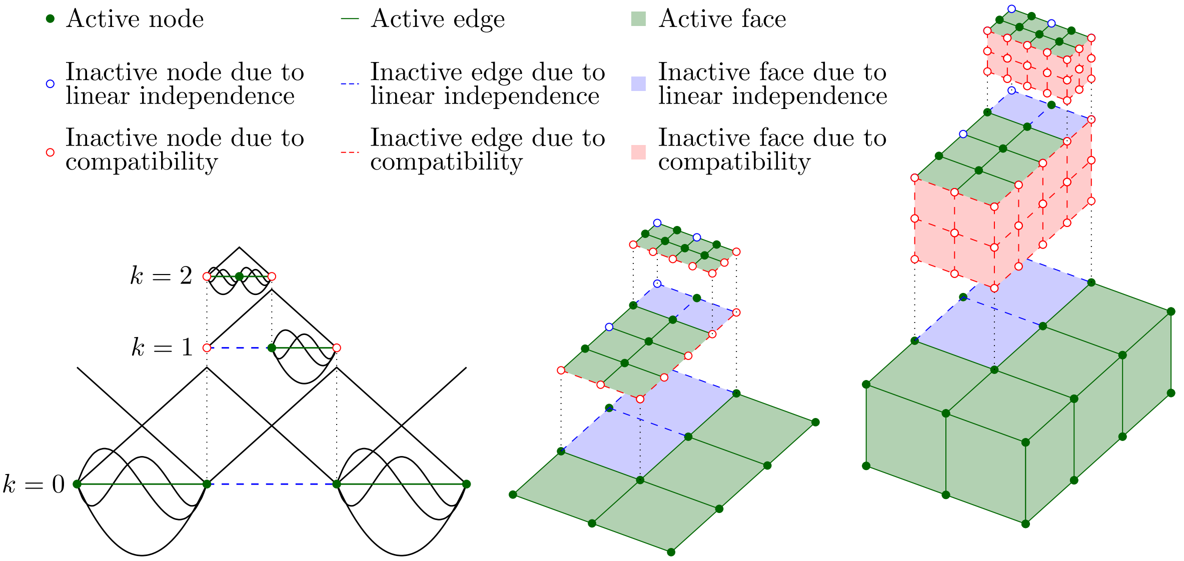
To overcome this problem, the multi-level *hp*-version of the finite element method uses a *refinement-by-superposition* approach [Zander et al., 2016]. A coarse mesh that captures the global solution characteristics is *superposed* by a finer mesh that increases the solution quality on the regions of interest. As a result, the solution is the sum of a base mesh solution $\mathbf{u}_\mathrm{b}$ and a fine mesh overlay solution $\mathbf{u}_\mathrm{o}$, i.e.,

$$\mathbf{u} = \mathbf{u}_\mathrm{b} + \mathbf{u}_\mathrm{o}. \tag{3.2}$$

In the multi-level *hp*-approach, fine elements can be refined again up to a depth k, creating a hierarchy of multiple levels of overlay meshes. In this context, it is essential to ensure two conditions:

1. **Linear independence**: coarse shape functions must not be expressible by a linear combination of fine overlay shape functions. Integrated Legendre shape functions can be associated directly with topological components such as nodes, edges, faces, and solids, as they take non-zero values in these components and zero on all others. Therefore, linear independence can be achieved by deactivating all topological components with active sub-components. Additionally, all direct successors of base nodes must be deactivated.
2. **Compatibility**: for a variational index m, compatibility requires C^m continuity within elements and C^{m-1} continuity across element boundaries. To achieve this, homogeneous Dirichlet boundary conditions are applied to the boundary of the overlay meshes. Only in cases where the boundary of the element coincides with the boundary of the physical domain, the respective components stay active. By doing so, hanging nodes are avoided by construction as no degrees of freedom are present on the boundary of the refinement zone.

The *hp*-refinement concept is depicted in Figure 3.3, where the deactivation of different topological components to achieve linear independence and compatibility of shape functions is shown.


● Active node
— Active edge
🟩 Active face
○ Inactive node due to linear independence
--- Inactive edge due to linear independence
▨ Inactive face due to linear independence
○ Inactive node due to compatibility
--- Inactive edge due to compatibility
🟥 Inactive face due to compatibility
k = 2
k = 1
k = 0

(a) One-dimensional case **(b)** Two-dimensional case **(c)** Three-dimensional case

Figure 3.3: Illustration of the multi-level *hp*-refinement scheme with two refinement levels, $k = 2$, in different spatial dimensions, adopted from [Zander et al., 2016]. The deactivation of specific topological components following a simple rule-set ensures compatibility and linear independence of the basis functions.

3.3 Solution of the coupled problem

Let $\mathcal{V}$ and $\tilde{\mathcal{V}}$ be the space of test functions for the displacements and phase-field solutions respectively, defined as

$$\mathcal{V} = \{\mathbf{v} : v_i \in H^1(\Omega), v_i|_{\Gamma_D} = \bar{u}_i\} \tag{3.3}$$

and

$$\tilde{\mathcal{V}} = \{q : q_i \in H^1(\Omega)\}, \tag{3.4}$$

where H^1 is a Sobolev space of degree one. The weak form of the coupled quasi-static problem in (2.6) in the context of the FCM reads:

Find $\mathbf{u} \in \mathcal{V}$ and $s \in \tilde{\mathcal{V}}$, such that

$$\begin{aligned}(\boldsymbol{\sigma}, \nabla\mathbf{w})_{\Omega_{\text{phy}}} + (\alpha_{\text{FCM}}\,\boldsymbol{\sigma}, \nabla\mathbf{w})_{\Omega_{\text{fict}}} + (\beta\,\mathbf{u}, \mathbf{w})_{\Gamma_D} = (\rho\,\mathbf{b}, \mathbf{w})_{\Omega_{\text{phy}}} + (\mathbf{h}, \mathbf{w})_{\Gamma_D} \\ + (\beta\,\mathbf{g}, \mathbf{w})_{\Gamma_D}\,, \quad \forall\,\mathbf{w} \in \mathcal{V}\end{aligned} \tag{3.5a}$$

$$\left(\left[\frac{4\,l_0}{G_c}(1-\eta)\mathcal{H}+1\right]\,s,q\right)_{\Omega_{\text{phy}}}+\left(\alpha_{\text{FCM}}\left[\frac{4\,l_0}{G_c}(1-\eta)\mathcal{H}+1\right]\,s,q\right)_{\Omega_{\text{fict}}}+\left(4\,l_0^2\,\nabla s,\nabla q\right)_{\Omega_{\text{phy}}}$$
$$+\left(\alpha_{\text{FCM}}\,4\,l_0^2\,\nabla s,\nabla q\right)_{\Omega_{\text{fict}}}=(1,q)_{\Omega_{\text{phy}}}\,,\quad\forall\,q\in\tilde{\mathcal{V}}. \tag{3.5b}$$

Here, $(\cdot,\cdot)$ represents the L^2-scalar product and β is the penalty parameter for the weak imposition of Dirichlet boundary conditions with the penalty method [Nagaraja et al., 2018]. These nonlinear equations are discretized in a finite element framework following the Bubnov-Galerkin formulation with Integrated Legendre shape functions, to arrive at the equation

$$\mathbf{K}^{uu}\Delta\mathbf{d}=\mathbf{F}^{u} \tag{3.6}$$

for displacements, and

$$\mathbf{K}^{ss}\Delta\Phi=\mathbf{F}^{s} \tag{3.7}$$

for the phase-field. The symbols $\mathbf{K}^{uu}$ and $\mathbf{K}^{ss}$ represent the stiffness matrices, $\mathbf{F}^{u}$ and $\mathbf{F}^{s}$ the force vectors, and $\mathbf{d}$ and Φ the nodal values. For a detailed derivation, the reader is referred to [Nagaraja et al., 2018].

The discretized system of equations is solved using a staggered scheme for each displacement step. In each step of the staggered scheme, displacements are obtained first using the current phase-field values, and then the phase-field is obtained assuming fixed displacements. The staggered steps continue until a maximum number of staggered steps is reached or when the criterion

$$\max\left[\left(\Delta\mathbf{d},\Delta\mathbf{d}\right),\left(\Delta\Phi,\Delta\Phi\right)\right]<\varepsilon_{\text{stag}} \tag{3.8}$$

is met, where $\varepsilon_{\text{stag}}$ is a predefined tolerance. Due to the nonlinearity of (3.5), the Newton-Raphson method is used for the incremental solution of each equation within staggered steps. This requires the consistent linearization of the terms in the weak form. Details about the nonlinear formulation can be found in [Ambati et al., 2015].

To implement the irreversibility condition with the history variables (2.9), a uniform voxel grid is used to store data at each integration point. Each voxel has two variables to store values of the tensile elastic strain energy density $\Psi^{+}(\boldsymbol{\varepsilon})$, one for the previous and the other for the current displacement step. At the beginning of a new displacement step, the two variables of each voxel are swapped, leaving the last converged values in the variable of the previous displacement step.

Chapter 4

Parallel phase-field computations

This chapter covers the implementational details for the efficient parallel solution of large phase-field fracture problems in computing clusters. The chapter starts with a brief review of basic parallel computing concepts: limits of serial computing and how parallel computing comes to the rescue, different hardware architectures and their related software models, and measures that allow the quantification of parallel performance. Then, a thorough explanation of the algorithms needed for efficient parallel mesh handling is given, covering the adaptive refinement based on the phase-field solution and the repartitioning of the meshes. Finally, the chapter discusses the selection of the linear solver and preconditioning technique used for the solution of the distributed linear systems arising from a phase-field analysis.

4.1 Parallel computing fundamentals

4.1.1 Serial vs. parallel computing

In *serial computing*, a program consists of a series of instructions that are executed on a single processor one after another, one at a time, to complete a certain task. This computing paradigm has reached its limits in the past decades for two main reasons:

- To reduce the run time of a serial program, one has to increase the clock frequency of the processor and perform serial optimizations of the code. However, data on the evolution of CPU technology shows that the clock frequency has plateaued in the past decade [Robey and Zamora, 2021]. This is unlikely to be reversed as the processor speed goes hand-in-hand with power consumption and therefore the generation of heat that is impossible to dissipate by traditional means.
- To compute larger problems, the amount of computing resources such as main memory and disk space has to be scaled as well, reaching a limit where it becomes impractical.

In *parallel computing*, a problem is broken down into smaller parts that are further broken down into instructions. All parts are executed simultaneously on different processors employing an overall coordination mechanism, enabling lower execution times. Moreover, exposing parallelism in a program allows it to operate on larger compute resources, overcoming the limit in problem size. In parallel computing, a *process* is defined as an instance of a program being executed, created by the operating system. A process consists of the executable program, a block of memory, descriptors of its resources, security information, and information about the state of the process. There are two approaches for the parallelization of programs. On one hand, there is *task parallelism*, which is the partition of various *tasks* to be done among several processes. On the other hand, there is *data parallelism*, which is the partition of the *data* among several processes, each of them performing similar tasks. In this work, only data parallelism is used. For the coordination of processes, a parallel program has to account for the following additional aspects not present on a serial program:

- **Communication**: sending/receiving data from one process to/from another.
- **Load balancing**: distribution of the computations such that all processes perform approximately the same amount of work.
- **Synchronization**: processes wait for other processes at some part of the program before continuing the execution.

4.1.2 Hardware and software models for parallel computing

The hardware used for parallel computing applications consists of a set of nodes that are connected with one or more networks, sometimes called interconnect. Each node contains many processors, Dynamic Random Access Memory (DRAM) modules, and other components to support the processing and transfer of data within the node. There are different parallel hardware architectures, the most basic ones being:

- **Distributed memory architecture**: each CPU has its own memory and is connected to other CPUs in a network. The access of an off-node memory location must be done explicitly by the programmer. Good scalability is achieved by adding more nodes to the network.
- **Shared memory architecture**: each CPU is connected to the same shared memory. While this simplifies programming, it can produce memory conflicts such as the correctness of the data and performance issues. The scalability is limited, as the addition of more CPUs does not increase the amount of available memory.
- **Hybrid distributed-shared memory architecture**: it is used by today's largest and faster computers and consists of several shared memory machines connected to others in a network, benefiting in that way of the advantages of both shared and distributed memory architectures.

The software models for parallel computing arise from the underlying hardware architectures. The most common ones are:

- **Process-based parallelization**: developed for distributed memory architectures, it consists of the division of the program into different processes, called ranks, each of which has its own memory space and set of instructions. The data is moved (communicated) between processes via *messages*. A common library that implements this model is the *Message Passing Interface* (MPI) [Clarke et al., 1994]. In the present work, the word *processes* refers to MPI processes unless stated otherwise.

- **Thread-based parallelization**: in this model several instruction pointers (threads) are spawned within the same process, allowing to easily share portions of the memory between threads without communication. One of the leading threading systems is *Open Multi-Processing* (OpenMP) [Dagum and Menon, 1998]. In the present work, the word *thread* refers to OpenMP threads unless stated otherwise.
- **Hybrid parallelization**: it is the combination of process-based and thread-based parallelization, where threads use on-node local data to perform computationally intensive operations and communication between nodes happens via message passing. The present work uses this approach with MPI and OpenMP.

4.1.3 Parallel performance

To measure how well the computational resources are utilized in a given program, parallel performance measures need to be defined. The most important measures are:

- **Speedup**: is the ratio between serial and parallel run times t of the same program.
- **Parallel efficiency**: is the ratio between the speedup and the number of processes.
- **Scalability**: is a measure of the quality of a parallel algorithm and can be distinguished between *strong scaling*, which considers the speedup for an increasing number of processes with the same problem size, and *weak scaling*, which considers the speedup when the problem size is increased proportionally to the number of processes.

In parallel computing, the potential speedup s for strong scaling can be calculated with Amdahl's law

$$S = \frac{t_{\text{serial}}}{t_{\text{parallel}}} = \frac{1}{s + \frac{p}{N}}, \tag{4.1}$$

where p and $s = 1-p$ are the parallel and serial fractions of the code, respectively, and N is the number of processes. Figure 4.1 shows the speedup curves for different parallel fractions based on Amdahl's law. It is evident that the speedup is limited by the serial fraction of the code, i.e., there is a point where using more processors does not provide a substantial reduction in run time. The ideal case, that corresponds to a code that is completely parallelized, is a line with a slope of 1, meaning that the speedup grows in the same proportion as the number of processes does. In practice, this cannot be attained, as there is always some communication overhead, load imbalance, or synchronization of the processes that reduce the speedup. However, there are cases where superlinear behavior can be observed [Ristov et al., 2016].

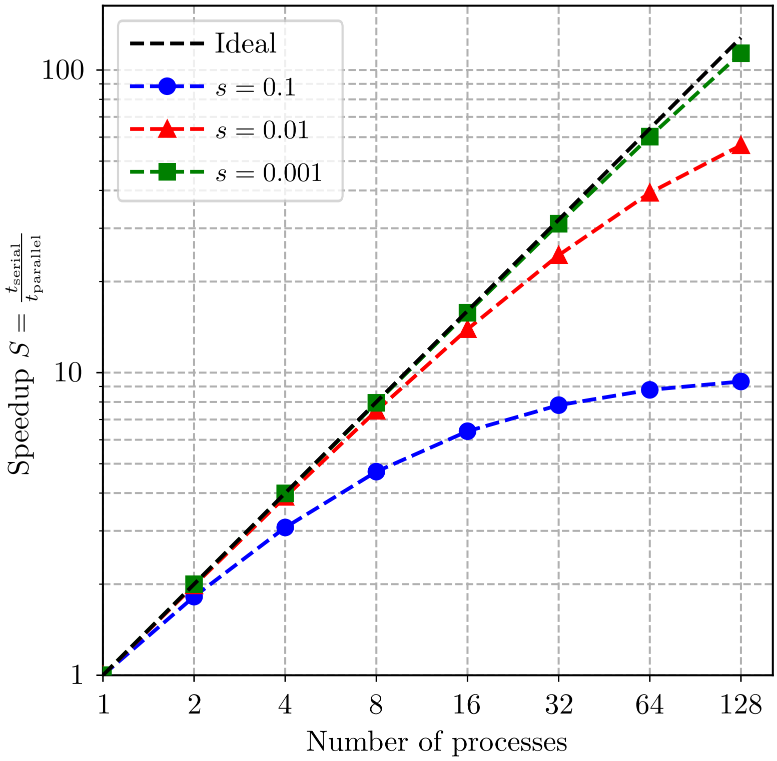


Figure 4.1: Plot of the speedup according to Amdahl's law for different serial fractions of the code.

To calculate the speedup in the case of weak scaling, Gustafson-Barsis's law [Gustafson, 1988] is used

$$S = s + p \times N. \tag{4.2}$$

This law addresses the fact that programmers tend to increase the problem size as more computational resources become available, which is in contradiction to Amdahl's fixed-size problem assumption. Looking at (4.2), it is clear that it is possible to achieve better parallel performance than is implied by (4.1).

4.2 Parallel mesh handling

4.2.1 Terminology

This section introduces the categories of cells used in this work and their nomenclature. Let $\mathcal{G}$ be a Cartesian grid composed of cells C with indices j on different refinement levels l. In the initial state, the Cartesian grid is denoted by $\mathcal{G}^{\text{init}}$, and every cell belonging to it has a refinement level $l = 0$. To identify a cell unambiguously, both the index and the refinement level need to be specified, as cells with the same index can appear in different refinement levels.

Grid cells of the Cartesian grid can be grouped into different categories, see Figure 4.2:

- **Base and leaf cells**: cells that are at the lowest refinement level $l = 0$ are called base cells, while cells that have no children are called leaf cells.

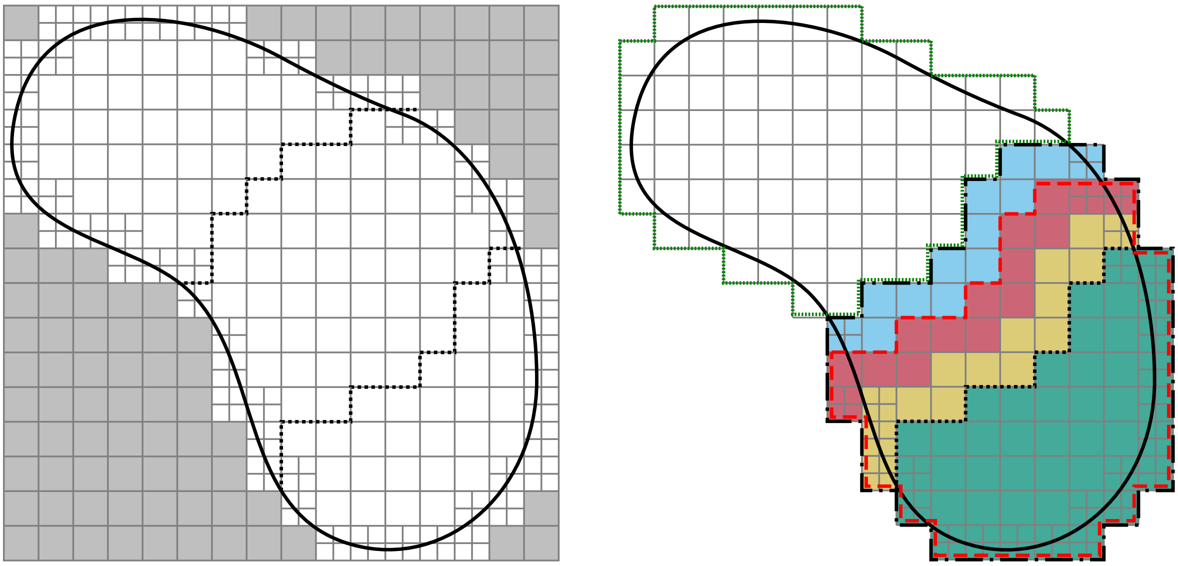
- **Inside and outside cells**: cells that are intersected by the domain boundary or completely inside of the physical domain are called inside cells and denoted by C^{in}. On the other hand, cells that lie completely outside of the physical domain are called outside cells and denoted by C^{out}, see Figure 4.2a. The set composed of all inside cells is denoted by $\mathcal{C}^{\text{in}}$, while $\mathcal{C}^{\text{out}}$ is used for the set of all outside cells. The union of these sets conform the complete Cartesian grid $\mathcal{C}^{\text{in}} \cup \mathcal{C}^{\text{out}} = \mathcal{G}$.

- **Owned and ghost cells**: cells of a distributed Cartesian grid are assigned to unique owning processes. Cells that belong to a particular process are called owned cells of that process and are denoted by C^{own}. Cells that are adjacent to owned cells and that are owned by remote processes are called ghost cells and denoted by C^{ghost}. The mesh handling algorithms developed in this work require three layers of ghost cells that are denoted by $C^{\text{ghost},0}$, $C^{\text{ghost},1}$, and $C^{\text{ghost},2}$, see Figure 4.2b. The set composed of all owned cells is denoted by $\mathcal{C}^{\text{own}}$, while $\mathcal{C}^{\text{ghost}}$ is used for the set of all ghost cells. It is important to note that ownership is defined at the base level $l = 0$, i.e., a cell and all its children are always owned by the same process.

- **Inner boundary cells**: owned cells that are adjacent to ghost cells are called inner boundary cells and denoted by C^{inner}. These cells coexist as ghost cells in remote processes and therefore three layers of inner boundary cells $C^{\text{inner},0}$, $C^{\text{inner},1}$, and $C^{\text{inner},2}$ are needed, see Figure 4.2c. The set composed of all inner boundary cells is denoted by $\mathcal{C}^{\text{inner}}$.

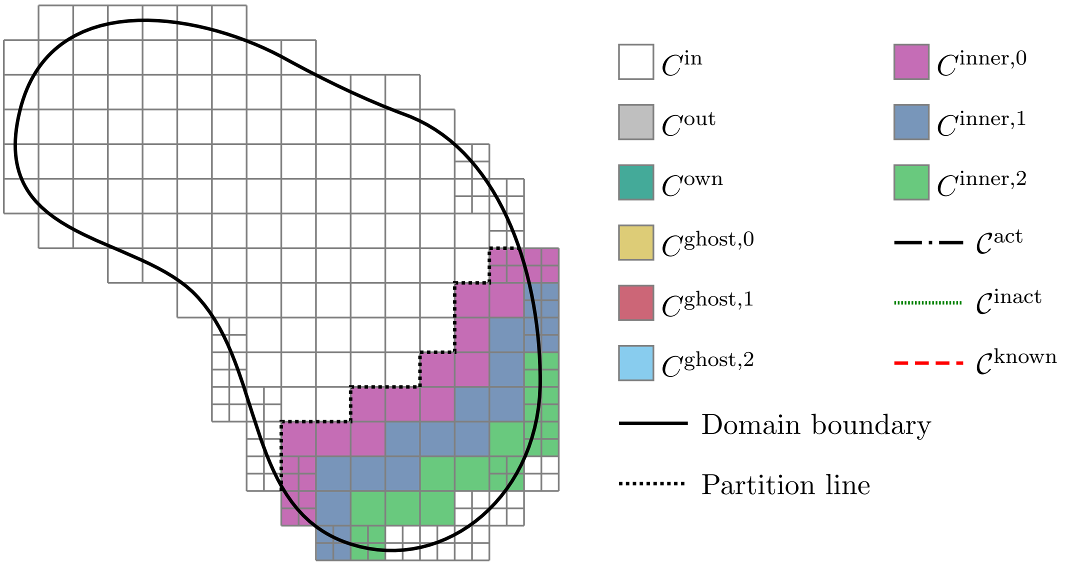
- **Active and inactive cells**: cells that are used for mesh generation in a particular process are called active cells and are denoted by C^{act}. Conversely, cells that are not used for mesh generation in a particular process are called inactive cells and are denoted by C^{inact}. The set composed of all active cells is formed by $\mathcal{C}^{\text{own}} \cup \mathcal{C}^{\text{ghost}}$ and denoted by $\mathcal{C}^{\text{act}}$, while $\mathcal{C}^{\text{inact}}$ is used for the set of all inactive cells, see Figure 4.2b.

- **Known cells**: cells that are used for mesh compatibility enforcement (see 4.2.3) are called known cells and are denoted by C^{known}. The set composed of all known cells is formed by $\mathcal{C}^{\text{own}} \cup \mathcal{C}^{\text{ghost},0} \cup \mathcal{C}^{\text{ghost},1}$ and denoted by $\mathcal{C}^{\text{known}}$, see Figure 4.2b.



(a) Inside and outside cells.

(b) Owned and ghost cells, active and known cell sets of the bottom right partition.

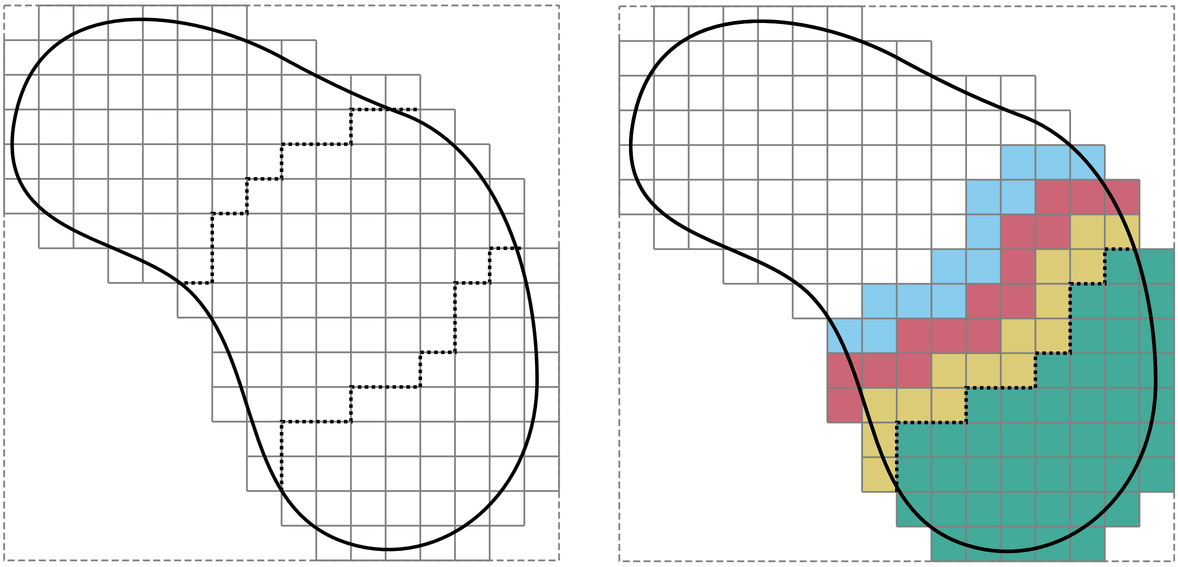

$\mathcal{C}^{\text{in}}$
$\mathcal{C}^{\text{out}}$
$\mathcal{C}^{\text{own}}$
$\mathcal{C}^{\text{ghost},0}$
$\mathcal{C}^{\text{ghost},1}$
$\mathcal{C}^{\text{ghost},2}$
$\mathcal{C}^{\text{inner},0}$
$\mathcal{C}^{\text{inner},1}$
$\mathcal{C}^{\text{inner},2}$
$\mathcal{C}^{\text{act}}$
$\mathcal{C}^{\text{inact}}$
$\mathcal{C}^{\text{known}}$
Domain boundary
Partition line

(c) Inner boundary cells of the bottom right partition.

Figure 4.2: Definition of different types of cells for a grid $\mathcal{G}$ partitioned among three processes, from the point of view of the bottom right partition.

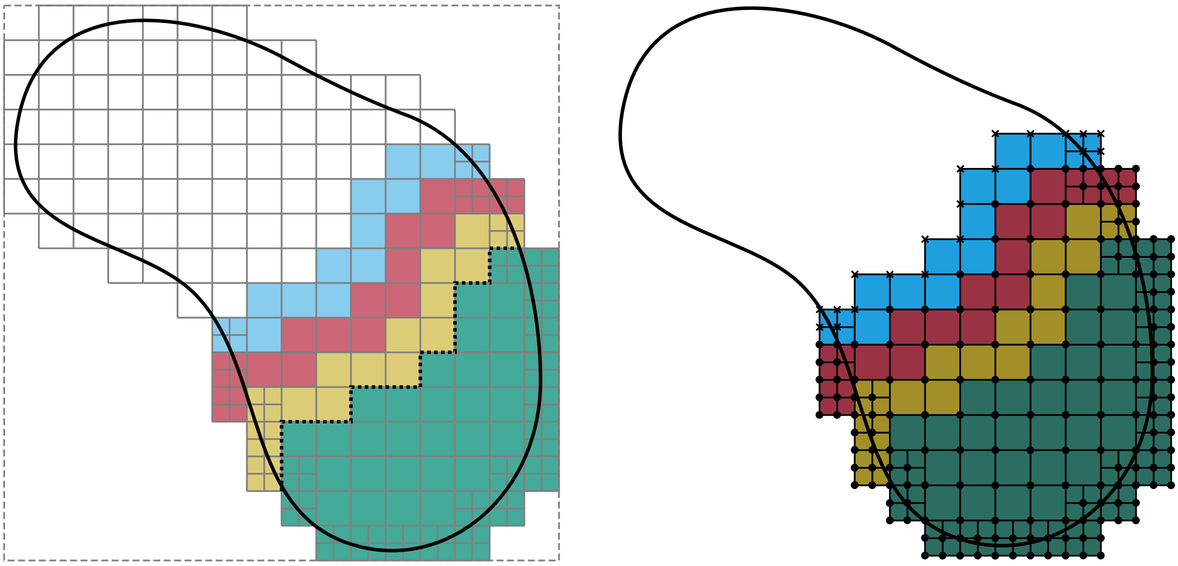
4.2.2 Mesh generation

In the present framework, a mesh is created from a Cartesian grid $\mathcal{G}$ that is structurally analogous but computationally cheaper than the former in the sense that it requires less storage capacity and allows for easier and faster refinement operations [Jomo, 2021]. Each of the processes stores the entire initial Cartesian grid $\mathcal{G}^{\text{init}}$ but only keeps track of the current state of its active cells $\mathcal{C}^{\text{act}}$. We denote the Cartesian grid stored in a given process by $\mathcal{G}_{\text{rank}}$. With the active cells in $\mathcal{G}_{\text{rank}}$, each process generates a high-order mesh $\mathcal{T}_{\text{rank}}$ at a given refinement level l_m. The full computational mesh is the union of all owned elements in local meshes among all processes. In the framework of [Jomo, 2021], meshes can be created at depths $l_m > 0$. However, the present work focuses on algorithms designed for $l_m = 0$ as problems commonly solved in fracture mechanics do not require higher mesh creation depths, avoiding unnecessary parallel overhead. There is always a one-to-one relationship between active cells and the generated mesh elements at every refinement level. In the context of a phase-field analysis, both the phase-field and the elastic meshes are constructed from the same Cartesian grid and the algorithm comprises the following steps, see Figure 4.3:



(a) An initial Cartesian grid $\mathcal{G}^{\text{init}}$ is generated on each process and the inside cells are partitioned.

(b) Three layers of ghost cells are determined.



(c) Active cells are refined according to predefined strategies.

(d) Two meshes are generated following the refined grid.

Figure 4.3: Distributed mesh generation starting from a Cartesian grid $\mathcal{G}^{\text{init}}$ where inside cells are partitioned among three processes, as seen from the point of view of the bottom right partition.

1. An initial Cartesian grid $\mathcal{G}^{\text{init}}$ is generated on each process, see Figure 4.3a.
2. The sets $\mathcal{C}^{\text{in}}$ and $\mathcal{C}^{\text{out}}$ are determined by an inside-outside test and the state of the cells is set accordingly.
3. Cells in $\mathcal{C}^{\text{in}}$ are partitioned among processes using the geometric `Zoltan` partitioner [Devine et al., 2002]. The result of this operation is the set of owned cells $\mathcal{C}^{\text{own}}$ in the calling process.
4. Cells in $\mathcal{C}^{\text{own}}$ are marked as active and each process registers its corresponding rank. All other (remote) cells are left with an invalid rank represented by -1.
5. Three levels of ghost cells $\mathcal{C}^{\text{ghost},0}$, $\mathcal{C}^{\text{ghost},1}$, and $\mathcal{C}^{\text{ghost},2}$ are determined, see Figure 4.3b. For this, the following steps are carried out, see Figure 4.4:
    (a) For each cell in $\mathcal{C}^{\text{own}}$, one layer of surrounding cells is determined, see Appendix A.
    (b) For each of the surrounding cells determined in 5a, the owning process' rank is compared to that of the current (calling) process. If they are different, then the surrounding cell belongs to $\mathcal{C}^{\text{ghost},0}$.
    (c) Steps 5a and 5b are repeated for each cell in $\mathcal{C}^{\text{ghost},0}$ to get $\mathcal{C}^{\text{ghost},1}$, excluding cells that are already in $\mathcal{C}^{\text{ghost},0}$.
    (d) Steps 5a and 5b are repeated for each cell in $\mathcal{C}^{\text{ghost},1}$ to get $\mathcal{C}^{\text{ghost},2}$, excluding cells that are already in $\mathcal{C}^{\text{ghost},0}$ and $\mathcal{C}^{\text{ghost},1}$.

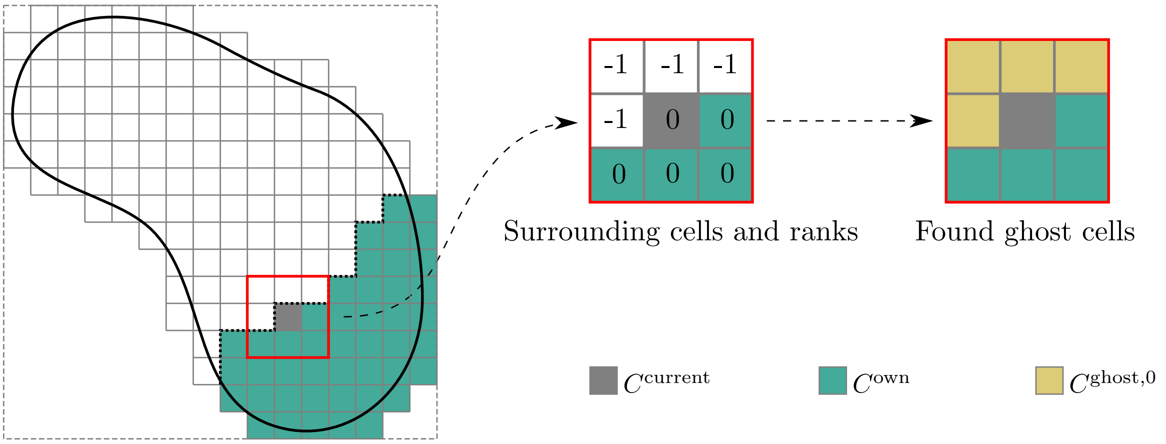

-1 -1 -1
-1 0 0
0 0 0
Surrounding cells and ranks
Found ghost cells
$\mathcal{C}^{\text{current}}$
$\mathcal{C}^{\text{own}}$
$\mathcal{C}^{\text{ghost},0}$

Figure 4.4: Scheme of the steps needed to find ghost cells of the first layer $\mathcal{C}^{\text{ghost},0}$.

6. Active cells $\mathcal{C}^{\text{act}}$ are refined according to the initial strategy, e.g. refinement towards the boundary of the domain, see Figure 4.3c.
7. Cells in $\mathcal{C}^{\text{ghost}}$ are marked as active and the corresponding ranks are registered. At this point, each process knows who owns each of its active cells, information that is essential to determine the communication patterns downstream in the simulation pipeline.
8. With the resulting grid collection two meshes are generated, one for the phase-field problem and another one for the elastic problem, denoted by $\mathcal{T}^{\text{PF}}$ and $\mathcal{T}^{\text{Elastic}}$ respectively, see Figure 4.3d.

9. The consistency of the degrees of freedom (DOFs) numbering across all processes is enforced, see 4.2.3.

4.2.3 Mesh compatibility

Meshes generated locally by each process as explained in 4.2.2 have to be connected so that analysis-suitable meshes are obtained. An analysis-suitable mesh, as noted in [Jomo, 2021], fulfills two conditions:

1. Each DOF has a unique global identifier and owning process,
2. The communication patterns used to update data of ghost elements' DOFs have been established.

The mesh connection process is formally known as *enforcement of parallel mesh compatibility*, and results in a consistent enumeration of DOFs among all processes. In the following section, basic aspects of mesh compatibility and some particularities that arise in the context of phase-field analyses are noted. For implementational details, the reader is referred to [Jomo, 2021].

DOFs terminology

In a distributed mesh, DOFs can be divided into the following categories depending on the region where the support of the associated basis function lies:

- **Valid and invalid DOFs**: DOFs associated with basis functions that intersect at least one basis function which is fully supported on elements corresponding to known cells $\mathcal{C}^{\text{known}}$ are valid DOFs. All remaining DOFs are regarded as invalid.
- **Internal and interface DOFs**: valid DOFs associated with basis functions fully supported on owned elements are internal DOFs. All remaining valid DOFs are interface DOFs.
- **Owned and remote DOFs**: DOFs of a distributed mesh are assigned to unique owning processes. DOFs that belong to a particular process are called owned DOFs, whereas those that are present in a local mesh but belong to remote processes are called remote DOFs. After solving the distributed linear system of equations of the problems, only owned DOFs have correct solution values. Remote DOFs get their values via inter-process communication. The ownership of internal DOFs is trivially assigned, as their basis functions exist in only one process. However, ownership of interface DOFs can be determined with different approaches [Jomo, 2021]. In the present work, ownership of an interface DOF is assigned to the element with the lowest grid index that contains the DOF's topological entity.

These categories can be seen in Figure 4.5 and are essential to understand the algorithms presented in the following sections.

Consistency of data in the context of a phase-field analysis

A process is responsible for handling the data of its owned DOFs, but it can also store data associated with remote DOFs. It is crucial that the data of remote DOFs stored in a process is the same as the data of the same DOF in the process that owns it. In contrast to [Jomo, 2021], the present work uses three layers of ghost elements. This ensures that all DOFs that belong to elements up the second ghost layer are valid (see Figure 4.5) and therefore the exact preconditioner for the linear solver (see 4.3.2) can be constructed without communication.

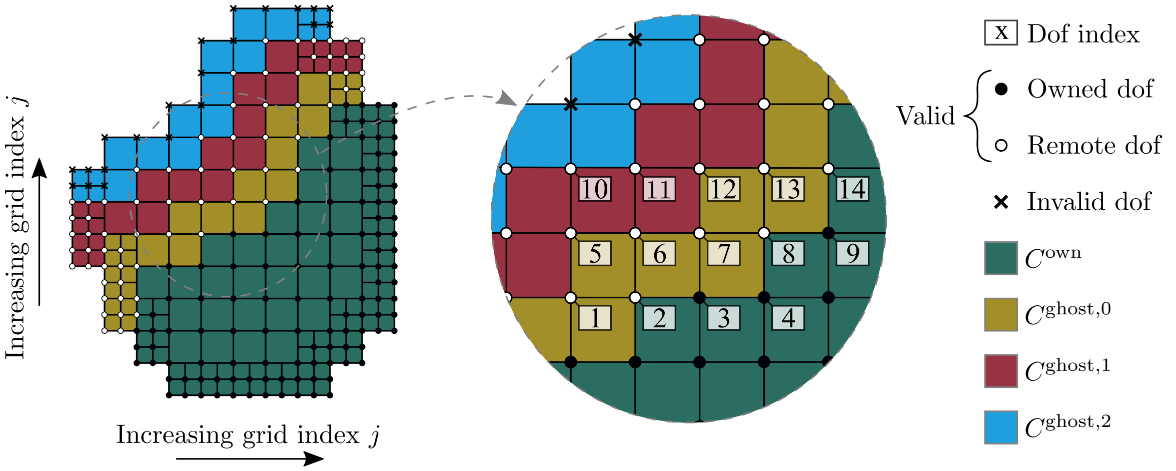


Figure 4.5: Scheme showing the categories of DOFs in a local mesh.

As explained in [Jomo et al., 2019], the first layer of ghost elements is needed for the setup of the stiffness matrix, while the second layer of ghost elements is needed for the setup of the preconditioner. For linear analyses, the DOFs interfacing with the third layer of ghost elements need not be valid, as they do not contribute to the corresponding matrix entries used for the preconditioner. However, in nonlinear phase-field analysis, the contribution to these entries depends on the value of the solution at the integration points of the second layer of ghost elements, which in turn depends on the value of the DOFs interfacing with the third layer of ghost elements. Consequently, consistency of data has to be enforced including DOFs up to the second ghost layer of elements.

To illustrate this, take the example of DOF 3 in Figure 4.5. To get the exact contribution of this DOF to the stiffness matrix, all elements surrounding the associated node have to be integrated. As the problem is nonlinear, the integration points of those elements must have the correct solution, and this, in turn, requires that the solution values of remote DOFs 2, 6, 7, and 8 must be correct as well. Additionally, the construction of the preconditioner requires correct stiffness matrix entries corresponding to DOFs that share support with DOF 3, i.e. DOFs 2, 6, 7, and 8, see 4.3.2. Again, all elements surrounding these DOFs have to be integrated, and therefore DOFs 1, 5, 10, 11, 12, 13, and 14 must have the correct solution values.

4.2.4 Adaptive mesh refinement

The solution of the phase-field around the crack has steep gradients that require a fine discretization to resolve them accurately, see Figure 2.2. To achieve this goal, the phase-field and the elastic mesh are refined adaptively depending on the current phase-field solution. Elements that are close to the crack, i.e. have a phase-field value lower than a predefined threshold are $h-$refined after each staggered step.

In a parallel setting, the refinement state of all active cells needs to be correct, so that mesh compatibility is maintained. However, this presents a major difficulty: the solution computed on elements that correspond to the third layer of ghost cells $\mathcal{C}^{\text{ghost},2}$ is incorrect, as some of the DOFs that belong to them are invalid, see Figure 4.6. Consequently, the refinement in these elements is incorrect and mesh compatibility is violated.

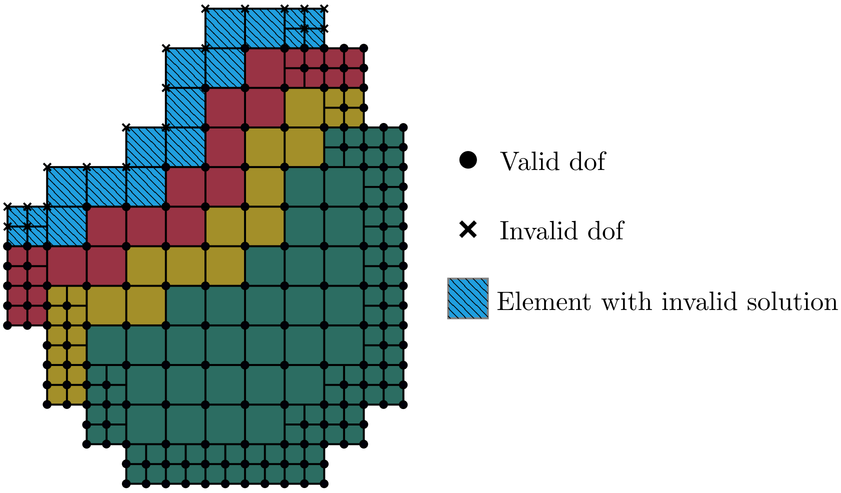


Figure 4.6: Elements of a local mesh with invalid solution.

To overcome this problem, the solution-based refinement strategy is modified. In a first step, refinement based on the solution is performed on owned cells. Then, ghost cells replicate the refinement from neighboring processes by communicating indices of cells to be refined and their refinement levels.

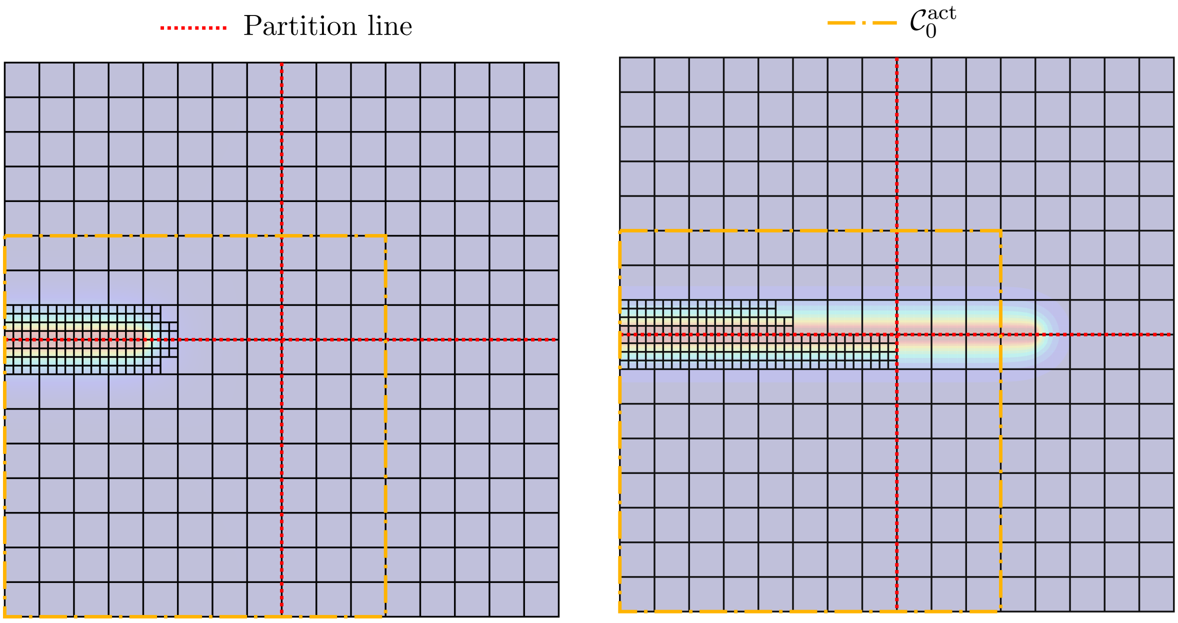
More precisely, the refinement is done in the following steps, see Figure 4.7:

1. All owned cells in $\mathcal{G}_{\text{rank}}$ with a phase-field value lower than a threshold, are refined up to a predefined depth l_r, see Figure 4.7b.

2. All processes wait for each other until they all finish the previous step. This is accomplished with an `MPI_Barrier` [Clarke et al., 1994] and is needed to ensure that the correct refinement levels are communicated between the processes.

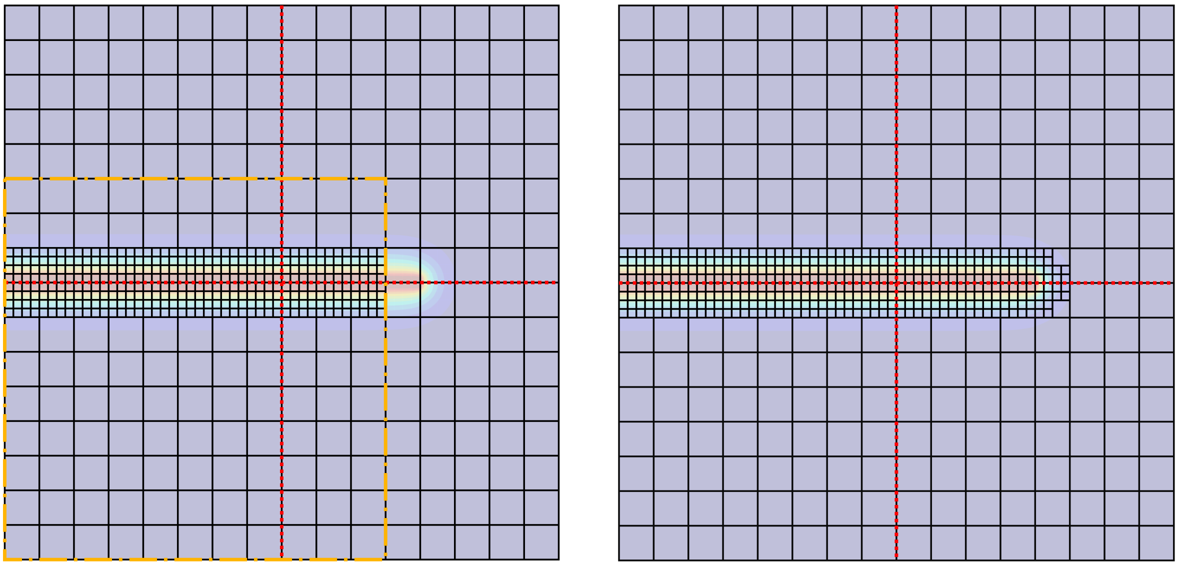
3. Three levels of inner boundary cells $\mathcal{C}^{\text{inner},0}$, $\mathcal{C}^{\text{inner},1}$, and $\mathcal{C}^{\text{inner},2}$ are determined. To this end, the following steps are carried out, see Figure 4.8:

    (a) For each cell in $\mathcal{C}^{\text{ghost},0}$, one layer of surrounding cells is determined, see Appendix A.



(a) Grid at displacement step u_0.

(b) Owned cells in $\mathcal{G}_0$ are refined based on the phase-field values at displacement step u_1.



(c) The refinement of the ghost cells at displacement step u_1 is replicated from owned cells of remote processes. Note that inactive cells do not change their refinement state.

(d) Global view of the grids after refinement at displacement step u_1.

Figure 4.7: Parallel adaptive refinement based on the phase-field solution for a crack propagating between displacement steps u_0 and u_1, as seen from the bottom left partition with rank 0.

(b) For each of the surrounding cells determined in 3a, the owning process' rank is compared to that of the calling process. If they are equal, then the surrounding cell belongs to $\mathcal{C}^{\text{inner},0}$.

(c) Steps 3a and 3b are repeated for each cell in $\mathcal{C}^{\text{inner},0}$ to get $\mathcal{C}^{\text{inner},1}$, excluding cells that are already in $\mathcal{C}^{\text{inner},0}$.

(d) Steps 3a and 3b are repeated for each cell in $\mathcal{C}^{\text{inner},1}$ to get $\mathcal{C}^{\text{inner},2}$, excluding cells that are already in $\mathcal{C}^{\text{inner},0}$ and $\mathcal{C}^{\text{inner},1}$.

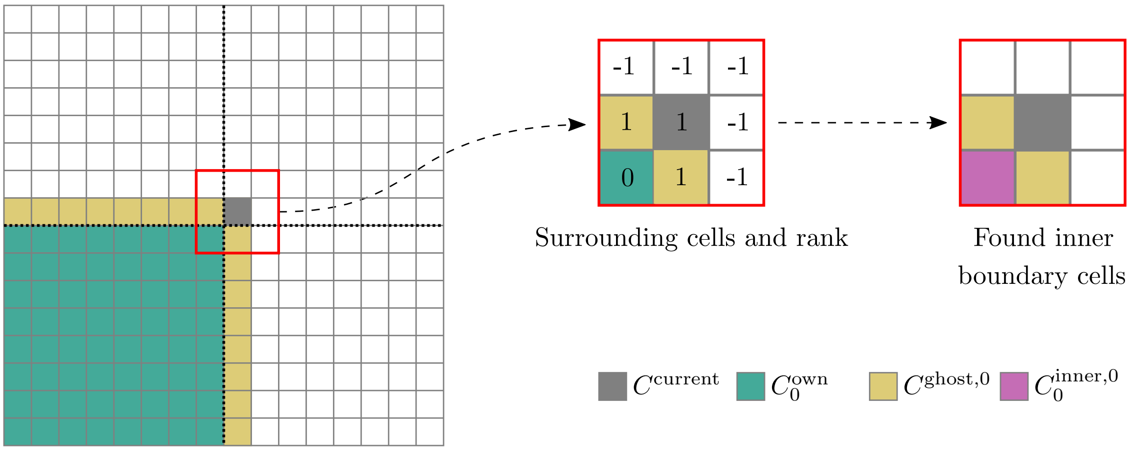


Figure 4.8: Scheme of the steps needed to find inner boundary cells of the first layer $C_0^{\text{inner},0}$, as seen from the bottom left partition with rank 0.

4. The refinement data (index j and refinement depth l) that each cell in $\mathcal{C}^{\text{inner}}$ needs to send to remote processes is determined. This is done as follows, see Figure 4.9:

   (a) Three layers of surrounding cells are determined, see Appendix A.

   (b) For each of the surrounding cells determined in 4a, the owning process' rank is compared to that of the current process. If they are different, then the current cell is a ghost cell in that remote process and therefore needs to send its refinement data. Figure 4.9 shows the case for a cell C^{current} owned by rank 1. Three layers of surrounding cells are determined, and as they are owned by either rank 0, rank 2, or rank 3, the refinement data has to be sent to the corresponding processes.

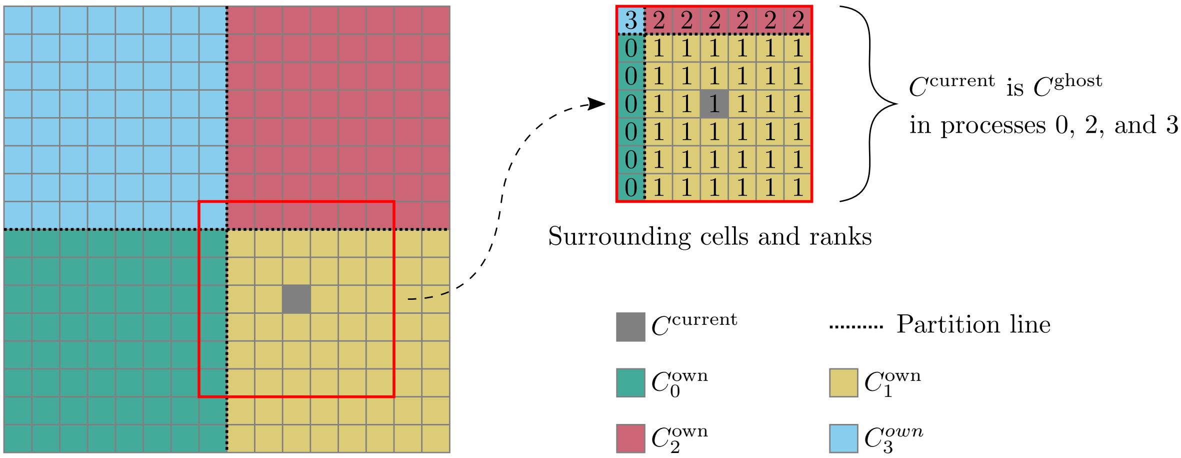


Figure 4.9: Scheme showing how to determine the processes to which the refinement data has to be sent to.

   (c) All indices and depths in the hierarchy of the current cell are extracted up to the second to deepest refinement level $l = l_{\max} - 1$, see Figure 4.10. The deepest refinement level indices need not be communicated, as it is only necessary to know the indices of their parent cells to perform the refinement and replicate the deepest level in the remote process.

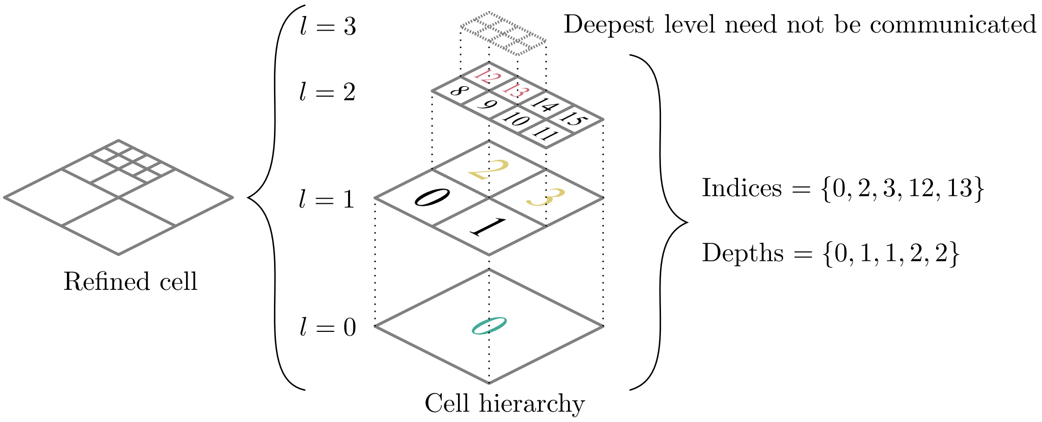

$l = 3$
$l = 2$
$l = 1$
$l = 0$
Deepest level need not be communicated
8
9
10
11
12
13
14
15
0
1
2
3
0
Refined cell
Indices $= \{0, 2, 3, 12, 13\}$
Depths $= \{0, 1, 1, 2, 2\}$
Cell hierarchy

Figure 4.10: Extraction of indices and depths from a refined cell for communication.

(d) For each of the cells obtained in 4c and each of the ranks determined in 4b, tuples (j, l, rank) are stored.

(e) Tuples defined in 4d are sorted by rank and then by depth. This is done to ensure that the cells are always found in the Cartesian grid of the remote process, as noted in 7.

5. The number of messages each processor needs to receive is determined in three steps, see Figure 4.11:

   (a) An array of zeros with as many entries as the number of processes is allocated in every process. Each entry of this array represents a process and the corresponding index is equal to the rank of that process.

   (b) Each process sets to 1 the entries corresponding to the processes it needs to send a message to. This information is taken from the tuples constructed in 4d.

   (c) An `MPI_Allreduce` is performed with an `MPI_SUM` operation [Clarke et al., 1994], giving, as a result, a vector where each entry represents the number of messages the calling process needs to receive.

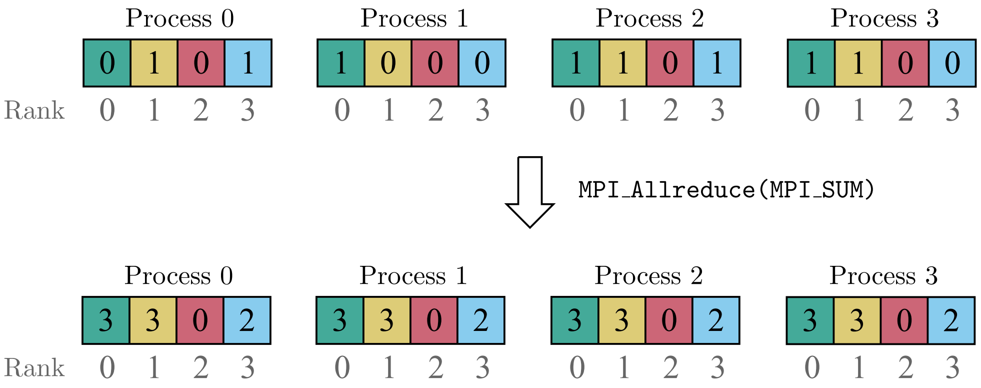


Figure 4.11: Scheme representing the operation to determine the number of messages to receive when a grid is partitioned among four processes.

These steps are exemplified in Figure 4.11 for a partition among four processes. Here, process 0 sends to processes 1 and 3, process 1 sends to process 0, process 2 sends to processes 0, 1, and 3, and process 3 sends to processes 0 and 1. After the `MPI_Allreduce` operation, it is determined that processes 0 and 1 receive 3 messages, process 2 receives no messages and process 3 receives 2 messages.

6. A non-blocking point-to-point communication of the cell indices j and depths l that were determined in 4d is performed. As a process does not know how many cells were refined in a remote process and therefore does not know the size of the messages it is receiving, message sizes are queried via `MPI_Probes` [Clarke et al., 1994].

7. Cells corresponding to received indices and depths are refined as shown in Figure 4.7c, from the lowest to the highest depths. This refinement order, which is a consequence of the pre-sorting of tuples in 4e, ensures that all received indices will be found on the local Cartesian grid $\mathcal{G}_{\text{rank}}$. At this point, all processes have refined their grids, see Figure 4.7d.

8. The phase-field and the elastic meshes are refined by following the refinement of the Cartesian grid.

4.2.5 Mesh repartitioning

When a crack propagates, the mesh is refined as explained in the previous section. If the initial partitioning of the meshes remains unchanged, it can lead to an uneven distribution of the workload among processes due to a higher integration effort on those processes that have more elements. Therefore, there is a detrimental effect on the parallel efficiency, as the processes with smaller loads finish the integration earlier and have to wait for the processes with higher loads to finish. To overcome this issue, a repartitioning of the meshes according to different strategies is proposed. Repartitioning means assigning new ownership to mesh elements, and it requires the generation of new meshes, both for the phase-field and the elastic problems, as well as the transfer of data from old meshes to new meshes, i.e. values of DOFs and values of the history variables. In the next sections, two proposed repartitioning strategies will be discussed first and then the algorithms for mesh repartitioning and transfer of data will be explained thoroughly.

Repartitioning strategies

The first and simplest repartitioning strategy, named *step-based repartitioning*, consists of waiting for a number of displacement steps predefined by the user to repartition the meshes. Though simple, this strategy can lead to the call of the repartitioning algorithm in cases where it is not needed at all, thus creating unnecessary overhead. Furthermore, the number of displacement steps to wait needs to be defined a-priori by the user, and thus requires a trial-and-error process to determine an acceptable value.

The second proposed repartitioning strategy, coined *dynamic repartitioning*, consists of measuring the load in each process and calling the repartitioning algorithm when the load in one or more processes deviates more than a predefined tolerance with respect to the ideal load. Based on the fact that the computational effort is proportional to the number of leaf cells, the proposed measure for the load consist of the count of owned leaf cells on a given process, divided by the total count of leaf cells among all processes. With this definition, the sum of the loads among all processes equals 1. The algorithm to determine whether meshes need to be repartitioned consists of the following steps:

1. The set of owned cells $\mathcal{C}^{\text{own}}$ is determined.
2. The weight of all owned cells is summed up to get the local load w_{rank} of the calling process. The weight of each cell is computed with a `cellWeightEvaluator`. By default, it counts the number of children, but there is also the possibility to implement user-defined evaluators with lambda expressions.
3. The total load among all processes is computed by summing up the local load of all processes
$$W = \sum_{\text{rank}=0}^{n_{\text{procs}}} w_{\text{rank}}. \tag{4.3}$$
This is materialized with an `MPI_Allreduce` of the local loads with an `MPI_SUM` operation.
4. The normalized load defined as the ratio between the load in the calling process and the total load is computed
$$\tilde{w}_{\text{rank}} = \frac{w_{\text{rank}}}{W}. \tag{4.4}$$
5. The ideal normalized load defined as the inverse of the number of processes is computed
$$\tilde{w}_{\text{ideal}} = \frac{1}{n_{\text{procs}}}. \tag{4.5}$$
6. The relative difference d of the normalized load with respect to the ideal normalized load is computed
$$d = \frac{|\tilde{w}_{\text{rank}} - \tilde{w}_{\text{ideal}}|}{\tilde{w}_{\text{ideal}}}. \tag{4.6}$$
7. If the relative difference computed with (4.6) is larger than a predefined tolerance ζ, then the meshes in the current process need to be repartitioned. Therefore, with $\zeta = 0$ the mesh is repartitioned in each staggered step. On the other hand, with a high enough ζ the mesh is never repartitioned.
8. To compute the total number of processes that need repartitioning, each process defines a variable with the value 1 in the case it has to repartition its meshes, and the value 0 otherwise. Then, an `MPI_Allreduce` of this variable with an `MPI_SUM` operation is performed to get the desired number.
9. The meshes in all processes are repartitioned as soon as any of the intervening processes need to repartition their meshes, i.e., the result of the reduction of the previous step is greater than zero.

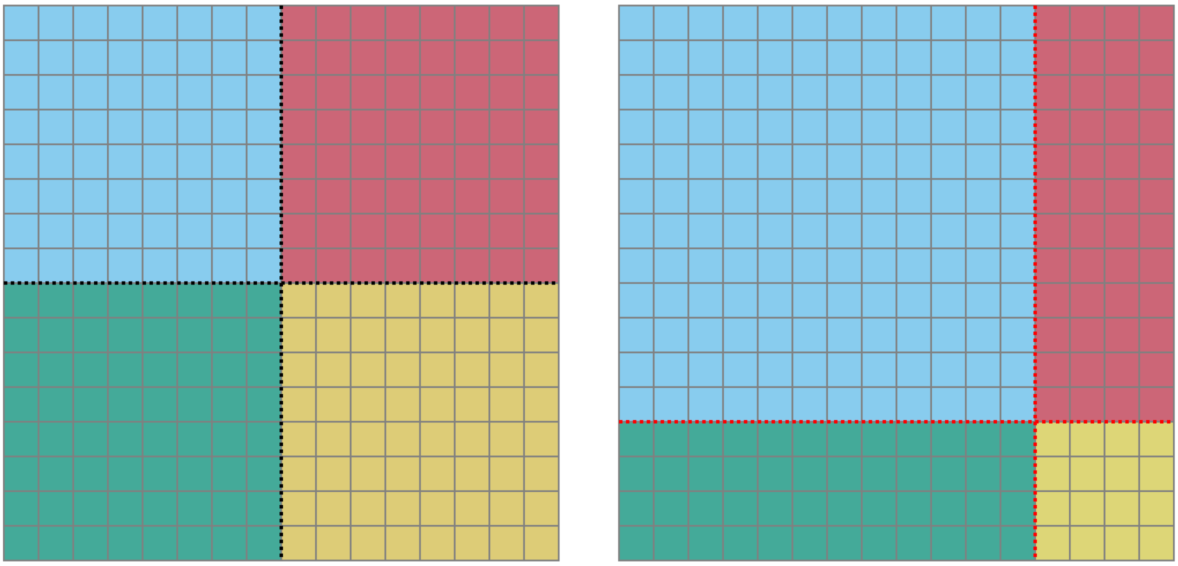
It is important to note that even though this algorithm requires collective communication among processes, it saves computational time by avoiding the call of the repartitioning algorithm with all the associated overhead when the load among processes is well balanced.

Repartitioning algorithm

The repartitioning algorithm can be broken down into two major parts. The first part consists of the redistribution of cells among processes and the generation of new elastic and phase-field meshes. The second part comprises the transfer of data from the old meshes to the new meshes, i.e. values of DOFs and values of the history variables. These two parts will be explained in detail in the following.

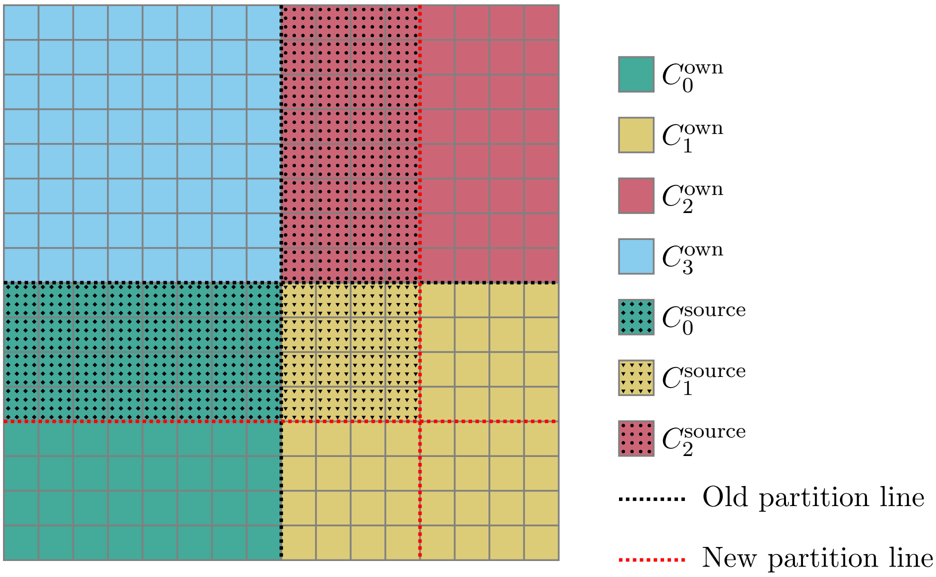
Redistribution of cells and generation of new meshes

1. Active cells $\mathcal{C}^{\text{act}}$ of the old partition are refined with the solution as explained in 4.2.4.
2. The set of owned cells of the old partition $\mathcal{C}^{\text{own}}_{\text{old}}$ is determined, see Figure 4.12a.
3. Three levels of ghost cells $\mathcal{C}^{\text{ghost},0}_{\text{old}}$, $\mathcal{C}^{\text{ghost},1}_{\text{old}}$, and $\mathcal{C}^{\text{ghost},2}_{\text{old}}$ of the old partition are looked up as explained in 4.2.2.
4. The cells in $\mathcal{C}^{\text{in}}$ are repartitioned among the intervening processes using the geometric `Zoltan` partitioner. The result of this operation is the set of new owned cells $\mathcal{C}^{\text{own}}_{\text{new}}$ of the calling process, see Figure 4.12b.
5. The cells in $\mathcal{C}^{\text{own}}_{\text{new}}$ are marked as active and each process registers its corresponding rank.
6. Three levels of ghost cells $\mathcal{C}^{\text{ghost},0}_{\text{new}}$, $\mathcal{C}^{\text{ghost},1}_{\text{new}}$, and $\mathcal{C}^{\text{ghost},2}_{\text{new}}$ are determined, marked as active, and their rank registered, analogously as in 4.2.2.
7. The set of cells that were owned by the process but are not owned anymore is called source grids and is determined as $\mathcal{C}^{\text{source}} = \mathcal{C}^{\text{own}}_{\text{old}} \setminus \mathcal{C}^{\text{own}}_{\text{new}}$, see Figure 4.12c.
8. The new rank of the source cells is queried.
9. The refinement state of the source cells $\mathcal{C}^{\text{source}}$ is sent to the new owning processes. For this, an algorithm analogous to the one in 4.2.4 is executed but in this case, using $\mathcal{C}^{\text{source}}$ instead of $\mathcal{C}^{\text{inner}}$ as “sending” cells. At this point, all new owned cells $\mathcal{C}^{\text{own}}_{\text{new}}$ have the correct refinement state.
10. The refinement of the new ghost cells $\mathcal{C}^{\text{ghost}}_{\text{new}}$ is performed as explained in 4.2.4.
11. With the resulting grid, two new meshes are generated, one for the phase-field problem and another one for the elastic problem, denoted by $\mathcal{T}^{\text{PF}}_{\text{new}}$ and $\mathcal{T}^{\text{Elastic}}_{\text{new}}$ respectively.
12. The consistency of the DOFs numbering across all processes is enforced, see 4.2.3. At this stage, the meshes are prepared for the transfer of data.



(a) Cell distribution before repartitioning.

(b) Cell distribution after repartitioning.



(c) Resulting source grids after repartitioning. These cells send their refinement data to neighboring processes.

Figure 4.12: Determination of source cells after repartitioning among four processes.

Transfer of data

1. The history variable values of $\mathcal{C}_{\text{old}}^{\text{ghost},2}$ is reset to 0. This step is needed because history values on the third layer are incorrect due to the existence of elements with invalid solutions, see Figure 4.6. After repartitioning, one of the grids of $\mathcal{C}_{\text{old}}^{\text{ghost},2}$ could fall inside $\mathcal{C}_{\text{new}}^{\text{known}}$ and therefore lead to an incorrect solution.
2. Old meshes are refined as explained in 4.2.4 to allow for consistent data transfer with the new meshes.

3. The set of cells that need to send its data to other processes is denoted by $\mathcal{C}^{\text{send}}$ and determined with the following steps, see Figure 4.13:
    (a) For each cell in $\mathcal{C}^{\text{own}}_{\text{old}}$, two layers of surrounding cells are determined, see Appendix A.
    (b) For each of the surrounding cells determined in 3a, the old and new ranks are queried. The set of old and new ranks for all surrounding cells of the current cell correspond to the processes that knew the cell before repartitioning and that know the cell after repartitioning, respectively.
    (c) The processes to which a given cell needs to send its data are the processes that know the cell after repartitioning but did not know about the cell before repartitioning. If there is at least one process to send the data to, then the current cell is added to $\mathcal{C}^{\text{send}}$, otherwise, it is not.

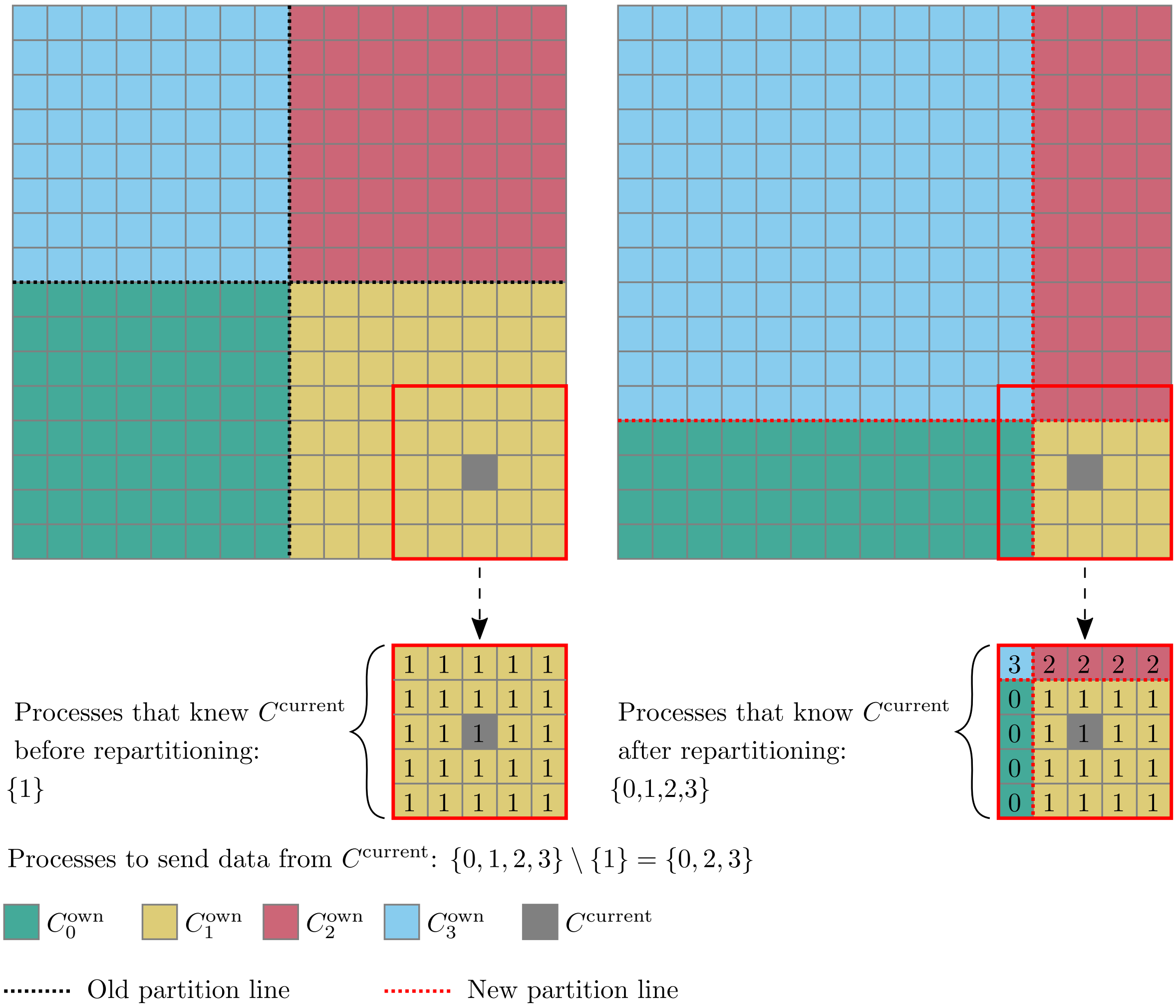
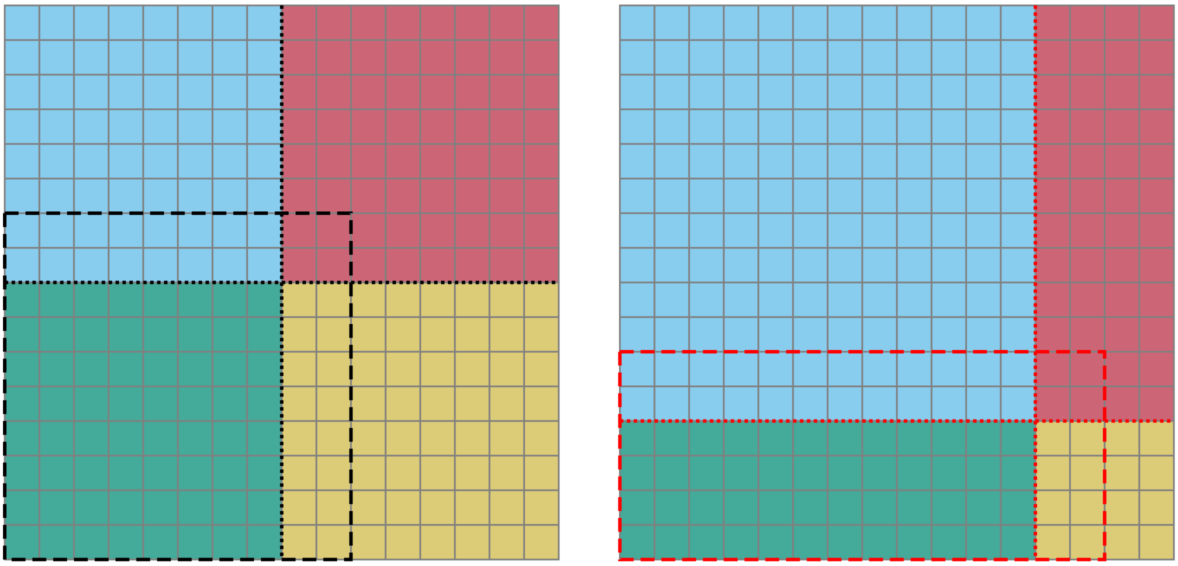

Processes that knew C^{current} before repartitioning: {1}
Processes that know C^{current} after repartitioning: {0,1,2,3}
Processes to send data from C^{current}: $\{0,1,2,3\} \setminus \{1\} = \{0,2,3\}$
C_0^{own}
C_1^{own}
C_2^{own}
C_3^{own}
C^{current}
Old partition line
New partition line

Figure 4.13: Scheme representing the steps to determine to which processes a cell has to send its associated data to.

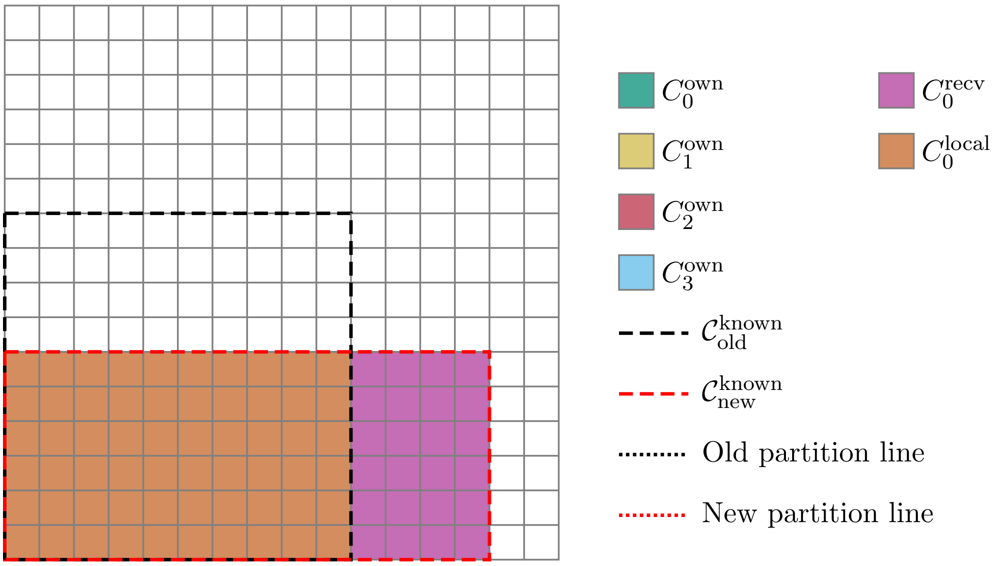
Figure 4.13 shows an example of how to determine the processes to which a cell C^{current} needs to send its data to, in a partition among four processes. Before repartitioning, only process 1 knew C^{current}, whereas after repartitioning, processes 0, 1, 2, and 3 know C^{current}. Therefore, C^{current} needs to send its data to processes 0, 2, and 3. As there is more than one process to which C^{current} needs to send data to, C^{current} belongs to $\mathcal{C}^{\text{send}}$.

4. The set of cells that need to receive data from other processes is conformed by cells that are known by the processes but were not previously known, and is determined as $\mathcal{C}^{\text{recv}} = \mathcal{C}^{\text{known}}_{\text{new}} \setminus \mathcal{C}^{\text{known}}_{\text{old}}$, see Figure 4.14c.



(a) Cell distribution before repartitioning.

(b) Cell distribution after repartitioning.



(c) Resulting C^{recv} and C^{local} after repartitioning. Receive cells C^{recv} get data from old meshes of remote processes, while local cells C^{local} get data from old meshes of the same process.

Figure 4.14: Determination of cells that receive data from other processes and cells that update locally after repartitioning among four processes, as seen from the bottom left partition with rank 0.

5. The cells that have not changed ownership after repartitioning update their data locally without communication. This set is determined as $\mathcal{C}^{\text{local}} = \mathcal{C}^{\text{known}}_{\text{new}} \setminus \mathcal{C}^{\text{recv}}$, see Figure 4.14c.
6. The DOFs values are transferred from the old meshes to the new meshes with the following procedure:
   (a) For each cell in $\mathcal{C}^{\text{send}}$, get the corresponding element in $\mathcal{T}_{\text{old}}$ and extract the values of the DOFs without sorting.
   (b) Perform non-blocking point-to-point communication of the DOFs values determined in 6a.
   (c) For each cell in $\mathcal{C}^{\text{recv}}$, get the corresponding element in $\mathcal{T}_{\text{new}}$, extract the DOFs without sorting, and set their values with the values that were received in 6b.
   (d) For each cell in $\mathcal{C}^{\text{local}}$, get the corresponding new and old elements in $\mathcal{T}_{\text{old}}$ and $\mathcal{T}_{\text{new}}$ respectively, extract the values of the DOFs of the old element and set them on the new element.
7. The history variable values are transferred with the following procedure:
   (a) For each cell in $\mathcal{C}^{\text{send}}$ get the corresponding element in $\mathcal{T}_{\text{old}}$ and extract the coordinates of the integration points.
   (b) For each of the coordinates determined in 7a, get the value of the history variable in the current and in the previous displacement step.
   (c) Perform non-blocking point-to-point communication of the history variable values determined in 7b.
   (d) For each cell in $\mathcal{C}^{\text{recv}}$, get the corresponding element in $\mathcal{T}_{\text{new}}$ and extract the coordinates of the integration points.
   (e) For each of the coordinates determined in 7d, set the value of the history variable corresponding to the current and the previous displacement steps with the values received in 7c.

4.3 Efficient parallel solvers

4.3.1 Parallel linear solvers

After setting up the matrices and vectors of the system of equations (3.6) and (3.7), a solution has to be found for $\Delta\mathbf{d}$ and $\Delta\mathbf{\Phi}$. For small-size problems, direct solvers can be applied to arrive at solutions in reasonable amounts of time. However, this type of solvers does not scale well due to their high memory requirements, rapidly increasing computational cost [Hackbusch, 1994], and difficulty to be parallelized [Dongarra et al., 1998; Saad, 2003]. Therefore, for large-size problems, iterative solvers are generally preferred. In this thesis, the Preconditioned Conjugate Gradient (PCG) method is used. This is a Krylov's subspace method that can be used for symmetric positive-definite matrices as the ones arising from a phase-field analysis.

To achieve good parallel scalability, it is common to utilize highly optimized third-party implementations rather than relying on an in-house implementation. With this in mind, the

`Trilinos` package from the Sandia National Laboratories [Heroux et al., 2005] is utilized in this work. In particular, `Trilinos` provides the `Epetra` and `Tpetra` packages for distributed linear algebra objects such as matrices and vectors, and the `AztecOO` linear solver package that implements the PCG solver, among others, and at the same time provides an interface for the application of user-defined preconditioners.

4.3.2 Preconditioning FCM analyses

The convergence of linear solvers depends on the condition number of the system's matrix, making it difficult or even impossible to arrive at a solution in the case of badly conditioned systems. The stiffness matrices arising from the FCM are an example of such ill-conditioning. This phenomenon was systematically studied in [de Prenter et al., 2017], where it was shown that the main cause is that basis functions in small cut cells can become arbitrarily small and also linearly dependent. In phase-field analyses, the problem becomes even worse, as the condition number of the system matrices steeply increases after the onset of crack propagation [Badri et al., 2021]. To improve the conditioning of the system and thus the convergence properties, preconditioning is used. Preconditioning involves the transformation of the original system of equations of the form $\mathbf{A}\mathbf{x} = \mathbf{b}$ to an equivalent one with the same solution but that is easier to solve. This entails solving

$$\mathbf{M}^{-1}\mathbf{A}\mathbf{x} = \mathbf{M}^{-1}\mathbf{b}, \tag{4.7}$$

where $\mathbf{M}$ is the preconditioning matrix. A good preconditioner renders the condition number of $\mathbf{M}^{-1}\mathbf{A}$ smaller than the one of $\mathbf{A}$. Particularly, a perfect preconditioner is obtained when $\mathbf{M}^{-1} = \mathbf{A}^{-1}$, but it is computationally costly and impractical to construct. In the case of the FCM, common preconditioners such as Jacobi or Gauss-Seidel are not robust as they only target partially the root causes of ill-conditioning mentioned before [de Prenter et al., 2017]. For this reason, the Additive Schwarz preconditioning technique for the FCM is used in this work [Jomo et al., 2019].

Additive Schwarz preconditioning

This section introduces the main aspects of the Additive Schwarz preconditioning that are important to understand the reason behind the need for a third layer of ghost elements in nonlinear phase-field analyses' distributed meshes, as presented in 4.2.3. For a detailed explanation of the method, the interested reader is referred to [Jomo et al., 2019].

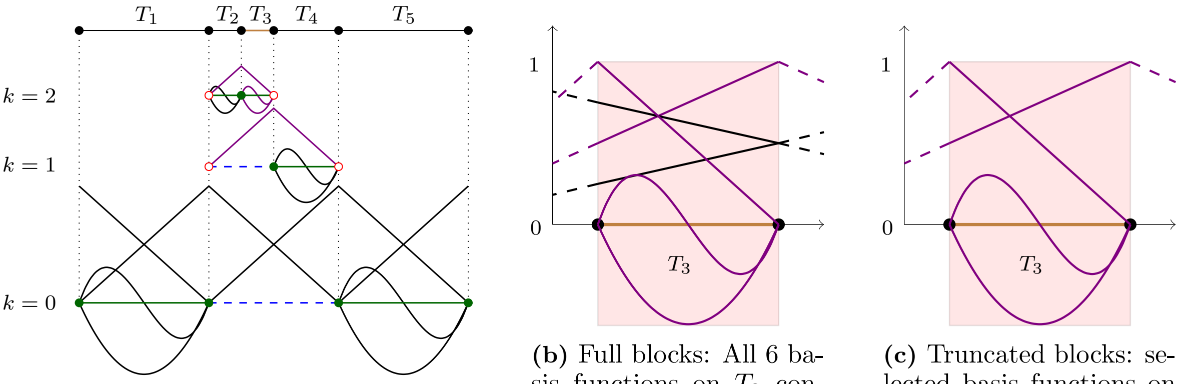
The idea of the Additive Schwarz preconditioning is to construct a preconditioner $\mathbf{M}^{-1}$ by inversion and summation of sub-matrices extracted from the system matrix $\mathbf{A}$. These sub-matrices correspond to groups or blocks of certain basis functions with intersecting support, that are extracted with restriction and prolongation operators $\mathbf{P}$ and $\mathbf{P}^T$ applied to $\mathbf{A}$. The formula for the preconditioner is then

$$\mathbf{M}^{-1} = \sum_{i=1}^{n_{\text{blocks}}} \mathbf{P}_i \underbrace{\left(\mathbf{P}_i^T \mathbf{A} \mathbf{P}_i\right)^{-1}}_{\mathbf{A}_i^{-1}} \mathbf{P}_i^T, \tag{4.8}$$

where n_{blocks} is the number of blocks of basis functions and i is the index of the i^{th} block containing m basis functions such that $\mathbf{A}_i^{-1} \in \mathbb{R}^{m \times m}$ and $\mathbf{P}_i \in \mathbb{R}^{n_{\text{DOFs}} \times m}$.

Different preconditioners with different properties can be constructed depending on the selection of the additive Schwarz blocks. Two of the available options are:

- **Selection of blocks based on base elements**: this is a straightforward approach where all basis functions that are supported on a base element and all its children form part of an additive Schwarz block, see Figure 4.15b.
- **Selection of blocks based on leaf elements**: this approach intends to build blocks containing basis functions that can potentially become almost linearly dependent, which is the case when an element is cut. As can be seen in Figure 4.15a, more than $p+1$ basis functions can be supported on a leaf element. However, only $p+1$ unique polynomial degrees of freedom exist. For efficient preconditioning, it is sufficient to construct additive Schwarz blocks of elements to span these $p+1$ polynomial degrees of freedom. The blocks are then built with the $p-1$ higher-order basis functions and two linearly independent basis functions of order $p=1$ that can be either both on the highest level or one on the highest level and another one down in the element hierarchy, as in Figure 4.15c. Important is to note that, for example, the block corresponding to the element T_2 in Figure 4.15a can end up having basis functions of the lowest refinement level, and therefore all related entries on the stiffness matrix must have correct values to be able to compute the exact preconditioner.



(a) One dimensional multi-level *hp*-mesh with 5 leaf elements.

(b) Full blocks: All 6 basis functions on T_3 considered in B_3.

(c) Truncated blocks: selected basis functions on T_3 considered in B_3.

Figure 4.15: Block selection for multi-level *hp*-grids with the full and truncated blocks of element T_3, adopted from [Jomo et al., 2019].

With the above explanation and referring again to Figure 4.5, it is clear that some additive Schwarz blocks containing the basis function that corresponds to DOF 3 also contain DOFs 2, 6, 7, and 8. Therefore, the stiffness matrix entries associated with DOFs 2, 6, 7, and 8 must be correct as they are used to compute the preconditioner. In a linear analysis, this is guaranteed without having all DOFs in the second layer of ghost elements with correct *solution* values, as the solution is not used for the computation of the stiffness matrix. However, in a nonlinear analysis, the solution values of all DOFs of elements in the second ghost layer must be correct as they are used for the computation of the stiffness matrix, requiring the existence of the third layer of ghost elements, see 4.2.3.

In the numerical examples presented in this thesis, the selection of blocks based on base elements is used, as it yields a good balance between the computational cost of the preconditioning matrix and reduction in solver iterations.

Chapter 5

Results

In this section, the performance of the proposed framework for parallel phase-field simulations is investigated. The first three examples were run on the CoolMUC2 Cluster of the Leibniz Supercomputing Centre (LRZ) in Garching, Germany [Leibniz Supercomputing Centre, 2021; Wilde et al., 2017]. The cluster is equipped with a total of 812 nodes, each having two Intel Xeon E5-2697 v3 14-core Haswell CPUs with 64 GB of main memory. The last example was run on the HTCE Cluster of the LRZ that is equipped with 4 nodes with Intel Xeon Gold 6126 CPUs. The code was compiled with the GNU compiler 7.0 and -O3 optimization flags, using the following library versions: IntelMPI compiler version 19.0, Trilinos 12.12.1 [Heroux et al., 2005], Insight Toolkit 4.12 [Ibanez et al., 2005], and Boost version 1.61 [Schäling, 2014]. The time measurements were taken with the `cpu_timer` implementation of the Boost library.

5.1 Square plate in tension with pre-existing crack

Problem setup

The first example consists of a square plate in plane strain with dimensions 1 × 1 mm and a 0.5 mm long crack in the middle of the height, as shown in Figure 5.1. The plate is subject to a vertical displacement $v = 7.4 \times 10^{-3}$ mm imposed on the top edge, applied incrementally in 6 steps of 1×10^{-3} mm and 14 steps of 1×10^{-4} mm. The bottom edge is fixed in the vertical direction, while the bottom left corner is fixed in both directions. The material has a Young's modulus $E = 210$ GPa, Poisson's ratio $\mu = 0.3$, and fracture toughness $G_c = 0.0027$ kN/mm. The length scale is chosen as $l_0 = \frac{1}{256}$ mm so that at least one element spans the half-width of the crack with the coarsest meshes used in the analyses. The pre-existing crack is defined by setting a history variable with the initial values

$$\mathcal{H}(x,y) = \begin{cases} \frac{4G_c}{l_0} & \forall x \in [0, 0.5] \wedge \forall y \in [0.5 - l_0, 0.5 + l_0] \\ 0 & \text{otherwise} \end{cases}. \tag{5.1}$$

The maximum number of staggered steps is set to 10 and a tolerance $\varepsilon_{\text{stag}} = 1 \times 10^{-4}$ is used for the stopping criterion (3.8).

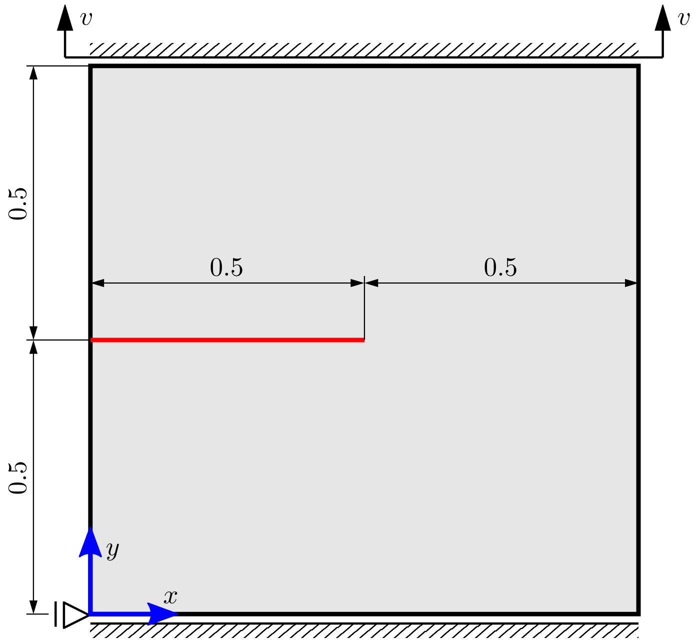


Figure 5.1: Square plate in tension with pre-existing crack. Geometry and boundary conditions.

Validation of the parallel implementation

To validate the parallel implementation presented in 4, the solution of the serial implementation from [Nagaraja et al., 2018; Hug et al., 2020] is compared to the solution of the parallel implementation using 64 processes divided among 16 nodes, and 7 threads per process. Cartesian meshes of 128×128 elements with a polynomial degree $p = 2$ and a refinement depth $k = 2$ were used in both cases. In the parallel case, a dynamic repartitioning strategy with a tolerance $\zeta = 1.0$ was utilized, see 4.2.5. Figure 5.2 shows the load-displacement curves for both cases.

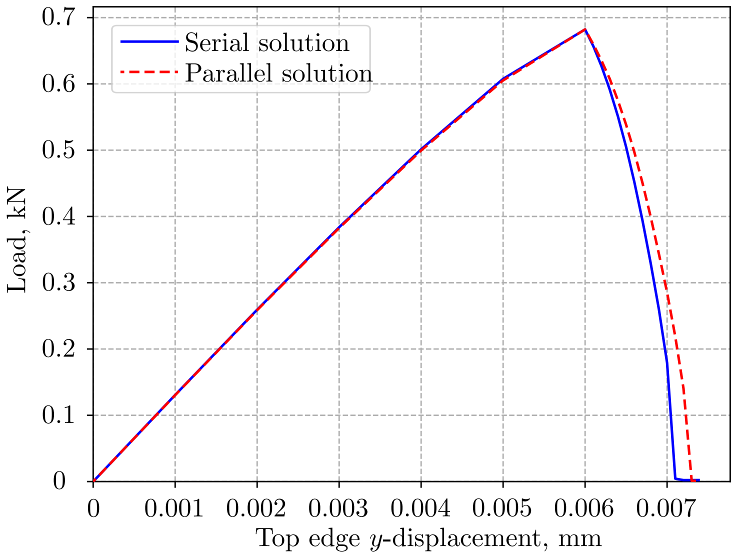


Figure 5.2: Square plate in tension with pre-existing crack. Comparison of the load-displacement curves of a serial and a parallel solution with 64 processes and $\zeta = 1.0$.

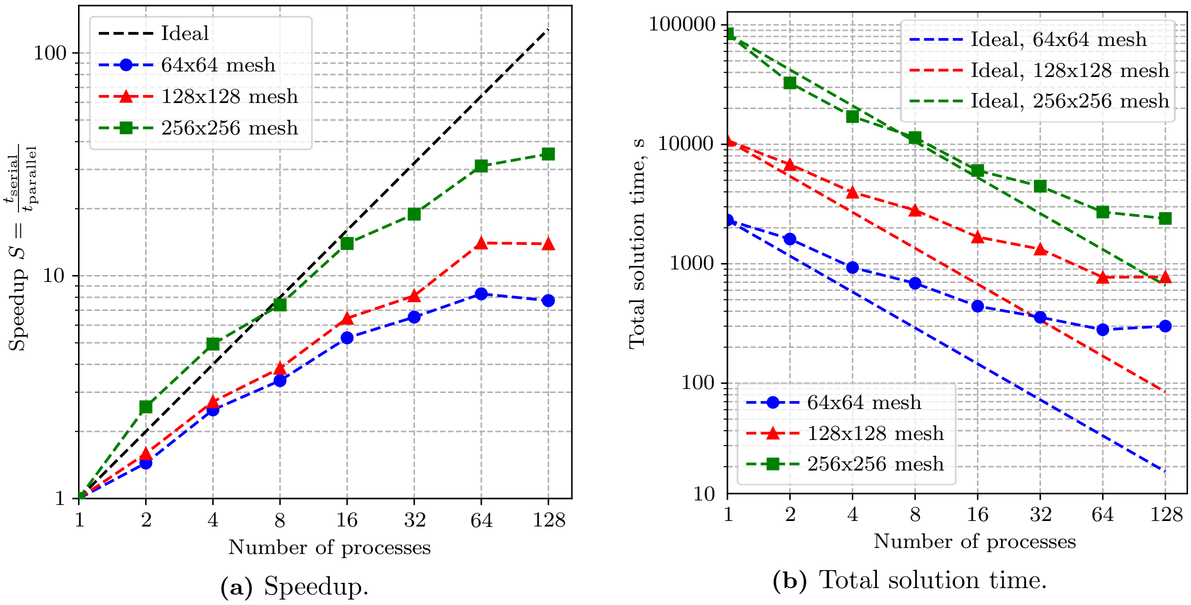
As it can be seen, both curves match closely as expected, thus validating the parallel implementation. The crack starts to propagate for $v = 6 \times 10^{-3}$ mm and the maximum load is 0.68 kN. In the crack propagation phase ($v > 6 \times 10^{-3}$ mm), there are minor differences between the curves that come from different convergence criteria used for the staggered iterations. While the parallel implementation used the absolute solution increments (3.8), the serial implementation used the relative solution increments [Nagaraja et al., 2018].

Strong scaling

To assess the strong scalability of the implementation, the analysis was run with a number of processes that varied from 1 to 128 while keeping the problem size fixed. 4 processes per node were used, except for the cases with 1 and 2 processes, and each process used 7 threads. The base case for the speedup measure was a serial run (1 process) of the parallel implementation. The time measurements were taken from the start until the end of the simulation, spanning all displacement steps.

Strong scaling for different meshes

First, the strong scaling for different meshes is investigated. Three analyses without repartitioning for Cartesian meshes of 64 × 64, 128 × 128, and 256 × 256 elements were run. In all cases, a polynomial degree $p = 2$ and a refinement depth $k = 2$ were used. Figure 5.3 shows the speedup and the total solution time vs. the number of processes for each of the simulations. The framework shows good scalability for up to 128 processes, reaching speedups of 8.4, 13.9, and 35.2 for the 64 × 64, 128 × 128, and 256 × 256 meshes respectively.



(a) Speedup.

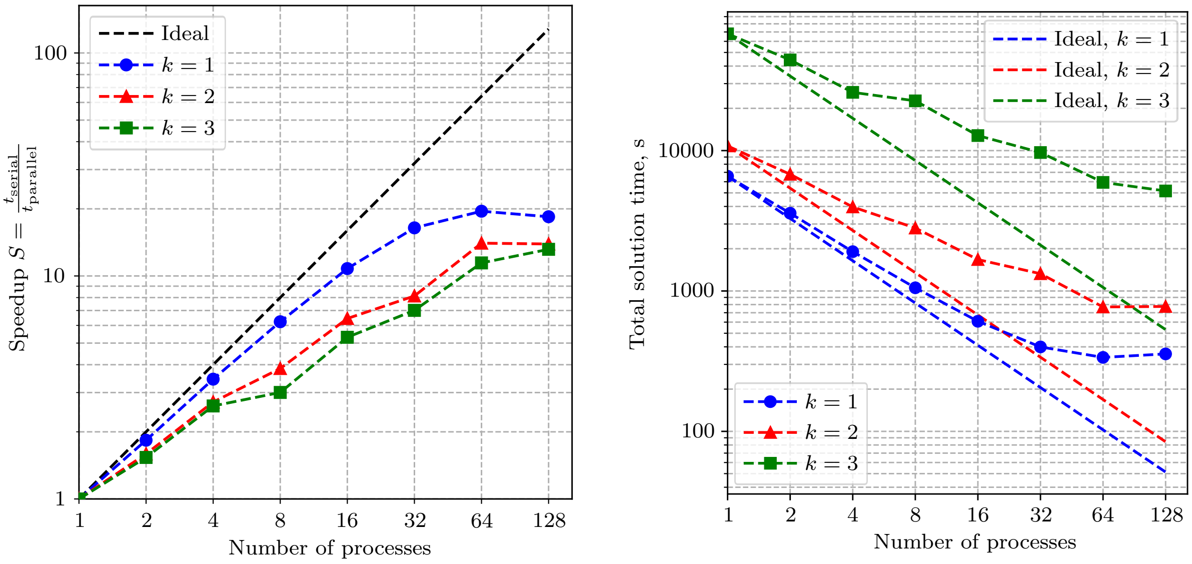
(b) Total solution time.

Figure 5.3: Square plate in tension with pre-existing crack. Strong scaling analysis for different meshes.

It can be observed that the increase in the speedup resulting from an increase in the number of processes becomes smaller for a higher number of processes, in agreement with Amdahl's law, see 4.1.3. Furthermore, coarser meshes result in smaller speedups, as the curves move farther away from the ideal line. This behavior is explained by the fact that increasing the number of processes or reducing the number of elements increases the ratio between the number of ghost and owned elements. The consequence is a higher number of redundant operations, e.g. integration of the elements matrices, that reduces the parallel efficiency. Moreover, a higher number of ghost elements requires the communication of more values between processes and therefore additional run time. The case of the 256×256 meshes exhibits a superlinear behavior with 2 and 4 processes, as the speedup curve is above the ideal line, see Figure 5.3a. Such a behavior could be caused by cache memory effects and other side effects [Ristov et al., 2016].

Strong scaling for different refinement depths

To assess the effect of different refinement depths on the scalability, three different analyses without repartitioning for Cartesian meshes of 128×128 elements, polynomial degree $p = 2$, and refinement depths $k = \{1, 2, 3\}$ were run. It is important to note that the length scale of the crack l_0 was fixed, and therefore the number of leaf elements and DOFs per process increased with increasing k. Figure 5.4 shows the speedup and the total solution time vs. the number of processes for different refinement depths. It is observed that an increase in the refinement depth reduces the speedup, as the curves move farther away from the ideal line. This effect can be attributed to the existence of a higher number of leaf elements in the ghost layers, which results in the communication of more values among processes and a higher number of redundant operations. Furthermore, higher refinement depths result in higher load imbalance among processes as the adaptive refinement is local to the crack zone, reducing the parallel efficiency. To counteract this behavior, one could reduce the length scale of the crack together with the increase in the refinement depth, keeping in mind that changing the length scale has an impact on the modeled crack response [Zhang et al., 2017].

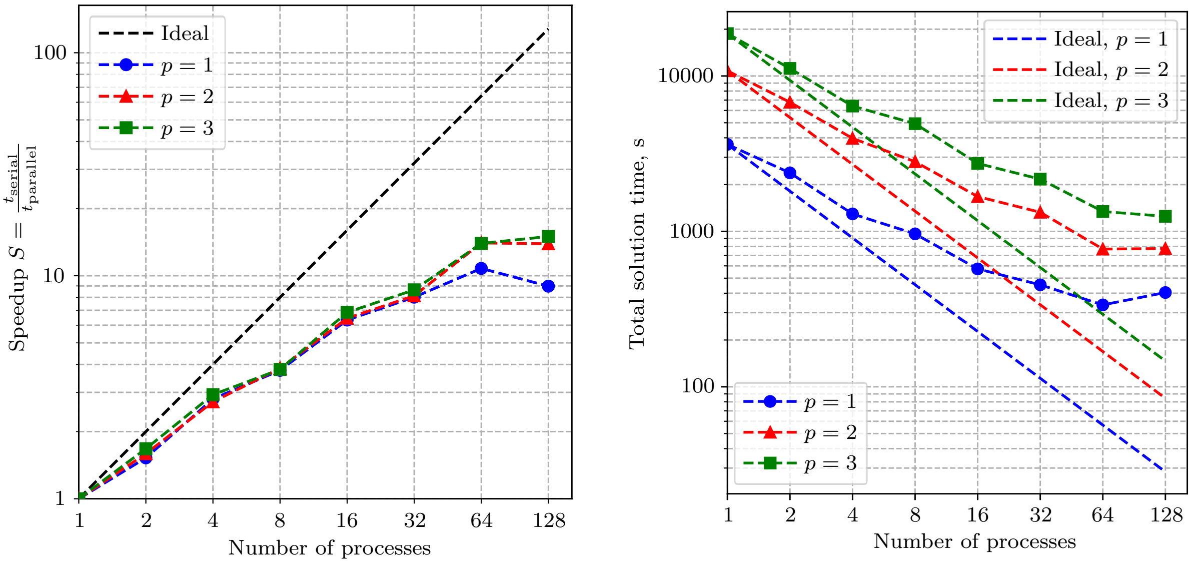


(a) Speedup. **(b)** Total solution time.

Figure 5.4: Square plate in tension with pre-existing crack. Strong scaling analysis for different refinement depths k.

Strong scaling for different polynomial degrees

The effect of different polynomial degrees on performance is investigated in this section. To this end, three different analyses without repartitioning for Cartesian meshes of 128 × 128, refinement depth $k = 2$, and polynomial degrees $p = \{1, 2, 3\}$ were run. Figure 5.5 shows the speedup and the total solution time vs. the number of processes for different polynomial degrees. It can be observed that increasing the polynomial degree does not have a considerable effect on the speedup, as the three curves are close to each other, see Figure 5.5a. By increasing the polynomial degree, all elements of the mesh are affected in a similar manner and the computational effort is increased in all processes. As a result, there is no noticeable load imbalance induced by a change in the polynomial degree. This is in contrast with the case of increasing refinement depth, where only elements in the crack zone are refined, unbalancing the load.



(a) Speedup. **(b)** Total solution time.

Figure 5.5: Square plate in tension with pre-existing crack. Strong scaling analysis for different polynomial degrees p.

Influence of repartitioning on the speedup

In section 4.2.5, two different strategies for repartitioning were presented. The goal of this section is to evaluate the improvements that can be achieved with each of them. For this study, Cartesian meshes of 128×128 elements with a polynomial degree $p = 2$ and a refinement depth $k = 2$ were used. The analysis was run with 4, 16, and 64 processes partitioned among 1, 4, and 16 nodes respectively, and with 7 threads per process. Figure 5.6 shows the evolution of the maximum relative difference of the normalized load

$$d_{\max} = \max_{\text{rank} \in [0, n_{\text{procs}}-1]} d(\text{rank}) \tag{5.2}$$

for every staggered step when no repartitioning is done. It is observed that the load imbalance increases with the number of processes.

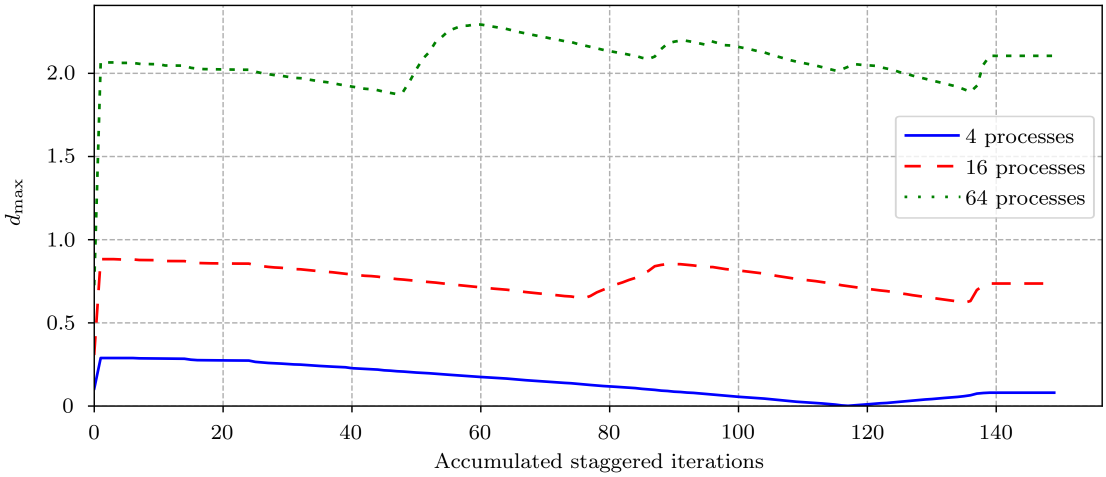


Figure 5.6: Square plate in tension with pre-existing crack. Evolution of the maximum relative difference of the normalized load $d_{\max}$ without repartitioning vs. the accumulated staggered iterations.

Step-based repartitioning For the step-based repartitioning strategy, the number of displacement steps to wait for repartitioning was varied in the range from 1 to 10, and the ratio between the speedup with and without repartitioning was measured. Figure 5.7 shows that only the case with 64 processes improves the speedup up to 9%. For cases with 4 and 16 processes, the reduction in computational effort caused by repartitioning is not enough to counteract the additional parallel overhead. For analyses with more displacement steps, it is expected that this behavior improves. However, this needs further investigation.

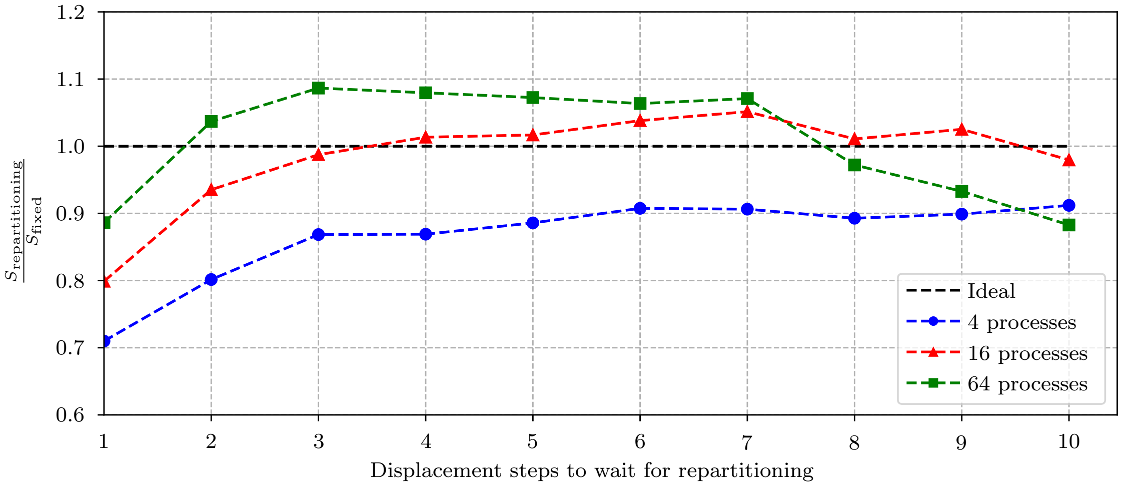


Figure 5.7: Square plate in tension with pre-existing crack. Influence of the number of steps to wait for repartitioning on the speedup.

Dynamic repartitioning

To assess the dynamic repartitioning strategy, several repartitioning tolerances ζ were used and the ratio between the speedup with and without repartitioning was measured. Figure 5.8 shows that improvements of up to 16% and 27% can be achieved for the partitioning among 16 and 64 processes respectively. For the partition among 4 processes, no improvements are observed because $d_{\max}$ is reduced as the analysis evolves (see Figure 5.6), and thus the load balances without external help. For the partitioning among 16 and 64 processes, however, $d_{\max}$ increases over the course of the analysis, and repartitioning helps to rebalance the load. It is clear that the benefits of repartitioning increase with the number of processes. This behavior is expected as the higher the number of processes is, the smaller is the number of base elements that each process owns. Therefore, it can happen that in a given process a great portion of elements is refined, while in other processes there is no refinement at all, yielding high d values. Figure 5.8 also shows that for small values of ζ repartitioning worsens the speedup, as it is performed too often and the gain of rebalancing the load is overshadowed by the additional parallel overhead. Furthermore, it is observed that the range of ζ values that result in an improvement of the speedup grows with the number of processes.

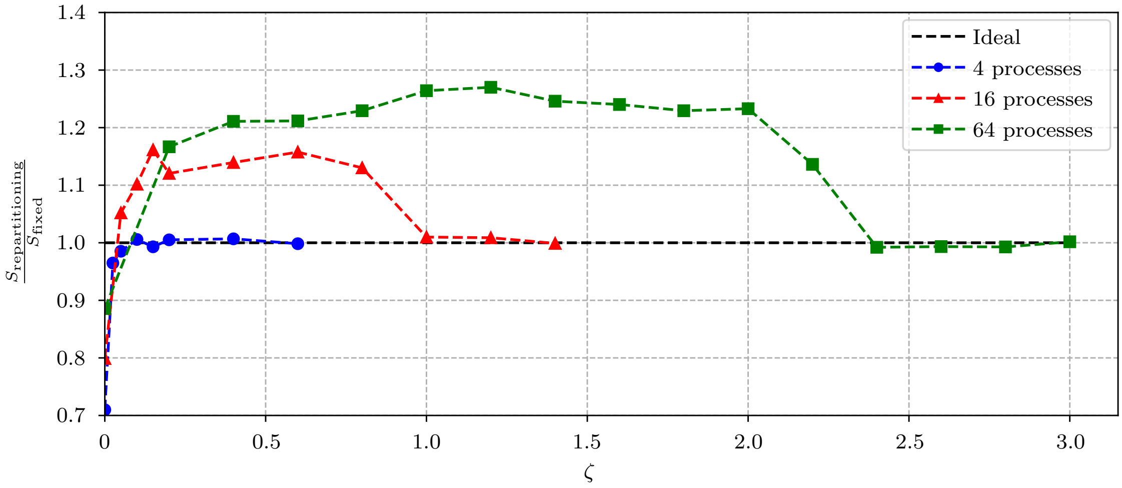


Figure 5.8: Square plate in tension with pre-existing crack. Influence of the repartitioning tolerance ζ on the speedup.

Figure 5.9 shows the evolution of the maximum relative difference of the normalized load $d_{\max}$ throught the simulation for a dynamic repartitioning with $\zeta = 0.2$.

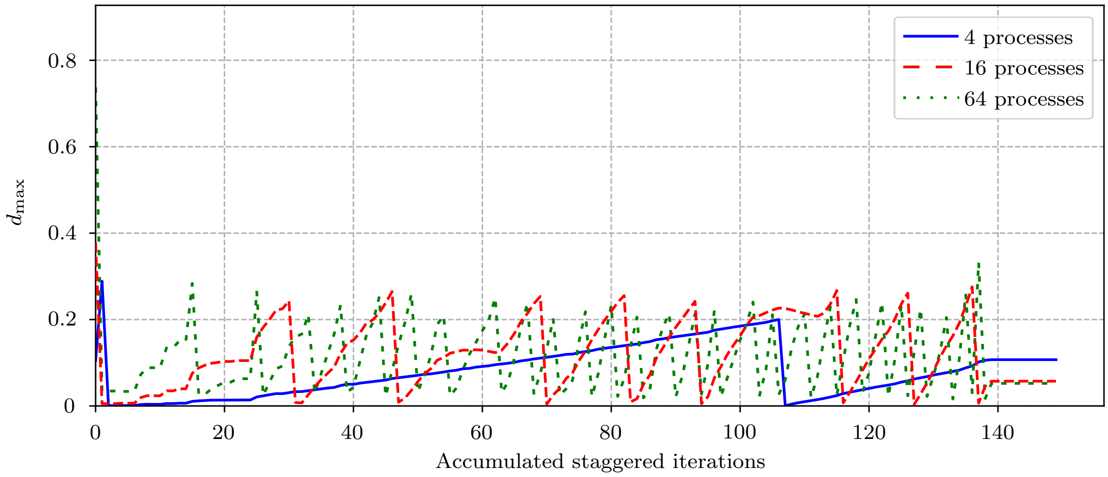
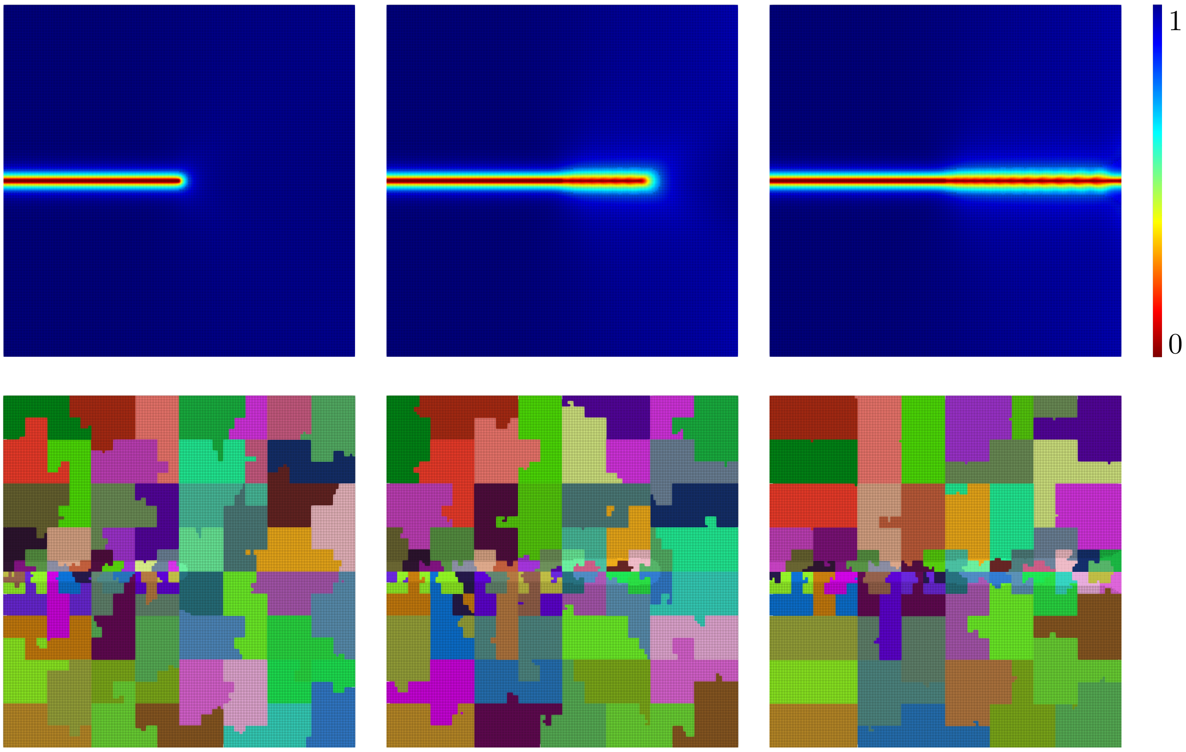


Figure 5.9: Square plate in tension with pre-existing crack. Evolution of the maximum relative difference of the normalized load $d_{\max}$ with dynamic repartitioning vs. the accumulated staggered iterations.

It is evident that repartitioning keeps the load balanced, helping to maintain a good parallel efficiency throughout the analysis.

Crack path and repartitioning visualization

Figure 5.10 shows the propagation of the crack and the evolution of the domain partitioning. Processes that own elements in the crack zone span smaller regions, as their elements are refined and therefore have a higher load density. Therefore, when the crack propagates more processes concentrate around it. It is worth noting that the geometric `Zoltan` partitioner does not give an optimal partitioning of the domain because some meshes are disjoint, i.e. owned elements form “islands” in the mesh, see 5.10 (bottom). For a given number of owned elements, disjoint meshes have a higher number of ghost elements than their contiguous counterparts, resulting in more redundant computations and inter-process communication.



(a) $v = 4 \times 10^{-3}$ mm **(b)** $v = 6.4 \times 10^{-3}$ mm **(c)** $v = 7.4 \times 10^{-3}$ mm

Figure 5.10: Square plate in tension with pre-existing crack. Phase-field (top) and partitioning (bottom) for different displacement steps.

5.2 Square plate in tension with rotated elliptical hole

Problem setup

The second example aims to test the FCM implementation in 2D and to provide a case where the propagating crack unbalances the load also in a partitioning among 4 processes. The example consists of a square plate in plane strain with dimensions 1 × 1 mm and a rotated elliptical hole in the center, as shown in Figure 5.11. The plate is subject to a vertical displacement $v = 7.2 \times 10^{-3}$ mm imposed on the top edge, applied incrementally in 6 steps of 1×10^{-3} mm and 6 steps of 2×10^{-4} mm. The boundary conditions, material properties, length scale, and values for the staggered convergence criterion are the same as in 5.1. In this case, there is no pre-existing crack.

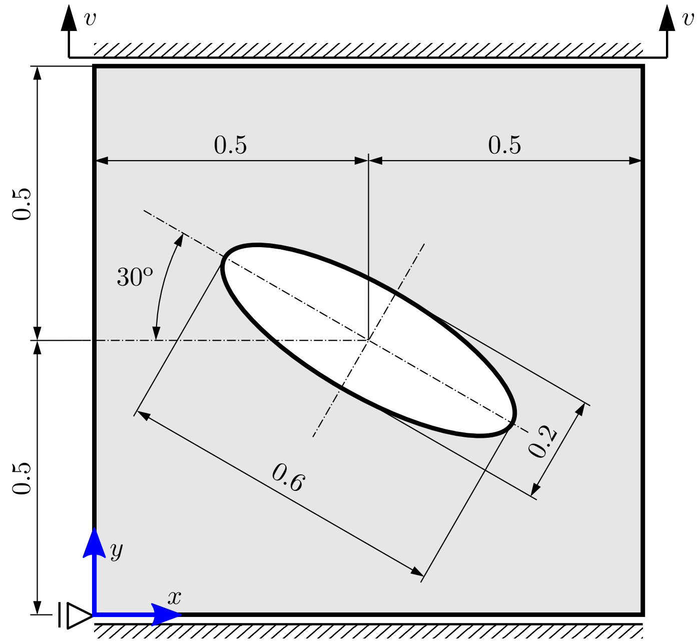
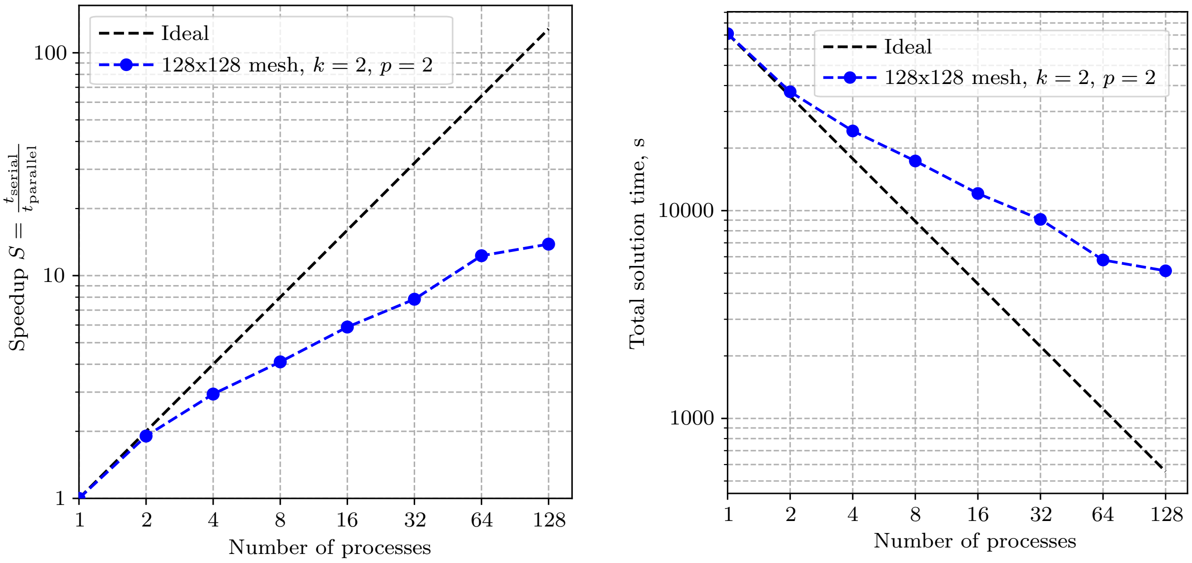


Figure 5.11: Square plate in tension with rotated elliptical hole. Geometry and boundary conditions.

Strong scaling

To assess the strong scalability of the implementation for the FCM, the analysis was run with a number of processes that varied from 1 to 128 while keeping the problem size fixed, analogously as in 5.1. Cartesian meshes of 128 × 128 elements with a polynomial degree $p = 2$ and a refinement depth $k = 2$ were used. Figure 5.12 shows the speedup and the total solution time vs. the number of processes. The framework shows good scalability for up to 128 processes, reaching a speedup of 13.8.



(a) Speedup.

(b) Total solution time.

Figure 5.12: Square plate in tension with rotated elliptical hole. Strong scaling analysis.

Influence of repartitioning on the speedup

To study the influence of repartitioning on the speedup, the analysis was run with 4, 16, and 64 processes partitioned among 1, 4, and 16 nodes respectively, and with 7 threads per process. Only dynamic repartitioning is assessed, as it was shown in Figure 5.7 that no significant improvement can be achieved with step-based repartitioning in problems with a small number of displacement steps. Figure 5.13 shows the evolution of the maximum relative difference of the normalized load d_{max} throughout the simulation when no repartitioning is done. It is observed that the load imbalance increases with the number of processes. In contrast to Figure 5.6, the case with 4 processes increases its imbalance as the analysis progresses, and thus an improvement with repartitioning is expected in this case.

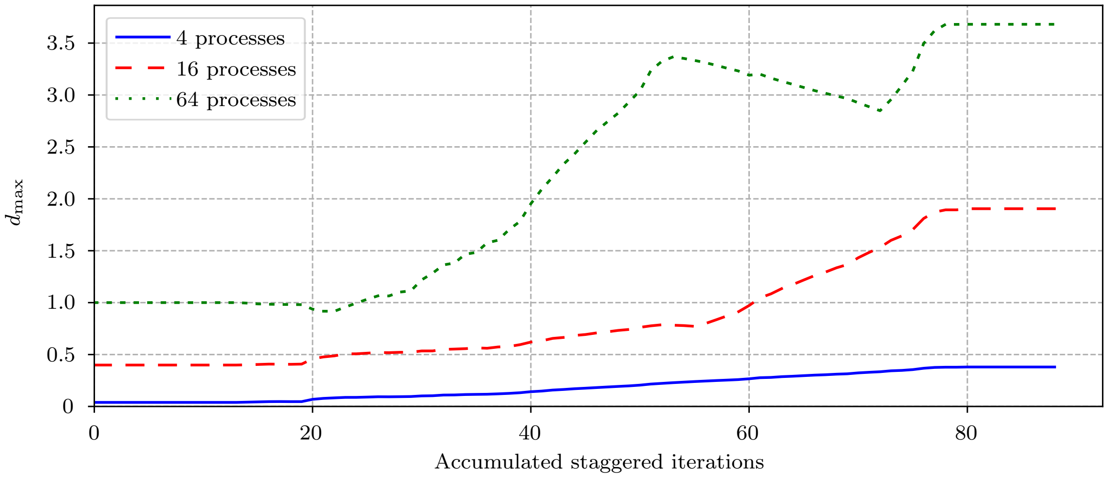


Figure 5.13: Square plate in tension with rotated elliptical hole. Evolution of the maximum relative difference of the normalized load d_{max} without repartitioning vs. the accumulated staggered iterations.

Figure 5.14 shows that improvements of up to 5%, 33% and 36% can be achieved for the partitioning among 4, 16, and 64 processes respectively. Again, for small values of ζ, repartitioning worsens the speedup due to the high parallel overhead it creates. The range of ζ values that result in an improvement of the speedup grows with the number of processes as in 5.1, see Figure 5.14.

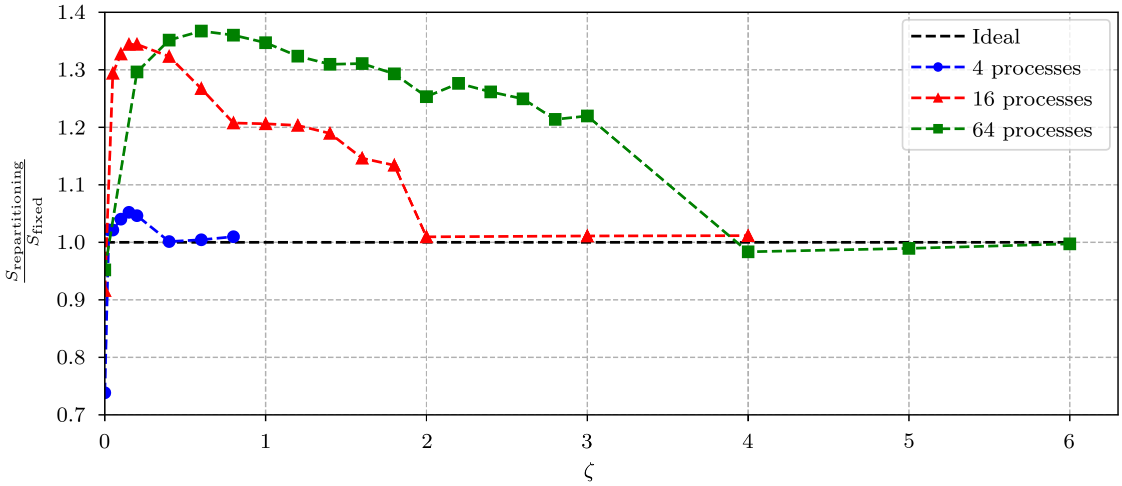


Figure 5.14: Square plate in tension with rotated elliptical hole. Influence of the repartitioning tolerance ζ on the speedup.

Figure 5.15 shows the evolution of the maximum relative difference of the normalized load d_{max} throught the simulation for a dynamic repartitioning with $\zeta = 0.2$. The same behavior as in 5.1 is observed, where the load is maintained at low values.

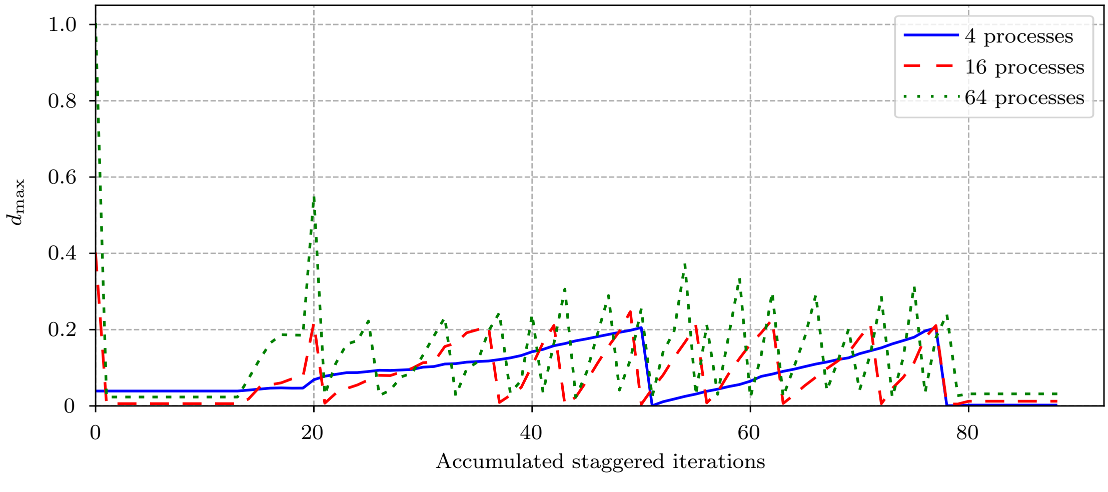
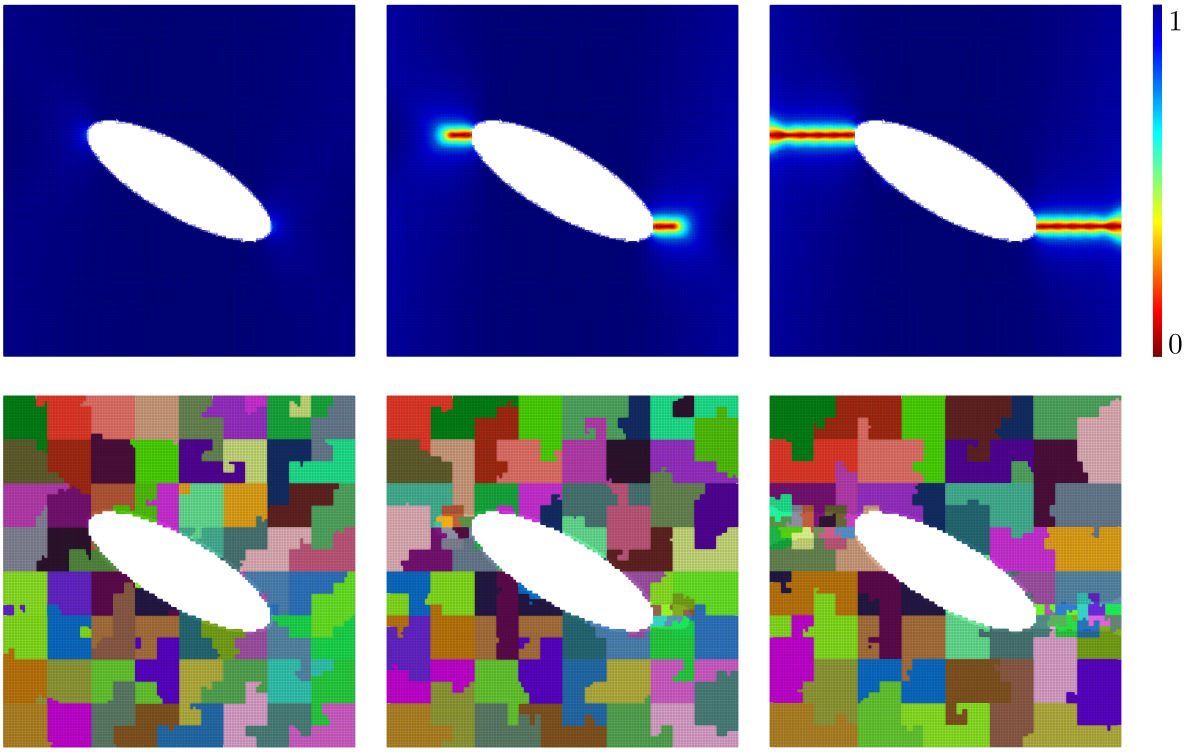


Figure 5.15: Square plate in tension with rotated elliptical hole. Evolution of the maximum relative difference of the normalized load d_{max} with dynamic repartitioning vs. the accumulated staggered iterations.

Crack path and repartitioning visualization

Figure 5.16 shows the propagation of the crack and the evolution of the domain partitioning. As there is no pre-existing crack, all processes span similar areas during the first steps of the simulation. When the crack propagates, the processes concentrate in regions around the crack due to the existence of refined elements. In this case, disjoint meshes are also observed.



(a) $v = 4 \times 10^{-3}$ mm **(b)** $v = 6.4 \times 10^{-3}$ mm **(c)** $v = 7.2 \times 10^{-3}$ mm

Figure 5.16: Square plate in tension with rotated elliptical hole. Phase-field (top) and partitioning (bottom) for different displacement steps.

5.3 Cubic body in tension with spherical inclusions

Problem setup

The goal of the third example is to test the parallel implementation for the FCM in 3D. The example consist of a cubic body with dimensions $1 \times 1 \times 1$ mm with 27 spherical inclusions of diameter $s = 0.2$ mm, as shown in Figure 5.17. The cube is subject to a vertical displacement $w = 1.15 \times 10^{-2}$ mm imposed on the top face, applied incrementally in 10 steps of 1×10^{-3} mm and 3 steps of 5×10^{-4} mm. The bottom face is completely fixed and the length scale is chosen as $l_0 = \frac{1}{64}$ mm so that at least one element spans the half-width of the crack. The material properties and values for the staggered convergence criterion are the same as in 5.1. In this case, there is no pre-existing crack.

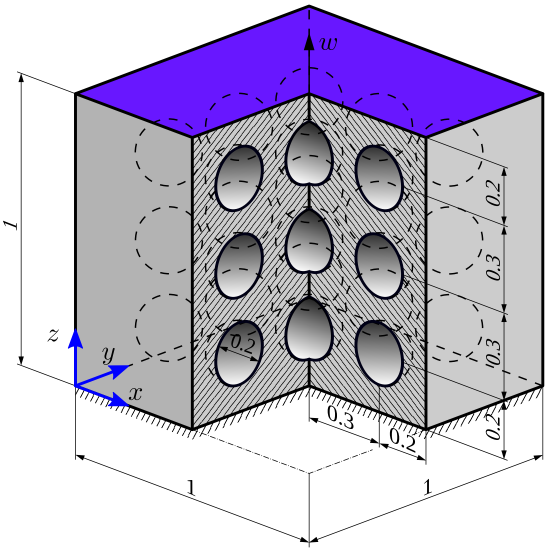
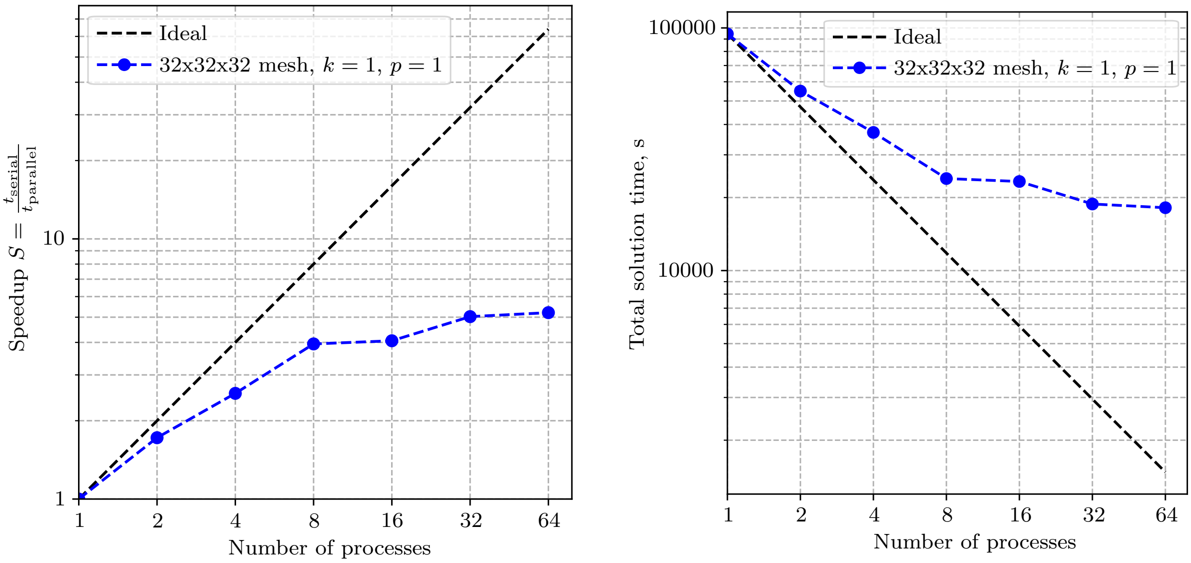


Figure 5.17: Cubic body in tension with spherical inclusions. Section view of the geometry and boundary conditions.

Strong scaling

To assess the strong scalability of the implementation for the FCM in 3D, the analysis was run with a number of processes that varied from 1 to 64 while keeping the problem size fixed. 4 processes per node were used, except for the cases with 1 and 2 processes, and each process used 7 threads. Cartesian meshes of $32 \times 32 \times 32$ elements with a polynomial degree $p = 1$ and a refinement depth $k = 1$ were used. A dynamic repartitioning strategy with $\zeta = 0.05$ was applied. Figure 5.18 shows the speedup and the total solution time vs. the number of processes.



(a) Speedup.

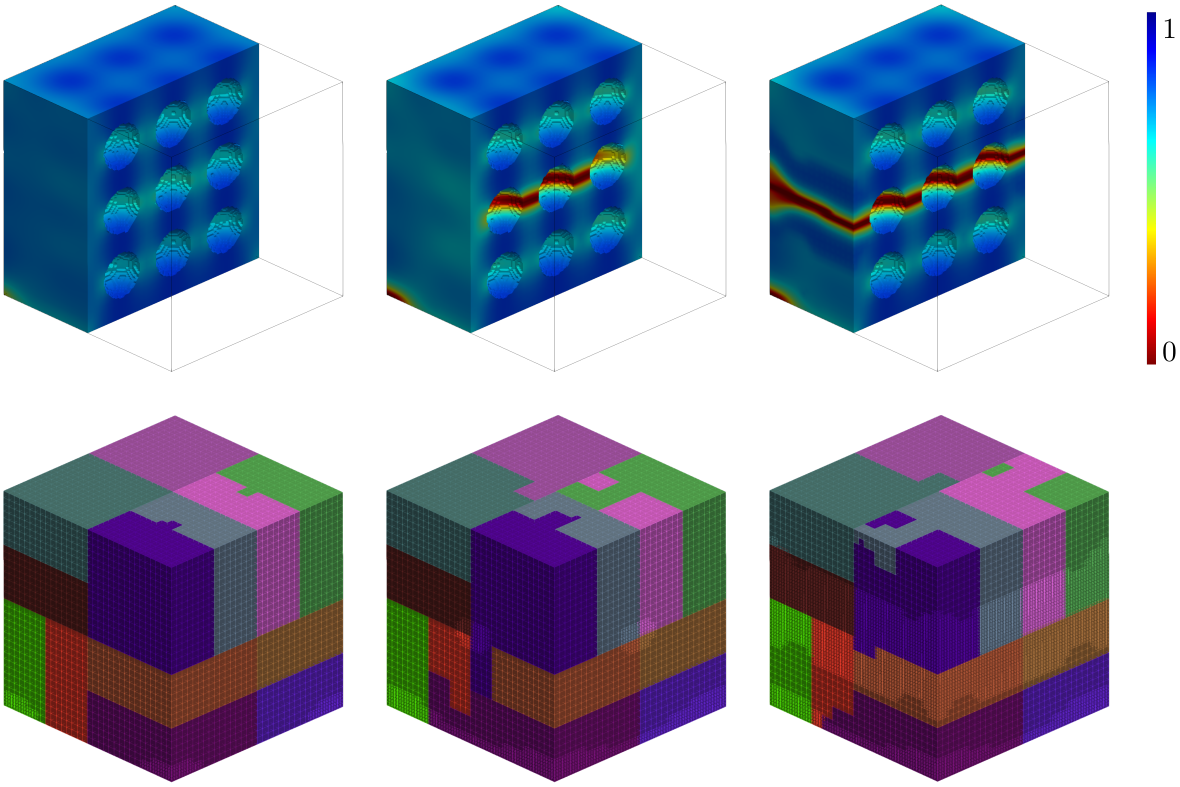
(b) Total solution time.

Figure 5.18: Cubic body in tension with spherical inclusions. Strong scaling analysis.

The framework shows good scalability for up to 64 processes, reaching a speedup of 5.2. For a larger number of processes, the parallel efficiency deteriorates as the ratio of ghost to owned elements becomes too large. By extrapolating the results from the 2D examples to 3D, it is expected that better speedups can be achieved for larger problems, i.e. finer meshes. However, this is not further investigated in this thesis due to time constraints.

Crack path and repartitioning visualization

Figure 5.19 shows the propagation of the crack and the evolution of the domain partitioning. The crack initiates in the central inclusion, propagating outwards in a plane perpendicular to the tensile load direction. It is clearly seen that as the crack grows, elements in the crack zone are refined and the meshes are repartitioned to keep the load balanced.



(a) $w = 0.9 \times 10^{-2}$ mm **(b)** $w = 1.05 \times 10^{-2}$ mm **(c)** $w = 1.15 \times 10^{-2}$ mm

Figure 5.19: Cubic body in tension with spherical inclusions. Phase-field (top) and partitioning (bottom) for different displacement steps. Displacements on the top face are applied in the vertically upward direction.

5.4 Compression test of a core sample

Problem setup

The last example aims to provide an application of practical relevance for the parallel framework. The example consists of the compression test of a core sample with a diameter $d \approx 54$ mm and a height $h = 117.4$ mm, as shown in Figure 5.20. The sample is subject to a vertical displacement $v = -0.22$ mm imposed on the top face, applied incrementally in 11 steps of 2×10^{-2} mm. The bottom face is completely fixed. The material has a Young's modulus $E = 54.3$ GPa, Poisson's ratio $\mu = 0.3$, and fracture toughness $G_c = 6.65 \times 10^{-4}$ kN/mm. The length scale is chosen as $l_0 = 1$ mm and there is no pre-existing crack. The input geometry of the model was obtained from a computerized tomography (CT) scan and the top and bottom surfaces were reconstructed for the application of the boundary conditions. The analysis was run with 16 processes partitioned among 4 nodes, and 10 threads per process. Cartesian meshes of $32 \times 60 \times 32$ elements with a polynomial degree $p = 2$ and a refinement depth $k = 1$ were used. A dynamic repartitioning strategy with $\zeta = 0.02$ was applied. Though it was shown that such a small value of ζ is not optimal (see figures 5.8 and 5.14), it is used here to ensure that repartitioning is performed at several steps of the analysis for illustration purposes.

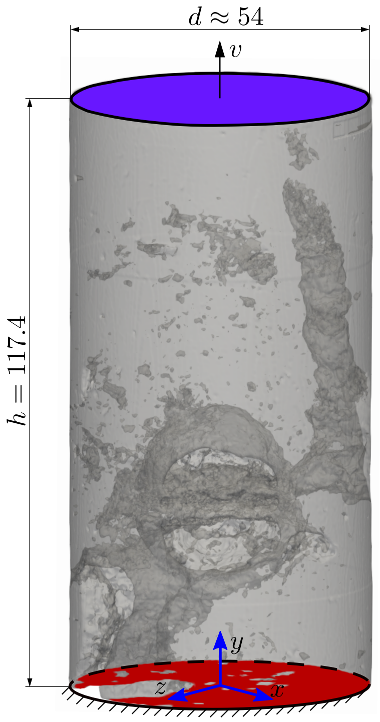
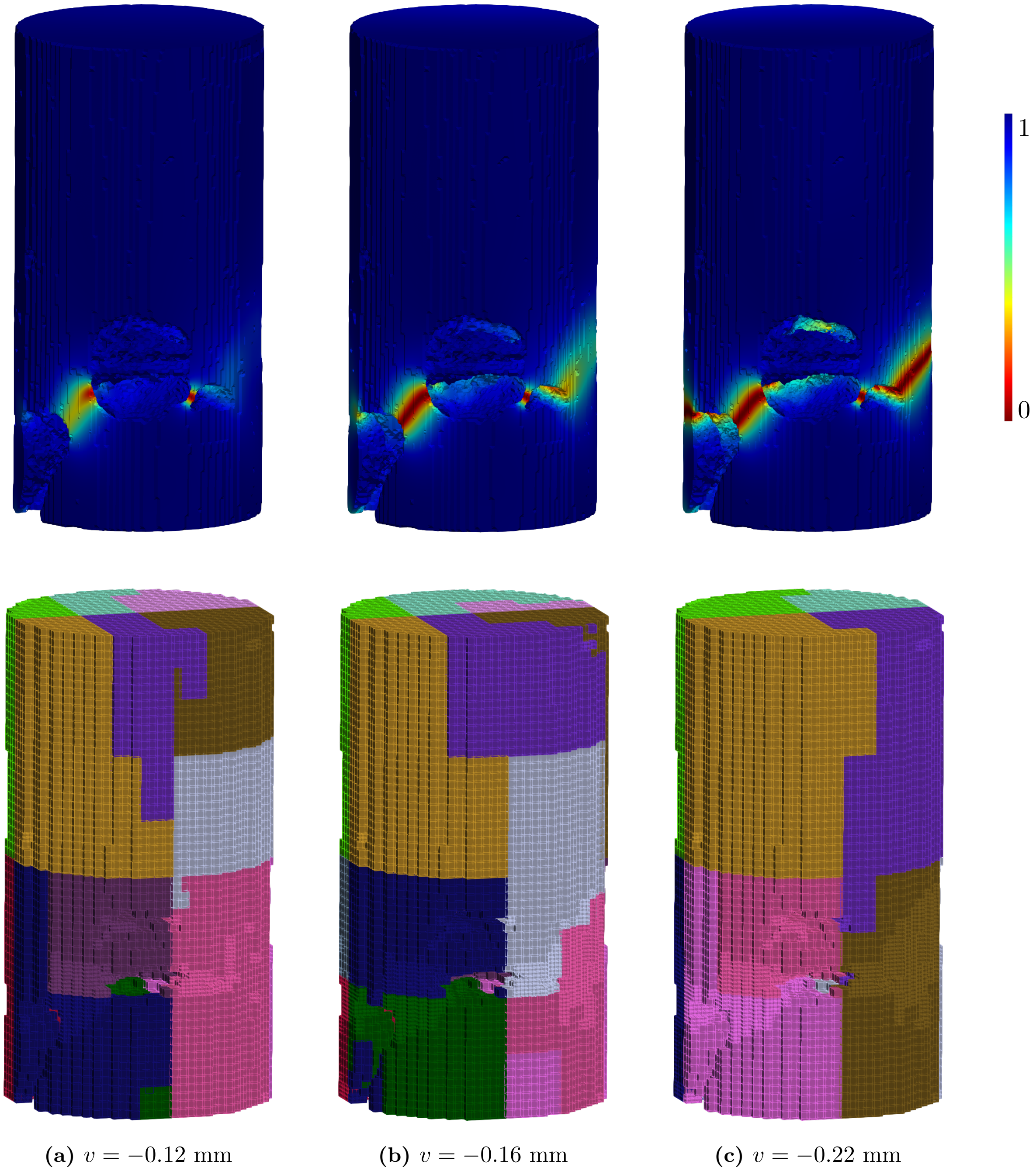


Figure 5.20: Compression test of a core sample. Geometry and boundary conditions.

Crack path repartitioning visualization

Figure 5.19 shows the propagation of the crack and the evolution of the domain partitioning. The crack nucleates around the hole on the front where stress concentrations are high and propagates towards the back of the sample. It is also observed in this case how the elements in the crack zone are refined and the domain is repartitioned. Comparing figures 5.21c (top) and 5.22, it is seen that the crack resulting from the phase-field analyses matches the crack of the real specimen, confirming the validity of the framework to solve complex practical fracture problems.



(a) $v = -0.12$ mm **(b)** $v = -0.16$ mm **(c)** $v = -0.22$ mm

Figure 5.21: Compression test of a core sample. Phase-field (top) and partitioning (bottom) for different displacement steps. Displacements on the top face are applied in the vertically downward direction.

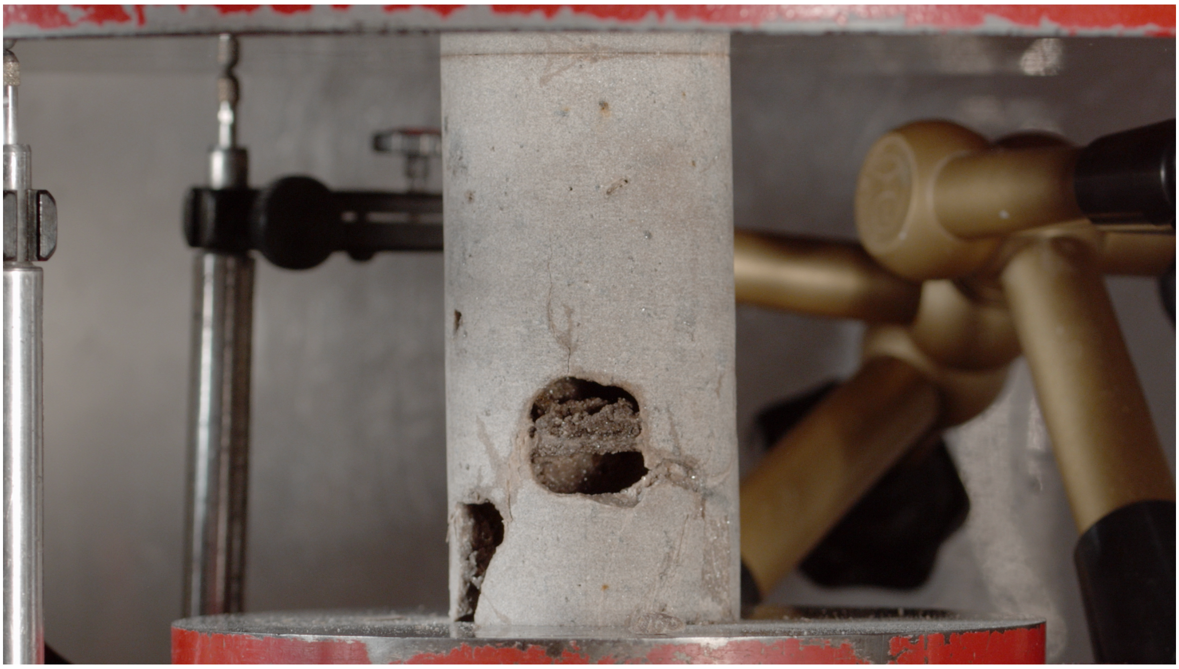


Figure 5.22: Compression test of a core sample. Photo showing the resulting crack.

Chapter 6

Conclusion and Outlook

6.1 Conclusion

This thesis successfully integrated the phase-field simulation model of [Nagaraja et al., 2018; Hug et al., 2020] with the parallel framework for large-scale finite cell analyses with *hp*-refined grids of [Jomo, 2021]. The new implementation reduces the execution time of phase-field analyses with iterative solvers and expands the limits in computable problem size by employing distributed data structures.

The mesh generation algorithms of [Jomo, 2021] were adapted to include a third layer of ghost elements, allowing the solution of nonlinear phase-field problems with the conjugate gradient solver and additive Schwarz preconditioners [Jomo et al., 2019]. The adaptive *hp*-refinement based on the phase-field solution of [Nagaraja et al., 2018] was modified to enable its use with distributed meshes. The new algorithms are based on the refinement of owned elements and the communication of refinement levels of ghost elements, where a solution-based refinement that ensures mesh compatibility is not possible. To handle the load imbalance created by the adaptive refinement, an algorithm to repartition the meshes and rebalance the load among processes was developed. The algorithm comprises the redistribution of cells, the generation of new meshes, and the transfer of solution and history variable values from the old to the new meshes.

Validation of the framework has been successfully done by comparing the load-displacement curves of a benchmark problem resulting from the parallel and the serial implementations. Performance studies were carried out, showing that good parallel scalability up to a modest number of processes. Results show that an increase in the mesh size improves the speedup and achieves good parallel scalability up to a higher number of processes. Furthermore, it was shown that increasing the refinement depth reduces the speedup, whereas changing the polynomial degree has virtually no effect on it. This suggests that an acceptable refinement strategy to use in the parallel framework consists in keeping a low refinement depth, e.g. $k \leq 3$, and increasing the polynomial degree if more accuracy is needed. The effect that the proposed repartitioning strategies have on speedup has been studied. Results show that a step-based repartitioning strategy does not yield a significant improvement of the speedup in the case with a small number of displacement steps. In contrast, a dynamic repartitioning strategy yields improvements on the speedup of up to 36% in the studied cases. It was observed that increasing the number of processes creates a higher imbalance of the load and thus improves the benefit of repartitioning on the speedup. The applicability of the parallel framework for problems of practical relevance was demonstrated with the example of a compression test of a core sample.

6.2 Outlook

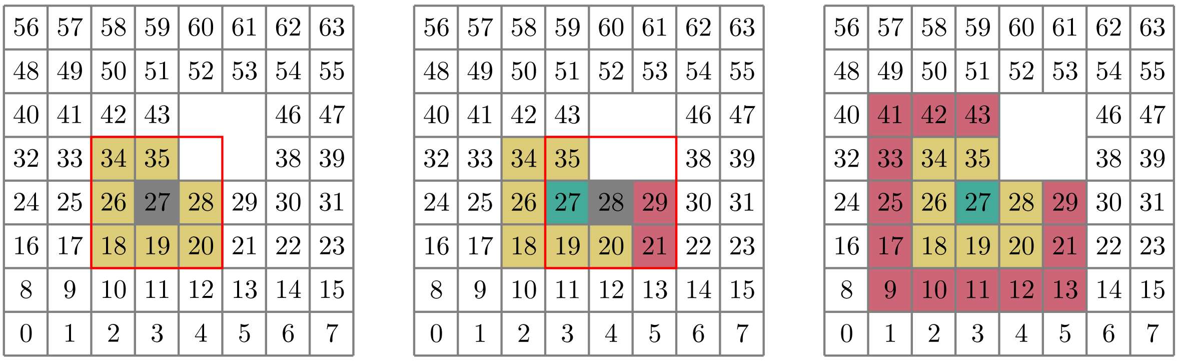
Although this thesis clearly illustrates the good scalability achieved with the parallel framework, there is still room for further improvements:

- The algorithms for parallel mesh generation, refinement, and repartitioning constitute the first step towards a highly efficient parallel framework. Several optimizations could be done, including but not limited to:
    - Adding thread-based parallelism to the layer-finding, the cell refinement, and the history update subroutines to take full advantage of hybrid distributed-shared memory architectures.
    - Refactoring algorithms to use sets instead of vectors to reduce the computational complexity.
- In this work, a voxel domain with the size of the global mesh was employed to store the history variables in each process. Therefore, several voxels are not used throughout the analysis, increasing the memory footprint unnecessarily. This is especially true when the number of processes is high and the ratio of local to global number of elements is small. The framework currently allows the use of a multi-level grid to avoid this problem. However, the implementation in a parallel setting remains to be validated.
- Additive Schwarz preconditioners [Jomo et al., 2019] have proved to be successful in helping to achieve convergence of the solution in the studied cases. However, its suitability for larger and more complex phase-field problems needs further investigation. The *hp*-multigrid preconditioner presented in [Jomo et al., 2021] could be adapted for such cases.
- The discussed examples have shown that the geometric `Zoltan` partitioner [Devine et al., 2002] sometimes results in disjoint meshes. For a given number of owned elements, a disjoint mesh has a higher number of ghost elements compared to a contiguous mesh, and the computations become more expensive. Other partitioners could be integrated to try to eliminate or reduce this negative effect.

Appendix A

Determination of surrounding cells

The set of surrounding cells $\mathcal{C}^{\text{surr}}$ is essential for the algorithms developed in this thesis, see 4.2.2, 4.2.4, and 4.2.5. To determine this set, the following steps are carried out, see Figure A.1:



56	57	58	59	60	61	62	63
48	49	50	51	52	53	54	55
40	41	42	43			46	47
32	33	34	35			38	39
24	25	26	27	28	29	30	31
16	17	18	19	20	21	22	23
8	9	10	11	12	13	14	15
0	1	2	3	4	5	6	7

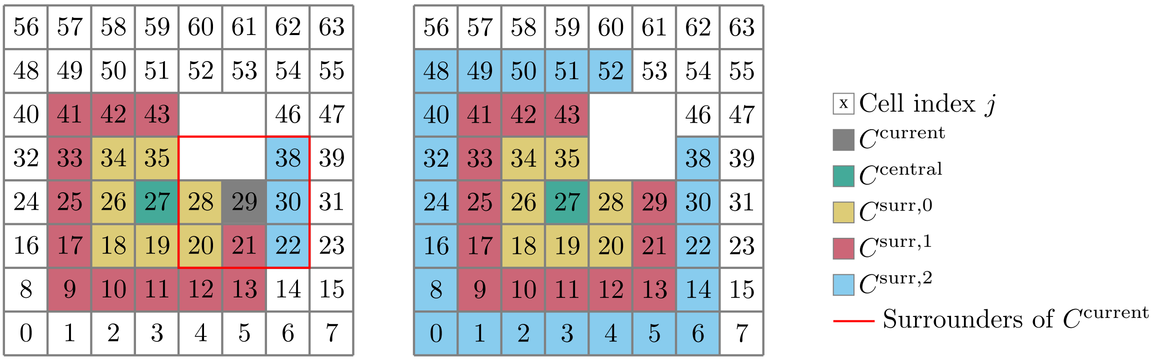
(a) First layer of surrounding cells of cell with index $j = 27$.

56	57	58	59	60	61	62	63
48	49	50	51	52	53	54	55
40	41	42	43			46	47
32	33	34	35			38	39
24	25	26	27	28	29	30	31
16	17	18	19	20	21	22	23
8	9	10	11	12	13	14	15
0	1	2	3	4	5	6	7

(b) Surrounding cells of cell with index $j = 28$.

56	57	58	59	60	61	62	63
48	49	50	51	52	53	54	55
40	41	42	43			46	47
32	33	34	35			38	39
24	25	26	27	28	29	30	31
16	17	18	19	20	21	22	23
8	9	10	11	12	13	14	15
0	1	2	3	4	5	6	7

(c) Second layer of surrounding cells.



56	57	58	59	60	61	62	63
48	49	50	51	52	53	54	55
40	41	42	43			46	47
32	33	34	35			38	39
24	25	26	27	28	29	30	31
16	17	18	19	20	21	22	23
8	9	10	11	12	13	14	15
0	1	2	3	4	5	6	7

(d) Surrounding cells of cell with index $j = 29$.

56	57	58	59	60	61	62	63
48	49	50	51	52	53	54	55
40	41	42	43			46	47
32	33	34	35			38	39
24	25	26	27	28	29	30	31
16	17	18	19	20	21	22	23
8	9	10	11	12	13	14	15
0	1	2	3	4	5	6	7

(e) Third layer of surrounding cells.

Figure A.1: Determination of surrounding cells in a Cartesian grid with a square hole.

1. For a given cell C^{central}, one layer of surrounding cells $C^{\text{surr},0}$ is determined, see Figure A.1a. This is done with a simple shift of cell indices and excluding the outside cells

C^{out}. The result of this operation is the set of all surrounding cells of the first layer $\mathcal{C}^{\text{surr},0}$.

2. For each cell C^{current} in $\mathcal{C}^{\text{surr},0}$, one layer of surrounding cells is determined analogously as in 1, see Figure A.1b. Surrounding cells that do not belong to $\mathcal{C}^{\text{surr},0}$ and that are different than C^{central} are added to the set of surrounding cells of the second layer $\mathcal{C}^{\text{surr},1}$.

3. Step 2 is repeated for cells in $\mathcal{C}^{\text{surr},1}$ excluding all previously found cells to get $\mathcal{C}^{\text{surr},2}$, see Figure A.1d.

Although the algorithms presented in this work require up to three layers of surrounding cells, more layers can be determined by repeating step 3 for each newly found layer of surrounding cells.

Figure A.1 shows an example of the determination of three layers of surrounding cells of a cell with index $j = 27$ in a Cartesian grid with a square hole (outside cells). In Figure A.1a, the set of surrounding cells of the first layer is determined as $\mathcal{C}^{\text{surr},0} = \{18, 19, 20, 26, 28, 34, 35\}$. Then, for a cell in $\mathcal{C}^{\text{surr},0}$, one layer of surrounding cells is determined (Figure A.1b), and the inside cells not included in $\mathcal{C}^{\text{surr},0} \cup C^{\text{central}}$ are added to $\mathcal{C}^{\text{surr},1}$. This is repeated for all cells in $\mathcal{C}^{\text{surr},0}$ until the full set $\mathcal{C}^{\text{surr},1} = \{9, 10, 11, 12, 13, 17, 21, 25, 29, 33, 41, 42, 43\}$ is determined. Finally, analogous steps are repeated to get the set of surrounding cells of the third layer $\mathcal{C}^{\text{surr},2} = \{0, 1, 2, 3, 4, 5, 6, 8, 14, 16, 22, 24, 30, 32, 38, 40, 48, 49, 50, 51, 52\}$, see figures A.1d and A.1e.

List of Figures

2.1 Modes of fracture. 6

2.2 Sharp crack topology and phase-field regularized crack surface, adopted from [Hug et al., 2020]. 6

3.1 Visualization of the FCM. 10

3.2 Comparison of the linear, quadratic and cubic one-dimensional integrated Legendre and Lagrange basis functions, adopted from [Jomo, 2021]. 10

3.3 Illustration of the multi-level hp-refinement scheme with two refinement levels, $k = 2$, in different spatial dimensions, adopted from [Zander et al., 2016]. The deactivation of specific topological components following a simple rule-set ensures compatibility and linear independence of the basis functions. 12

4.1 Plot of the speedup according to Amdahl's law for different serial fractions of the code. 18

4.2 Definition of different types of cells for a grid $\mathcal{G}$ partitioned among three processes, from the point of view of the bottom right partition. 20

4.3 Distributed mesh generation starting from a Cartesian grid $\mathcal{G}^{\text{init}}$ where inside cells are partitioned among three processes, as seen from the point of view of the bottom right partition. 21

4.4 Scheme of the steps needed to find ghost cells of the first layer $C^{\text{ghost},0}$. 22

4.5 Scheme showing the categories of DOFs in a local mesh. 24

4.6 Elements of a local mesh with invalid solution. 25

4.7 Parallel adaptive refinement based on the phase-field solution for a crack propagating between displacement steps u_0 and u_1, as seen from the bottom left partition with rank 0. 26

4.8 Scheme of the steps needed to find inner boundary cells of the first layer $C_0^{\text{inner},0}$, as seen from the bottom left partition with rank 0. 27

4.9 Scheme showing how to determine the processes to which the refinement data has to be sent to. 27

4.10 Extraction of indices and depths from a refined cell for communication. . . . 28

4.11 Scheme representing the operation to determine the number of messages to receive when a grid is partitioned among four processes. 28

4.12 Determination of source cells after repartitioning among four processes. . . . 32

4.13 Scheme representing the steps to determine to which processes a cell has to send its associated data to. 33

4.14 Determination of cells that receive data from other processes and cells that update locally after repartitioning among four processes, as seen from the bottom left partition with rank 0. 34

4.15 Block selection for multi-level *hp*-grids with the full and truncated blocks of element T_3, adopted from [Jomo et al., 2019]. 37

5.1 Square plate in tension with pre-existing crack. Geometry and boundary conditions. 40

5.2 Square plate in tension with pre-existing crack. Comparison of the load-displacement curves of a serial and a parallel solution with 64 processes and $\zeta = 1.0$. 40

5.3 Square plate in tension with pre-existing crack. Strong scaling analysis for different meshes. 41

5.4 Square plate in tension with pre-existing crack. Strong scaling analysis for different refinement depths k. 42

5.5 Square plate in tension with pre-existing crack. Strong scaling analysis for different polynomial degrees p. 43

5.6 Square plate in tension with pre-existing crack. Evolution of the maximum relative difference of the normalized load $d_{\max}$ without repartitioning vs. the accumulated staggered iterations. 44

5.7 Square plate in tension with pre-existing crack. Influence of the number of steps to wait for repartitioning on the speedup. 44

5.8 Square plate in tension with pre-existing crack. Influence of the repartitioning tolerance ζ on the speedup. 45

5.9 Square plate in tension with pre-existing crack. Evolution of the maximum relative difference of the normalized load $d_{\max}$ with dynamic repartitioning vs. the accumulated staggered iterations. 45

5.10 Square plate in tension with pre-existing crack. Phase-field (top) and partitioning (bottom) for different displacement steps. 46

5.11 Square plate in tension with rotated elliptical hole. Geometry and boundary conditions. 47

5.12 Square plate in tension with rotated elliptical hole. Strong scaling analysis. . 48

5.13 Square plate in tension with rotated elliptical hole. Evolution of the maximum relative difference of the normalized load d_{max} without repartitioning vs. the accumulated staggered iterations. 48

5.14 Square plate in tension with rotated elliptical hole. Influence of the repartitioning tolerance ζ on the speedup. 49

5.15 Square plate in tension with rotated elliptical hole. Evolution of the maximum relative difference of the normalized load d_{max} with dynamic repartitioning vs. the accumulated staggered iterations. 49

5.16 Square plate in tension with rotated elliptical hole. Phase-field (top) and partitioning (bottom) for different displacement steps. 50

5.17 Cubic body in tension with spherical inclusions. Section view of the geometry and boundary conditions. 51

5.18 Cubic body in tension with spherical inclusions. Strong scaling analysis. . . . 51

5.19 Cubic body in tension with spherical inclusions. Phase-field (top) and partitioning (bottom) for different displacement steps. Displacements on the top face are applied in the vertically upward direction. 52

5.20 Compression test of a core sample. Geometry and boundary conditions. . . . 53

5.21 Compression test of a core sample. Phase-field (top) and partitioning (bottom) for different displacement steps. Displacements on the top face are applied in the vertically downward direction. 54

5.22 Compression test of a core sample. Photo showing the resulting crack. 55

A.1 Determination of surrounding cells in a Cartesian grid with a square hole. . . 59

Bibliography

Alberti, G. (2000). *Variational models for phase transitions, an approach via Γ-convergence*, pages 95–114. Springer Berlin Heidelberg, Berlin, Heidelberg.

Ambati, M., Gerasimov, T., and De Lorenzis, L. (2015). A review on phase-field models of brittle fracture and a new fast hybrid formulation. *Computational Mechanics*, 55(2):383–405.

Babuska, I. and Guo, B. (1992). The h, p and hp version of the finite element method: basis theory and applications. *Advances in Engineering Software*, 15:159–174.

Badri, M., Rastiello, G., and Foerster, E. (2021). Preconditioning strategies for vectorial finite element linear systems arising from phase-field models for fracture mechanics. *Computer Methods in Applied Mechanics and Engineering*, 373:113472.

Bourdin, B., Francfort, G., and Marigo, J.-J. (2000). Numerical experiments in revisited brittle fracture. *Journal of the Mechanics and Physics of Solids*, 48(4):797–826.

Carpenter, W., Read, D., and Dodds, R. (1986). Comparison of several path independent integrals including plasticity effects. *International journal of fracture*, 31(4):303–323.

Clarke, L., Glendinning, I., and Hempel, R. (1994). The MPI Message Passing Interface Standard. In Decker, K. M. and Rehmann, R. M., editors, *Programming Environments for Massively Parallel Distributed Systems*, pages 213–218, Basel. Birkhäuser Basel.

Dagum, L. and Menon, R. (1998). OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55.

de Prenter, F., Verhoosel, C., van Zwieten, G., and van Brummelen, E. (2017). Condition number analysis and preconditioning of the finite cell method. *Computer Methods in Applied Mechanics and Engineering*, 316:297–327. Special Issue on Isogeometric Analysis: Progress and Challenges.

Devine, K., Boman, E., Heaphy, R., Hendrickson, B., and Vaughan, C. (2002). Zoltan data management services for parallel dynamic applications. *Computing in Science Engineering*, 4(2):90–96.

Dongarra, J. J., Duff, L. S., Sorensen, D. C., and Vorst, H. A. V. (1998). *Numerical Linear Algebra for High Performance Computers*. Society for Industrial and Applied Mathematics, USA.

Düster, A., Parvizian, J., Yang, Z., and Rank, E. (2008). The finite cell method for three-dimensional problems of solid mechanics. *Computer Methods in Applied Mechanics and Engineering*, 197(45):3768–3782.

Elhaddad, M., Zander, N., Bog, T., Kudela, L., Kollmannsberger, S., Kirschke, J., Baum, T., Ruess, M., and Rank, E. (2017). Multi-level *hp*-finite cell method for embedded interface problems with application in biomechanics. *International Journal for Numerical Methods in Biomedical Engineering*, 34.

Elices, M., Guinea, G., Gómez, J., and Planas, J. (2002). The cohesive zone model: advantages, limitations and challenges. *Engineering Fracture Mechanics*, 69(2):137–163.

Griffith, A. A. (1921). VI. The phenomena of rupture and flow in solids. *Philosophical transactions of the royal society of london. Series A, containing papers of a mathematical or physical character*, 221(582-593):163–198.

Gustafson, J. L. (1988). Reevaluating Amdahl's Law. *Commun. ACM*, 31(5):532–533.

Hackbusch, W. (1994). *Iterative solution of large sparse systems of equations (1st ed.)*, volume 95.

Hendrickson, B. and Devine, K. (2000). Dynamic load balancing in computational mechanics. *Computer Methods in Applied Mechanics and Engineering*, 184:485–500.

Heroux, M. A., Bartlett, R. A., Howle, V. E., Hoekstra, R. J., Hu, J. J., Kolda, T. G., Lehoucq, R. B., Long, K. R., Pawlowski, R. P., Phipps, E. T., Salinger, A. G., Thornquist, H. K., Tuminaro, R. S., Willenbring, J. M., Williams, A., and Stanley, K. S. (2005). An Overview of the Trilinos Project. *ACM Trans. Math. Softw.*, 31(3):397–423.

Hug, L., Kollmannsberger, S., Yosibash, Z., and Rank, E. (2020). A 3D benchmark problem for crack propagation in brittle fracture. *Computer Methods in Applied Mechanics and Engineering*, 364:112905.

Ibanez, L., Schroeder, W., Ng, L., Cates, J., and Consortium, I. (2005). The ITK Software Guide, Second Edition, Updated for ITK version 2.4.

Irwin, G. R. (1960). Plastic zone near a crack and fracture toughness. In *Proceedings of the 7th Sagamore Ordnance Materials Research Conference*, volume 4, pages 63–78, New York, NY. Syracuse University Press.

Jomo, J., de Prenter, F., Elhaddad, M., D'Angella, D., Verhoosel, C., Kollmannsberger, S., Kirschke, J., Nübel, V., van Brummelen, E., and Rank, E. (2019). Robust and parallel scalable iterative solutions for large-scale finite cell analyses. *Finite Elements in Analysis and Design*, 163:14–30.

Jomo, J., Oztoprak, O., de Prenter, F., Zander, N., Kollmannsberger, S., and Rank, E. (2021). Hierarchical multigrid approaches for the finite cell method on uniform and multi-level *hp*-refined grids. *Computer Methods in Applied Mechanics and Engineering*, 386:114075.

Jomo, J. N. (2021). *Towards scalable finite cell computations on massively parallel systems*. PhD thesis, Technical University of Munich.

Korshunova, N., Papaioannou, I., Kollmannsberger, S., Straub, D., and Rank, E. (2021). Uncertainty quantification of microstructure variability and mechanical behavior of additively manufactured lattice structures. *Computer Methods in Applied Mechanics and Engineering*, 385:114049.

Leibniz Supercomputing Centre (2021). CoolMUC-2. `https://doku.lrz.de/display/PUBLIC/CoolMUC-2`. [Online; accessed 25.10.2021].

Miehe, C., Hofacker, M., and Welschinger, F. (2010). A phase field model for rate-independent crack propagation: Robust algorithmic implementation based on operator splits. *Computer Methods in Applied Mechanics and Engineering*, 199(45):2765–2778.

Moes, N., Dolbow, J., and Belytschko, T. (1999). A Finite Element Method for Crack Growth without Remeshing. *International Journal for Numerical Methods in Engineering*, 46:131–150.

Nagaraja, S., Elhaddad, M., Ambati, M., Kollmannsberger, S., De Lorenzis, L., and Rank, E. (2018). Phase-field modeling of brittle fracture with multi-level *hp*-FEM and the finite cell method. *Computational Mechanics*, 63(6):1283–1300.

Parvizian, J., Düster, A., and Rank, E. (2007). Finite cell method. *Computational Mechanics*, 41.

Ristov, S., Prodan, R., Gusev, M., and Skala, K. (2016). Superlinear speedup in HPC systems: Why and when? In *2016 Federated Conference on Computer Science and Information Systems (FedCSIS)*, pages 889–898.

Robey, R. and Zamora, Y. (2021). *Parallel and High Performance Computing*. Manning.

Saad, Y. (2003). *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, second edition.

Schäling, B. (2014). *The boost C++ libraries*. XML Press.

Szabó, B. and Babuška, I. (1991). *Finite element analysis*. John Wiley & Sons.

Wilde, T., Ott, M., Auweter, A., Meijer, I., Ruch, P., Hilger, M., Kühnert, S., and Huber, H. (2017). CooLMUC-2: A supercomputing cluster with heat recovery for adsorption cooling. In *2017 33rd Thermal Measurement, Modeling Management Symposium (SEMI-THERM)*, pages 115–121.

Zander, N., Bog, T., Elhaddad, M., Frischmann, F., Kollmannsberger, S., and Rank, E. (2016). The multi-level *hp*-method for three-dimensional problems: Dynamically changing high-order mesh refinement with arbitrary hanging nodes. *Computer Methods in Applied Mechanics and Engineering*, 310:252–277.

Zander, N., Bog, T., Kollmannsberger, S., Schillinger, D., and Rank, E. (2015). Multi-level *hp*-adaptivity: high-order mesh adaptivity without the difficulties of constraining hanging nodes. *Computational Mechanics*, 55:499–517.

Zhang, X., Vignes, C., Sloan, S., and Sheng, D. (2017). Numerical evaluation of the phase-field model for brittle fracture with emphasis on the length scale. *Computational Mechanics*, 59.