Parallel Re-Initialization of Level Set Functions and Load Balancing for Two-Phase Flow Simulations

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften
der RWTH Aachen University zur Erlangung des akademischen Grades eines
Doktors der Naturwissenschaften genehmigte Dissertation

vorgelegt von

Diplom-Mathematiker Oliver Christian Fortmeier
aus Duisburg

Berichter: apl. Professor Dr.-Ing. H. Martin Bücker
Universitätsprofessor Dr. Arnold Reusken
Universitätsprofessor Dr. Uwe Naumann

Tag der mündlichen Prüfung: 17. November 2011

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über
http://dnb.d-nb.de abrufbar.

ISBN 978-3-8439-0227-4

D 82 (Diss. RWTH Aachen University, 2011)

Sternstr. 18, 80538 München
Tel.: 089/66060798
www.dr.hut-verlag.de

1. Auflage 2011

Für Caro

Zusammenfassung

Die vorliegende Arbeit beschäftigt sich mit parallelen Algorithmen, welche bei der Simulation von dreidimensionalen Zweiphasenströmungen auf adaptiv verfeinerten, unstrukturierten Tetraedergittern eingesetzt werden. Ziel dabei ist es, diese Strömungen von zwei unmischbaren Phasen auf modernen Hochleistungsrechnerarchitekturen zu simulieren, welche derzeit in der Regel aus einer großen Anzahl von vernetzten Mehrkernprozessoren bestehen.

Die mathematische Modellierung der Zweiphasenströmungen beruht auf den Navier–Stokes-Gleichungen zur Beschreibung der Strömungsdynamik und der Levelset-Methode zur Charakterisierung der beiden Phasen. Um diese partiellen Differentialgleichungen rechnergestützt zu lösen, werden für die Ortsdiskretisierung Finite-Elemente-Funktionen und für die Zeitdiskretisierung ein implizites Theta-Schema angewandt. Dadurch können Zweiphasenströmungsprobleme in Form einer Sequenz von großen, dünnbesetzten linearen Gleichungssystemen dargestellt werden, welche iterativ durch Krylov-Teilraumverfahren gelöst werden.

Um die Rechenarbeit der Simulation auf Rechenkerne mit verteiltem Speicher aufzuteilen, wird ein Gebietszerlegungsansatz gewählt, in dem – basierend auf einer Partitionierung des Rechengebiets – die zugrundeliegende Hierarchie von Tetraedergittern verteilt wird. In dieser Arbeit werden Graph- und Hypergraph-Partitionierungsmodelle eingeführt, um eine Zerlegung des Rechengebiets für Zweiphasenströmungssimulationen zu bestimmen. Ein zentraler Algorithmus bei der Verwendung des Levelset-Ansatzes ist die Reinitialisierung der Levelset-Funktion, welche ihre numerisch wichtige Eigenschaft einer vorzeichenbehafteten Abstandsfunktion wiederherstellt. Hierfür kommt ein neuer paralleler Algorithmus auf einem verteilt gespeicherten, unstrukturierten Tetraedergitter zum Tragen.

Alle in dieser Arbeit vorgestellten Konzepte wurden in dem Software-Werkzeug DROPS implementiert, das in einer Zusammenarbeit mit dem Lehrstuhl für numerische Mathematik der RWTH Aachen University entwickelt wird. Die parallele Skalierbarkeit dieser Methoden wird durch detaillierte numerische Experimente auf bis zu 1 024 Rechenkernen demonstriert. Zudem werden die parallelen Konzepte in einer ingenieurs-relevanten Fallstudie kombiniert, welche die hochaufgelöste Simulation eines n-Butanol-Tropfens in einer wässrigen Phase beinhaltet. Diese Studie entstammt dem Sonderforschungsbereich SFB 540 der RWTH und wurde erst durch den Einsatz paralleler Algorithmen auf modernen Hochleistungsrechnern ermöglicht, die den nötigen Speicher und Rechenleistung zur Verfügung stellen.

Abstract

This thesis addresses parallel algorithms for three-dimensional two-phase flow simulations on adaptively refined unstructured tetrahedra grids. These algorithms are designed to simulate the fluid dynamics of two immiscible phases on recent high-performance computer architectures which, in general, consist of clusters of a large number of multi-core processors.

The underlying mathematical model of these two-phase flows is based on the Navier–Stokes equations to describe the fluid dynamics. The level set approach is employed to characterize the two phases. The spatial discretization of these partial differential equations is given by the finite element method whereas the time discretization is performed by an implicit theta scheme. This approach facilitates the description of two-phase flow problems as a sequence of large and sparse systems of linear equations which are efficiently solved by Krylov subspace methods.

The computational work of two-phase flow simulations is decomposed among compute cores with distributed memory. To this end, a domain decomposition approach is pursued where the tetrahedra of the underlying hierarchy of triangulations are accordingly distributed. In this thesis, graph and hypergraph partitioning models are introduced which determine tetrahedral decompositions. These models are specifically designed for two-phase flow simulations. A major algorithmic element in such simulations is constituted by the re-initialization algorithm that periodically rebuilds a numerically crucial property of the level set function, namely the signed distance property. This task is addressed by a novel parallel algorithm which is capable of re-initializing level set functions on distributed unstructured triangulations.

The numerical results of the presented parallel concepts are gathered by the software toolkit DROPS which is being developed in a collaboration with the Chair of Numerical Mathematics at RWTH Aachen University. The parallel scalability of the methods is demonstrated by detailed numerical experiments on up to 1 024 compute cores. Furthermore, all parallel concepts are combined in an engineering relevant case study that is concerned with the analysis of an n-butanol drop in an aqueous phase on a triangulation with a high resolution. This study originates from the collaborative research center SFB 540 at RWTH. Its simulation is too large—in terms of memory and compute time—for sequential computing. Thus, only the parallel techniques presented in this thesis allow to perform this detailed analysis.

Acknowledgments

The Deutsche Forschungsgemeinschaft (DFG) supported many parts of my research work through the SFB 540 "Model-based experimental analysis of kinetic phenomena in fluid multi-phase reactive systems." Besides the financial support, its seminars provided a forum for many fruitful discussion and applications that have driven many of my research.

Special thanks go to thank my supervisor Prof. Martin Bücker. He has spent much time in introducing me into the world of parallel algorithms, methods, literature, and scientific writing. He has provided me with excellent support in the last years. Thank you, Martin. I would also like to express my deep gratitude to Prof. Reusken for being the co-advisor of this thesis and for showing much interest in my work. Moreover, I thank Prof. Uwe Naumann having reviewed my thesis.

The atmosphere at the Institute for Scientific Computing has stimulated me for many research issues. I really appreciate the—not only—scientific advises of Prof. Christian Bischof. Furthermore, I would like to thank all my colleagues and friends at SC, Michael Lülfesmann, Arno Rasch, Monika Petera, Andre Vehreschild, Johannes Willkomm, Andreas Wolf, and all the student workers who have supported me. The researches at the LNM are developing the software DROPS which provides a brilliant basis for my research. They have also helped me in understanding the mathematics in the interesting field of two-phase flows. Therefore, I am grateful to Patrick Esser, Jörg Grande, and Eva Loch. In particular, I want to express my thankfulness to Sven Groß. Furthermore, there are many people at the Center for Computing and Communication who have helped my throughout my research activities. I thank the HPC group for providing me such a detailed knowledge and insight of the HPC architectures at RWTH Aachen University. Thank you Christian Terboven, Dieter an Mey, Tim Cramer, Christian Iwainsky, Paul Kapinos, Samuel Sarholz, Dirk Schmidl, and Sandra Wienke. Bernd Hentschel has pointed me to the technology of k-d trees which forms one key ingredient of the parallel re-initialization algorithm. Thank you. The administrators of the RWTH Aachen's HPC computers have never stopped their effort to get my jobs through the batch queue. Prof. Rob Bisseling and Bas Fagginger Auer at Utrecht University have helped me partitioning an enormous number of hypergraphs in collaboration. Without their effort, the novel hypergraph model could not have been evaluated in this thesis.

All the proof readers deserve big thanks for spending their time revising my manuscripts. Besides the fantastic reviews of Patrick McCormick and Vera Peters, much beneficial assistance has been supplied by Patrick Esser, Sven Groß, Bernd Hentschel, Christian Iwainsky, Carolin Jacob, Michael Lülfesmann, Monika Petera, Dirk Schmidl, and Sandra Wienke.

Fabian Birchel and Vera Peters helped me to free my mind as friends during my researches. “Vielen Dank!” I am deeply grateful to my parents Hubert and Anne Marie as well to my sister Conny. Thank you for so much support and, thus, for the possibility performing this research. I thank Irene und Jörg Jacob for many fruitful discussions and mental support. Finally, I express my deep gratitude to Caro for encouraging me during the whole research process and during authoring this thesis. I love you.

Contents

1 Introduction

Two-phase flow problems constitute a key element in various areas of computational science and engineering. These flows occur in systems that contain two spatially separated regions which vary over time. In each spatial region, all physical properties are homogeneous. Illustrating examples are water and steam in power plant boilers, oil slicks in coastal waters, air and water in groundwater flow, liquid-liquid extraction columns, or falling films. Up to now, a detailed knowledge of such systems' behavior still poses a challenge for scientists. To gain a deeper insight in multi-phase and particularly two-phase flow problems, an interdisciplinary team of researchers from computer science, mathematics, chemistry and engineering at RWTH Aachen University concentrates on analyzing these flows, e.g., in single drops [20,87,88] and in falling films [32,55,56,103–105]. The work was part of the collaborative research center SFB 540 "Model-based experimental analysis of kinetic phenomena in fluid multi-phase reactive systems" [119] where a central goal was to derive mathematical models describing two-phase flows. To predict physical phenomena, these models are then implemented in a simulation software.

In addition to algorithms used to simulate one-phase flows, the complex phenomena of two-phase flows demand special, sophisticated numerical techniques [85]: the level set approach [160] appropriately describes the two phases; the extended finite element method [83,135] adequately captures the pressure discontinuity between the two phases; adaptive tetrahedral grid refinement [82] facilitates a high spatial resolution in domains of interest; and the continuous surface force term [84] couples the two phases at the interface. If these methods are applied in a finite element simulation, then solving two-phase flow problems typically results in large amount of computational work and memory requirements. To cope with these requirements for meaningful problem sizes, employing high-performance computers is indispensable as they provide both, the necessary compute power and sufficient memory.

If we merely consider one-phase flow simulations, the literature includes a vast variety of serial and parallel algorithms for simulating the computational fluid dynamics by the finite element method on unstructured grids. However, a quite different picture is observed for two-phase flow simulations based on the widely applied level set approach when following a domain decomposition approach [39]. Serial algorithms [85] exist to simulate these flows by the finite element method on unstructured grids. Although some parallel one-phase flow algorithms can be

adapted to employ a domain decomposition approach, there remains a remarkable lack of parallel algorithms. To tackle this challenge, the major contributions of this thesis address the gap of missing parallel algorithms by focusing on load balance models for parallel two-phase flow simulations and on the re-initialization of level set functions.

Modeling Load Balancing To efficiently solve the partial differential equations describing a two-phase flow problem, an adaptive refined tetrahedral grid is employed to discretize the computational domain yielding a hierarchy of triangulations. Pursuing a domain-decomposition approach to address a distributed-memory parallelization raises the question of how to distribute the tetrahedra among the processes. We have presented solutions for two-phase flow problems on these meshes in [72] and [67]. To this end, the tetrahedral hierarchy is described by various graph models specifically designed for two-phase flow simulations. A partitioning of these graphs results in a decomposition of the set of tetrahedra. Chapter 4 shows how to extend and to combine the graph partitioning models presented in [72] and [67]. Furthermore, that chapter introduces an innovative model that relies on a hypergraph formulation. In contrast to "standard" graph models, this novel hypergraph model is capable of exactly expressing the communication volume among neighboring processes for linear algebra operations during a distributed two-phase flow simulation. Although all graph models in this thesis are particularly designed for two-phase flow simulations, they can also be employed in other simulations where the computational loads per compute element vary, e.g., in simulations where turbulences occur in subdomains.

Parallel Re-Initialization If simulating two-phase flow problems by the level set approach, the numerically crucial "signed distance property" of the level set function is not preserved in general. This property ensures that the sign of the level set function characterizes the two phases while its absolute value describes the distance between any point in the computational domain and the interface. To re-establish the signed distance property, the level set function is re-initialized. Typically, only a small fraction of the execution time is spent for this task in a serial simulation. However, in parallel simulations dealing with large meaningful problem instances, a parallelized version of a fast serial re-initialization algorithm is increasingly getting a major performance bottleneck when adding more and more processes. The problem of re-establishing the signed distance property of level set functions occurs—besides in two-phase flow problems—in many applications which are based on a level set formulation, e.g., propagating interfaces, optimal path planning and construction of shortest geodesic paths [153]. In the literature, several serial and parallel algorithms exist for re-initializing level set functions: PDE-based methods [155], the fast marching method [152], the fast

iterative method [99], the fast sweeping method [178], and the Euclidean distance transform method [61]. Nevertheless, none of these methods is capable of re-initializing level set functions on distributed unstructured grids. We introduced a parallel algorithm in [70] which is specifically designed for distributed-memory computations. This novel algorithm is also well suited in the context of a hybrid parallel approach where a distributed- and a shared-memory parallelization [69] are nested to exploit the structure of today's high-performance computers which commonly consist of clusters of multi-core processors. For the parallel re-initialization algorithm, we combined two known algorithmic elements in a novel way. The first element consists of a brute-force re-initialization strategy. The second element is a suitable multidimensional tree data structure which is employed to reduce the time complexity. Chapter 5 focuses on this novel parallel re-initialization algorithm.

We present a detailed evaluation to demonstrate the general applicability of these new algorithmic approaches in the context of two-phase flow simulations. Therefore, the algorithms are implemented in the software DROPS [117] which is being developed in a collaboration with the Chair for Numerical Mathematics at RWTH Aachen University. In this software, the distributed-memory parallelization is implemented by the Message Passing Interface (MPI) [175] and the shared-memory parallelization is based on OPENMP [42, 129]. Numerical results are gathered on recent high-performance compute clusters located at the Center for Computing and Communication with up to 1 024 processes. In previous work [20], the predictive results of the parallel two-phase flow solver DROPS are used to answer engineering questions. Besides forward simulations where for given input parameters a solution is simulated, this software is also employed in optimizations [30, 32, 86] where, in contrast, the input parameters are sought. For instance, in [30], the inflow velocity into a measurement cell is estimated such that the simulated velocities inside the cell [68] match the measured data which are obtained by nuclear magnetic resonance measurements [5, 6].

In summary, the main new contributions of this thesis are as follows. First, models for load-balancing of two-phase flow finite element simulations are systematically developed and derived. This is the first time, a hypergraph model is used in this context to determine a decomposition of a tetrahedral hierarchy. Second, the re-initialization of level-set functions on distributed unstructured triangulations based on [69, 70] is presented in detail.

The outline of this thesis is as follows. In Chap. 2, we briefly summarize the governing mathematical equations describing a two-phase flow problem when considering the level set approach. Furthermore, we recapitulate a finite-element based algorithm [80, 81, 85] to simulate such flow problems on a hierarchy of tetrahedral grids. In Chap. 3, we outline a distributed-memory parallelization of this basic algorithm. Thereby, we describe a domain-decomposition approach to distribute the computational domain among processes, and we present the effect of this de-

composition on linear algebra operations. A new data format for storing vectors allows us to increase the accuracy of parallel linear algebra operations. The following two chapters represent the main novel contribution of this thesis. In Chap. 4, we focus on answering the question how to decompose the tetrahedral hierarchy that discretizes the computational domain. To this end, we formulate various graph partitioning problems which are specifically designed for parallel two-phase flow simulations. We summarize the approach of Groß [80] and formulate our two approaches originally presented in [72] and [67], respectively. Additionally, we introduce th novel hypergraph model. Chapter 5 deals with the re-initialization of level set functions. Here, we present our algorithm introduced in [70] and [69]. Furthermore, we analyze this algorithm theoretically and numerically and compare it with an existing algorithm. Its parallel scalability is demonstrated by numerical experiments on up to 1 024 compute cores. In Chap. 6, we employ all presented parallel algorithms and techniques to analyze the effect of grid refinements on the rising velocity of an n-butanol drop surrounded by an aqueous phase. We also show that this detailed study is infeasible when solely considering a serial simulation. Chapter 7 finalizes this thesis by summarizing the main contributions of the present work, stating some open questions for future research, and drawing some conclusions.

2 Modeling and Simulating Two-Phase Flow Problems

The numerical simulation of two-phase flow problems is complicated by the fact that, besides the modeling of the fluid dynamics, the interface between the two phases needs to be represented for the reconstruction of the interfacial movement. In addition, the physical phenomena in the vicinity of the interface demand advanced techniques from computer sciences and mathematics. To numerically represent the two phases and the interface between these phases, two major approaches are available in the open literature, namely interface-tracking and interface-capturing.

Interface-tracking methods explicitly track the interface by computational elements that follow its movement. Here, either the grid on which the fluid dynamics are simulated is adapted to the interface (Lagrangian approach) [12] or an additional grid located at the interface is introduced (Eulerian approach) [169]. Commonly, the former one is applied if simulating free surface flows [17]. Nevertheless, this approach is also employed in two-phase flow simulations [165]. When using the latter approach joining or breaking regions is difficult to realize.

Interface-capturing methods represent the interface implicitly by suitably defined functions. This class mainly consists of the volume of fluid method and the level set approach. In the volume of fluid method [95,114], each grid cell contains information about the phases. For two-phase flows, this information is given by a fraction describing the ratio of the volume of the fluids in that particular cell. This method is mass conservative; however, the reconstruction of smooth interfaces requires advanced interpolation techniques. An overview of the volume of fluid method is presented for instance in [141]. The level set approach, presented in [41, 130], is another interface-capturing method eliminating this inconvenience. Herein, the position of the interface is captured by the zero level of a scalar-valued function, called the level set function, that splits the computational domain into two subdomains. A drawback of this method is that it does not conserve mass and that the shape of the function may become degenerated when evolving in simulation time. Yet, an advantage of the level set approach is its elegance and simplicity to handle complicated problems involving breaking or joining regions.

Besides pure interface-tracking and interface-capturing methods, there exist variants and combinations of both approaches. For instance in [93], two grids

for simulating two-phase flow problems are used. A first, unstructured grid is employed for discretizing the fluid dynamics and a second, structured grid is introduced for representing the level set function.

In this thesis, we use the level set approach for representing the two phases. Hereby, the same underlying computational grid is used for both, the fluid dynamics and the representation of the phases. The remainder of this chapter is organized as follows. Section 2.1 presents a mathematical model for simulating two-phase flow problems. Afterwards, in Sect. 2.2, we outline an algorithm for numerically solving the mathematical model.

2.1 A Mathematical Model for Two-Phase Flow Problems

In this section, a mathematical model for describing two-phase flows is presented which relies on the level set approach [159, 160]. The coupling of the two phases at the interface is based on a continuous surface force term [27, 41]. A detailed derivation of the mathematical model is given in [85]. This section is organized as follows. In Sect. 2.1.1, we outline the level set approach for representing two phases and describing the evolution of the interface. Afterwards, we summarize equations that model the underlying fluid dynamics in the context of two-phase flows in Sect. 2.1.2.

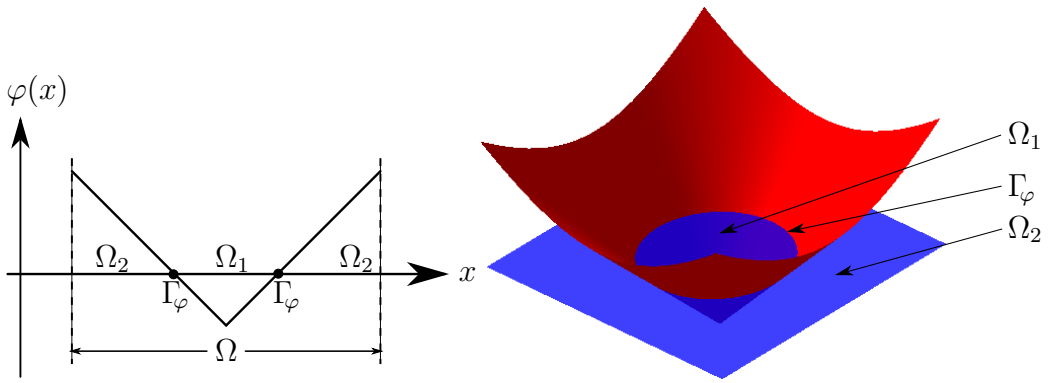
2.1.1 The Level Set Approach

We first introduce some notations to describe two incompressible and immiscible phases in a domain $\Omega \subset \mathbb{R}^3$ where we assume that the location of the phases changes with respect to time $\tau \in [0, \tau_e]$. To this end, let $\Omega_1(\tau) \subset \Omega$ and $\Omega_2(\tau) \subset \Omega$ denote the two subdomains where both phases are located. The domain consists of two immiscible phases which results in $\overline{\Omega} = \overline{\Omega_1}(\tau) \cup \overline{\Omega_2}(\tau)$ and $\Omega_1(\tau) \cap \Omega_2(\tau) = \emptyset$. The interface between both phases is given by the manifold $\Gamma(\tau) = \overline{\Omega_1}(\tau) \cap \overline{\Omega_2}(\tau)$.

The level set approach characterizes the two phases and their evolution. To this end, a scalar-valued function $\varphi : \Omega \times [0, \tau_e] \to \mathbb{R}$, called the *level set function*, is introduced. Hereby, at time τ, the interface between both phases is given by the zero level

$$\Gamma_\varphi(\tau) := \{\mathbf{x} \in \Omega \mid \varphi(\mathbf{x}, \tau) = 0\}$$

of φ. Since, at time τ, the interface $\Gamma(\tau)$ equals $\Gamma_\varphi(\tau)$, in the remainder of this theses, we denote the interface by $\Gamma_\varphi(\tau)$ to emphasize the dependence between the interface and the level set function. To distinguish between the two phases,



(a) One-dimensional example. (b) Two-dimensional example.

Figure 2.1: Examples of signed distance functions.

the sign of $\varphi(\cdot, \tau)$ is used. That is, a negative value of $\varphi(\mathbf{x}, \tau)$ indicates that the point $\mathbf{x}$ is located in the first phase at time τ and, vice versa, a positive value characterizes $\mathbf{x}$ in the second phase, in formula

$$\Omega_1(\tau) = \{\mathbf{x} \in \Omega \mid \varphi(\mathbf{x}, \tau) < 0\} \quad \text{and} \quad \Omega_2(\tau) = \{\mathbf{x} \in \Omega \mid \varphi(\mathbf{x}, \tau) > 0\}\,.$$

Moreover, it is convenient for numerical reasons that the absolute value of $\varphi(\mathbf{x}, \tau)$ equals the distance of $\mathbf{x}$ to the interface $\Gamma_\varphi(\tau)$. That is, φ fulfills the *distance property*

$$|\varphi(\mathbf{x}, \tau)| = \min_{\mathbf{y} \in \Gamma_\varphi} \|\mathbf{x} - \mathbf{y}\|_2 \tag{2.1}$$

for all $\mathbf{x} \in \Omega$ and $\tau \in [0, \tau_\mathrm{e}]$. We detail the term of the signed-distance property in Sect. 5.1.1 and show that this property implies $\|\nabla\varphi(\mathbf{x}, \tau)\|_2 = 1$.

Figure 2.1 shows two examples of level set functions bisecting a computational domain. Figure 2.1(a) presents a one-dimensional example, where the level set function is used to decompose an interval. In Fig. 2.1(b), the level set function describes a circle by its zero level $\Gamma_\varphi(\tau)$ in a square-shaped domain. Here, the first phase Ω_1 is located inside and the second phase Ω_2 is located outside of that circle.

The evolution of the interface $\Gamma_\varphi(\tau)$ follows the approach presented in [130]. In that paper, Osher and Sethian devised a numerical algorithm for surface motion problems. The approach is driven by the idea that, if a particle $\mathbf{x}(0) \in \Gamma_\varphi(0)$ resides at time $\tau = 0$ at the interface, then it stays on the interface for all $\tau \in [0, \tau_\mathrm{e}]$. Using the level set notation, this reads as

$$\varphi(\mathbf{x}(\tau), \tau) = \varphi(\mathbf{x}(0), 0) = 0. \tag{2.2}$$

Extending (2.2) to the computational domain Ω implies that a particle $\mathbf{x}(\tau)$ stays at a given level of φ for $\tau \in [0, \tau_\mathrm{e}]$, i.e.,

$$\varphi(\mathbf{x}(\tau), \tau) = \varphi(\mathbf{x}(0), 0) = \text{const}.$$

Differentiating this equation with respect to τ leads to

$$\frac{\partial}{\partial\tau}\varphi(\mathbf{x}(\tau),\tau) + \nabla\varphi(\mathbf{x}(\tau),\tau)\cdot\frac{\partial}{\partial\tau}\mathbf{x}(\tau) = 0. \tag{2.3}$$

Let $\mathbf{u}(\mathbf{x},\tau)$ denote the velocity in the computational domain. Then, the motion of the particles is given by this velocity, i.e.,

$$\frac{\partial}{\partial\tau}\mathbf{x}(\tau) = \mathbf{u}(\mathbf{x},\tau). \tag{2.4}$$

The equations (2.3) and (2.4) yield the level set equation

$$\frac{\partial}{\partial\tau}\varphi + \mathbf{u}\cdot\nabla\varphi = 0, \tag{2.5}$$

which describes the evolution of the interface $\Gamma_\varphi(\tau)$. Next, we couple this level set equation with a term determining the forces at the interface $\Gamma_\varphi(\tau)$ and the Navier–Stokes equations describing the fluid dynamics.

2.1.2 The Fluid Dynamics

To describe the underlying fluid dynamics, the velocity $\mathbf{u} = \mathbf{u}(\mathbf{x},\tau)$ and the pressure $p = p(\mathbf{x},\tau)$ are modeled by combining the Navier–Stokes equations with the continuum surface force (CSF) model [27, 41]. This model introduces a local force term

$$f_\Gamma := \gamma\kappa_\Gamma\delta_\Gamma\mathbf{n}_\Gamma \tag{2.6}$$

at the interface $\Gamma_\varphi(\tau)$. Here, γ is the surface tension coefficient which is given by the material properties of the two-phase system, κ_Γ the curvature of $\Gamma_\varphi(\tau)$, δ_Γ the Dirac delta function, and $\mathbf{n}_\Gamma$ the unit normal on $\Gamma_\varphi(\tau)$ pointing from $\Omega_1(\tau)$ to $\Omega_2(\tau)$. Note that the curvature κ_Γ and the unit normal $\mathbf{n}_\Gamma$ are time dependent and can be evaluated in terms of the level set function by

$$\mathbf{n}_\Gamma(\tau) = \frac{\nabla\varphi(\tau)}{\|\nabla\varphi(\tau)\|_2} \quad\text{and}\quad \kappa_\Gamma(\tau) = -\operatorname{div}(\mathbf{n}_\Gamma(\tau)) = -\operatorname{div}\left(\frac{\nabla\varphi(\tau)}{\|\nabla\varphi(\tau)\|_2}\right). \tag{2.7}$$

The combination of the Navier–Stokes equations and the CSF model yields the following equations in $\Omega\times[0,\tau_e]$,

$$\varrho\left(\frac{\partial}{\partial\tau}\mathbf{u} + (\mathbf{u}\cdot\nabla)\mathbf{u}\right) = -\nabla p + \varrho\,\mathbf{g} + \operatorname{div}(\mu\mathbf{D}(\mathbf{u})) + f_\Gamma \qquad \text{in } \Omega, \tag{2.8}$$

$$\operatorname{div}(\mathbf{u}) = 0 \qquad \text{in } \Omega \tag{2.9}$$

with suitable boundary and initial conditions. In these equations, the symbol $\mathbf{g}$ denotes gravity and $\mathbf{D}(\mathbf{u}) := \nabla\mathbf{u} + (\nabla\mathbf{u})^T$ the deformation tensor. We assume

that the density ϱ and dynamic viscosity μ are constant in each phase and are given by ϱ_i and μ_i, with $i = 1, 2$, for the first and second phase, respectively. Then the density ϱ and dynamic viscosity μ in (2.8) are represented by piece-wise constant functions which depend on the phases and, hence, on the sign of the level set function, i.e.,

$$\varrho(\mathbf{x}, \tau) = \varrho(\varphi) = \begin{cases} \varrho_1, & \mathbf{x} \in \Omega_1(\tau) \\ \varrho_2, & \mathbf{x} \in \Omega_2(\tau) \end{cases} \quad \text{and} \tag{2.10}$$

$$\mu(\mathbf{x}, \tau) = \mu(\varphi) = \begin{cases} \mu_1, & \mathbf{x} \in \Omega_1(\tau) \\ \mu_2, & \mathbf{x} \in \Omega_2(\tau) \end{cases}. \tag{2.11}$$

Overall, the Navier–Stokes equations (2.8)–(2.9) with the CSF term (2.6) and the level set equation (2.5) constitute a mathematical model for describing two-phase flow problems in $\Omega \times [0, \tau_e]$:

$$\varrho \left(\frac{\partial}{\partial \tau} \mathbf{u} + (\mathbf{u} \cdot \nabla)\mathbf{u} \right) = -\nabla p + \varrho\, \mathbf{g} + \operatorname{div}\left(\mu \mathbf{D}(\mathbf{u})\right) + \gamma \kappa_\Gamma \delta_\Gamma \mathbf{n}_\Gamma, \tag{2.12}$$

$$\operatorname{div}(\mathbf{u}) = 0, \text{ and} \tag{2.13}$$

$$\frac{\partial}{\partial \tau} \varphi + \mathbf{u} \cdot \nabla \varphi = 0. \tag{2.14}$$

In the next section, we present an algorithm that solves this system of coupled partial differential equations (PDEs) by the finite element method.

2.2 Simulating Two-Phase Flow Problems

The algorithm presented in this section is based on [81] and a detailed description of the algorithm can be found in [85]. In [20], the algorithm has been validated by comparing its results to the (measured) rising velocity of an n-butanol drop in water.

Algorithm 1 summarizes the proposed algorithm. Here, a hierarchy of tetrahedral grids, denoted by $\mathcal{M}$, is employed to numerically describe the computational domain Ω. In Sect. 2.2.1, we introduce this hierarchy and discuss an algorithm for suitably modifying it. The velocity, pressure and level set function are discretized by finite element functions $\mathbf{u}_h$, p_h and φ_h, whose vector representations at a given time τ_i are denoted by $\overline{\mathbf{u}}^i$, $\overline{p}^i$, and $\overline{\varphi}^i$, respectively. The resulting, discrete two-phase flow problem and an algorithm to solve the discrete two-phase flow problem are sketched in Sect. 2.2.2.

Although the continuous formulation of level set method is mass conservative, the discretization of the underlying partial differential equations may introduce a

loss of mass [128,167]. To address this drawback, we pursue the following strategy. If, the difference between the volume of Ω_1 and its initial volume is larger or smaller than a given threshold β_V in a time step τ_i, the level set function is modified such that the initial volume of the phase Ω_1 is recovered. This is implemented by shifting the level set function. Therefore, let $V_1(\varphi)$ denote the volume of Ω_1, then, at time τ_i, we move φ by $d \in \mathbb{R}$ such that

$$V_1(\varphi_h(\cdot, \tau_i) + d) \;=\; V_1(\varphi(\cdot, 0)) \;=:\; V_0 \;=\; \text{const.}$$

This equation is numerically solved by the Anderson–Björck method [8]. Note that this method cannot be applied in simulations where Ω_1 is a domain which is not connected.

Another challenge in the level set approach is given by keeping the level set function smooth. While evolving in simulation time, the signed distance property (2.1) of the level set function is typically lost. To recover this property, a re-initialization algorithm is employed, if the gradient of φ is larger or smaller than a given threshold β_φ. This topic is addressed in Chap. 5 in detail.

Algorithm 1: Simulating time dependent two-phase flows.

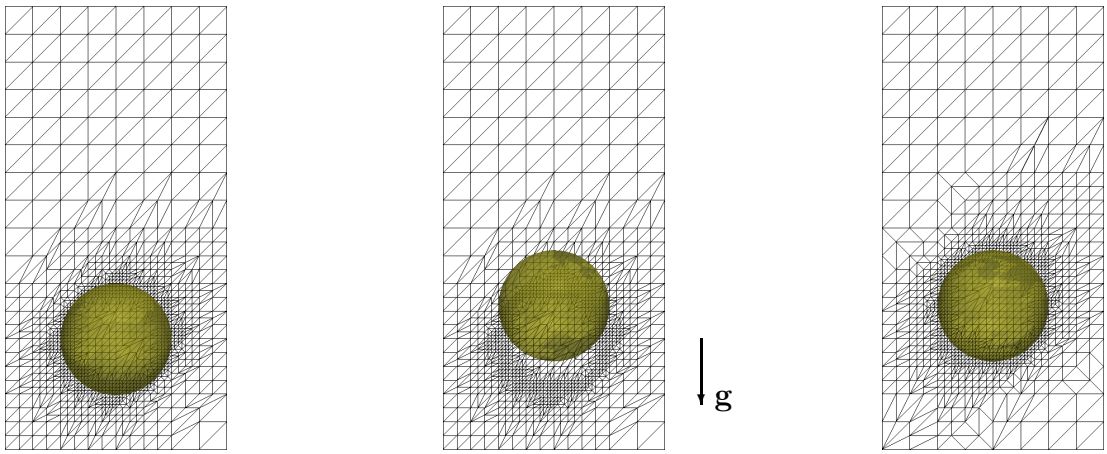
```
   // Initialization
1  M^0 ← Discretize (Ω)
2  k ← 0
3  τ_0 ← 0
4  (u^0, p^0, φ^0) ← Initialize (M^0, τ_0)
   // Evolve in simulation time
5  for τ_i ← τ_1, τ_2, ..., τ_N = τ_e do
6      (u^i, p^i, φ^i) ← SolveDiscreteTwoPhaseFlowProb (M^k, u^{i-1}, p^{i-1}, φ^{i-1})
7      if NeedsUpdate (M^k) then
8          M^{k+1} ← Update (M^k)
9          u^i ← Update (u^i)
10         p^i ← Update (p^i)
11         φ^i ← Update (φ^i)
12         k ← k + 1
       // Conserve mass
13     if V_1(φ^i) < V_0 − β_V or V_1(φ^i) > V_0 + β_V then
14         φ^i ← UpdateMass (φ^i)
       // Recover the distance property
15     if min ‖∇φ^i‖_2 < 1 − β_φ or max ‖∇φ^i‖_2 > 1 + β_φ then
16         φ^i ← ReInitialize (φ^i)
```

2.2.1 Hierarchy of Tetrahedral Grids

We consider a tetrahedral grid to represent the computational domain Ω. More precisely, a hierarchy $\mathcal{M}$ of tetrahedral grids is employed. This hierarchy facilitates the high resolution of Ω in domains of interest by discretizing these subdomains by a large number of small tetrahedra. And, vice versa, only a small number of large tetrahedra is employed to discretize subdomains of less interest. For instance, we resolve the domain in the vicinity of Γ_φ with high resolution and only use a small number of tetrahedra far from Γ_φ to save memory and computing time. In the relative literature, there exist two main classes of refinement algorithms. The first class relies on bisecting triangles or tetrahedra [115, 136]. The second class regularly refines the triangles or tetrahedra and uses a set of irregular refinement rules to make the mesh conforming [11,21,22,116]. In [123], a comparison of these methods is given when considering elliptic problems. A data structure for storing, refining and coarsening a three-dimensional tetrahedral grid is presented in detail in [25]. In this thesis, we only consider the second class. After demonstrating the necessity of the grid adaption, we outline the data structures for $\mathcal{M}$ that are used in the implementation of the refinement algorithm [22].

Consider an n-butanol drop in a cuboid shaped domain filled with water as depicted in Fig. 2.2. The initialization of the drop and a triangulation is given in Fig. 2.2(a). In this figure, the triangulation is refined in the vicinity of the interface of the drop. When evolving in simulation time, the n-butanol drop ascends because it has a lower density than the surrounding water. After a few time steps, the drop reaches a position as illustrated in Fig. 2.2(b). Here, we notice that the triangulation is not suitably refined because some small tetrahedra are located far from the interface and some large tetrahedra are located at Γ_φ. Therefore, the grid is adapted to the location of the drop. The new, modified triangulation is shown in Fig. 2.2(c). Next, we formalize the underlying tetrahedral hierarchy and outline the grid refinement algorithm.

Let $\mathcal{T}_0$ denote a coarse triangulation representing the computational domain Ω. A finer triangulation $\mathcal{T}_1$ is obtained by refining some tetrahedra of $\mathcal{T}_0$ by a red/green refinement algorithm. Within this algorithm a parent tetrahedron t on the coarse level is refined to at most eight child tetrahedra $\mathrm{Ch}(t)$ which are located on the finer level. The parent tetrahedron of a child tetrahedron t' is denoted by $\mathrm{Pa}(t')$. We distinguish between a regular (red) and an irregular (green) refinement of a tetrahedron. The former one is used to refine a tetrahedron that is marked for refinement by some algorithm. The latter one is applied to prevent so-called "hanging nodes." Multiple recursive refinements of tetrahedra lead to a hierarchy $\mathcal{M} = (\mathcal{T}_0, \ldots, \mathcal{T}_{k-1})$ of k triangulations, where the triangulation $\mathcal{T}_{k-1}$ represents the finest triangulation. Note that a child resulting from an irregular refinement of t is not allowed to be further refined. Instead, first, the parent tetrahedron t is regularly refined before refining its children $\mathrm{Ch}(t)$. Hence, the red/green



(a) Initialization. (b) Moved drop. (c) Grid adaption.

Figure 2.2: Serial simulation of a rising n-butanol drop.

algorithm ensures that all tetrahedra are not degenerated. Additionally, the refinement algorithm is capable of adaptively modifying a hierarchy of tetrahedral grids. The algorithm then transforms a hierarchy $\mathcal{M} = (\mathcal{T}_0, \ldots, \mathcal{T}_{k-1})$ of k levels to a different hierarchy $\mathcal{M}' = (\mathcal{T}'_0, \ldots, \mathcal{T}'_{k-1})$ of k levels where, in general $\mathcal{T}'_l \neq \mathcal{T}_l$ for $l = 1, \ldots, k-1$.

The hierarchy $\mathcal{M} = (\mathcal{T}_0, \ldots, \mathcal{T}_{k-1})$ of triangulations is called *admissible* if the following two conditions hold for all levels $l \in \{1, \ldots, k-1\}$:

(i) A tetrahedron $t \in T_l$ is either in $\mathcal{T}_{l-1}$ or it is obtained by a refinement of a tetrahedron $t' \in \mathcal{T}_{l-1}$.

(ii) If $t \in \mathcal{T}_{l-1} \cap T_l$ then $t \in \mathcal{T}_{l+1}, \ldots, T_{k-1}$. That is, if the tetrahedron t is not refined in level l then it stays unrefined in all finer levels $l+1, \ldots, k-1$.

Due to these two conditions on $\mathcal{M}$, we can assign each tetrahedron t a unique level

$$\text{level}(t) := \min \{m \mid t \in \mathcal{T}_m\}.$$

The set of all tetrahedra on level l is denoted by $\mathcal{G}_l$ and is called the *hierarchical surplus*, i.e.,

$$\mathcal{G}_l := \{t \mid \text{level}(t) = l\}.$$

The hierarchical decomposition $\mathcal{H}$ is defined by a k-tuple of these hierarchical surpluses, i.e., $\mathcal{H} := (\mathcal{G}_0, \ldots, \mathcal{G}_{k-1})$. Since each tetrahedron is located on exactly one hierarchical surplus this decomposition is used to efficiently store all tetrahedra. A detailed description of the data structures used to represent the hierarchy is given in [80]. However, both representations of the hierarchy are equivalent.

Given a hierarchy of triangulations $\mathcal{M}$, then the *hierarchical decomposition* $\mathcal{H}$ can be constructed by

$$\mathcal{G}_0 = \mathcal{T}_0 \quad \text{and} \quad \mathcal{G}_l = \mathcal{T}_l \setminus \mathcal{T}_{l-1} \quad \text{for } l = 1 \ldots, k-1.$$

Vice versa, given the decomposition $\mathcal{H}$, then the hierarchy $\mathcal{M}$ can be recursively generated by

$$\mathcal{T}_0 = \mathcal{G}_0 \quad \text{and} \quad \mathcal{T}_l = \mathcal{G}_l \cup \{t \in \mathcal{T}_{l-1} \mid \mathrm{Ch}(t) = \emptyset\}.$$

The hierarchy in Fig. 2.3 serves as an illustrating example throughout this thesis. This hierarchy consists of three triangulations $\mathcal{T}_0, \mathcal{T}_1$, and $\mathcal{T}_2$ in—for the sake of simplicity—two spatial dimensions. The triangulations $\mathcal{T}_{i+1}$ are obtained by refining $\mathcal{T}_i$, $i = 0, 1$. In this figure, we label each tetrahedron by t_j^l, where l denotes its level and j is a consecutive number in each level. The coarsest triangulation $\mathcal{T}_0$ is shown in Fig. 2.3(a) and consists of four tetrahedra. The triangulation $\mathcal{T}_1$ results from $\mathcal{T}_0$ by regularly refining the tetrahedra t_0^0 and t_1^0. To avoid hanging nodes, some tetrahedra are irregularly refined, e.g., in that example t_2^0 is refined to t_8^1 and t_9^1 in Fig. 2.3(b). The refinement continues in Fig. 2.3(c) where the triangulation $\mathcal{T}_2$ is shown. This triangulation is obtained by regularly refining t_1^1. The triangulations are given by

$$\mathcal{T}_0 = \{t_0^0, \ldots, t_3^0\}, \quad \mathcal{T}_1 = \{t_3^0, t_0^1, \ldots, t_9^1\}, \text{ and } \quad \mathcal{T}_2 = \{t_3^0, t_4^1, \ldots, t_9^1, t_0^2, \ldots, t_9^2\}$$

whereas the hierarchical surpluses are

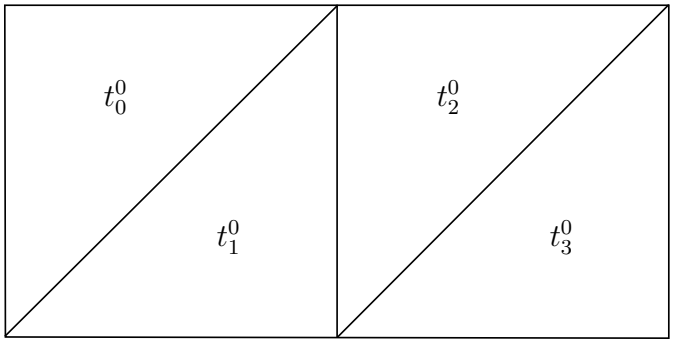
$$\mathcal{G}_0 = \{t_0^0, \ldots, t_3^0\}, \quad \mathcal{G}_1 = \{t_0^1, \ldots, t_9^1\}, \text{ and } \quad \mathcal{G}_2 = \{t_0^2, \ldots, t_9^2\}.$$

In each triangulation depicted in Fig. 2.3, only the tetrahedra located in the hierarchical surpluses $\mathcal{G}_0$, $\mathcal{G}_1$, and $\mathcal{G}_2$ are labeled.

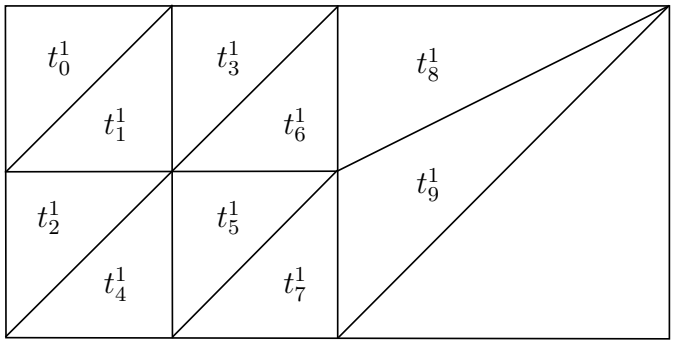
A hierarchy of triangulations can also be regarded as a forest. Here, each tetrahedron on level 0 spans a tree which consists of all its descendants, i.e., children, grandchildren, and so forth. Although no adjacencies are represented by this forest representation, it simplifies the consideration of the parent-child relation between tetrahedra. In Fig. 2.4, the forest representation is shown which corresponds to the hierarchy of triangulation presented in Fig. 2.3.

2.2.2 Solving the Discrete Two-Phase Flow Problem

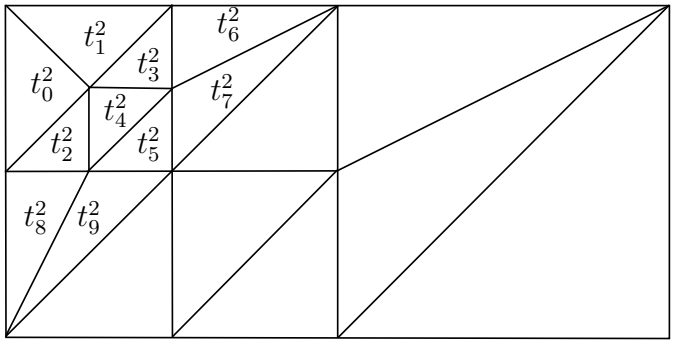
Next, we concentrate on simulating the two-phase flow problem (2.12)–(2.14) by means of the finite element method. This method is applied to these equations on the finest level $\mathcal{T}_{k-1}$ of the hierarchy of triangulations. Therefore, to simplify notations, we here only consider the triangulation $\mathcal{T} := \mathcal{T}_{k-1}$ rather than a hierarchy of triangulations.



(a) Triangulation $\mathcal{T}_0$.



(b) Triangulation $\mathcal{T}_1$.



(c) Triangulation $\mathcal{T}_2$.

Figure 2.3: Triangulation hierarchy $\mathcal{M} = (\mathcal{T}_0, \mathcal{T}_1, \mathcal{T}_2)$ of three triangulations. The tetrahedra in the hierarchical decomposition $\mathcal{H} = (\mathcal{G}_0, \mathcal{G}_1, \mathcal{G}_2)$ are labeled.

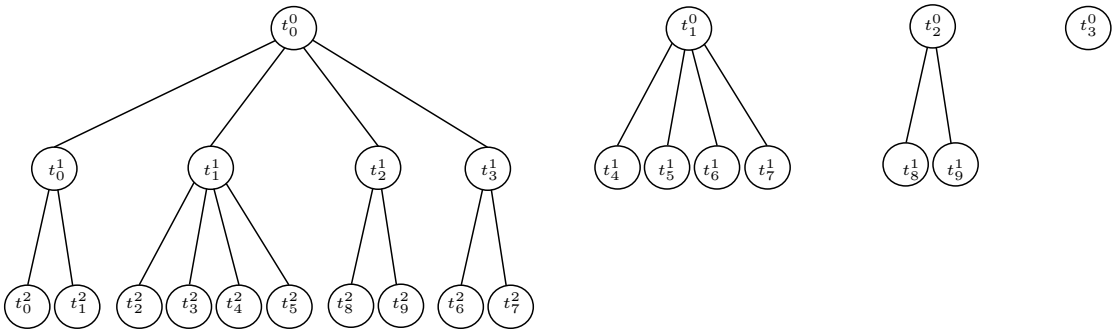


Figure 2.4: Forest representation of the triangulation hierarchy $\mathcal{M} = (\mathcal{T}_0, \mathcal{T}_1, \mathcal{T}_2)$ which is depicted in Fig. 2.3.

The time is discretized by the time steps $0 = \tau_0, \ldots, \tau_N = \tau_e$. Here, we consider an implicit time integration, e.g., a fractional theta scheme [134], a one step theta scheme, or an operator splitting scheme [29]. The spatial discretization is based on finite elements. To this end, the three unknown functions in (2.12)–(2.14), namely the velocity $\mathbf{u}$, the level set function φ and the pressure p, are discretized by the finite element functions $\mathbf{u}_h$, p_h, and φ_h. Here, the functions $\mathbf{u}_h$ and φ_h are represented by quadratic (P_2) finite element functions whereas p_h is given by so-called extended linear finite element functions (P_1^Γ) [18, 83, 124, 135]. Thus, degrees of freedom (DOF) of all finite element functions are located at vertices and, additionally for $\mathbf{u}_h$ and φ_h, at mid-vertices of edges of $\mathcal{T}$. Let $n_\mathbf{u}$, n_p, and n_φ denote the number of DOF to represent the three functions, respectively. Then, the corresponding vector representations of the finite element functions are given by $\overline{\mathbf{u}}^i \in \mathbb{R}^{n_\mathbf{u}}$, $\overline{p}^i \in \mathbb{R}^{n_p}$, and $\overline{\varphi}^i \in \mathbb{R}^{n_\varphi}$ at a given time τ_i. Note that the length of these vectors may vary in time due to adaptive grid modifications and the time-dependent P_1^Γ space. Applying the implicit time integration scheme and the finite element method to equations (2.12)–(2.14) leads to a discrete two-phase flow problem which consists of a system of non-linear equations

$$\left(\begin{array}{c|c} A(\overline{\varphi}^i) + N(\overline{\mathbf{u}}^i, \overline{\varphi}^i) & B^T(\overline{\varphi}^i) \\ \hline B(\overline{\varphi}^i) & 0 \end{array}\right) \cdot \begin{pmatrix} \overline{\mathbf{u}}^i \\ \overline{p}^i \end{pmatrix} = \begin{pmatrix} \mathbf{b}(\overline{\varphi}^i) \\ \mathbf{c} \end{pmatrix} \quad \text{and} \tag{2.15}$$

$$L(\overline{\mathbf{u}}^i) \cdot \overline{\varphi}^i = \mathbf{d}. \tag{2.16}$$

where $\mathbf{b} \in \mathbb{R}^{n_\mathbf{u}}$, $\mathbf{c} \in \mathbb{R}^{n_p}$ and $\mathbf{d} \in \mathbb{R}^{n_\varphi}$ are determined by physical properties of the system, the CSF term (2.6), the boundary conditions of equations (2.12)–(2.14), and the time integration scheme. The CSF term is discretized by an improved Laplace–Beltrami technique [84] which contributes to the vector $\mathbf{b}$. Note that this technique does not need second derivatives of φ to evaluate the curvature κ_Γ on Γ_φ as suggested by (2.7). In (2.15), the matrices $A \in \mathbb{R}^{n_\mathbf{u} \times n_\mathbf{u}}$, $N \in \mathbb{R}^{n_\mathbf{u} \times n_\mathbf{u}}$, and $B \in \mathbb{R}^{n_p \times n_\mathbf{u}}$ depend on the position of the interface and on the material

properties ϱ and μ, cf. (2.10)–(2.11). Thus, the values of the level set function play an important role when assembling these matrices. Note that the matrix B does not depend on $\overline{\varphi}^i$ if only considering linear finite elements (P_1) functions rather than P_1^Γ functions to represent the pressure. Discretizing and stabilizing the level set equation (2.14) by the streamline diffusion method (SDFEM) [66,97,140] yields the matrix $L \in \mathbb{R}^{n_\varphi \times n_\varphi}$.

The coupling of the equations (2.15) and (2.16) is handled by a fixed-point iteration. The non-linearity $(\mathbf{u}\cdot\nabla)\mathbf{u}$ in the Navier–Stokes equation (2.8) is present in the matrix N in (2.15). Numerically, this non-linearity is decoupled by an adaptive fixed-point defect correction scheme [85, 171]. This scheme includes a linearization of the Navier–Stokes equation which results in the Oseen equation

$$\left(\begin{array}{c|c} \tilde{A}(\overline{\varphi}^i) & B^T(\overline{\varphi}^i) \\ \hline B(\overline{\varphi}^i) & 0 \end{array}\right) \cdot \begin{pmatrix} \overline{\mathbf{u}}^i \\ \overline{p}^i \end{pmatrix} = \begin{pmatrix} \mathbf{b} \\ \mathbf{c} \end{pmatrix}. \tag{2.17}$$

This equation is either solved by an inexact Uzawa algorithm [132] or by applying an iterative Krylov subspace method [142] to solve the whole system of linear equations using special, blocked preconditioners for the matrix $\tilde{A}$ and the Schur complement $B\tilde{A}^{-1}B^T$.

This approach of solving the discrete two-phase flow problem (2.15)–(2.16) results in a variety of systems of linear equations. We approximate a solution of these systems by preconditioned Krylov subspace methods. In the current implementation of the presented algorithm the following Krylov subspace methods are applied: CG [94], GMRES [143], GCR [60], and QMR [73].

Overall, the solution of the discrete two-phase flow problem (2.15)–(2.16) includes a nested hierarchy of solvers. Figure 2.5 summarizes this section by depicting this hierarchy. In this figure, the boxes represent different parts and solvers of the simulation. If an arrow points from box A to box B, then the content of box A is used by the part represented by box B. For instance, the preconditioners are used withing the Krylov subspace methods. The inexact Uzawa box is printed dashed because it can be substituted by solving the Oseen equation (2.17) by a Krylov subspace method without exploiting its structure. Note that the solvers are strongly coupled because the quality of the respective approximated solution is adaptively determined when solving the system of non-linear equations. For instance, consider the decoupling the non-linearity in the Navier–Stokes equations by a fixed-point approach. Here, in the first iterations, the linear Oseen equation is approximated with a large tolerance whereas the quality of the approximation is successively increased with respect to the fixed-point iterations.

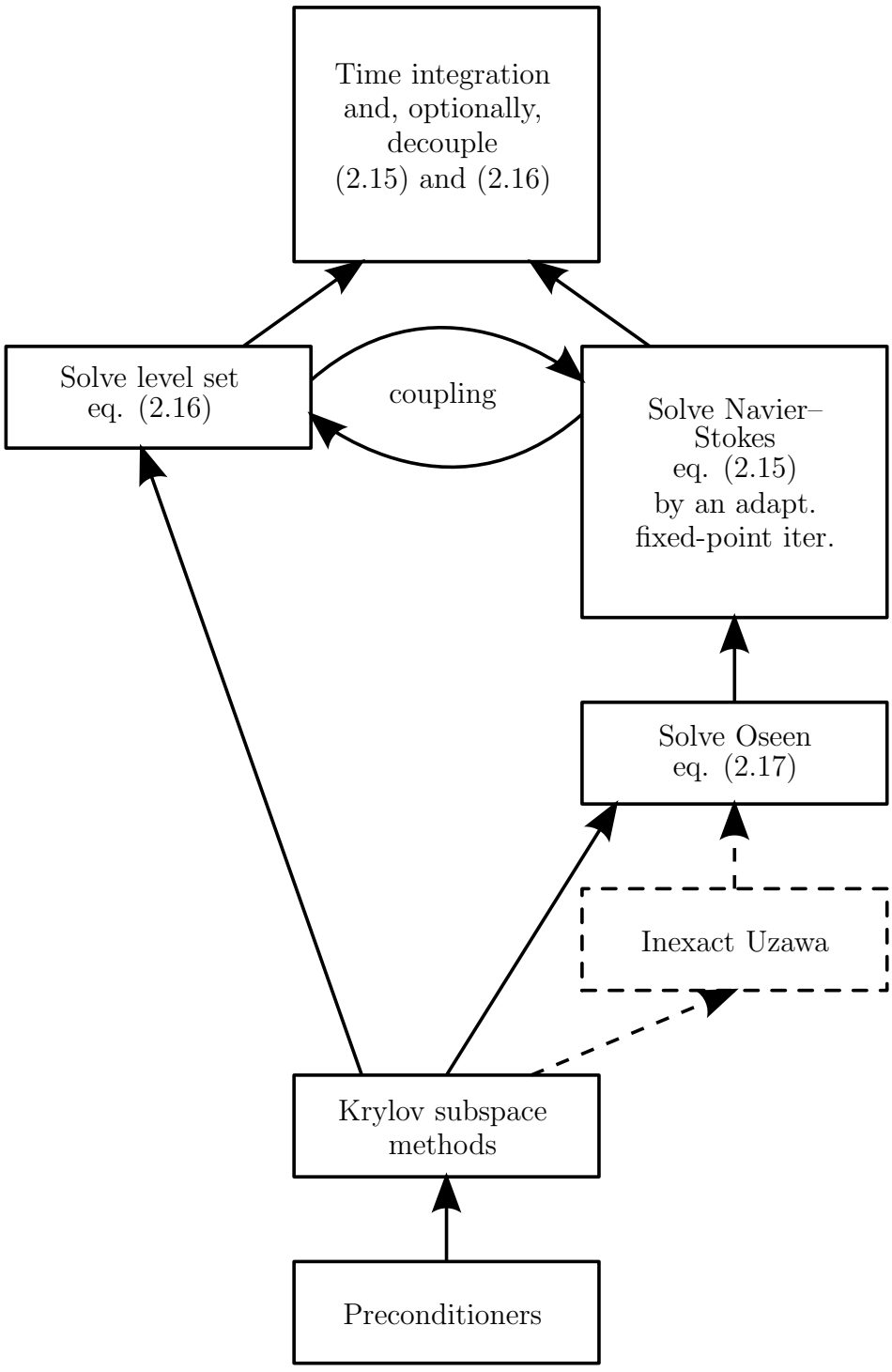


Figure 2.5: Nested solvers for the solution of the coupled discrete Navier–Stokes equations (2.15) and the level set equation (2.16).

3 Data Structures for Parallel Computing

Having described the governing equations and an algorithm for simulating two-phase flow problems by the level set approach, we now focus on a parallel strategy for simulating two-phase flow problems where our primary concern is a distributed-memory approach for the parallel algorithm. Nevertheless, most of the ideas are also applicable to shared-memory parallelization. As in [78], we distinguish between the two terms "processor" and "process." Here, a processor is a hardware unit that physically performs computations whereas the process is a logical unit that performs tasks. We follow a domain decomposition approach [39,58,59,156] to distribute the computational work among a set of processes. Therefore, the tetrahedra of the hierarchy of tetrahedral grids are distributed among the processes. In this chapter, we are concerned with defining suitable data structures for representing a distributed hierarchy of tetrahedral grids instead of finding a suitable decomposition of this hierarchy. The latter topic is addressed in Chap. 4. In the literature, various data structures and algorithms have been presented to perform adaptive mesh refinements in parallel. Recall from Sect. 2.2.1 that two different strategies for grid refinement exist, namely bisection and red-green refinements. Parallel algorithms based on the bisection approach can be found in [10, 100, 137]. However, we do not consider a parallel bisection algorithm here, because we aim to incorporate the ideas and algorithms from the previous chapter. A parallel red-green refinement algorithm for a hierarchy of triangulations has been introduced in [82]. The authors consider parallel data structures and analyze the parallel refinement algorithm with respect to important data partitioning properties such as data locality and storage overhead. Their work is based on previous work presented in [14, 15, 21].

Decomposing the set of tetrahedra implies that the numerical data, which are located at entities of the triangulations, are also spread among the processes. To perform the linear algebra operations in parallel, we define data structures for describing distributed numerical data on multiple processes. In Sect. 3.2, we define three different representations of these data where two of them are described in [80, 89, 121]. These two distributions pursue an overlapped representation, i.e., numerical data on "process boundaries" are stored by all bordering processes. However, these representations may result in an inaccurate determination of inner

products which may yield a negative squared 2-norm of vectors due to round-off errors. Since these negative squared norms are erroneous and unallowable in numerical simulations, we introduce in Sect. 3.2.2 a third, non-overlapped representation to overcome this difficulty. Defining algorithms to execute linear algebra operations, such as matrix-vector products, vector updates, and inner products on these distributions, leads to a parallel implementation of Krylov subspace methods. In this thesis, we do not focus on variants of Krylov subspace methods which are specially designed for parallel computing. In general, these parallel methods aim at reducing the synchronization of and communication among processes. Examples of methods reducing the number of synchronizations are the modified QMR method [31] or so-called s-step methods [46, 47]. A strategy to reduce the number of communications while allowing redundant computations is given in [52, 125]. We also do not focus on parallel preconditioning methods. However, we interface DROPS to the library HYPRE [62] which provides a variety of serial and parallel preconditioners.

This chapter is organized as follows. We first present an admissible distributed hierarchical decomposition in Sect. 3.1 to distribute a hierarchy of tetrahedral grids among processes. Afterwards, we concentrate on distributing the numerical data, such as matrices and vectors in Sect. 3.2. Based on the algorithm in Sect. 2.2.2, this distribution leads to a parallel algorithm that is capable of simulating two-phase flow problems. The scalability of the introduced data structures is demonstrated in Sect. 3.3 where this parallel approach is exemplified by the finite element simulation of a Poisson equation using up to 1 024 processes. Finally, we conclude this chapter in Sect. 3.4 by briefly discussing the implementation of the parallel data structures and ongoing work about a hybrid distributed-/shared-memory parallelization.

3.1 Distributing a Hierarchy of Tetrahedral Grids

To distribute the hierarchy $\mathcal{M} = (\mathcal{T}_0, \ldots, \mathcal{T}_{k-1})$ of tetrahedral grids among P processes, a subset of tetrahedra is assigned to each process. Therefore, let $\mathcal{G}_k^p$ denote the set of tetrahedra on level k that is assigned to a process $p \in \{1, \ldots, P\}$. Then, the union of all $\mathcal{G}_k^p$ of all processes results in the hierarchical surplus of level k, in formula

$$\mathcal{G}_k = \bigcup_{p=1}^{P} \mathcal{G}_k^p.$$

Moreover, the *distributed hierarchical decomposition* $\mathcal{H}^p$ of process p is defined as

$$\mathcal{H}^p := (\mathcal{G}_0^p, \ldots, \mathcal{G}_{k-1}^p).$$

It is crucial for the refining and coarsening algorithm that parent and child tetrahedra can easily access each other. Therefore, we require that all children are stored by the same process as the parent tetrahedron. If we do not allow copies of tetrahedra on more than one process, then this request will result in a coarse granularity. For instance, if a process p stores a tetrahedron t, then p also has to store all children of t as well as all descendants, i.e., grandchildren, great-grandchildren, and so on. To prevent a process storing a complete tetrahedron family, we introduce so-called *master* and *ghost* copies of tetrahedra. In principle, each tetrahedron is exactly represented by a master copy whereas a ghost copy is introduced for reproducing a parent tetrahedron whose master copy is located on another process. This classification has been adapted from [23, 24].

We formalize the terms ghost and master by introducing an *admissible distributed hierarchical decomposition* [82]. Therefore, let $\mathrm{Ma}_k^p \subset \mathcal{G}_k^p$ and $\mathrm{Gh}_k^p \subset \mathcal{G}_k^p$ respectively denote the master and ghost copies of tetrahedra on level k stored by process p. Then, a distributed hierarchical decomposition $(\mathcal{H}^1, \ldots, \mathcal{H}^P)$ is called admissible, if the following conditions are fulfilled.

(i) On each level l, the set $\{\mathrm{Ma}_l^1, \ldots \mathrm{Ma}_l^P\}$ forms a partition of the set $\mathcal{G}_l$, i.e.,

$$\bigcup_{p=1}^{P} \mathrm{Ma}_l^p = \mathcal{G}_l \quad \text{and} \quad \mathrm{Ma}_l^p \cap \mathrm{Ma}_l^q = \emptyset \quad \text{for all } p \neq q.$$

(ii) Let $t \in \mathcal{G}_l$ with $l \in \{0, \ldots, k-2\}$ be a refined tetrahedron and $\mathrm{Ch}(t) \subset \mathcal{G}_{l+1}$ its children. Then, all children of t are stored as a master copy on exactly one process p, in formula

$$\mathrm{Ch}(t) \subset \mathrm{Ma}_{l+1}^p \quad \text{for one and only one } p.$$

Furthermore, the parent tetrahedron t on process p is either stored

a) as a master copy and no ghost copy of t exists, i.e.,

$$t \in \mathrm{Ma}_l^p \;\Rightarrow\; t \notin \mathrm{Gh}_l^q \quad \text{for all } q \in \{1, \ldots, P\},$$

b) or as a ghost copy and no other ghost copy exist on another process, i.e.,

$$t \in \mathrm{Gh}_l^p \;\Rightarrow\; t \notin \mathrm{Gh}_l^q \quad \text{for all } q \neq p.$$

(iii) Let $t \in \mathrm{Gh}_l^p$ with $l \in \{0, \ldots, k-2\}$ be a ghost copy of a tetrahedron $t \in \mathcal{G}_l$ on process p. Then, t is refined and p stores all children of t, i.e.,

$$t \in \mathrm{Gh}_l^p \;\Rightarrow\; \emptyset \neq \mathrm{Ch}(t) \subset \mathrm{Ma}_{l+1}^p\,.$$

The admissible distributed hierarchical decomposition aims at describing an efficient data structure to store a hierarchy of triangulations by multiple processes. In particular, due to (i), each tetrahedron of the hierarchical decomposition $\mathcal{H}$ is stored exactly once as a master copy and, due to (iii) ghost copies are only introduced whenever they are needed to represent parents. Moreover, due to (ii), this data structure allows the reuse of the serial refinement algorithm with only a few modifications because parents and all their children are stored on the same process. A detailed discussion of this data structure and the modifications of the serial refinement algorithm has been presented and analyzed in [82]. The authors have additionally proven that their parallel refinement algorithm transforms an admissible distributed hierarchical decomposition into a new hierarchical decomposition which is also admissible. Note that not only tetrahedra are classified as master and ghost copies but also their subsimplices, i.e., faces, edges, and vertices of tetrahedra. For a detailed description, we refer the reader to [79].

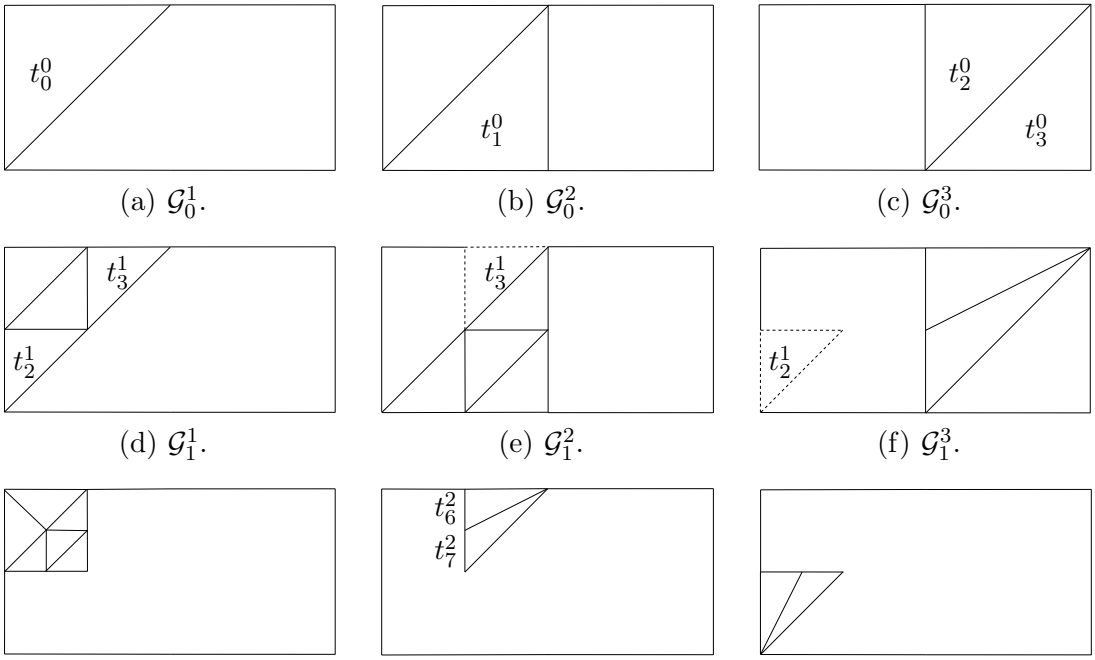
Distributing the hierarchical decompositions also leads to distributed triangulations among processes. Therefore, we do not consider the ghost copies when referring to a triangulation $\mathcal{T}_l^p$ on level l stored by process p, i.e.,

$$\mathcal{T}_l^p := \mathcal{T}_l \cap \mathrm{Ma}_l^p \,.$$

We comment that the local hierarchy $(\mathcal{T}_0^p, \ldots, \mathcal{T}_{k-1}^p)$ of one process p does not define an admissible hierarchy as defined in Sect. 2.2.1. Therefore, recall that each tetrahedron in $\mathcal{T}_l$ either exists in $\mathcal{T}_l$ or is a child of a parent tetrahedron in $\mathcal{T}_{l-1}$ in a sequential admissible hierarchy. However, in a local hierarchy of a process p, a tetrahedron $t \in \mathcal{T}_l^p$ may first occur on level $0 < l < k$ without existing in $\mathcal{T}_{l-1}^p$ or having a parent element in $\mathcal{T}_{l-1}^p$ because the parent tetrahedron is stored as a ghost copy on p.

As an example of a decomposition of tetrahedra, we reconsider the hierarchy of triangulations depicted in Fig. 2.3 in the previous chapter. An admissible distributed hierarchical decomposition among three processes is shown in Fig. 3.1. In this figure, the three left figures illustrate the tetrahedra assigned to process 1, the three figures in the middle those of process 2, and the figures on the right those of process 3. Note that overlaying the figures in one row results in a hierarchical surplus, e.g., Fig. 3.1(a)–(c) results in $\mathcal{G}_0$. The ghost copies of tetrahedra are depicted by dashed lines in these figures. For instance, the tetrahedron $t_3^1 \in \mathcal{G}_1^2$ in Fig. 3.1(e) is stored by process 2 as a ghost copy whereas its master copy $t_3^1 \in \mathcal{G}_1^1$ is located on process 1, cf. Fig. 3.1(d). Since t_3^1 is assigned—with all its siblings $\{t_0^1, t_1^1, t_2^1\}$—to process 1 and its children $\mathrm{Ch}(t_3^1) = \{t_6^2, t_7^2\}$ are located at process 2, it is indispensable to store a ghost copy t_3^1 on process 2.

The corresponding forest representation is given in Fig. 3.2, where, again, the ghosts are illustrated by dashed lines. This figure clarifies that all children of a



(a) $\mathcal{G}_0^1$. (b) $\mathcal{G}_0^2$. (c) $\mathcal{G}_0^3$.

(d) $\mathcal{G}_1^1$. (e) $\mathcal{G}_1^2$. (f) $\mathcal{G}_1^3$.

(g) $\mathcal{G}_2^1$. (h) $\mathcal{G}_2^2$. (i) $\mathcal{G}_2^3$.

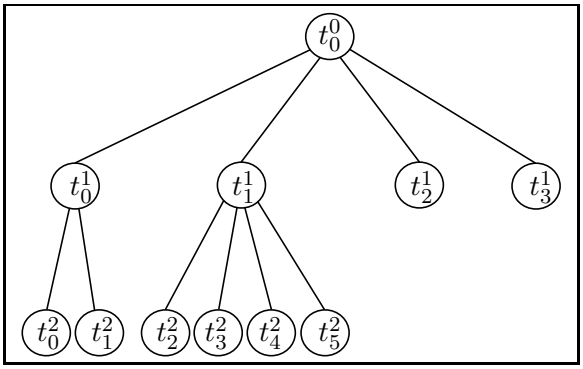
Figure 3.1: Admissible distributed hierarchical decomposition among three processes of the triangulation hierarchy $\mathcal{M} = (\mathcal{T}_0, \mathcal{T}_1, \mathcal{T}_2)$ which is depicted in Fig. 2.3.

tetrahedron are stored as master copies on one process. This figure also demonstrates that a local hierarchy of triangulations is not admissible as defined in Sect. 2.2.1. For instance, the tetrahedron $t_6^2 \in \mathcal{T}_2^2$ in Fig. 3.2(b) is a child of t_3^1 which is not located in $\mathcal{T}_1^2$ (but in the triangulation $\mathcal{T}_1^1$ on process 1).

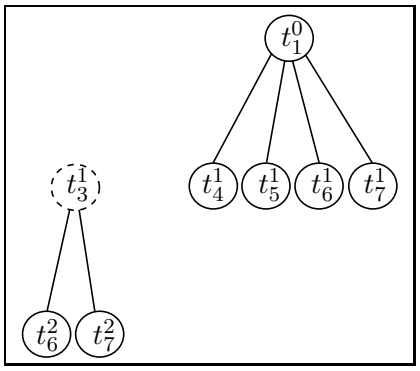
In this thesis, the mathematical model of a two-phase flow problem is numerically solved on the finest triangulation, in this example on level 2. The triangulation $\mathcal{T}_2$ is decomposed among the three processes as follows

$$\mathcal{T}_2^1 = \{t_0^2, t_1^2, t_2^2, t_3^2, t_4^2, t_5^2\}, \quad \mathcal{T}_2^2 = \{t_4^1, t_5^1, t_6^1, t_7^1, t_6^2, t_7^2\}, \quad \text{and}$$
$$\mathcal{T}_2^3 = \{t_3^0, t_8^1, t_9^1, t_8^2, t_9^2\}.$$

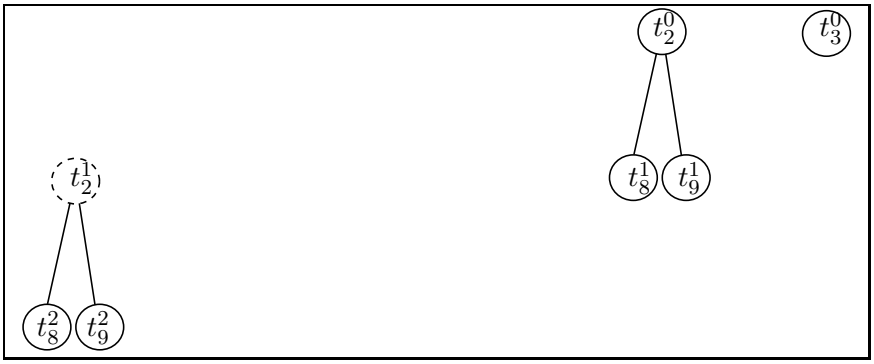
Hence, the triangulation $\mathcal{T}_2$ is almost evenly distributed among the three processes if considering the number of tetrahedra on the finest triangulation per process. More sophisticated models to determine a decomposition of the tetrahedra will be presented in Chap. 4 in detail.



(a) Process 1.



(b) Process 2.



(c) Process 3.

Figure 3.2: Distributed forest representation of the hierarchy $\mathcal{M} = (\mathcal{T}_0, \mathcal{T}_1, \mathcal{T}_2)$ of triangulations which is depicted in Fig. 2.3.

3.2 Distributed Numerical Data

Next, we focus on the distribution of numerical data. Recall from the previous chapter that we are concerned with the finite element method, where the solution of the underlying PDEs are discretized by finite element functions. Therefore, degrees of freedom (DOFs) are introduced on entities of a given triangulation $\mathcal{T}$. We consider quadratic, linear and extended linear finite element functions in this thesis. Thus, DOFs are located on vertices and edges of $\mathcal{T}$. The set of DOFs is distributed among the processes because a decomposition $(\mathcal{T}^1, \ldots, \mathcal{T}^P)$ of $\mathcal{T}$ also results in a decomposition of the vertices and edges of $\mathcal{T}$. We describe in Sect. 3.2.1 three formats representing finite element functions on the tetrahedral decomposition $(\mathcal{T}^1, \ldots, \mathcal{T}^P)$ and how to obtain these formats. Afterwards, in Sect. 3.2.2, we employ these data formats to present an approach for solving systems of linear equations in parallel by Krylov subspace methods.

3.2.1 Parallel Representations of Finite Element Functions

First, consider a serial representation of a finite element function of a given triangulation $\mathcal{T}$ on a fixed level. Let $\mathcal{I} = \{1, \ldots, n\}$ denote an ordering of n DOFs used to represent a finite element function on that triangulation. For instance, if a scalar, linear (P_1) finite element function is investigated then $\mathcal{I}$ is given by an ordering of the triangulation vertices. Furthermore, let $\mathbf{x} \in \mathbb{R}^n$ be the vector which contains the DOFs of a finite element function when using the ordering $\mathcal{I}$.

Now, consider a distributed triangulation $(\mathcal{T}^1, \ldots, \mathcal{T}^P)$ among P processes and let $\mathcal{I}^p = \{1, \ldots, n^p\}$ denote an ordering of the DOFs located at a process p. We introduce three representations of the vector $\mathbf{x}$ by P processes in the following. The first one $\overline{\mathbf{x}}$ is called the *accumulated representation*, the second one $\widetilde{\mathbf{x}}$ the *distributed representation*, and the third one $\widehat{\mathbf{x}}$ the *exclusive representation*. The first two formats have been introduced in [121] where these representations are referred to as "Type II-" and "Type I-distribution." These formats are also used in [89] and [80]. The third representation $\widehat{\mathbf{x}}$ of $\mathbf{x}$ is primarily employed when determining inner products. We next describe all three representations..

Accumulated Representation In this representation, each process p stores a vector $\overline{\mathbf{x}}^p \in \mathbb{R}^{n^p}$ containing the global values of all DOFs located on p. That is, if a process p stores a DOF with the number $i \in \mathcal{I}^p$ then the global value of this DOF is given by $(\overline{\mathbf{x}}^p)_i$. In the remainder, the representation

$$\overline{\mathbf{x}} = (\overline{\mathbf{x}}^1, \ldots, \overline{\mathbf{x}}^P) \in \mathbb{R}^{n^1} \times \cdots \times \mathbb{R}^{n^P} \tag{3.1}$$

of a vector $\mathbf{x} \in \mathbb{R}^n$ is called *accumulated representation of* $\mathbf{x}$.

From a mathematical point of view, transforming the global vector $\mathbf{x}$ to a local vector $\overline{\mathbf{x}}^p$ can be expressed by a linear mapping. To this end, we define the *coincidence matrix* $I^p \in \{0,1\}^{n^p \times n}$ whose entries are given by

$$(I^p)_{i,j} = \begin{cases} 1, & \text{if a DOF with global number } j \in \mathcal{I} \text{ exists on} \\ & \text{processor } p \text{ with local number } i \in \mathcal{I}^p, \\ 0, & \text{else} \end{cases} . \tag{3.2}$$

This matrix is also called the Boolean connectivity matrix in [121]. The accumulated representation of $\overline{\mathbf{x}}$ in (3.1) becomes

$$\overline{\mathbf{x}} = (I^1 \cdot \mathbf{x}, \ldots, I^P \cdot \mathbf{x}). \tag{3.3}$$

Since some vertices and edges of the triangulation are stored overlapped, i.e., by multiple processes, some DOFs are located at multiple processes. Hence, the accumulated number of DOFs is larger than the global number of DOFs, i.e.,

$$\overline{n} := \sum_{p=1}^{P} n^p \geq n.$$

Figure 3.3 illustrates this situation for two tetrahedra and two processes p_1 and p_2. The first tetrahedron t_1 is stored by a process p_1 and the second tetrahedron t_2 by a process p_2, indicated by gray-shading. In this example, four global DOFs exist, namely $x_1, \ldots, x_4$. These DOFs are distributed among the two processes, where each process p determines an ordering $\mathcal{I}^p = \{1, \ldots, 3\}$ of its three DOFs whose values are given by x_1^p, x_2^p, and x_3^p. In this example, both processes assign a number for the global DOF x_1 which is locally stored as x_3^1 and x_1^2 on p_1 and p_2, respectively. Thus, the two entries $(I^{p_1})_{3,1}$ and $(I^{p_2})_{1,1}$ are nonzero. The coincidence matrices for p_1 and p_2 read as

$$I^{p_1} = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \quad \text{and} \quad I^{p_2} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}.$$

Distributed Representation Discretizing a PDE by the finite element method is commonly performed by a loop over all tetrahedra of a given triangulation level. This loop assembles a system of linear equations which discretely represents the PDE. That is, in general, the loop sets up a matrix and a vector. Here, each tetrahedron contributes data to the matrix and the vector. Since the tetrahedra of the triangulation are distributed among processes, each process assembles parts of the resulting system of linear equations. For instance, reconsider the example in

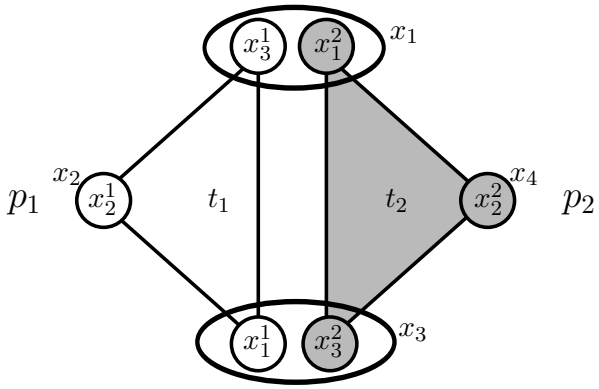


Figure 3.3: Distributed DOFs among two processes.

Fig. 3.3 where the two adjacent tetrahedra t_1 and t_2 contribute partial information to the DOF x_1 while assembling a system of linear equations. The global values of DOFs can be obtained by summing up the corresponding local values of the DOFs. More precisely, a representation $\widetilde{\mathbf{x}} = (\widetilde{\mathbf{x}}^1, \ldots, \widetilde{\mathbf{x}}^P) \in \mathbb{R}^{n^1} \times \cdots \times \mathbb{R}^{n^P}$ of a vector $\mathbf{x} \in \mathbb{R}^n$ is called *distributed*, if the following equation holds

$$\mathbf{x} = \sum_{p=1}^{P} (I^p)^T \, \widetilde{\mathbf{x}}^p. \tag{3.4}$$

Note that transforming a distributed vector into an accumulated vector involves so-called neighbor communication, i.e., all processes which share at least one DOF communicate their values among each other. Although equation (3.4) contains a sum over all processes, no global communication and synchronization, e.g., a reduce-type operation, is needed. Moreover, the neighbor communication can be overlapped by computations when determining linear algebra operations, as we will see later in Sect. 3.2.2.

A similar result holds when considering matrices. Let $\widetilde{A}^p \in \mathbb{R}^{n^p \times n^p}$ denote the local matrix of process p which is assembled by p when iterating over its tetrahedra $\mathcal{T}^p$. Then, this matrix is stored in a distributed fashion and the global matrix $A \in \mathbb{R}^{n \times n}$ can be obtained by summing up the distributed matrix entries. Employing the coincidence matrix (3.2) yields the formula

$$A = \sum_{p=1}^{P} (I^p)^T \cdot \widetilde{A}^p \cdot I^p. \tag{3.5}$$

A slightly modified version of this equation also holds for rectangular matrices where different numberings are used to number the rows and columns of the matrix. For instance, the rows of the matrix B in (2.15) are numbered by the pressure

DOFs and the columns of B by the velocity DOFs. Here, the local matrices $\widetilde{B}^p$ stored by processes $p = 1, \ldots, P$ are transformed to the global matrix B by using the coincidence matrices corresponding to the pressure DOFs and the velocity DOFs in (3.5), respectively.

Exclusive Representation Finally, the third representation aims at assigning each DOF to exactly one process. This can be seen as a conversion from an accumulated representation to a distributed one. Moreover, the exclusive representation is advantageous to determine inner products as we will see later on. To this end, we consider a partitioning $\{\widehat{\mathcal{I}}^1, \ldots, \widehat{\mathcal{I}}^P\}$ of the global numbering $\mathcal{I}$, i.e.,

$$\bigcup_{p=1}^{P} \widehat{\mathcal{I}}^p = \mathcal{I} \quad \text{with} \quad \widehat{\mathcal{I}}^p \cap \widehat{\mathcal{I}}^q = \emptyset,\ p \neq q \quad \text{and}$$
$$\widehat{\mathcal{I}}^p \subset \mathcal{I}^p. \tag{3.6}$$

Note that (3.6) assures that a DOF is exclusively assigned to a process which stores a copy of the DOF in the accumulated or distributed representation. We define the number of exclusive DOF of process p by $\widehat{n}^p = |\widehat{\mathcal{I}}^p|$. Then, the global number of DOFs is given by $n = \sum_{n=1}^{P} \widehat{n}^p$. Moreover, let $\widehat{I}^p \in \mathbb{R}^{\widehat{n}^p \times n}$ denote the *exclusive coincidence matrix* of process p, whose entries are given by

$$(\widehat{I}^p)_{i,j} = \begin{cases} 1, & \text{if a DOF with global number } j \in \mathcal{I} \text{ exclusively exists} \\ & \text{on processor } p \text{ with local number } i \in \widehat{\mathcal{I}}^p, \\ 0, & \text{else} \end{cases} .$$

The representation $\widehat{\mathbf{x}} = (\widehat{\mathbf{x}}^1, \ldots, \widehat{\mathbf{x}}^P) \in \mathbb{R}^{\widehat{n}^1} \times \cdots \times \mathbb{R}^{\widehat{n}^P}$ is called *exclusive* if

$$\widehat{\mathbf{x}}^p = \widehat{I}^p \, \mathbf{x} \tag{3.7}$$

holds for all $p = 1, \ldots, P$. In the style of (3.4), this formula equals

$$\mathbf{x} = \sum_{p=1}^{P} \left(\widehat{I}^p\right)^T \widehat{\mathbf{x}}^p. \tag{3.8}$$

Note that the exclusive representation can be easily determined from the accumulated representation without communication by restricting the accumulated DOF to the exclusive ones.

3.2.2 Parallel Linear Algebra

We next focus on executing linear algebra operations in parallel. Recall that the discrete formulation of the two-phase flow problem (2.15)–(2.16) consists of

a system of non-linear equations for each time step. When following the method presented in the previous chapter, that system is solved by a hierarchy of various solvers, see Fig. 2.5, which—besides operations on scalars—primarily employ the following linear algebra operations:

(i) matrix-vector products, e.g., $\mathbf{y} \leftarrow A \cdot \mathbf{x}$,

(ii) vector updates, e.g., $\mathbf{z} \leftarrow \alpha\mathbf{x} + \mathbf{y}$, and

(iii) inner products, e.g., $\alpha \leftarrow \mathbf{x}^T \cdot \mathbf{y}$

for $\alpha \in \mathbb{R}$, $\mathbf{x}, \mathbf{y}, \mathbf{z} \in \mathbb{R}^n$, and $A \in \mathbb{R}^{n \times n}$. We describe methods in the following to execute these operations in parallel when using the three representations of vectors and the distributed matrix representation presented above.

Matrix-Vector Product Assume that a matrix A is given in the distributed format, cf. (3.5), and the vector $\mathbf{x}$ is available by the accumulated representation $\overline{\mathbf{x}}$. Then, the following chain of equations holds:

$$\begin{aligned} A \cdot \mathbf{x} &\overset{(3.5)}{=} \left(\sum_{p=1}^{P} (I^p)^T \cdot \widetilde{A}^p \cdot I^p \right) \cdot \mathbf{x} \;=\; \sum_{p=1}^{P} (I^p)^T \cdot \widetilde{A}^p \cdot (I^p \cdot \mathbf{x}) \\ &\overset{(3.3)}{=} \sum_{p=1}^{P} (I^p)^T \cdot \underbrace{\widetilde{A}^p \cdot \overline{\mathbf{x}}^p}_{:=\widetilde{\mathbf{y}}^p} \;=\; \sum_{p=1}^{P} (I^p)^T \cdot \widetilde{\mathbf{y}}^p \overset{(3.4)}{=} \mathbf{y}. \end{aligned}$$

Thus, if a process p multiplies an accumulated vector with its local part of a distributed matrix then the result is a distributed vector. No communication is necessary for this matrix-vector product. In contrast, if the product of a distributed matrix and a distributed vector is requested, then the vector has to be first transformed into an accumulated representation which includes neighbor communication.

Vector updates Computing $\mathbf{z} \leftarrow \alpha\mathbf{x} + \mathbf{y}$ does not involve any communication if the vectors $\mathbf{x}$ and $\mathbf{y}$ have the same representation. Hereby, the result is represented in the same format. Since for all $p = 1, \ldots, P$ the equation

$$I^p \left(\alpha\mathbf{x} + \mathbf{y}\right) \;=\; \alpha I^p\mathbf{x} + I^p\mathbf{y} \;=\; \underbrace{\alpha\overline{\mathbf{x}}^p + \overline{\mathbf{y}}^p}_{:=\overline{\mathbf{z}}^p} \;=\; \overline{\mathbf{z}}^p$$

holds, the result of a vector update of accumulated vectors results in an accumulated vector $\overline{\mathbf{z}}$. For the distributed representation, we find

$$\begin{aligned}\alpha\mathbf{x}+\mathbf{y} &= \alpha\left(\sum_{p=1}^{P}(I^p)^T\,\widetilde{\mathbf{x}}^p\right)+\left(\sum_{p=1}^{P}(I^p)^T\,\widetilde{\mathbf{y}}^p\right)\\ &= \sum_{p=1}^{P}(I^p)^T\underbrace{\left(\alpha\widetilde{\mathbf{x}}^p+\widetilde{\mathbf{y}}^p\right)}_{:=\widetilde{\mathbf{z}}^p} = \sum_{p=1}^{P}(I^p)^T\,\widetilde{\mathbf{z}}^p = \mathbf{z}.\end{aligned}$$

When replacing I^p by $\widehat{I}^p$, these equations prove that updating two exclusive vectors also results in an exclusive representation of $\mathbf{z}$.

Inner products We present two approaches to determine the inner product of two vectors $\mathbf{x}$ and $\mathbf{y} \in \mathbb{R}^n$. The first algorithm is also described in [80, 121] and relies on the following chain of equations

$$\mathbf{x}^T\cdot\mathbf{y} \overset{(3.4)}{=} \mathbf{x}^T\cdot\sum_{p=1}^{P}(I^p)^T\,\widetilde{\mathbf{y}}^p = \sum_{p=1}^{P}\mathbf{x}^T\cdot(I^p)^T\,\widetilde{\mathbf{y}}^p \tag{3.9}$$

$$= \sum_{p=1}^{P}(I^p\mathbf{x})^T\cdot\widetilde{\mathbf{y}}^p \overset{(3.3)}{=} \sum_{p=1}^{P}(\overline{\mathbf{x}}^p)^T\cdot\widetilde{\mathbf{y}}^p. \tag{3.10}$$

Thus, the inner product is computed by, first, locally multiplying the accumulated representation $(\overline{\mathbf{x}}^p)^T$ and the distributed representation $\widetilde{\mathbf{y}}^p$ by each process and, afterwards, summing up the results among all processes. This summation includes a synchronization of all processes which becomes very expensive when considering a large number of processes.

If both vectors $\mathbf{x}$ and $\mathbf{y}$ are given in the distributed representation, a preceding step is necessary to transform one vector into the accumulated form yielding in neighbor communication. However, this communication can be overlapped by computations on only locally stored vector entries such as x_2^1 in Fig. 3.3.

However, this algorithm may result in negative squared 2-norms of vectors. For instance, consider the following situation, where $P = 4$ processes determine the squared norm $\|\mathbf{x}\|_2^2 = \mathbf{x}^T\cdot\mathbf{x}$ of a one-dimensional vector $\mathbf{x} \in \mathbb{R}^1$. Furthermore, assume that each process computes with two significant bits. The values of the distributed representation are given in the second column of Table 3.1 for each process $p = 1, \ldots, 4$. First, an accumulated representation of $\mathbf{x}$ is required to compute $\mathbf{x}^T\cdot\mathbf{x}$. Since the only DOF x_1 is distributed among four processes, each process sends its distributed entry to all other processes. Afterwards, each process p sums up the distributed entries—in this example—in the following order:

$$\widetilde{x}_1^p + \widetilde{x}_1^{p+1 \bmod P} + \widetilde{x}_1^{p+2 \bmod P} + \widetilde{x}_1^{p+3 \bmod P}. \tag{3.11}$$

p	$\widetilde{\mathbf{x}}^p$	$\overline{\mathbf{x}}^p$	$(\overline{\mathbf{x}}^p)^T \cdot \widetilde{\mathbf{x}}^p$
1	0.95	$0.95 - 0.95 - 0.001 + 0.001 = 0$	0
2	-0.95	$-0.95 - 0.001 + 0.001 + 0.95 = 0$	0
3	-0.001	$-0.001 + 0.001 + 0.95 - 0.95 = 0$	0
4	0.001	$0.001 + 0.95 - 0.95 - 0.001 \approx -0.001$	$-1 \cdot 10^{-6}$

Table 3.1: Computing $\|\mathbf{x}\|_2$ by $P = 4$ processes.

This results in an accumulated representation of $\overline{\mathbf{x}}^p$. In the third column of Table 3.1, the values of this representation are shown for each process when determining the sum in (3.11) with two significant bits. Then, the product $(\overline{\mathbf{x}}^p)^T \cdot \widetilde{\mathbf{x}}^p$ is determined by each process leading to the values in the fourth column of Table 3.1. Finally, the squared norm $\|\mathbf{x}\|_2^2$ is determined by the sum of the local products, i.e., $(-1) \cdot 10^{-6}$. Hence, in this example, the algorithm determines a negative squared norm of a real-valued vector. This is numerically unacceptable.

To overcome this issue, we next present an algorithm which makes use of the exclusive representation. Therefore, consider the following chain of equations

$$\mathbf{x}^T \cdot \mathbf{y} \overset{(3.8)}{=} \mathbf{x}^T \cdot \sum_{p=1}^{P} \left(\widehat{I}^p\right)^T \widehat{\mathbf{y}}^p = \sum_{p=1}^{P} \mathbf{x}^T \cdot \left(\widehat{I}^p\right)^T \widehat{\mathbf{y}}^p \tag{3.12}$$

$$= \sum_{p=1}^{P} \left(\widehat{I}^p \mathbf{x}\right)^T \cdot \widehat{\mathbf{y}}^p \overset{(3.7)}{=} \sum_{p=1}^{P} (\widehat{\mathbf{x}}^p)^T \cdot \widehat{\mathbf{y}}^p. \tag{3.13}$$

In contrast to the algorithm of (3.9)–(3.10), the local inner products are determined by using the exclusive representation of the vectors. This algorithm results in positive squared norms of a vector $\mathbf{x}$ since each summand of $(\widehat{\mathbf{x}}^p)^T \cdot \widehat{\mathbf{x}}^p$ in (3.13) is greater or equal zero. To even increase the accuracy of determining inner products, especially for large vectors, we use the Kahan's summation algorithm [102] for building the local sums. Note that both vectors are assumed to have an exclusive representation which might imply an additional communication step if at least one vector is given in distributed format. However, this communication can be overlapped by computation in the same way as for the algorithm above.

We recall the example from Table 3.1 to illustrate the algorithm based on the exclusive representation (3.12)–(3.13). If the only entry of $\mathbf{x}$ is exclusively assigned to process 4, the squared 2-norm $\|\mathbf{x}\|_2^2$ is determined as $(-0.001)^2 = 1 \cdot 10^{-6}$. If another process exclusively owns the entry, the algorithm correctly computes 0 as the norm of $\mathbf{x}$.

3.3 Numerical Results

Having described the parallel data structures and how to execute linear algebra operations, we now present numerical results showing that the decomposition of the triangulation and the parallel representation of numerical data is well suited for solving PDEs in parallel. To this end, consider the following Poisson equation

$$\begin{aligned} -\Delta u &= 128(f(y,z)+f(x,z)+f(x,y)) && \text{in } \Omega^C := [0,1]^3 \text{ and} \\ u &= 1 && \text{on } \partial[0,1]^3 \end{aligned}, \tag{3.14}$$

where $f(x_1, x_2) := x_1 x_2 (1 - x_1)(1 - x_2)$. We do not consider a simulation of a two-phase flow problem because the behavior of the nested solvers is highly interdependent. When using different numbers of processes, sums are evaluated in different orderings which causes different results in floating point arithmetic. Additionally, a different partitioning of the matrices and vectors commonly results in a modified convergence behavior of numerical algorithms [49]. Thus, using different numbers of processes may result in different iteration numbers of the solvers presented in Sect. 2.2. In Chap. 6, we contemplate this issue when simulating two-phase flow problems. Although the Poisson equation (3.14) is numerically easier to treat than the coupled, non-linear two-phase flow equations (2.12)–(2.14), from a computational point of view, its simulation consists of similar parts which dominates the compute time. These parts are the grid generation, the assembly of systems of linear equations, and the solution of these systems. Therefore, we here investigate those three parts.

The numerical results of this and the following chapters are gathered by the software DROPS [81, 117] on two different clusters located at the Center for Computing and Communication of RWTH Aachen University. The first one is based on quad-core Harpertown (E5450) processors whereas the second one is assembled by quad-core Nehalem (X5570) processors. In both clusters, two processors are located on one node and, thus, up to eight cores can access the shared memory. Furthermore, the nodes are connected by an InfiniBand network. Table 3.2 presents the characteristics of both clusters. In this section, we only consider the Harpertown cluster. However, the results gathered on the Nehalem cluster show essentially the same behavior.

We inspect results where (3.14) is discretized by quadratic (P_2) finite element functions on three different tetrahedral grids which vary in their number of tetrahedra. A load-balancing algorithm is employed aiming at minimizing faces between processes and evenly distributing the number of tetrahedra of the finest level among the processes. We place eight MPI processes per compute node. Hence, at least eight processes are employed. The size of the grid

$$|\mathcal{H}| = \sum_{l=0}^{k-1} \sum_{p=1}^{P} |\mathcal{G}_k^p|$$

	Harpertown	Nehalem
processor	Intel's E5450	Intel's X5570
cores per processor	4	4
clock rate [Ghz]	3.0	2.93
processors per node	2	2
number of nodes	50	192
shared memory per node [GB]	16	24
network	DDR InfiniBand	QDR InfiniBand

Table 3.2: Characteristics of the two clusters used to gather the numerical results.

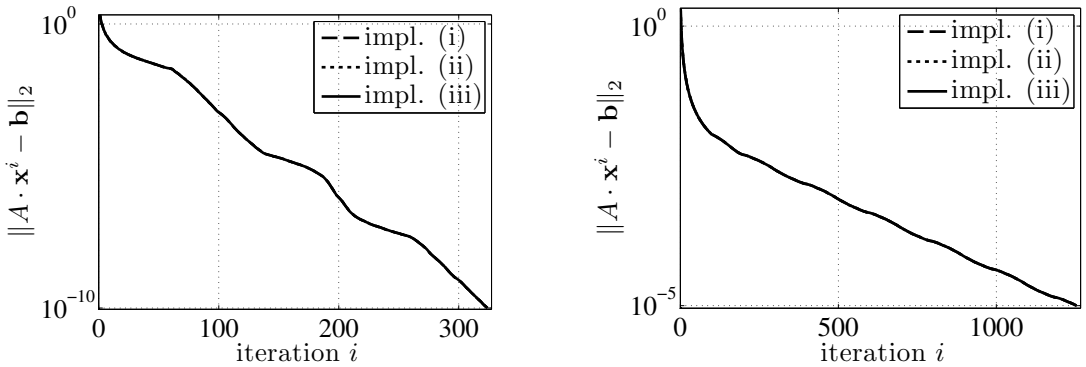
Problem	$\lvert\mathcal{H}\rvert$	n	$P_{\min}$	$M_{\text{DROPS}}(P_{\min})$ [GB]
$(\Omega^{\text{C}}, \text{small})$	234 686	249 954	8	3.2
$(\Omega^{\text{C}}, \text{medium})$	1 863 999	2 047 587	8	8.5
$(\Omega^{\text{C}}, \text{large})$	15 020 070	16 573 239	32	57.6

Table 3.3: Problem characteristics.

and the size of the corresponding system of linear equations n is given in Table 3.3. Here, the symbol Ω^{C} is used to denote the computational domain $[0,1]^3$. This table also shows the minimal number of processes $P_{\min}$—as a multiple of eight—that are needed to represent the corresponding problem in terms of available memory. The actual memory consumption $M_{\text{DROPS}}(P_{\min})$ to store all data on $P_{\min}$ processes is presented in the last column of this table.

We start the discussion of the performance results by considering the time for solving the system of linear equations $A \cdot \mathbf{x} = \mathbf{b}$ when discretizing (3.14). In two-phase flow simulations, this is the main time consuming computational part. The system of linear equations corresponding to (3.14) is symmetric and positive definite and can be solved by CG-type algorithms. In contrast, most systems of linear equations of the discrete two-phase flow problem (2.15)–(2.16) do not exhibit this property. Therefore, we consider instead a restarted, Jacobi-preconditioned [142] GMRES method rather than a CG-type method. Figure 3.5 depicts performance results for three different implementations of the GMRES method.

(i) The serial implementation is parallelized "line-by-line," i.e., each matrix-vector multiplication, inner product, and vector update is performed in par-



(a) $(\Omega^{\mathrm{C}}, \mathrm{small})$ by $P = 8$ processes. (b) $(\Omega^{\mathrm{C}}, \mathrm{large})$ by $P = 32$ processes.

Figure 3.4: Convergence of $\|A \cdot \mathbf{x}^i - \mathbf{b}\|_2$ when solving $A \cdot \mathbf{x} = \mathbf{b}$ by the three implementations of the GMRES method

allel. Here, the orthogonalization within the GMRES method is carried out by the modified Gram–Schmidt [77] method.

(ii) As in the implementation (i), the modified Gram–Schmidt method is employed. However, a more efficient implementation of the GMRES method is applied, i.e., intermediate accumulated vectors are stored if they can be reused later in the algorithm.

(iii) In contrast to the previous two implementations (i) and (ii), the standard Gram–Schmidt [77] method is used to orthogonalize vectors. This is advantageous since this orthogonalization includes only one synchronization of processes opposed to the modified Gram–Schmidt method which performs multiple synchronizations. As in (ii), an efficient implementation is chosen.

Although the standard Gram–Schmidt method is known to be less numerically stable than the modified method, all implementations are capable of solving the system of linear equations $A \cdot \mathbf{x} = \mathbf{b}$ with the same number of iterations. Let $\mathbf{x}^i$ denote the approximation of the solution $\mathbf{x}$ in the i-th iteration of the GMRES method. In Fig. 3.4, the Euclidean norm of the residual vectors $\|A \cdot \mathbf{x}^i - \mathbf{b}\|_2$ are depicted for all three implementations of the GMRES method. Here, $P = 8$ and $P = 32$ processes are employed to solve the problems $(\Omega^{\mathrm{C}}, \mathrm{small})$ and $(\Omega^{\mathrm{C}}, \mathrm{large})$, respectively. These plots exemplarily illustrate the same convergence behavior of all three implementations.

The performance results in Fig. 3.5 consist of two parts. First, in the left three figures (a),(c), and (e), the time $T(P)$ of all three implementations is shown when using up to $P = 1\,024$ processes. Second, the related speedup

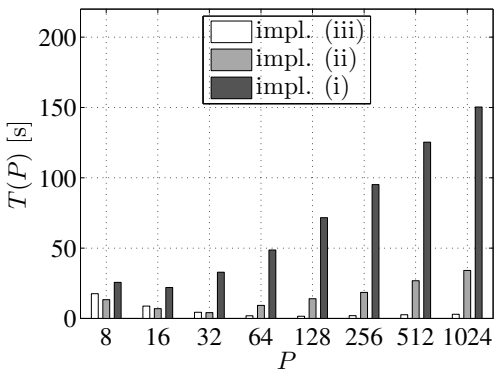
$$S_{P_{\min}}(P) := \frac{T(P_{\min})}{T(P)} \cdot P_{\min} \tag{3.15}$$

is illustrated in the right three plots of Fig. 3.5. Note that the speedup for $P_{\min}$ processes is defined as $S_{P_{\min}}(P_{\min}) := P_{\min}$.

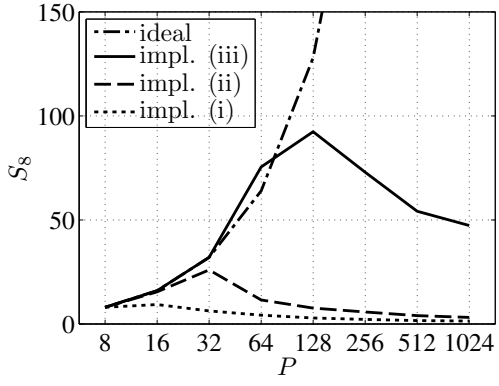
We observe that, in almost all cases, the implementation using the standard Gram–Schmidt method (iii) results in smaller execution times than the other two implementations. The difference in the run times increases when solving the Poisson equation by larger numbers of processes. For instance, in problem $(\Omega^{\mathrm{C}}, \mathrm{large})$, the run time of (ii) and (iii) is almost equal on 128 processes whereas a factor of approximately 3.4 is observed on $P = 1\,024$ processes. An explanation is given by the reduced number of synchronizations in implementation (iii). In particular, in problem $(\Omega^{\mathrm{C}}, \mathrm{small})$ on $P = 1\,024$ processes, the implementation (iii) is 11.5 times faster than (ii) and even 50.7 times faster than (i) where a larger computational overhead is present. The reduced communication and synchronization overhead also implies a better scalability of the implementation (iii). In problem $(\Omega^{\mathrm{C}}, \mathrm{medium})$, the implementations (i) and (ii) scale only up to 64 processes before a slowdown is observed. However, the implementation (iii) is capable of efficiently solving the system of linear equations by the preconditioned GMRES method on 1 024 processes. Here, a speedup of about $S_{32}(1\,024) = 482$ is achieved. Increasing the problem size results in a speedup of $S_{64}(1\,024) = 1\,581$ for problem $(\Omega^{\mathrm{C}}, \mathrm{large})$. Note that a super linear speedup is observed due to cache effects and due to the definition of the speedup in (3.15).

Second, we consider the assembly of the system of linear equations which comprises the second most time consuming part of simulating two-phase flow problems. In Table 3.4, the corresponding timing results are presented in the columns labeled by "assemb." The run time for assembling the system decreases for all three problem sizes when increasing the number of processes. The speedup is very good for all problem instances $(\Omega^{\mathrm{C}}, \mathrm{small})$, $(\Omega^{\mathrm{C}}, \mathrm{medium})$, and $(\Omega^{\mathrm{C}}, \mathrm{large})$, i.e., a speedup of $S_8(1\,024) = 2\,039$, $S_{32}(1\,024) = 1\,409$, and $S_{64}(1\,024) = 890$ is achieved, respectively.

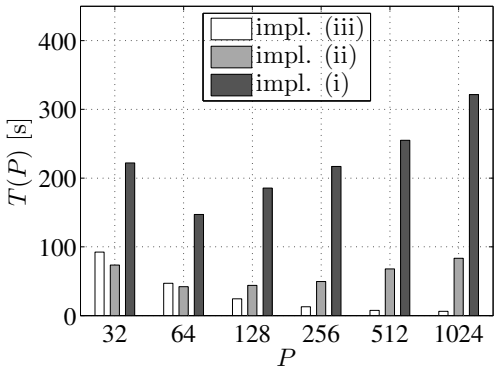
Third, we consider the time for generating the hierarchy of triangulations which is depicted in Table 3.4 in the columns labeled by "grid." This time includes the refinement and migration algorithm. Here, we observe that this computational part scales up to 64 processes for problem $(\Omega^{\mathrm{C}}, \mathrm{small})$ and up to 256 processes for the problems $(\Omega^{\mathrm{C}}, \mathrm{medium})$ and $(\Omega^{\mathrm{C}}, \mathrm{large})$. The time increases when considering larger number of processes. In these cases, the problem size is too small for such large number of processes, i.e., the communication time dominates the runtime and avoids a scaling. However, the time for generating the grid for the problem $(\Omega^{\mathrm{C}}, \mathrm{large})$ by $P = 1\,024$ processes is smaller by a factor of 1.4 than the time solving the corresponding system of linear equations by the fastest implementation (iii) of GMRES. In this example, the time for generating the parallel triangulation hierarchy is larger than assembling the matrices. In constrast, this does not hold for two-phase flow simulations. Moreover, the time spent in gener-



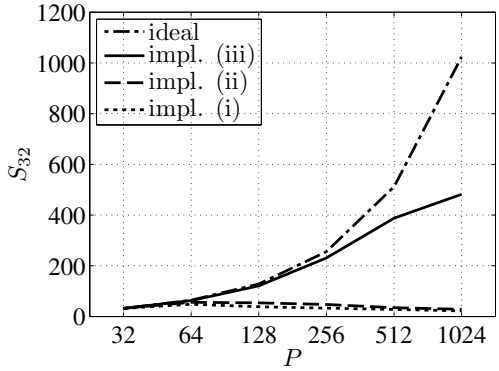
(a) Time (Ω^{C}, small).



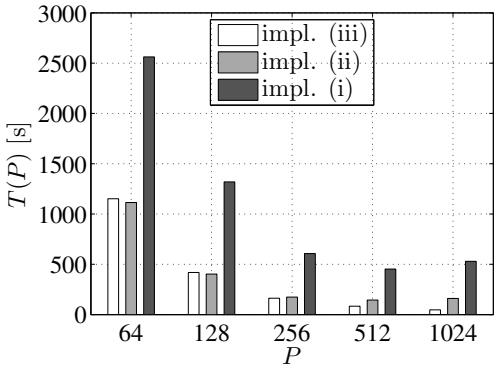
(b) Speedup (Ω^{C}, small).



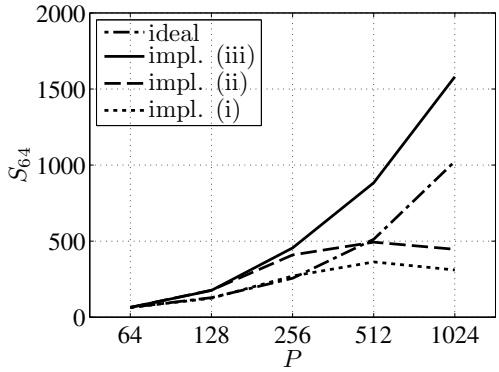
(c) Time (Ω^{C}, medium).



(d) Speedup (Ω^{C}, medium).



(e) Time (Ω^{C}, large).



(f) Speedup (Ω^{C}, large).

Figure 3.5: Performance results for solving the Poisson equation (3.14) when using the Harpertown cluster.

P	(Ω^C, small)		$(\Omega^C, \text{medium})$		(Ω^C, large)	
	assemb	grid	assemb	grid	assemb	grid
8	1.672	2.304	—	—	—	—
16	0.8272	1.488	—	—	—	—
32	0.3930	1.013	3.736	6.701	—	—
64	0.1692	0.7675	1.824	4.004	14.73	33.55
128	0.06410	1.173	0.9051	2.598	7.607	17.55
256	0.02548	1.024	0.4220	1.678	3.670	9.774
512	0.01230	1.357	0.2016	3.972	2.214	28.84
1 024	0.006560	3.038	0.08480	4.630	1.059	33.76

Table 3.4: Time in seconds for assembling the system of linear equation (assemb) and generating the grid (grid).

ating and modifying the grid tends to be negligible for two-phase flow problems because a large number of systems of linear equations are solved on one grid and only small modifications of the grid are made during the course of simulating these flows.

Overall, this example demonstrates that the data structures and algorithms presented in this chapter are capable of efficiently solving PDEs by the finite element method. We conclude this chapter by a brief discussion of the implementation of this parallel approach.

3.4 Discussion

A key ingredient of the parallel refinement algorithm and the admissible distributed hierarchical decomposition is managing and tracking the information about distributed elements of the triangulations, such as tetrahedra, faces, edges, and vertices. That is, within the refinement algorithm each process queries information about each of its locally stored elements, such as existence of that element on other processes or a list of processes owning a copy. Furthermore, in some stages of the refinement algorithm, so-called interface-communication is necessary. Here, information is communicated among all copies of an element. For instance, deciding if a face needs to be refined involves information from all adjacent tetrahedra that are possibly located at different processes. Another type of communication is performed, when tetrahedra are migrated among processes. Here, elements of

the triangulation must be transferred, forming a delicate task. For instance, if a tetrahedron should be transferred from process p_1 to p_2, then its faces, edges, vertices, and, eventually, its children, need to be sent, too. However, this implies that, in some cases, not only p_1 and p_2 exchange information but other processes as well. For instance, if there is a third process p_3 which also stores a copy of an element of t, e.g., a vertex, then p_3 must be informed about the transfer from p_1 to p_2.

To address those tasks, the parallel implementation of DROPS intensively uses the library DDD [24] (Dynamic Distributed Data) of the software toolkit UG [16] (Unstructured Grids). This library is not supported any more and due to performance issues when considering larger number of processes and longer simulation times, a replacement of the library DDD is currently being developed. This new object-orientated module of DROPS basically relies on the methods that are used for the library PMDB which is presented in [65, 144, 151] and is called DISTS (Distributed Simplex Types). Briefly, the functionality of DISTS addresses three tasks. First, it manages the additional information for the distribution of the tetrahedral hierarchy and allows queries of this information. Second, it implements the interface communication. And, third, this module supports the transfer of tetrahedra with all their subsimplices among processes. In the current state, DISTS is not capable of adaptively simulating two-phase flow problems, yet. A detailed description of DISTS is not a topic of this thesis. However, some preliminary results of a two-phase simulation obtained with this module are presented in the case study chapter, i.e., Chap. 6.

A shared-memory parallelization of DROPS has been devised in [163] where the authors employ OPENMP to distribute the computational work among multiple threads. Here, the main focus lies on a parallelization strategy for assembling matrices of the two-phase flow problem and for solving the resulting systems of linear equations, cf. (2.15)–(2.16). Currently, the ideas of using an OPENMP parallelization in combination with object-oriented programming [162] are analyzed to speed up linear algebra computations in DROPS. That is, the underlying classes, which implement matrix- and vector-operations, execute OPENMP-parallel algorithms [161] to compute matrix-vector products, inner products and vector updates. This approach aims at encapsulating sophisticated, additional code for an advanced shared-memory parallelization that considers various memory architectures. The shared-memory parallelization does not contradict the development of the distributed-memory parallelization. Yet, combining both approaches aims at exploiting the architecture of today's high-performance computer systems which primarily consist of clusters of multi-core processors. This so-called hybrid parallel approach uses the distributed-memory parallelization to decompose the computational work among compute nodes whereas the shared-memory parallelization utilizes the compute cores on each node to speedup the computations. In Chap. 5, we go into detail of such a hybrid distributed-/shared-memory parallel approach.

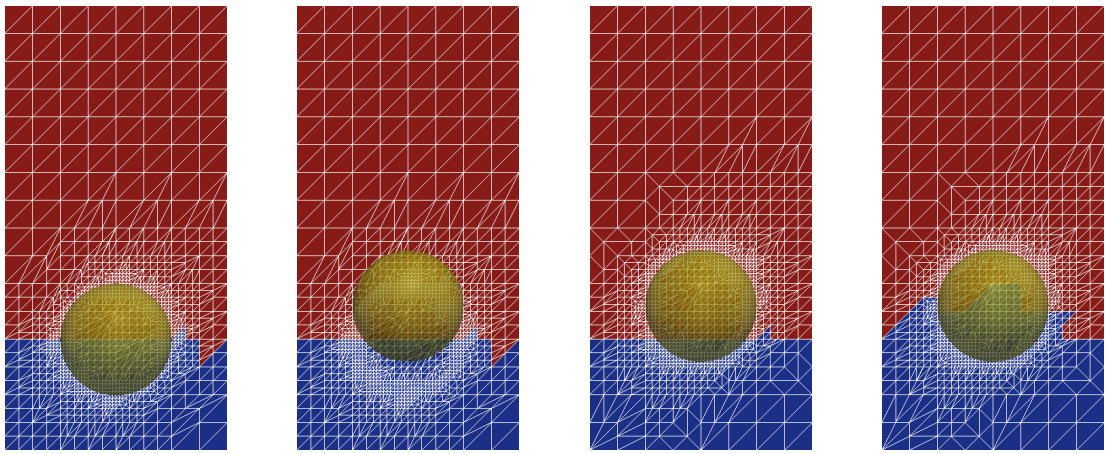
4 Load-Balancing Strategies

In the previous chapter, an admissible distributed hierarchical decomposition has been introduced. In this chapter, we are concerned with approaches to find a suitable decomposition of the computational load among the processes. To this end, we follow a domain decomposition approach where the tetrahedra of the triangulation hierarchies are distributed among the processes. We first give an illustrating example showing that load balancing is of particular interest when simulating two-phase flow problems.

Recall the example of a rising n-butanol drop that has been illustrated in Chap. 2 in Fig. 2.2. Here, first, a triangulation hierarchy $\mathcal{M}^0$ is determined to discretize the computational domain and resolve the physical phenomena in the beginning of the simulation. Afterwards, this hierarchy $\mathcal{M}^0$ is used to solve the two-phase flow problem for a few number of discrete time steps. When the phases and characteristics of the flow have been changed, $\mathcal{M}^0$ is modified by the refinement and coarsening algorithm resulting in a new hierarchy of tetrahedral grids which is denoted by $\mathcal{M}^1$. This procedure of solving the fluid dynamics and modifying the hierarchy is repeatedly applied until the simulation ends. Hence, a sequence of $m \geq 0$ tetrahedral hierarchies $(\mathcal{M}^0, \ldots, \mathcal{M}^{m-1})$ is generated during a two-phase flow simulation.

Next, consider the same example of a rising n-butanol drop simulated by two processes which is illustrated in Fig. 4.1. In Fig. 4.1(a), the initial hierarchy $\mathcal{M}^0$ is distributed among two processes. The tetrahedra stored by the first process are colored blue and those of the second process are highlighted in red. The n-butanol drop is colored in yellow. Recall from Chap. 3 that the computational work and the memory consumption of one process corresponds to the number of tetrahedra which are stored by this process. However, the work and memory do not depend on the size of a tetrahedron. Thus, in this figure, roughly the same number of tetrahedra is colored blue or red. The size of the process boundary approximates the communication volume between adjacent processes. Therefore, the process boundary is determined as small as possible in this example. The distribution of tetrahedra on all levels of the triangulation hierarchy can be expressed by the partitioning function

$$\pi^0_{\mathcal{M}} : \bigcup_{\mathcal{T} \in \mathcal{M}^0} \mathcal{T} \to \{1, \ldots, P\}$$



(a) Initialization. (b) Moved drop. (c) Grid adaption. (d) Load balance.

Figure 4.1: Parallel simulation of a rising n-butanol drop by two processes. The tetrahedra stored by the first process are colored in red and those of the second process in blue. The n-butanol drop is given in a yellow color.

that maps the tetrahedra of $\mathcal{M}^0$ to $P \geq 2$ processes. That is, if $\pi^0_{\mathcal{M}}(t) = p$ holds for a tetrahedron $t \in \mathcal{M}^0$, then t is assigned to process $1 \leq p \leq P$. As in the sequential case, after a few time steps, the hierarchy is modified by the parallel refinement algorithm to adequately resolve the problem characteristics. Figure 4.1(b) and (c) respectively show the grid before and after the modifications. The refinement algorithm may cause an imbalance in the number of tetrahedra which are assigned to the processes. For instance, there are more red than blue tetrahedra in Fig. 4.1(c). Hence, the grid modification may lead to a discrepancy in the computational work load of the processes in contrast to the sequential case. If this imbalance in the number of tetrahedra is too large, then we seek for a new partitioning function $\pi^1_{\mathcal{M}}$ that decomposes the tetrahedra of $\mathcal{M}^1$. This partitioning again aims at reducing the communication volume while balancing the computational load. Afterwards, according to the mapping $\pi^1_{\mathcal{M}}$, the tetrahedra of $\mathcal{M}^1$ are migrated among the processes. For the present example, the result of the migration is depicted in Fig. 4.1(d).

From a theoretical point of view, we seek for a sequence of m partitioning functions $(\pi^0_{\mathcal{M}}, \ldots, \pi^{m-1}_{\mathcal{M}})$ that describes the decomposition of the m triangulation hierarchies $(\mathcal{M}^0, \ldots, \mathcal{M}^{m-1})$ among P processes while simulating a two-phase flow problem. Commonly , the triangulation hierarchy $\mathcal{M}^i$ with $0 < i < m$ differs only slightly from $\mathcal{M}^{i-1}$ because coarsening and refinement operations are only locally performed on small subdomains of the whole computational domain. Furthermore, the triangulation $\mathcal{M}^i$ results from the parallel refinement of the distributed

hierarchy $\mathcal{M}^{i-1}$. Thus, the hierarchy $\mathcal{M}^i$ is also distributed among the processes. This leads to two observations. First, if $\mathcal{M}^i$ and $\mathcal{M}^{i-1}$ differ only slightly, the imbalance in tetrahedra among the processes may be insignificant and there is no need for migrating tetrahedra. Second, the distribution of tetrahedra in $\mathcal{M}^{i-1}$ may serve as a good "initial guess" to search for the partitioning $\pi^i_{\mathcal{M}}$ to decompose $\mathcal{M}^i$. This issue of dynamically or adaptively finding P-way partitioning functions has been extensively investigated in, e.g., [54,57,65,146,147]. These models include in general a parameter specifying a trade-off between the computational load and the migration costs for the computational elements (in our case the tetrahedra). We investigate two-phase flow simulations in this thesis, where the migration costs for a tetrahedron are negligible compared to the computational load that is caused by a single tetrahedron. Therefore, we concentrate on finding a good decomposition of a single tetrahedral hierarchy $\mathcal{M}$ rather than determining partitionings for a sequence of hierarchies.

In a general two-phase flow simulation, most of the computational load originates from assembling various systems of linear equations and solving these systems. Furthermore, storing these systems needs a huge amount of memory. To simulate large and meaningful instances of two-phase flow problems, parallel computing is indispensable as they provide sufficient memory and necessary compute power. However, the parallelization commonly results—besides communication—in a memory overhead compared to the storage of a sequential simulation. For instance, in the parallel approach presented in Chap. 3, degrees of freedom (DOFs) at process boundaries are stored by multiple processes. Hence, the goal of this chapter is to determine a decomposition of the tetrahedra aiming at

- minimizing the storage overhead and the communication volume for parallel computing, and
- balancing the computational load of all processes.

To determine such a decomposition of tetrahedra, we follow the common load-balancing approach where the triangulation is represented by a graph and a partitioning of this graph yields a decomposition of the tetrahedra among the processes. Overall, the load-balancing algorithm redistributes a decomposed hierarchy $\mathcal{M}$ of tetrahedral grids among processes and consists of four steps:

(i) represent the tetrahedral hierarchy $\mathcal{M}$ by a graph G;

(ii) partition the graph G;

(iii) interpret the partitioning of the graph G as a partitioning function $\pi_{\mathcal{M}}$ for the tetrahedral hierarchy $\mathcal{M}$; and

(iv) migrate tetrahedra according to $\pi_{\mathcal{M}}$.

An overview of literature concerning this approach is given in [164] and, additionally, diverse graph partitioning algorithms and libraries are compared for various

PDE applications. However, these graph models do not take two-phase flow simulations into account. In this chapter, we focus on the first step of the load-balancing algorithm and devise graph models to accurately model a triangulation hierarchy when particularly simulating two-phase flow problems. The second step of the above algorithm is surveyed in, e.g., [64] and the third and fourth step is detailed in [79].

The outline of this chapter is as follows. We pursue two different approaches for determining a decomposition of the tetrahedra among the processes. In the first approach which is presented in Sect. 4.1, the tetrahedral hierarchy is described by "standard" graphs. The underlying graph has been presented in [79, 80]. This graph represents the tetrahedra of the triangulation hierarchy by vertices and their adjacencies by the graph edges. Thus, a partitioning of the graph vertices results in a decomposition of the tetrahedra. We advance this graph model for especially addressing two-phase flow simulations. To this end, we employ specially designed weighting functions for the vertices and for the edges to represent certain properties of two-phase flow simulations. These properties include the varying computational load of each tetrahedron, communication patterns when solving the linear equation systems, and the storage overhead for representing numerical data in parallel. These weighting functions have been introduced in [67, 72]. The second approach is introduced in Sect. 4.2. In contrast to the standard graph models, a hypergraph model is investigated to exactly model the communication volume among neighbor processes while performing linear algebra operations on distributed vectors. That is, the hypergraph representing the hierarchy of triangulations contains information about the communication pattern that is applied in the course of simulating two-phase flow problems. In Sect. 4.3, we conclude this chapter by a discussion on the presented approaches and an outlook on future work.

Throughout this chapter, we exemplify all presented graph models by the same hierarchy of triangulations depicted in Fig. 2.3. Recall that this hierarchy consists of three levels. In Fig. 4.2, this hierarchy is presented again where all tetrahedra of the finest triangulation are labeled.

4.1 Modeling the Tetrahedral Hierarchy by Graphs

To model the hierarchy of tetrahedral grids by graphs, we first summarize the approach presented by Sven Groß in [79,80]. In these theses, the vertices and edges of the graph are defined such that the graph vertices represent the tetrahedra and the edges their adjacencies in the tetrahedral hierarchy. Furthermore, the vertices and edges are weighted by information only considering the tetrahedral hierarchy. The resulting graph is called *tetrahedral graph* and is denoted by $G_{\mathcal{M}}$. A partitioning of this graph aims at minimizing the number of faces at process boundaries while

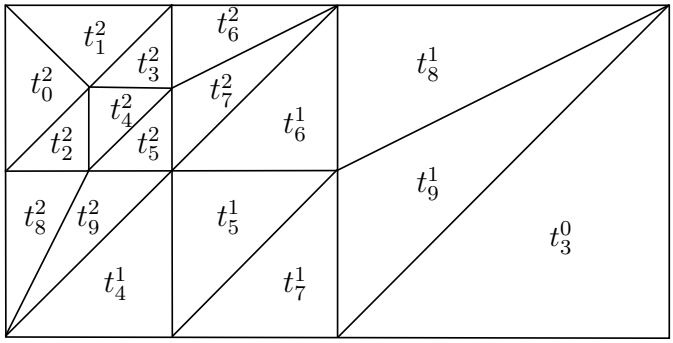


Figure 4.2: Finest triangulation of the hierarchy in Fig. 2.3.

balancing the number of tetrahedra of the finest triangulation. This approach is reasonable for finite element simulations due to the following observation stated in [79]. Let n_T denote the number of tetrahedra in a triangulation $\mathcal{T}$ used to discretize a connected domain without holes and n_{F_B} the number of faces at the boundary. In addition, let $A \in \mathbb{R}^{n \times n}$ denote the stiffness matrix for linear P_1 finite element functions on $\mathcal{T}$. Then, the number of floating point operations for computing $A \cdot \mathbf{x}$ with $\mathbf{x} \in \mathbb{R}^n$ is bounded by

$$16\, n_T + 7/2\, n_{F_B} + 2. \tag{4.1}$$

However, it turns out that these weights only roughly approximate the computational load and data dependencies for the finite element solution of a two-phase flow problem. We cope with these problems by developing two graph models which rely on the same definition of vertices and edges but different weighting functions [67, 72]. In [72], we presented a new weighting function for the vertices to better model the computational load corresponding to a vertex. These vertex weights are based on information about DOFs and yield the *DOF graph* G_{D}. We modify this model in [67] by empirically determining the computational load per vertex yielding a weighting function for the graph vertices. Moreover, we evaluate the weight of the edges by considering the memory overhead when simulating two-phase flow problems by multiple processes. This overhead is given by the DOFs that are redundantly stored on process boundaries. Modeling the storage overhead by the weight of edges also results in small communication volume when the discrete two-phase flow problem is simulated. Overall, we call the resulting graph the *two-phase graph* which is denoted by G_Γ.

This section is organized as follows. In Sect. 4.1.1, we briefly outline the standard graph partitioning problem. Afterwards, in Sect. 4.1.2, we summarize the approach given by Groß in [79, 80] yielding the tetrahedral graph $G_{\mathcal{M}}$. The DOF graph G_{D} and the two-phase graph G_Γ are detailed in Sect. 4.1.3 before we present numerical results on up to 512 processes in Sect. 4.1.4.

4.1.1 Graph Partitioning

First, we define the general P-way graph partitioning problem before describing the graph models. To this end, let $G = (V, E, \varrho, \sigma)$ be a weighted graph, where V denotes the vertices and E the edges of the graph G. The weights of the vertices and edges are given by the two functions $\varrho : V \to \mathbb{R}^+$ and $\sigma : E \to \mathbb{R}^+$, respectively. In a more general setting, both weighting functions can be multidimensional. If each vertex is mapped to multiple weights, the corresponding partitioning problem is called multi-constrained. If the edge-weighting function σ maps each edge to more than one weight, the partitioning problem is called multi-objective. However, opposed to Sect. 4.2, we are concerned with one-dimensional weights for both vertices and edges in this section. The objective of the graph partitioning problem is to find a *P-way graph partitioning function* $\pi_G : V \to \{1, \ldots, P\}$. This function decomposes the set of vertices V into $P \geq 2$ disjoint subsets $V_1, \ldots, V_P$, i.e., the sets $V_p = \{v \in V \mid \pi_G(v) = p\}$, for $1 \leq p \leq P$, fulfill the two conditions

$$\bigcup_{p=1}^{P} V_p = V \quad \text{and} \quad V_p \cap V_q = \emptyset,\ p \neq q.$$

Using these notations, we introduce the *P-way graph partitioning problem* for a graph by the following definition.

P-way Graph Partitioning

For a given weighted graph $G = (V, E, \varrho, \sigma)$, find a P-way graph partitioning function $\pi_G : V \to \{1, \ldots P\}$ such that

$$\textit{minimize} \quad E_{cut} := \sum_{\substack{e=(v,w)\in E \\ \pi_G(v)<\pi_G(w)}} \sigma(e), \tag{4.2}$$

$$\textit{s.t.} \quad \sum_{\pi_G(v)=p} \varrho(v) \approx \frac{1}{P} \sum_{v \in V} \varrho(v) \quad \textit{for } p = 1, \ldots, P. \tag{4.3}$$

That is, find a decomposition of the vertices V into P partitions, such that the edge cut E_{cut} is minimized (4.2) while balancing the accumulated vertex weights of each partition (4.3). From a theoretical point of view, if considering two equal sized partitions with a minimal edge cut, the graph partitioning problem is known to be NP-complete [75, 76]. Hence, these types of problems are solved in general by heuristics or approximation algorithms which can be found in, e.g., [126, 148].

4.1.2 Tetrahedral Graph Model

Next, we describe the tetrahedral graph $G_{\mathcal{M}}$ that aims at modeling a hierarchy of tetrahedral grids $\mathcal{M} = (\mathcal{T}_0, \ldots, \mathcal{T}_{k-1})$. Since we simulate two-phase flow problems on the finest triangulation level, the objective of the corresponding P-way partitioning problem is to decompose the tetrahedra of the finest level among P processes. In the graph $G_{\mathcal{M}}$, the vertices in V represent the tetrahedra and the edges in E correspond to adjacencies between the tetrahedra. A vertex $v \in V$ is introduced for each parent tetrahedron t that has at least one child $\mathrm{Ch}(t)$ in the finest triangulation. Formally, we introduce a vertex $v \in V$ for each tetrahedron t with

$$t \in \bigcup_{l=0}^{k-1} \mathcal{T}_l \text{ and } \begin{cases} \mathrm{level}(t) = 0, & \text{if } t \text{ is unrefined or} \\ \mathrm{Ch}(t) \cap \mathcal{T}_{k-1} \neq \emptyset, & \text{if } t \text{ is refined} \end{cases}. \tag{4.4}$$

That is, a vertex $v \in V$ either corresponds to an unrefined tetrahedron in the coarsest triangulation or to a parent tetrahedron with its children where at least one child is located in the finest triangulation. The second condition ensures that all children of a parent tetrahedron are represented by a single vertex. The advantage is twofold. First, using a single vertex to represent a set of tetrahedra rather than a single tetrahedron reduces the number of vertices in the graph $G_{\mathcal{M}}$. Therefore, the graph is called reduced in [79]. Second, if a vertex v is assigned to a process p then all tetrahedra represented by v are assigned to p, too. Recall that this is crucial for the implementation of the refinement algorithm, cf. Sect. 3.1. In the remainder, let $\mathcal{P}(S)$ denote the power set of a set S and

$$T : V \to \mathcal{P}(\mathcal{T}_{k-1}) \tag{4.5}$$

express the mapping of a vertex v to the set of tetrahedra on the finest triangulation which are represented by v.

Although $T(u) \cap T(v) = \emptyset$ holds for all $u \neq v$, there may exist tetrahedra—not located at the finest triangulation—that are represented by more than one vertex. For instance, consider the forest representation of a tetrahedral hierarchy which is displayed in Fig. 4.3. The tetrahedron t^0 has two children: the left child t^1_{l} and the right child t^1_{r}. The tetrahedron t^1_{l} is located on the finest triangulation and, thus, a vertex v is introduced to represent the tetrahedra t^0, t^1_{l} and t^1_{r}. The mapping in (4.5) reads as $T(v) = \{t^1_{\mathrm{l}}\}$ for this vertex v. The right child t^1_{r} of t^0 is further refined into its left t^2_{l} and right child t^2_{r}. Since both children are located in the finest triangulation, a vertex v' is introduced which represents t^1_{r} and its children and thus $T(v') = \{t^2_{\mathrm{l}}, t^2_{\mathrm{r}}\}$. In this example, the intersection of the sets $T(v)$ and $T(v')$ is empty, however, both vertices v and v' represent the tetrahedron t^1_{r}.

The edges E of the graph $G_{\mathcal{M}}$ are introduced as follows. Two vertices v and w of $G_{\mathcal{M}}$ are adjacent if there is at least one common face in the sets of tetrahedra $T(v)$ and $T(w)$. To this end, let $t_1 \sqcap t_2$ denote either the common face between

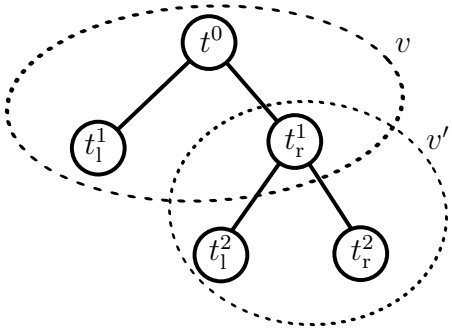


Figure 4.3: Forest representation and vertices of $G_\mathcal{M}$ for a tetrahedral hierarchy.

two tetrahedra t_1 and t_2 or the empty set, if no common face exists. This notation allows us to define the edges of the graph by

$$(u, v) \in E \quad :\Leftrightarrow \quad \exists\, t_u \in T(u) \text{ and } t_v \in T(v) \text{ such that } t_u \sqcap t_v \neq \emptyset. \tag{4.6}$$

In the style of (4.5), the mapping $F : E \to \mathcal{P}\left(\{f \mid f \text{ is face in } \mathcal{T}_{k-1}\}\right)$ assigns each graph edge $(v, w) \in E$ to the set of faces that occur in both $T(v)$ and $T(w)$, in formula

$$F(u, v) = \{f \text{ is face in } \mathcal{T}_{k-1} \mid \exists\, t_u \in T(u) \text{ and } t_v \in T(v) : t_u \sqcap t_v = f\}\,.$$

An intuitive way of weighting the vertices and the edges of the graph is as follows. Since each vertex $v \in V$ represents a set $T(v)$ of tetrahedra, a reasonable weight for a vertex is given by

$$\varrho_\mathrm{T} : V \to \mathbb{R}^+,\ v \mapsto |T(v)|\,. \tag{4.7}$$

That is, each vertex is weighted by the number of represented tetrahedra in the finest triangulation. Note that the vertex weights proposed in [79] slightly differ from the vertex weights ϱ_T defined in (4.7). To illustrate the difference, consider a vertex v that is introduced for a parent tetrahedron t. Then, the weight of v is determined as the number of all children of t in [79]. In contrast, the weighting function ϱ_T only considers the children of t that are located on the finest triangulation level because we aim at evenly distributing these tetrahedra. For instance, we weight the vertex v in Fig. 4.3 by $\varrho_\mathrm{T}(v) = 1$ whereas in [79] the weight of v is computed as 2.

For weighting the edges, we observe the following. The faces in the finest triangulation form the dependencies between the tetrahedra because DOFs are located at the corner vertices and edges of the faces. If a face of the finest triangulation is stored on two processes, the adjacent DOFs cause communication. Thus, a larger number of faces between processes yields a larger communication volume. Hence,

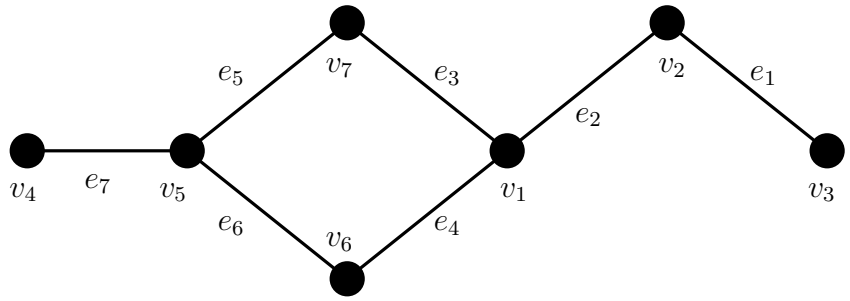


Figure 4.4: Graph $G_{\mathcal{M}}$ representing the triangulation hierarchy $\mathcal{M}$ given in Fig. 2.3.

a meaningful weight for an edge $e = (v, w)$ is given by the number of common faces between the tetrahedron families $T(v)$ and $T(w)$, in formula

$$\sigma_{\mathrm{T}} : E \to \mathbb{R}^+, \quad e \mapsto |F(e)|\,.$$

So, if an edge e represents more faces than another edge e', i.e., $|F(e)| > |F(e')|$, then the weight of e is larger than the weight of e', i.e., $\sigma_{\mathrm{T}}(e) > \sigma_{\mathrm{T}}(e')$. These vertex and edge weights aim at minimizing the number of floating point operations for determining the sparse matrix vector product of a stiffness matrix for P_1 functions, cf. (4.1).

Next, as an illustrating example, we present the graph $G_{\mathcal{M}}$ of the hierarchy of triangulations $\mathcal{M}$ in Fig. 4.2. The graph $G_{\mathcal{M}}$ is depicted in Fig. 4.4 whereas the vertices and edges are listed in Table 4.1. Although the corresponding hierarchy consists of 24 tetrahedra and 21 interior faces, $G_{\mathcal{M}}$ has only seven vertices and seven edges. For instance, the vertex v_5 is introduced to represent the parent tetrahedron t_1^1 and its four children $t_2^2, \ldots, t_5^2$ on the finest triangulation $\mathcal{T}_2$. Hence, on the finest triangulation level, the vertex v_5 represents the tetrahedra $T(v_5) = \{t_2^2, \ldots, t_5^2\}$. The weight of vertex v_5 is determined by $\varrho_{\mathrm{T}}(v_5) = 4$. The parent tetrahedron and the mapping T for all vertices are presented in the second and third column of Table 4.1(a), respectively. This table also includes the weights $\varrho_{\mathrm{T}}(v)$ for each vertex $v \in V$. In Table 4.1(b), the adjacent vertices and the corresponding weights $\sigma_{\mathrm{T}}(e)$ are displayed for each graph edge $e \in E$. For instance, the edge $e_6 = (v_5, v_6)$ is introduced because the two tetrahedron families $T(v_5)$ and $T(v_6) = \{t_8^2, t_9^2\}$ have two common faces, i.e., $t_2^2 \sqcap t_8^2$ and $t_5^2 \sqcap t_9^2$.

Using the tetrahedral graph to model a tetrahedral hierarchy $\mathcal{M}$, we arrive at the problem of *tetrahedral graph partitioning for a tetrahedral hierarchy (T-GPTH)* that states the problem of finding a decomposition of $\mathcal{M}$ as a P-way

v	parent	$T(v)$	$\varrho_{\mathrm{T}}(v)$
v_1	t_1^0	$\{t_4^1, t_5^1, t_6^1, t_7^1\}$	4
v_2	t_2^0	$\{t_8^1, t_9^1\}$	2
v_3	t_3^0	$\{t_3^0\}$	1
v_4	t_0^1	$\{t_0^2, t_1^2\}$	2
v_5	t_1^1	$\{t_2^2, t_3^2, t_4^2, t_5^2\}$	4
v_6	t_2^1	$\{t_8^2, t_9^2\}$	2
v_7	t_3^1	$\{t_6^2, t_7^2\}$	2

(a) List of vertices V.

e	vertices	$\sigma_{\mathrm{T}}(e)$
e_1	(v_2, v_3)	1
e_2	(v_1, v_2)	2
e_3	(v_1, v_7)	1
e_4	(v_1, v_6)	1
e_5	(v_5, v_7)	2
e_6	(v_5, v_6)	2
e_7	(v_4, v_5)	2

(b) List of edges E.

Table 4.1: Weights ϱ_{T} and σ_{T} of the graph $G_{\mathcal{M}}$ given in Fig. 4.4.

graph partitioning problem.

Tetrahedral Graph Partitioning for a Tetrahedral Hierarchy (T-GPTH)

Given a hierarchy $\mathcal{M}$ of tetrahedral grids and the corresponding tetrahedral graph model $G_{\mathcal{M}} = (V, E, \varrho_T, \sigma_T)$. Then, the T-GPTH consists of finding a P-way graph partitioning function $\pi_G : V \to \{1, \ldots, P\}$ that solves the P-way graph partitioning problem for $G_{\mathcal{M}}$.

Although the weights ϱ_{T} and σ_{T} for the vertices and edges of the graph $G_{\mathcal{M}}$ are quite meaningful, these weights do not distinguish between different types of tetrahedra or faces. For instance, a tetrahedron located at the boundary of the computational domain may cause less computational load than a tetrahedron intersected by the interface Γ_φ between both phases. In addition, the communication volume via a single face may vary with respect to the location of the face. Therefore, we present two more sophisticated approaches to weight the vertices and edges of the graph $G_{\mathcal{M}}$ in the following. These approaches rely on the research presented in [67, 72].

4.1.3 Two-Phase Flow Graph Models

Recall that the discrete two-phase flow problem consists of a system of coupled non-linear equations, cf. (2.15)–(2.16). This system is assembled by a loop over all tetrahedra of the finest triangulation $\mathcal{T}_{k-1}$. Each tetrahedron updates a (small) subset of nonzeros in the matrices. The size of the subset depends on the DOFs

	$t \in \mathcal{T}_{k-1} \setminus \mathcal{T}_\Gamma$		$t \in \mathcal{T}_\Gamma$	
	vertex	edge	vertex	edge
$\mathbf{u}$	3	3	3	3
φ	1	1	1	1
p	1	0	2	0

Table 4.2: Number of DOFs to represent the velocity, pressure and level set function. The numbers are given for vertices and edges of a tetrahedron t residing on the finest triangulation.

located at the corner vertices and edges of the tetrahedron. In the approach to simulate a two-phase flow problem presented in Chap. 2, the velocity $\mathbf{u}$ is given by a vector-valued quadratic (P_2) function, the pressure by a scalar-valued extended linear (P_1^Γ) function, and the level set function is discretized by a scalar-valued quadratic (P_2) function. In the remainder, let $\mathcal{T}_\Gamma$ denote the tetrahedra on the finest triangulation that are intersected by the interface Γ_φ. Table 4.2 summarizes the number of DOFs for all three finite element functions located at corner vertices and edges of a tetrahedron $t \in \mathcal{T}_{k-1}$ in the interior of the computational domain Ω. Note that these numbers may decrease for tetrahedra that are located at a domain boundary where Dirichlet boundary conditions are employed. Not only the storage requirement of a single tetrahedron t depends on its location, but also the computational work caused by t. For instance, determining the values of nonzeros related to a tetrahedron of $\mathcal{T}_\Gamma$ involve more computational work than evaluating the nonzeros of a tetrahedron in $\mathcal{T}_{k-1} \setminus \mathcal{T}_\Gamma$. Overall, this results in the following observation:

> *A tetrahedron intersected by the interface Γ_φ causes a higher memory consumption and computational work than a tetrahedron which is not intersected by Γ_φ.*

In [72], we presented a first approach that considers this difference in memory and work while determining a suitable graph. To this end, we employ a new weighting function ϱ_D for the vertices which considers information about DOFs. Let $\mathsf{dof}(t)$ denote the set of velocity, pressure and level set DOFs located at the vertices and edges of a tetrahedron $t \in \mathcal{T}_{k-1}$ as given in Table 4.2. Then, we determine the weight of a vertex by

$$\varrho_\mathrm{D} : V \to \mathbb{R}^+, \quad v \mapsto \Big| \bigcup_{t \in T(v)} \mathsf{dof}(t) \Big|.$$

That is, the weight of a vertex v is evaluated by the number of DOFs located at all vertices and edges of the tetrahedron family $T(v)$. Using the same ver-

tices V, edges E, and edge weighting function σ_{T} as for the tetrahedral graph $G_{\mathcal{M}}$, we arrive at the *DOF graph* $G_{\mathrm{D}} = (V, E, \varrho_{\mathrm{D}}, \sigma_{\mathrm{T}})$ for representing a tetrahedral hierarchy. This model aims at evenly balancing the number of DOFs among the processes. The P-way graph partitioning problem for G_{D} results in the *DOF-graph partitioning for a tetrahedral hierarchy (T-GPTH).*

DOF Graph Partitioning for a Tetrahedral Hierarchy (D-GPTH)

Given a hierarchy $\mathcal{M}$ of tetrahedral grids and the corresponding DOF graph denoted by $G_D = (V, E, \varrho_D, \sigma_T)$. Then, the D-GPTH problem consists of finding a P-way graph partitioning function $\pi_G : V \to \{1, \ldots, P\}$ that solves the P-way graph partitioning problem for G_D.

However, the numerical results in Sect. 4.1.4 will show that employing the D-GPTH problem to determine a decomposition of the tetrahedra among the processes is inferior compared to the T-GPTH. Indeed, employing the T-GPTH model yields a better balance of DOFs as the D-GPTH. An explanation is as follows. Consider a triangulation vertex u that is adjacent to a large number of tetrahedron families. Then, the number of DOFs located at u influences the weight of all graph vertices representing the adjacent tetrahedron families. Thus, in the D-GPTH model, the DOFs located at u are represented multiple times for various graph vertices. However, in the triangulation, these DOFs occur only once, if u resides on one process.

To overcome this drawback, we have introduced a new graph model [67] which is better suited for representing a tetrahedral hierarchy employed in two-phase flow simulations. Therefore, we now investigate the assembly of the systems of nonlinear equations (2.15)–(2.16) to represent the discrete two-phase flow problem. We then use this observation to derive a graph partitioning model. Recall that DOFs located at process boundaries are stored redundantly. Hence, these DOFs cause a storage overhead. Furthermore, the computational load introduced by a single tetrahedron varies with respect to its location. We formalize this observation in the following problem definition.

Parallel Finite Element Assembly (PFEA)

Find a P-way partitioning function $\pi_{\mathcal{M}}$ that decomposes the set of tetrahedra in the hierarchy $\mathcal{M} = (\mathcal{T}_0, \ldots, \mathcal{T}_{k-1})$ of triangulations to $P \geq 2$ processes such that the amount of redundant storage for the matrices A, N, B, L and the vectors $\mathbf{b}$, $\mathbf{c}$, $\mathbf{d}$ of the discrete two-phase flow problem (2.15)–(2.16) is minimized and the computational work for assembling these data is evenly balanced among the processes.

Note that finding a partitioning function $\pi_{\mathcal{M}}$ that approximates a solution to the PFEA problem also provides a "good" partitioning when considering the so-

lution process of the non-linear equations systems. Two reasons are given in the following. First, a small storage overhead implies less communication volume since the communication takes place on the DOFs that are redundantly stored. Second, tetrahedra intersected by the interface Γ_φ are linked to more DOFs than other tetrahedra and, thus, more rows of the systems(2.15)–(2.16) are assigned to tetrahedra intersected by the interface. So, a tetrahedron intersected by Γ_φ causes more computational load when solving these systems of equations of the discrete two-phase flow problem. To transform the PFEA problem to a suitable P-way graph partitioning problem, we consider the same vertices V and edges E as for the graphs $G_\mathcal{M}$ and G_D and define suitable weights ϱ_Γ for the vertices and σ_Γ for the edges. This results in the *two-phase graph* G_Γ.

The difference in the computational work caused by a single tetrahedron t results whether t is intersected by Γ_φ or t is not located in the vicinity of Γ_φ. Since the vertices V of the graph represent the tetrahedra, we introduce a parameter $\varrho_\mathrm{I} \in \mathbb{R}^+$ for weighting the vertices that correspond to intersected tetrahedra. That is, the parameter ϱ_I represents the computational overhead that is caused if a tetrahedron is intersected by Γ_φ. For instance, $\varrho_\mathrm{I} = 2$ states that a tetrahedron $t \in \mathcal{T}_\Gamma$ causes twice the computational work than a tetrahedron which is not intersected by Γ_φ. To weight a vertex v, we count the number of intersected tetrahedra, weight these tetrahedra by ϱ_I, and add the number of non-intersected tetrahedra. This results in the definition of ϱ_Γ by

$$\varrho_\Gamma : V \to \mathbb{R}^+, \quad v \mapsto \varrho_\mathrm{I} \cdot |T(v) \cap \mathcal{T}_\Gamma| + |T(v) \setminus \mathcal{T}_\Gamma| \,. \tag{4.8}$$

We use the edge weighting function to model the storage requirements for the DOFs. Since a graph edge $e \in E$ represents the faces $F(e)$ between tetrahedron families, e also describes the vertices and edges of the finest triangulation where DOFs are located. So, we define the weighting function for graph edges by

$$\sigma_\Gamma : E \to \mathbb{R}^+, \quad e \mapsto \Big| \bigcup_{f \in F(e)} \mathsf{dof}(f) \Big|.$$

Here, $\mathsf{dof}(f)$ denotes the set of DOFs located at the triangulation vertices and edges at the face f. This implies that an edge e with a larger weight $\sigma_\Gamma(e)$ represents more DOFs than an edge e' with $\sigma_\Gamma(e') < \sigma_\Gamma(e)$.

To present an example for both weights ϱ_Γ and σ_Γ, we again investigate the triangulation hierarchy of Sect. 2.2.1, whose finest triangulation is given in Fig. 4.2. We assume that an interface Γ_φ is given as illustrated by the dashed line in Fig. 4.5. Since the P_1^Γ pressure DOF are extended at intersected tetrahedra, the vertices containing the extended pressure DOF are highlighted by a circle $\circ$. Recall that the vertices and the edges of G_Γ are identical to those of $G_\mathcal{M}$ and G_D. Hence, the graph in Fig. 4.4 also displays V and E of G_Γ. The vertex weights $\varrho_\Gamma(v)$

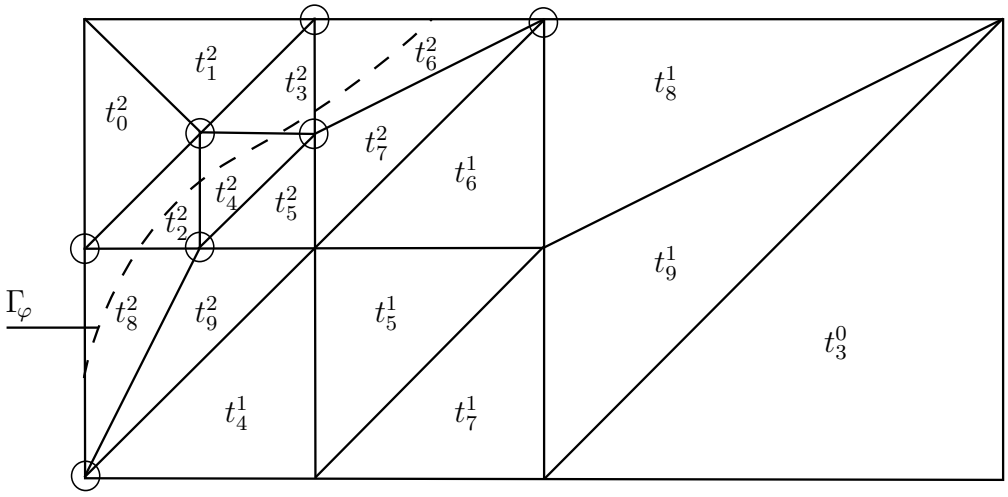


Figure 4.5: Triangulation $\mathcal{T}_2$ of Fig. 4.2 and an interface Γ_φ.

for all $v \in V$ are displayed in Table 4.3(a). For instance, the weight of the graph vertex $v_5 \in V$ of G_Γ is obtained as follows. This vertex represents the four tetrahedra $T(v_5) = \{t_2^2, \ldots, t_5^2\}$ on the finest triangulation. Here, the three tetrahedra $\{t_2^2, t_3^2, t_4^2\}$ are intersected by Γ_φ whereas $t_5^2 \notin \mathcal{T}_\Gamma$. Hence, in this example, the weight of vertex v_5 is determined as $\varrho_\Gamma(v_5) = 3\varrho_\mathrm{I} + 1$. The edge weights of the graph in Fig. 4.4 are presented in Table 4.3(b). The second, third, and fourth column of this table respectively display the number of pressure, velocity and level set DOFs on vertices and edges of the triangulation, cf. Table 4.2. For instance, the graph edge $e_5 = (v_5, v_7)$ represents the two triangulation faces $t_5^2 \sqcap t_7^2$ and $t_3^2 \sqcap t_6^2$. These two faces consist of two edges and three vertices two of which are extended. So, by Table 4.2, the resulting edge weight of e_5 is given by

$$\sigma_\Gamma(e_5) \;=\; \underbrace{5 \cdot 3}_{\mathbf{u}} + \underbrace{2 \cdot 2 + 1 \cdot 1}_{p} + \underbrace{5 \cdot 1}_{\varphi} \;=\; 25.$$

Overall, we arrive at the parametrized *two-phase graph partitioning for a tetrahedral hierarchy (TP-GPTH)* problem to model the PFEA problem in terms of a P-way graph partitioning problem.

Two-Phase Graph Partitioning for a Tetrahedral Hierarchy (ϱ_I-TP-GPTH)

Given a hierarchy $\mathcal{M}$ of tetrahedral grids, the parameter ϱ_I, and the corresponding two-phase graph $G_\Gamma = (V, E, \varrho_\Gamma, \sigma_\Gamma)$. Then, the ϱ_I-TP-GPTH problem consists of finding a P-way graph partitioning function $\pi_G : V \to \{1, \ldots, P\}$ that solves the P-way graph partitioning problem for G_Γ.

We refer to Table 4.10 in the end of this chapter for a comparison of all present graph and partitioning approaches. Next, we present results when using the above

v	$T(v)$	$\varrho_\Gamma(v)$
v_1	$\{t_4^1, t_5^1, t_6^1, t_7^1\}$	4
v_2	$\{t_8^1, t_9^1\}$	2
v_3	$\{t_3^0\}$	1
v_4	$\{t_0^2, t_1^2\}$	2
v_5	$\{t_2^2, t_3^2, t_4^2, t_5^2\}$	$3\varrho_\mathrm{I}$ +1
v_6	$\{t_8^2, t_9^2\}$	ϱ_I +1
v_7	$\{t_6^2, t_7^2\}$	ϱ_I +1

(a) List of vertices V.

e	p-DOF	**u**-DOF	φ-DOF	$\sigma_\Gamma(e)$
e_1	2	9	3	14
e_2	4	15	5	24
e_3	3	9	3	15
e_4	3	9	3	15
e_5	5	15	5	25
e_6	5	15	5	25
e_7	6	15	5	26

(b) List of edges E.

Table 4.3: Weights ϱ_Γ and σ_Γ of the graph G_Γ given in Fig. 4.4 when considering the interface depicted in Fig. 4.5.

defined graph partitioning problems to decompose a hierarchy $\mathcal{M}$ of tetrahedral grids among P processes.

4.1.4 Numerical Results

The graph models are investigated for a two-phase flow problem that consists of a spherical phase in a cuboid shaped domain $\Omega^\mathrm{C} = [0,1]^3$ as illustrated in Fig. 4.6. Two problems are investigated which differ in the number of refinement steps in the vicinity of Γ_φ. Applying three or five consecutive refinements leads to different number of tetrahedra to represent Ω^C and, thus, two different problem sizes. The corresponding setups are displayed in Table 4.4 and are denoted by S and L indicating a small and large problem size, respectively. In Table 4.4(a), the number of tetrahedra $|\mathcal{T}_{k-1}|$ on the finest triangulation level and the number of intersected tetrahedra $|\mathcal{T}_\Gamma|$ are given besides the size of the graph. In this section, we consider the Nehalem cluster whose characteristics are detailed in Table 3.2. We chose this cluster because each node provides more memory than the Hapertwon cluster. For each of the two problems, a minimal number of processes P_min is needed to assemble the matrices due to the available memory. As in Sect. 3.3, P_min is a multiple of eight to efficiently utilize all cores of a single compute node. This number is given in the fifth column in Table 4.4(b). Its last column lists the accumulated memory $M_\mathrm{DROPS}(P_\mathrm{min})$ used by P_min processes to store all data of the two-phase flow simulation. This table also shows the number of DOF $n_\mathbf{u}$, n_p, and n_φ needed to represent the velocity, the pressure, and the level set function, respectively. To find an approximate solution of the P-way graph partitioning

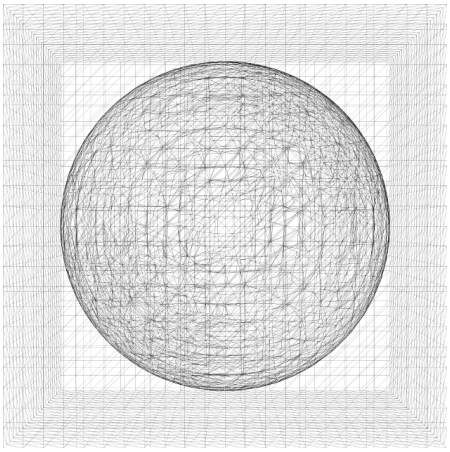


Figure 4.6: Computational domain Ω^{C} and interface Γ_φ.

problems, we employ the library ParMETIS [107]. In particular, we use the function `ParMETIS_V3_AdaptiveRepart` which is a parallel implementation of the so-called "unified repartitioning algorithm" [147]. This function [109] allows to specify an acceptable load imbalance by a parameter ε. This parameter is used to formalize the balance constraint (4.3) of the graph partitioning problem by

$$\sum_{\pi_G(v)=p} \varrho(v) \leq (1+\varepsilon) \cdot \frac{1}{P} \sum_{v \in V} \varrho(v) \quad \text{for } p = 1, \ldots, P. \tag{4.9}$$

The parameter ε can also be seen as a trade-off between the quality of the objective function in (4.2) and the constraints (4.3) of the partitioning problem. For instance, if choosing $\varepsilon = P - 1$, we allow that all vertices are located in one partition and, hence, the edge cut is zero. In the following experiments, this parameter is set to $\varepsilon = 5\,\%$.

Before evaluating the graph models, we empirically estimate the parameter ϱ_{I} that is used to determine the vertex weights ϱ_Γ of the two-phase graph G_Γ in (4.8). Most of the computational work to assemble the matrices in (2.15)–(2.16) is spent in determining A and B. Thus, we focus on these two matrices in this section and measure

(i) the time for determining the nonzeros of A and B using 1 000 tetrahedra located in $\mathcal{T}_\Gamma$. This timing yields 0.8643 s.

(ii) the time for determining the nonzeros of A and B using 1 000 tetrahedra located in $\mathcal{T}_{k-1} \setminus \mathcal{T}_\Gamma$. This timing yields 0.076 s.

These results imply that 1 000 tetrahedra of $\mathcal{T}_\Gamma$ cause 11.37 more computational work than 1 000 tetrahedra of $\mathcal{T}_{k-1} \setminus \mathcal{T}_\Gamma$. This leads to an estimation of the parameter ϱ_{I} by

$$\varrho_{\mathrm{I}} := 11.37. \tag{4.10}$$

Prob.	Ref.	$\lvert\mathcal{T}_{k-1}\rvert$	$\lvert\mathcal{T}_\Gamma\rvert$	$\lvert V\rvert$	$\lvert E\rvert$
S	3	166 164	42 012	38 238	159 540
L	5	2 997 450	667 224	582 678	2 468 820

(a) Number of tetrahedra and size of the graph.

Prob.	$n_{\mathbf{u}}$	n_p	n_φ	$P_{\min}$	$M_{\text{DROPS}}(P_{\min})$ [GB]
S	662 631	28 354	223 783	8	8.4
L	12 012 120	501 627	4 006 946	64	138.4

(b) Number of DOF and memory consumption.

Table 4.4: Characteristics of the two problem setups S and L.

Since the library ParMETIS does not support floating point numbers as weights, we only use integer values for this parameter. We perform the experiments with three different values for ϱ_{I}, namely $\varrho_{\text{I}} \in \{2, 11, 20\}$.

As a first result, we compare the solution of the three graph partitioning problems T-GPTH, D-GPTH, and 11-TP-GPTH, i.e., we employ these problems to determine a data decomposition for the two-phase flow simulations S and L. In particular, we focus on the balance constraints of these problems. To this end, we consider three different balance numbers. First, we define the *partition balance*

$$\alpha_{\text{P}}(P;\varrho) \;:=\; \frac{\max_{p=1,\dots,P} \varrho(V_p)}{\min_{p=1,\dots,P} \varrho(V_p)}, \tag{4.11}$$

where the vertex weighting function ϱ is chosen corresponding to the partitioning problem, e.g., ϱ_{D} when considering the D-GPTH formulation. This balance number compares the weighted size of the graph partitionings and strictly corresponds to the underlying graph partitioning problem. Thus, α_{P} is determined by the graph partitioning algorithm for different choices of the weighting functions. Opposed to this balance number, the next two balance numbers are related to the resulting data decomposition of the implementation in the two-phase flow solver Drops when employing the three different partitioning problems T-GPTH, D-GPTH, and 11-TP-GPTH. We investigate the *tetrahedron balance*

$$\alpha_{\mathcal{T}}(P) \;:=\; \frac{\max_{p=1,\dots,P} \lvert\mathcal{T}_{k-1}^p\rvert}{\min_{p=1,\dots,P} \lvert\mathcal{T}_{k-1}^p\rvert} \tag{4.12}$$

and the *DOF balance*

$$\alpha_{\text{D}}(P) \;:=\; \frac{\max_{p=1,\dots,P} \lvert\mathsf{dof}(p)\rvert}{\min_{p=1,\dots,P} \lvert\mathsf{dof}(p)\rvert}. \tag{4.13}$$

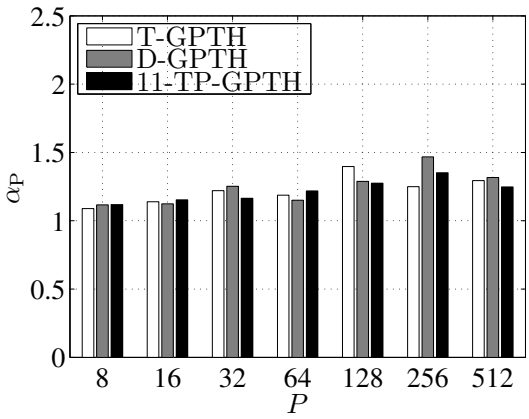
In (4.13), the symbol $\mathsf{dof}(p)$ denotes the set of DOFs stored by process p. For instance, if $\alpha_{\mathcal{T}}(P) = 1.5$ or $\alpha_{\mathrm{D}}(P) = 1.5$ holds, then there is a process that stores 1.5 times more tetrahedra or DOFs than another process, respectively.

Figure 4.7 displays all three balance numbers for all three partitioning problems and both problem sizes. Figures 4.7(a) and (b) show the partition balance number α_{P}. This result only corresponds to the quality of the partitioning library ParMETIS. The greatest imbalance in the partitioning is observed in Fig. 4.7(b) for the D-GPTH where on 256 processes a partition balance number $\alpha_{\mathrm{P}}(256, \varrho_{\mathrm{D}}) > 2$ is detected. The balance of the number of tetrahedra $\alpha_{\mathcal{T}}$ does only indirectly depend on the graph partitioner ParMETIS but depend on the decomposition of the tetrahedral hierarchy. This balance number is illustrated in Fig. 4.7(c) and (d). If using the T-GPTH or 11-TP-GPTH formulation to compute a decomposition of the tetrahedra, the balance $\alpha_{\mathcal{T}}$ is significantly better than this balance number for the D-GPTH problem. Although the T-GPTH is designed to evenly balance the number of tetrahedra among the processes, in most cases, the 11-TP-GPTH gives a better balance of the tetrahedra than the T-GPTH. However, this difference is only small and is probably caused by the heuristic that is used to compute a graph partitioning. In Fig. 4.7(e) and (f), the DOF balance number α_{D} is presented for the problems S and L. For both problem sizes, the D-GPTH problem gives the worst results. In Sect. 4.1.3, we presented the explanation that the vertex weights does not suitably represent the number of DOFs. The numerical results for α_{D} corroborates this explanation. However, the two other graph models, T-GPTH and 11-TP-GPTH, distribute the number of DOFs among the processes much better. Overall, the best results are obtained by the 11-TP-GPTH problem. Therefore, in the rest of this section, we investigate the ϱ_{I}-TP-GPTH problem for $\varrho_{\mathrm{I}} \in \{2, 11, 20\}$ in more detail. However, we refer the reader to [72] for a discussion of the T-GPTH and D-GPTH model.

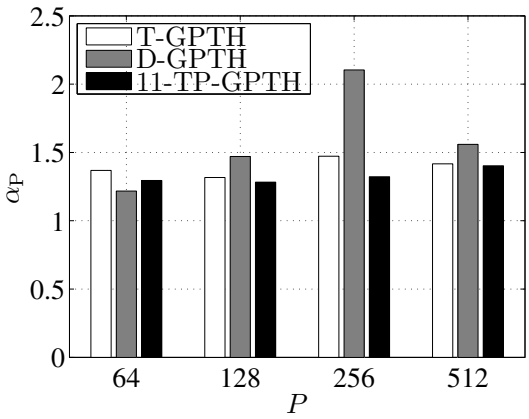
The objective of the ϱ_{I}-TP-GPTH problem is to reduce the storage overhead when representing the discrete two-phase flow problem by $P \geq 2$ processes. This objective is given by the edge cut E_{cut} for the two-phase graph G_{Γ}. Therefore, first, we simultaneously investigate the memory overhead and E_{cut}. For a distributed hierarchy of tetrahedral grids obtained by solving the ϱ_{I}-TP-GPTH problem, we define the *storage overhead* $o(P; \varrho_{\mathrm{I}})$ for P processes by

$$o(P) := \underbrace{\left(\sum_{p=1}^{P} \mathsf{dof}(p) \right)}_{\text{parallel}} - \underbrace{(n_{\mathbf{u}} + n_p + n_{\varphi})}_{\text{sequential}}.$$

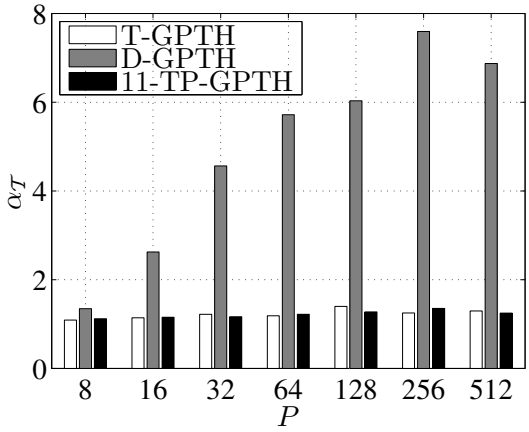
The overhead $o(P)$ denotes the number of DOFs which are additionally needed to represent the finite element functions by P processes compared to represent these functions sequentially. Note that, in comparison to the partition balance



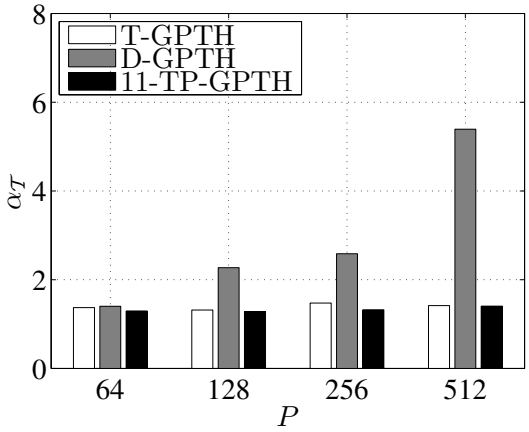
(a) Partition balance for problem S.



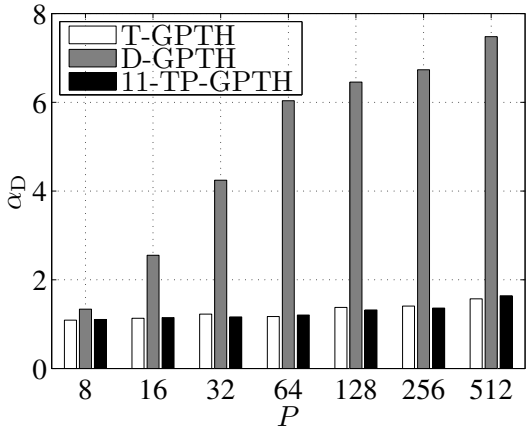
(b) Partition balance for problem L.



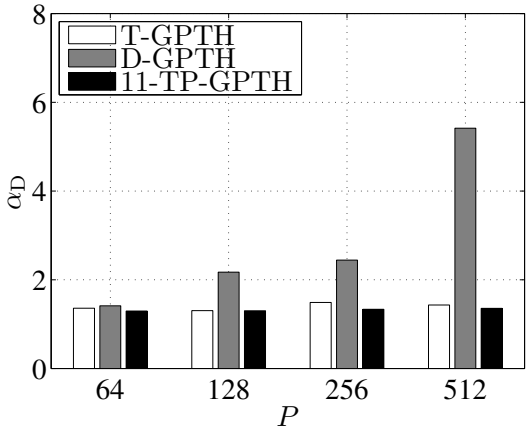
(c) Tetrahedron balance for problem S.



(d) Tetrahedron balance for problem L.



(e) DOF balance for problem S.



(f) DOF balance for problem L.

Figure 4.7: Comparison of balance numbers for the partitioning problems.

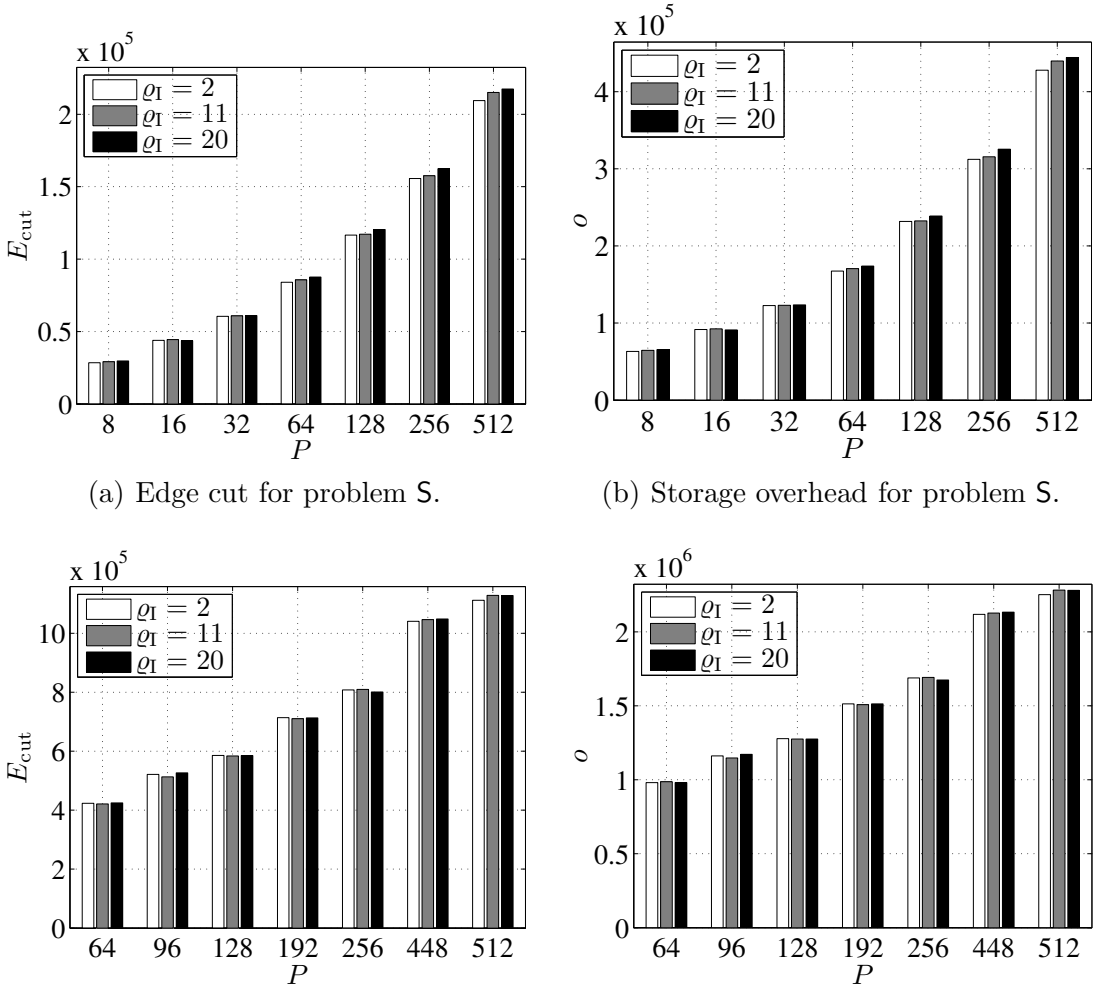
number α_{P} in (4.11), the edge cut E_{cut} corresponds to the graph partitioning library ParMETIS. In contrast—as for the tetrahedral balance $\alpha_{\mathcal{T}}$ in (4.12) and the DOF balance α_{D} in (4.13)—the storage overhead is related to the data distribution obtained from the solution of the ϱ_{I}-TP-GPTH problem. In Fig. 4.8, the edge cut E_{cut} and the storage overhead $o(P)$ are printed side-by-side. Obviously, the edge cut and the storage overhead increase with growing P. In this figure, using $\varrho_{\mathrm{I}} = 11$ for problem L, about $o(512) \approx 2.3 \cdot 10^6$ additional DOFs are stored on 512 processes compared to a sequential data structure. Although this is an overhead of approximately 14 % in comparison to the sequential number of DOFs in Table 4.4, the storage overhead on 512 processes is only about $2.3 \cdot 10^6/512 \approx 4\,500$ DOFs per process which is rather moderate. However, this overhead can be further reduced by allowing a greater intolerance for the balancing contraints of the underlying graph partitioning problem in (4.9). This overhead does not strongly depend on the choice of the parameter ϱ_{I} for both problems and all number of processes. However, we want to point out that $o(P)$ is approximately twice as large as $E_{\mathrm{cut}}(P)$ for all shown cases. This implies that the edge cut is proportional to the storage overhead, indicating that the ϱ_{I}-TP-GPTH problem describes the storage overhead, adequately.

Next, we present results illustrating that the computational work is evenly balanced among the processes in Fig. 4.9. In contrast to Fig. 4.7, we here present results with varying ϱ_{I}. In the style of the partition balance number $\alpha_{\mathrm{P}}(P, \varrho_{\Gamma})$ in (4.11), we define a time balance number. To this end, let $T_{\mathrm{AB}}(p; \varrho_{\mathrm{I}})$ denote the time of process p spent in assembling the matrices A and B when using the solution of the ϱ_{I}-TP-GPTH problem to distribute the tetrahedral hierarchy. Then, the *time balance* for P processes is defined by

$$\alpha_{\mathrm{T}}(P; \varrho_{\mathrm{I}}) \;:=\; \frac{\max_{p=1,\dots,P} T_{\mathrm{AB}}(p; \varrho_{\mathrm{I}})}{\min_{p=1,\dots,P} T_{\mathrm{AB}}(p; \varrho_{\mathrm{I}})}.$$

The two balance numbers α_{P} and α_{T} are depicted side-by-side in Fig. 4.9 for both problem sizes. The partition balance number $\alpha_{\mathrm{P}}(P, \varrho_{\Gamma})$ tends to be smaller for problem S than for problem L. For instance, its maximal value for problem S is given by 1.37 whereas its maximal value is about 1.84 for problem L. In contrast, the time balance number $\alpha_{\mathrm{T}}(P; \varrho_{\mathrm{I}})$ shows a tendency to decrease while going from problem S to problem L. Here, the maximal value of 1.49 for problem S decreases to a maximum of 1.24 for problem L. These results indicate that the computational work is evenly balanced among up to 512 processes for problem L.

Now, we demonstrate that the good load balancing and minimization of the storage overhead also leads to favorable time for assembling the matrices A and B. The timings are displayed in Table 4.5. The estimation of ϱ_{I} in (4.10) suggests that the smallest timings should be achieved by setting $\varrho_{\mathrm{I}} = 11$. For problem L, this holds in most cases. However, there are some timings in Table 4.5 where the execution time for assembling is smaller if choosing $\varrho_{\mathrm{I}} = 2$ or $\varrho_{\mathrm{I}} = 20$. There



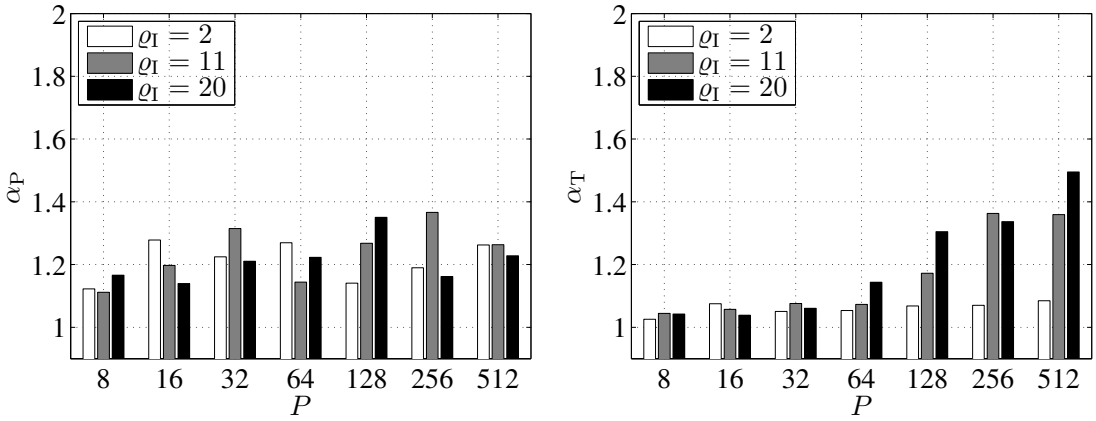
(a) Edge cut for problem S.

(b) Storage overhead for problem S.

(c) Edge cut for problem L.

(d) Storage overhead for problem L.

Figure 4.8: Edge cut of the graph model and storage overhead of the implementation using the ϱ_I-TP-GPTH problem for $\varrho_I \in \{2, 11, 20\}$.



(a) Partition balance number for problem S. (b) Time balance number for problem S.

(c) Partition balance number for problem L. (d) Time balance number for problem L.

Figure 4.9: Partition balance number of the graph model and time balance number of the implementation using the ϱ_I-TP-GPTH problem with the parameter $\varrho_I \in \{2, 11, 20\}$.

Prob.	P	Time [s] $\varrho_{\mathrm{I}} = 2$	$\varrho_{\mathrm{I}} = 11$	$\varrho_{\mathrm{I}} = 20$
S	8	6.216	6.368	6.484
	64	0.7960	0.7613	0.8163
	512	0.1328	0.1405	0.2219
L	64	14.32	14.24	14.34
	256	3.830	3.681	3.677
	512	1.892	1.817	1.972

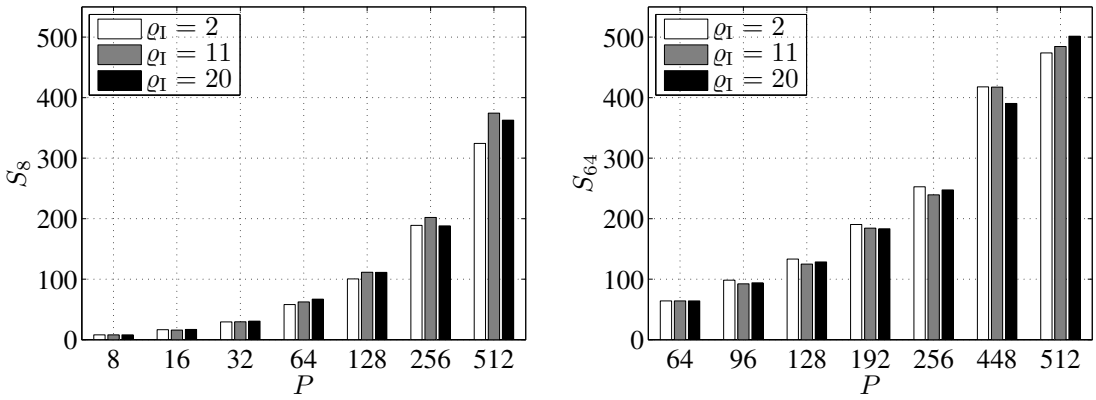
Table 4.5: Time for assembling the matrices A and B using the ϱ_{I}-TP-GPTH problem for $\varrho_{\mathrm{I}} \in \{2, 11, 20\}$.

are two reasonable arguments that help to explain this behavior. First, the parameter ϱ_{I} is estimated by measuring the time for determining the nonzeros of the matrices A and B. The assembly process also includes the time-consuming construction of data structures for these two sparse matrices. This part does not depend on the location of the interface Γ_φ. The second argument is based on investigating the solution of the graph partitioning problem. Suppose that the ϱ_{I}-TP-GPTH problem perfectly describes the PFEA problem. Then there are two observations. First, rather than computing an exact solution to the problem, the library ParMETIS uses a heuristic computing an approximate solution. Second, this library provides a tolerance parameter to specify the imbalance of the partition sizes that is acceptable for a solution. Hence, the graph partitioning problem is solved only approximately leading to load imbalances which, in turn, result in larger execution times. This effect interferes with the study of the parameter ϱ_{I}.

Finally, we present the speedup for assembling the matrices A and B. Recall that at least $P_{\min} = 8$ and $P_{\min} = 64$ processes are needed to store the data for problems S and L, respectively. Hence, we define the speedup for P processes as in Sect. 3.3 by

$$S_{P_{\min}}(P; \varrho_{\mathrm{I}}) := \frac{T(P_{\min}; \varrho_{\mathrm{I}})}{T(P; \varrho_{\mathrm{I}})} \cdot P_{\min}.$$

That is, the speedup for P processes $S_{P_{\min}}(P; \varrho_{\mathrm{I}})$ is relatively defined with respect to the time $T(P_{\min}; \varrho_{\mathrm{I}})$. This speedup does not compare the execution times for different choices of the vertex weight parameter ϱ_{I}. Hence, the speedup does not strongly depend on this parameter. This effect is observed in Fig. 4.10 where the speedup $S_{P_{\min}}(P; \varrho_{\mathrm{I}})$ is depicted for both problems S and L and for all parameters $\varrho_{\mathrm{I}} \in \{2, 11, 20\}$. In all cases, the assembly scales on up to 512 processes. For



(a) Speedup for problem S.

(b) Speedup for problem L.

Figure 4.10: Speedup of assembling the matrices A and B using the ϱ_I-TP-GPTH problem for $\varrho_I \in \{2, 11, 20\}$.

problem S, a speedup of $S_8(512, 11) = 374.4$ is observed and, for the larger problem L, a noticeable speedup of $S_{64}(512; 11) = 484.5$ is achieved which correspondes to a parallel efficiency of 73.1 % and 94.6 %, respectively.

4.2 Modeling the Tetrahedral Hierarchy by Hypergraphs

In this section, we model the triangulation hierarchy by hypergraphs rather than by standard graphs as in the previous section. Using this type of graphs allows us to formulate a hypergraph partitioning problem that exactly describes the communication volume among neighbor processes when performing linear algebra operations on the distributed numerical data introduced in Sect. 3.2.

Hypergraph models have successfully been employed in various computational combinatorial fields, e.g., Very Large Scale Integration (VLSI) [4, 34], database storage and data mining [40, 131], and mesh reordering [157]. In particular, hypergraph models are used to partition data when considering sparse matrix-vector products [38, 90, 173, 174] and their application in preconditioned iterative methods [172]. For instance, the drawback of the standard graph model is that these models commonly only approximate the amount of data that is transferred during one sparse matrix-vector product. In contrast, the advantage of hypergraph models consists of exactly modeling the amount of transferred data. In this section, we extend the ideas of the hypergraph model for sparse matrix-vector multiplications to represent a hierarchy of triangulations.

This section is organized as follows. In Sect. 4.2.1, we first define hypergraphs and formulate the P-way hypergraph partitioning problem corresponding to the P-way graph partitioning problem for standard graphs. Then, in Sect. 4.2.2, we model the problem of finding a distribution of the hierarchy of triangulations by a hypergraph partitioning problem. Hereby, we state the main result of this section that the hypergraph model is capable of exactly representing the communication volume and the storage overhead. We conclude this section with numerical results demonstrating that the novel hypergraph model is capable of saving communication volume and also communication time for linear algebra operations.

4.2.1 Hypergraph Partitioning Problem

A hypergraph is a generalization of a graph and can be expressed by the quadruple $H = (V, N, \varrho, \sigma)$. Here, the set V denotes the vertices and $N \subset \mathcal{P}(V)$ the *nets* of the hypergraph H. Each net $n \in N$ connects a set of vertices, where the vertices $v \in n$ are called *pins*. Note that, opposed to standard graphs, each net can express adjacencies between more than one or two vertices. In the style of graphs, the vertices and the nets are weighted by the functions ϱ and σ, respectively. In this section, we consider multiple weights on the nets, i.e., $\sigma : N \to (\mathbb{R}^+)^m$, $m > 1$. For a hypergraph H, a *P-way hypergraph partitioning function* $\pi_H : V \to \{1, \ldots, P\}$ decomposes the set of vertices V into $P \geq 2$ disjoint subsets $V_1, \ldots, V_P$. The P sets $V_p = \{v \in V \mid \pi_H(v) = p\}$, with $1 \leq p \leq P$, fulfill the two conditions

$$\bigcup_{p=1}^{P} V_p = V \quad \text{and} \quad V_p \cap V_q = \emptyset \quad \text{for} \quad p \neq q.$$

Opposed to standard graphs, a single net can connect multiple parts. This is expressed by the *connectivity* λ of a net $n \in N$ which is defined by

$$\lambda(n) \;:=\; \Big| \bigcup_{v \in n} \{p \in \{1, \ldots, P\} \mid \pi_H(v) = p\} \Big|,$$

i.e., $\lambda(n)$ denotes the number of parts which are connected by n.

As a counterpart of the edge cut E_{cut} defined in (4.2) for standard graphs, we introduce the cut size N_{cut} for a partitioned hypergraph H. Various approaches are available to model N_{cut}. These approaches can be generalized by

$$N_{\text{cut}} \;:=\; \sum_{n \in N} \sigma(n)\, c(\lambda(n)), \tag{4.14}$$

where the *cost function* $c : \mathbb{N} \to \mathbb{N}_0$ is used to model the costs for a net $n \in N$ that connects different parts. Note that N_{cut} is an m-dimensional vector when

considering m-dimensional weights on the nets. For a general cost function c in (4.14), we assume two conditions. First, if all pins of a net n are located in one partition, i.e., $\lambda(n) = 1$, the cost function equals zero, in formula $c(1) = 0$. Second, if a net n connects more parts of the partition then another net n', then the value of the cost function for n is larger than the value corresponding to n'. Hence, the cost function c is a monotonically increasing function, i.e.,

$$\lambda(n) > \lambda(n') \;\Rightarrow\; c(\lambda(n)) \geq c(\lambda(n')) \quad \text{for all } n, n' \in N.$$

In the literature, the cost function c is commonly determined either as the *cut-net metric*

$$c_{\text{cn}} : \lambda(n) \mapsto \begin{cases} 0 & ,\lambda(n) = 1 \\ 1 & ,\lambda(n) \geq 2 \end{cases} \tag{4.15}$$

or as the *connectivity−1 cut metric*

$$c_{\text{C}-1} : \lambda(n) \mapsto \lambda(n) - 1. \tag{4.16}$$

The cut-net metric does not consider the number of parts connected by the nets when evaluating the cut size in (4.14). Therefore, the connectivity−1 cut metric weights each net n by $\lambda(n) - 1$ which commonly better models the communication volume, i.e., this cost function exactly models the communication volume for one sparse matrix-vector product. However, in Sect. 4.2.2, we introduce another cost function that is well suited for modeling a tetrahedral hierarchy.

With these definitions, we can state the problem of finding an "optimal" P-way hypergraph partitioning function.

P-way hypergraph partitioning

Let $H = (V, N, \varrho, \sigma)$ be a given weighted hypergraph and $c : \mathbb{N} \to \mathbb{N}_0$ be a cost function. Then, for $P \geq 2$, the P-way hypergraph partitioning problem consists of finding a P-way hypergraph partitioning function $\pi_H : V \to \{1, \ldots, P\}$ such that

$$\textit{minimize} \quad N_{cut} := \sum_{n \in N} \sigma(n)\, c(\lambda(n)), \tag{4.17}$$

$$\textit{s.t.} \quad \sum_{\pi_H(v) = p} \varrho(v) \approx \frac{1}{P} \sum_{v \in V} \varrho(v) \quad \textit{for } p = 1, \ldots, P. \tag{4.18}$$

Here, minimizing N_{cut} in (4.17) corresponds to finding a minimal edge cut E_{cut} in the P-way graph partitioning problem, cf. (4.2). The constraints in (4.18) are equal to the constraints in the partitioning problem for standard graphs in (4.3).

As for standard graphs, this problem is called multi-objective [1, 150] or multi-constrained [9,106] if multiple weights of nets or vertices are defined, respectively. In the following, we will set up a multi-objective, single-constrained partitioning problem to model the decomposition of a tetrahedral hierarchy. However, in this thesis, we do not consider methods to solve multi-objective optimization problems.

Note that—for some definitions of the cost function c and balancing constraints in (4.18)—the P-way hypergraph partitioning problem [113] and its variants [118] are NP-hard. Therefore, this problem is generally solved by heuristics or approximation algorithms. Examples of serial heuristics are given in [4, 63, 108] whereas a parallel heuristic can be found in [168].

4.2.2 Hypergraph Model

Next, we formulate a hypergraph $H_{\mathcal{M}} = (V, N, \varrho_{\mathrm{H}}, \sigma_{\mathrm{H}})$ and a corresponding partitioning problem that aim at determining a suitable decomposition of a tetrahedral hierarchy $\mathcal{M}$ among P processes. Recall that the objective of such a model is to distribute the tetrahedron families among the processes such that the resulting communication volume is minimized and the computational load is balanced. In Sect. 3.2, we described that neighbor communication occurs when transforming a vector from a distributed into an accumulated representation. We exactly model this neighbor communication by first defining the vertices and nets of a hypergraph $H_{\mathcal{M}}$. Afterwards, we introduce a suitable cost function c for the cut size in (4.14), and weight the vertices and nets by suitable weighting functions ϱ_{H} and σ_{H}, respectively. The result is a P-way hypergraph partitioning problem that exactly models the volume of neighbor communication when transforming a distributed vector into an accumulated representation.

Since we seek for a partitioning model to distribute tetrahedron families, we model the vertices V of $H_{\mathcal{M}}$ in the same way as for the graph $G_{\mathcal{M}}$ in Sect. 4.1. Thus, the vertices V of the hypergraph are defined by (4.4), i.e., the vertices of the hypergraph represents the same tetrahedron families as those of the standard graph models. In this section, we also employ the mapping $T(v)$ of (4.5) to denote the tetrahedra of the finest triangulation that are represented by a vertex v. The vertices can be weighted by all vertex weighting functions which are presented in Sect. 4.1 for the standard graphs.

To introduce the nets, recall that DOFs, which are located at process boundaries, cause a storage overhead and neighbor communication when accumulating a vector, i.e., transforming a distributed vector into an accumulated vector. In this neighbor communication, each process that stores a distributed DOF exchanges its local values of the DOF with adjacent processes. In the remainder, let $C^{p \to q}(\mathbf{x})$ denote the number of DOFs that process p sends to process q during the accumulation of the vector $\mathbf{x}$. Then, the total communication volume for accumulating

the vector $\mathbf{x}$ by P processes is defined by

$$C^{\text{total}}(\mathbf{x}, P) \;:=\; \sum_{p=1}^{P} \sum_{q=1}^{P} C^{p \to q}(\mathbf{x}). \tag{4.19}$$

We use this consideration of transferred DOFs to define the nets N and the weighting function σ_{H} of the hypergraph $H_{\mathcal{M}}$ as well as for defining a cost function c. For a given hierarchy of triangulations $\mathcal{M}$, let $\mathcal{V}$ denote the set of triangulation vertices and edges where DOFs are located. Then, for each $u \in \mathcal{V}$, a corresponding net $n_u \in N$ is introduced. The net $n_u \subset V$ connects these hypergraph vertices that represent tetrahedra which have the DOF located at u in common. This results in $|\mathcal{V}|$ nets. We arrive at the formal definition of the net n_u for $u \in \mathcal{V}$ by

$$n_u \;:=\; \{v \in V \;\mid\; u \text{ is located at a tetrahedron of } T(v)\}\,.$$

Since for each $u \in \mathcal{V}$ a net is introduced, the set of nets N is defined by

$$N \;:=\; \{n_u \subset V \;\mid\; u \in \mathcal{V}\}\,.$$

To define a suitable cost function c, consider a DOF located at $u \in \mathcal{V}$ and the corresponding net n_u whose pins are located in $\lambda(n_u)$ partitions. For a triangulation, this means that the DOF located at u is distributed among $\lambda(n_u)$ processes. When transforming this DOF to an accumulated representation, each of the $\lambda(n_u)$ processes sends its local DOF value to $\lambda(n_u) - 1$ adjacent processes. Overall, the DOF at u causes $\lambda(n_u) \cdot (\lambda(n_u) - 1)$ information packages which need to be transferred. This observation results in the definition of the cost function

$$c_{\mathcal{M}} : \lambda(n) \mapsto \lambda(n) \cdot (\lambda(n) - 1)\,. \tag{4.20}$$

Note that if a net is cut into more than two parts, this novel cost function penalizes this net more than the cut-net or the connectivity-1 metric, defined in (4.15) and (4.16). That is, we state for each net $n \in N$

$$\lambda(n) > 2 \quad \Rightarrow \quad c_{\mathcal{M}}(\lambda(n)) > c_{\text{C}-1}(\lambda(n)) > c_{\text{cn}}(\lambda(n)).$$

For evaluating the weight of a net, we consider the number of velocity, pressure and level set DOFs located at $u \in \mathcal{V}$ which are denoted by $\mathsf{dof}_{\mathbf{u}}(u)$, $\mathsf{dof}_{p}(u)$ and $\mathsf{dof}_{\varphi}(u)$, respectively. These notations allows us to exactly state the length of each of the $c_{\mathcal{M}}(\lambda(n_u))$ information packages that are transferred. If we accumulate the, e.g., velocity DOFs at u, each adjacent process sends $\mathsf{dof}_{\mathbf{u}}(u)$ local DOFs to its neighbors. Hence, we weight the net n_u by the number of DOFs which reside at u. Since we have three different types of DOFs to represent a two-phase flow problem, we define the weighting function for the nets by

$$\sigma_{\text{H}} : N \to (\mathbb{R}^{+})^3, \quad n_u \mapsto \begin{pmatrix} \mathsf{dof}_{\mathbf{u}}(u) \\ \mathsf{dof}_{p}(u) \\ \mathsf{dof}_{\varphi}(u) \end{pmatrix}.$$

So, the cost function $c_{\mathcal{M}}$ models the number of information packages among different processes whereas the weight σ_{H} determines the length of each of the packages. Thus, we finally arrive at the definition of the cut size by

$$N_{\mathrm{cut}} \;:=\; \sum_{u\in\mathcal{V}} \begin{pmatrix} \mathsf{dof}_{\mathbf{u}}(u) \\ \mathsf{dof}_{p}(u) \\ \mathsf{dof}_{\varphi}(u) \end{pmatrix} \lambda(n_u)\,(\lambda(n_u)-1)\,. \tag{4.21}$$

Due to the above considerations, we state the main result of this section that this hypergraph model exactly expresses the total communication volume during the simulation of a two-phase flow problem.

Cut size equals total communication volume

Let N_{cut} in (4.21) *be the cut size of a P-way partitioned hypergraph $H_{\mathcal{M}}$. Moreover, let $C^{total}(\cdot, P)$ be the total communication volume defined in* (4.19)*. Then,*

$$N_{cut} \;=\; \begin{pmatrix} C^{total}(\overline{\mathbf{u}}, P) \\ C^{total}(\overline{p}, P) \\ C^{total}(\overline{\varphi}, P) \end{pmatrix} \tag{4.22}$$

holds, where $C^{total}(\overline{\mathbf{u}}, P)$, $C^{total}(\overline{p}, P)$, and $C^{total}(\overline{\varphi}, P)$ denote the total communication volume for accumulating the velocity $\overline{\mathbf{u}}$, pressure $\overline{p}$ and level set $\overline{\varphi}$ vector representation, respectively.

Next, we present an illustrating example for a hypergraph. To this end, we model the triangulation hierarchy in Fig. 2.3 and Fig. 4.2 by a hypergraph. For the sake of presentation, we only consider the scalar-valued P_1^{Γ} extended finite element function for the pressure, whose DOFs reside at the vertices of the triangulation. Figure 4.11 depicts a numbering of the 14 pressure DOFs that is also employed to number the nets of the corresponding hypergraph $H_{\mathcal{M}}$. The triangulation vertices, that contain an extended DOF to represent the pressure discontinuity at the interface, are highlighted by the symbol $\circ$. The corresponding hypergraph is presented in Fig. 4.12. The seven hypergraph vertices are symbolized by a circle and the 14 nets by squares. We observe that the hypergraph consists of the same vertices as the standard graph in Fig. 4.4. Table 4.6 summarizes the nets of the hypergraph which correspond to the numbering of the 14 pressure DOFs. In the second column of that table, the pins of the nets n_u are given for all DOFs located at the triangulation vertices $u \in \mathcal{V}$. For example, consider the DOF which is located at the triangulation vertex numbered by 5. This DOF is adjacent to the tetrahedra t_0^2, t_2^2 and t_8^2 which are represented by v_4, v_5 and v_6, respectively. Thus, the set of pins of the net n_5 is determined by $\{v_4, v_5, v_6\}$. Additionally, the third column of Table 4.6 shows the weight $(\sigma_{\mathrm{H}}(n_u))_2 = \mathsf{dof}_p(u)$ of each net n_u,

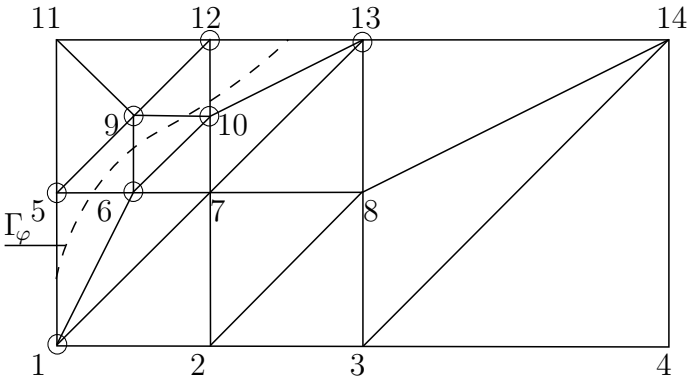


Figure 4.11: Numbering of the DOFs of a P_1^Γ finite element function on the finest triangulation $\mathcal{T}_2$ and interface Γ_φ depicted in Fig. 4.5.

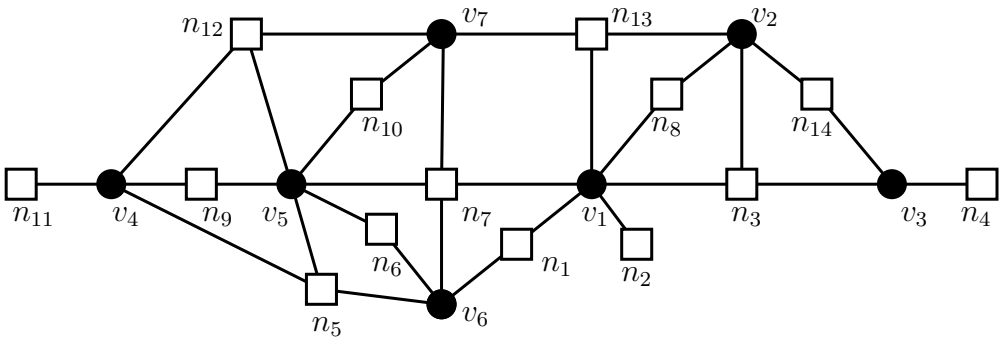


Figure 4.12: Hypergraph $H_\mathcal{M}$ representing the triangulation hierarchy $\mathcal{M}$ given in Fig. 2.3 when using the DOF numbering in Fig. 4.11.

if we solely take the pressure DOF into account. For instance, the weight of the net n_5 is evaluated by $(\sigma_\mathrm{H}(n_5))_2 = 2$ because the corresponding scalar pressure DOF is extended. To present an example of determining the cut size, assume that the pins v_4, v_5 and v_6 of the net n_5 are located at three different processes p_1, p_2 and p_3, respectively. Thus, the connectivity of n_5 is given by $\lambda(n_5) = 3$. For accumulating the corresponding DOF, each of the three processes sends its local values to its two neighbors. Hence, $3 \cdot 2 = \lambda(n_5)\,(\lambda(n_5) - 1) = 6$ message packages exist. Since this pressure DOF is extended, each message package consists of two values. Overall, the communication volume caused by the DOF at the triangulation vertex numbered by 5 is evaluated as

$$\underbrace{2}_{(\sigma_\mathrm{H}(n_5))_2} \cdot \underbrace{(3 \cdot (3-1))}_{c_\mathcal{M}(\lambda(n_5))} = 12.$$

We conclude this section by stating the problem of the *hypergraph partitioning*

$u \in \mathcal{V}$	n_u	$(\sigma_{\mathrm{H}}(n_u))_2$
1	$\{v_1, v_6\}$	2
2	$\{v_1\}$	1
3	$\{v_1, v_2, v_3\}$	1
4	$\{v_3\}$	1
5	$\{v_4, v_5, v_6\}$	2
6	$\{v_5, v_6\}$	2
7	$\{v_1, v_5, v_6, v_7\}$	1
8	$\{v_1, v_2\}$	1
9	$\{v_4, v_5\}$	2
10	$\{v_5, v_7\}$	2
11	$\{v_4\}$	1
12	$\{v_4, v_5, v_7\}$	2
13	$\{v_1, v_2, v_7\}$	2
14	$\{v_2, v_3\}$	1

Table 4.6: Nets N of the hypergraph $H_{\mathcal{M}}$ presented in Fig. 4.12.

for a tetrahedral hierarchy (HPTH).

Hypergraph Partitioning for a Tetrahedral Hierarchy (HPTH)

Given a hierarchy $\mathcal{M}$ of triangulations and the corresponding weighted hypergraph $H_{\mathcal{M}} = (V, N, \varrho_H, \sigma_H)$. Then, the HPTH problem consists of finding a P-way hypergraph partitioning function $\pi_H : V \to \{1, \ldots, P\}$ that solves the P-way hypergraph partitioning problem for $H_{\mathcal{M}}$.

Note, that the corresponding P-way hypergraph partitioning problem is a multi-objective, single-constrained optimization problem.

4.2.3 Numerical Results

We now present results when using the HPTH problem to decompose the set of tetrahedra among processes. There mainly exist the following libraries which are capable of finding an approximate solution of a hypergraph partitioning problem:

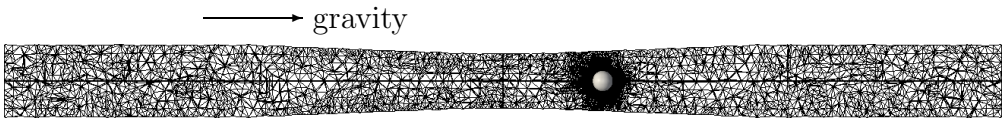


Figure 4.13: Vertical slice through the computational domain Ω^{M} and a drop described by φ.

PaToH [37, 38]; hMetis [108]; Mondriaan [174] (for sparse matrices); MLpart [35] (for circuits); and PHG [53]. However, these libraries do not provide a functionality that is capable of solving the hypergraph partitioning problem HPTH defined in Sect. 4.2.2. There are two reasons. First, the HPTH problem is a multiobjective problem and none of the above libraries support this type of problems. Second, we use a novel, self-defined cost function $c_{\mathcal{M}}$ in (4.20) to determine the communication when cutting a net into multiple parts. However, these libraries only provide the cut-net metric (4.15) or the connectivity-1 cut metric (4.16).

In a collaboration with Rob Bisseling and Bas Fagginger Auer at Utrecht University, we are developing a method and an implementation to decompose a tetrahedral hierarchy $\mathcal{M}$ obtained by Drops among processes using the HPTH approach. The corresponding P-way hypergraph partitioning problem with the novel cost function $c_{\mathcal{M}}$ is solved by a heuristic which is implemented in the library Mondriaan. This preliminary work is—right now—not capable of considering user defined weighting functions ϱ_{H} for the vertices and σ_{H} for the nets and, in particular multidimensional weights for the nets. We thus weight all vertices and nets by one, i.e., $\varrho_{\mathrm{H}} \equiv 1$ and $\sigma_{\mathrm{H}} \equiv 1$. We allow the same imbalance of $\varepsilon = 5\,\%$ for the balancing constraint in (4.18) for the P-way hypergraph partitioning problem as for the graph partitioning problems, cf. (4.9). We investigate the quality and the performance of a hypergraph partitioning when transforming a vector $\mathbf{x} \in \mathbb{R}^n$ from its distributed representation into its accumulated representation. This transformation includes the only neighbor communication when performing linear algebra operations in Drops. The DOFs represented by $\mathbf{x}$ reside at vertices and edges of a triangulation hierarchy that discretizes the computational domain Ω^{M}. This domain is schematically illustrated in Fig. 4.13. The figure shows a vertical slice through the domain Ω^{M} and the position of a spherical drop. The domain Ω^{M} describes an hourglass-shaped measurement cell of 10 cm length and a maximal diameter of 0.72 cm. The domain Ω^{M} is an optimized measurement cell used to investigate levitated drops, see [88]. Refining the coarsest triangulation three or four times results in hierarchies of triangulations with $k = 4$ and $k = 5$ levels and, thus, in two different problem sizes which are detailed in Table 4.7.

We compare the decomposition of a tetrahedral hierarchy obtained by the HPTH problem with a decomposition obtained by the T-GPTH problem that is solved by ParMETIS. To this end, we present three different metrics: first, we consider the total communication volume that occurs when transforming the distributed representation into the accumulated representation of $\mathbf{x}$; second, we analyze the

Prob.	k	$\lvert\mathcal{T}_{k-1}\rvert$	n	$\lvert V\rvert$	$\lvert N\rvert$
S	4	74 354	101 462	16 247	101 462
L	5	411 423	551 425	66 616	551 425

Table 4.7: Size of problems used to investigate the HPTH.

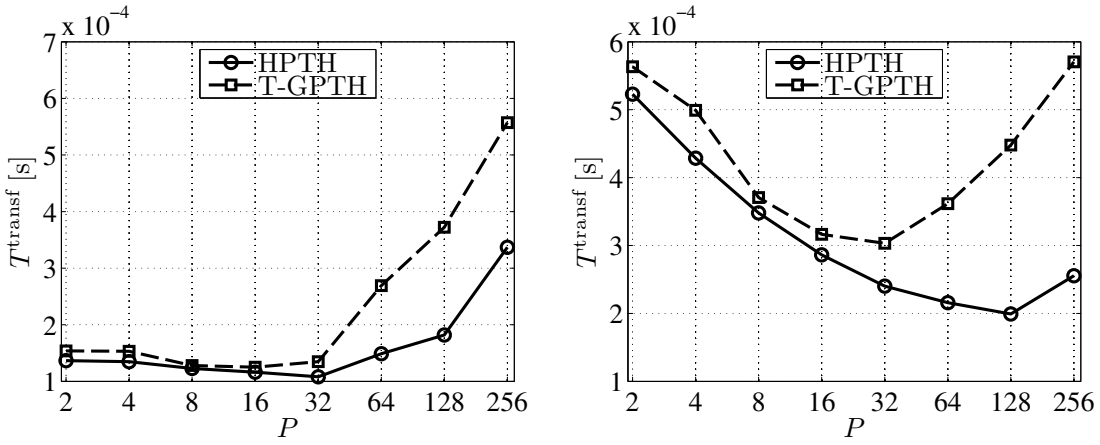
Prob.	Partit. prob.	P 2	4	8	16	32	64	128	256
S	T-GPTH	5 152	11 310	17 852	28 236	40 602	57 996	81 992	175 946
S	HPTH	4 836	9 762	16 310	24 982	38 174	55 504	78 976	112 356
L	T-GPTH	15 326	33 718	48 112	82 626	119 210	171 030	236 984	315 652
L	HPTH	14 250	27 366	45 988	72 692	109 294	160 032	217 088	305 530

Table 4.8: Total communication volume $C^{\text{total}}(\mathbf{x}, P)$ for transforming a distributed vector into an accumulated one.

time for this transformation; and third, we investigate the number of messages during this operation.

The objective of the hypergraph formulation is to minimize the total communication volume $C^{\text{total}}(\mathbf{x}, P)$. Therefore, we present this volume for both problems and various numbers of processes in Table 4.8. The results demonstrate that all decompositions of the tetrahedral hierarchies obtained by the novel HPTH problem cause less total communication volume than the corresponding decompositions obtained by the standard graph-based T-GPTH formulation. For instance, the communication volume $C^{\text{total}}(\mathbf{x}, 4)$ for problem S on four processes is reduced by approximately 13.7 % and for problem L by approximately 18.8 %. The smallest reduction is observed for problem L on 256 processes where the communication volume decreases by 3.2 % while the largest reduction is achieved for S on 256 processes and accounts for 36.1 %.

Next, we show that a reduced total communication volume also decreases the time for transforming a distributed representation into an accumulated representation of a vector. The time for this transformation is dominated by the time for the neighbor communication. Note that a reduced communication volume must not necessarily yield smaller transformation times because latency also contributes to the communication time [48]. We analyze the transformation time on a Nehalem type cluster by performing 1 000 accumulations of a distributed



(a) Problem S (b) Problem L

Figure 4.14: Time $T^{\text{transf}}(P)$ for transforming a distributed vector into an accumulated one.

vector. Figure 4.14 displays the average time $T^{\text{transf}}(P)$ of one transformation. The timings show the same behavior for all cases. First, the time is reduced for a growing number of processes. Afterwards, the time increases when increasing the number of processes further. If determining a decomposition of the tetrahedral hierarchy by the T-GPTH problem, the minimal transformation time T^{transf} is obtained by 16 processes for problem S and by 32 processes for L. The corresponding minimal timing for a decomposition of tetrahedra determined by the HPTH problem is observed for 32 and 128 processes for S and L, respectively. Thus, $T^{\text{transf}}(P)$ with respect to the HPTH problem scales to a higher number of processes than the time $T^{\text{transf}}(P)$ for the T-GPTH problem. The greatest difference in the transformation time is noticed for problem L on 256 processes, i.e., instead of $T^{\text{transf}}(P)$ =0.57 ms only $T^{\text{transf}}(P)$ =0.26 ms are spent in one transformation. Here, the hypergraph formulation is capable of more than bisecting the transformation time.

The reduction in the transformation time is primarily caused by minimizing the communication volume and not by reducing the number of messages which are sent among processes. This is corroborated by Table 4.9 where the numbers of messages are presented for all transformations. For instance, although the number of messages determined by the HPTH problem for L on 64 processes is larger than the corresponding number for the T-GPTH problem, the transformation time is significantly smaller in Fig. 4.14. However, there is no clear tendency which partitioning model yields smaller numbers of messages. For instance, the number of messages in problem S is the same for both partitionings when we consider 2, 4, or 64 processes. Since neither the HPTH nor the T-GPTH formulation addresses the reduction of number of messages, this observation is reasonable.

Prob.	Partit. prob.	P 2	4	8	16	32	64	128	256
S	T-GPTH	2	12	50	154	396	922	2 176	4 910
S	HPTH	2	12	48	174	392	922	2 020	4 590
L	T-GPTH	2	12	48	158	402	926	2 042	4 388
L	HPTH	2	12	50	146	412	952	1 988	4 312

Table 4.9: Number of messages for transforming a distributed vector into an accumulated one.

4.3 Discussion

In this chapter, we introduced various models to determine a decomposition of tetrahedral hierarchies. In Table 4.10, we present a summary of the graph models and the corresponding partitioning problems to determine a decomposition of tetrahedral hierarchies. All of these models result in good performance when distributing the tetrahedra for a two-phase flow simulation. The major reason is that these novel models distinguish between computational loads caused by different types of tetrahedra. These differences will even increase in the future. For instance, if surfactants are also considered an additional equation on the interface between both phases will be introduced. Obviously, solving these equations increases the computational load of the tetrahedra intersected by the interface and, thus, the difference in the computational load between different tetrahedron types even grows. Since we parameterized the difference in the computational load with respect to different types of tetrahedra, the models in this section are already capable of handling these future requirements. Besides balancing the computational load, the novel models aim at minimizing the storage overhead and communication for parallel simulations. To this end, the introduced two-phase graph and hypergraph partitioning problem uses information of the degrees of freedom to address this objective.

However, there is room for improvements. In the field of high-performance computing, the time for transferring messages is modeled by the latency, the bandwidth, and the amount of transferred data [48]. That is, if one process sends information to another processes, the communication time is determined by setting up the communication and the amount of transferred data divided by the bandwidth. On recent high-performance computers, the latency dominates the communication time for moderate message lengths. However, all of the presented graph and hypergraph models aim at minimizing the length of the messages and

do not consider the latency. Therefore, future research is required that considers both, the message length and the latency. One possible approach may be to minimize the number of neighbor processes. This results in a reduced number of messages and, hence, aims at minimizing the number of communications, i.e., latency. Furthermore, the models neglect the topology of processors when determining a decomposition of the computational work. For instance, in the underlying cluster of processors, there may be differences in the number of compute cores of the processors or in the connection between the nodes. Right now, this is not considered by the models. However, this becomes even more important when taking a hybrid distributed-/shared-memory parallelization into account. Therefore, future research work should be devoted to map the topology of the processors into the graph partitioning problems.

We introduced an innovative hypergraph partitioning model to determine a decomposition of the tetrahedra. The advantage of this model over the standard graph models consists in its capability of describing neighbor communication on numerical data, exactly. Although this novel model aims at distributing the tetrahedra among the processes, it contains all necessary information to express the neighbor communication involved in linear algebra operations. We implemented the hypergraph partitioning formulation in a collaboration with Rob Bisseling and Bas Fagginger Auer at Utrecht University. The numerical results demonstrate that the hypergraph model is superior to a standard graph model when considering the transformation of a distributed vector into an accumulated vector. Since this is the only neighbor communication involved in linear algebra operations, this new approach seems to be superior to the standard graph models. Ongoing work addresses the partitioning of a tetrahedral hierarchy specifically designed for two-phase flow problems. The corresponding two-phase flow simulations includes different neighbor-communication patterns, e.g., transforming a velocity vector yield a different pattern than transforming a pressure vector. This results in a multi-objective hypergraph partitioning problem which is topic of the ongoing work.

	tetrahedral graph $G_{\mathcal{M}}$	DOF graph G_{D}	two-phase graph G_{Γ}	hypergraph $H_{\mathcal{M}}$
Vertices	tetrahedron families	see $G_{\mathcal{M}}$	see $G_{\mathcal{M}}$	see $G_{\mathcal{M}}$
Edges	common faces	see $G_{\mathcal{M}}$	see $G_{\mathcal{M}}$	DOF information
Vertex weight	number of tetrahedra on finest level	number of DOFs	number of intersected tetrahedra	see $G_{\mathcal{M}}$, G_{D}, and G_{Γ}
Edge weight	number of common faces	see $G_{\mathcal{M}}$	number of DOFs	DOF information
Objective of the corresponding partition problem	minimize number of faces at process boundaries	see $G_{\mathcal{M}}$	minimize storage overhead	minimize total communication volume
Constraint of the corresponding partition problem	balance number of tetrahedra on finest level	balance number of DOFs	balance computational work for assembly	see $G_{\mathcal{M}}$, G_{D}, and G_{Γ}

Table 4.10: Comparison of the graph models and their corresponding partitioning problems.

5 Re-Initializing Level Set Functions

When solving the partial differential equations (PDEs) which correspond to the level set approach for two-phase flows, it is numerically essential that the level set function is smooth in the vicinity of the interface. In particular, retaining the level set function "close" to a signed distance function with respect to the interface is advantageous [44] for numerical methods determining the interface and approximating physical phenomena in the vicinity between two phases. However, the signed distance property of level set functions is typically not preserved when evolving in simulation time; even if a two-phase flow simulation is set up with a signed distance function as initial level set function.

In the context of two-phase flow problems, various methods have been developed or adapted to keep the underlying level set function close to a signed distance function. In particular, in [70], we have recently published and analyzed a parallel method which is capable of efficiently recover the signed distance property of level set functions on an unstructured grid using a distributed-memory parallelization approach. In contrast to other standard serial re-initialization techniques, our approach is based on direct distance computations. Our novel method has been advanced in [69] in order to exploit recent high-performance computer architectures consisting of clusters of multi-core processors. We want to point out that, to our knowledge, there is no other algorithm in the literature available for efficiently re-initializing level set functions on distributed, unstructured grids.

This chapter is an extended version of our articles [70] and [69] where the novel re-initialization algorithm based on direct distances was introduced. This chapter is organized as follows. In Sect. 5.1, the signed distance property is defined and an illustrating example is given that highlights the advantage of this property. This section additionally contains an overview of the open literature concerning the re-initialization of level set functions. In Sect. 5.2, the serial version of the re-initialization algorithm is presented. The parallel algorithm, both distributed- and hybrid distributed-/shared-memory parallel, are described in Sect. 5.3. Afterwards, the algorithm is theoretically analyzed in Sect. 5.4. Section 5.5 focuses on a detailed numerical investigation of the novel re-initialization algorithm based on direct distances. Finally, Sect. 5.6 concludes this chapter by a discussion and an outlook.

5.1 Preserving the Signed Distance Property

In this section, we first introduce a mathematical formulation of the signed distance property. Then, we explain why this property is advantageous for numerical methods determining the interface. Finally, we conclude this section by an overview of the open literature on methods that re-construct the signed distance property for level set functions.

5.1.1 The Signed Distance Property

We clarify the term "signed distance function." A level set function φ is called a *signed distance function* for a given interface Γ_φ in the computational domain Ω, if it satisfies the following two properties:

(i) The sign of φ bisects Ω into two domains Ω_1 and Ω_2, i.e.,

$$\begin{aligned} \varphi(\mathbf{x}) < 0 \quad &\Leftrightarrow \quad \mathbf{x} \in \Omega_1 \quad \text{and} \\ \varphi(\mathbf{x}) > 0 \quad &\Leftrightarrow \quad \mathbf{x} \in \Omega_2. \end{aligned}$$

(ii) The distance between the interface Γ_φ and a point $\mathbf{x} \in \Omega$ is given by the absolute value of $\varphi(\mathbf{x})$, i.e.,

$$|\varphi(\mathbf{x})| \;=\; \min_{\mathbf{p} \in \Gamma_\varphi} \|\mathbf{p} - \mathbf{x}\|_2. \tag{5.1}$$

Note that (5.1) implies that the interface between both subdomains Ω_1 and Ω_2 is characterized by the zero level of φ, i.e.,

$$\varphi(\mathbf{x}) = 0 \quad \Leftrightarrow \quad \mathbf{x} \in \Gamma_\varphi.$$

Since the interface Γ_φ is the zero level of φ and the gradient $\nabla\varphi$ is orthogonal to the levels of φ, the distance in (5.1) can be expressed by the distance property which reads as

$$\|\nabla\varphi(\mathbf{x})\|_2 = 1, \quad \mathbf{x} \in \Omega. \tag{5.2}$$

Next, we present an illustrating example demonstrating that a signed distance function is advantageous for numerically determining Γ_φ. Consider the following one-dimensional signed distance function

$$\varphi^{\text{ex}} : [0,1] \to \mathbb{R}, \quad x \mapsto x - 0.25.$$

This function describes the signed distance between each point $x \in [0,1]$ and the interface $\Gamma_\varphi = 0.25$. Assume that this function changes while evolving in simulation time and results in the disturbed level set function

$$\varphi^{\text{dist}} : [0,1] \to \mathbb{R}, \quad x \mapsto \begin{cases} \left(\varphi^{\text{ex}}(x)\right)^2, & \varphi^{\text{ex}}(x) \geq 0 \\ \left(\varphi^{\text{ex}}(x)\right)^3, & \varphi^{\text{ex}}(x) < 0 \end{cases}.$$

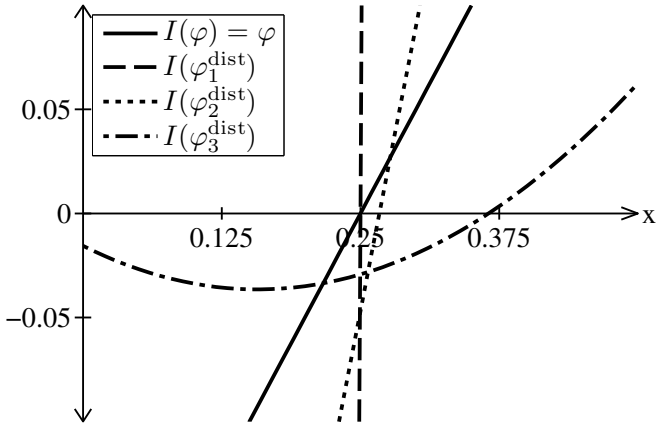


Figure 5.1: Disturbed level set function and its polynomial interpolation.

Note that both functions φ^{ex} and φ^{dist} have the same zero level and, hence, describe the same interface $\Gamma_{\varphi^{\text{ex}}} = \Gamma_{\varphi^{\text{dist}}} = \{0.25\}$. However, this does not hold when considering their interpolations. For instance, we quadratically interpolate both functions using data points at 0, 0.5 and 1. Let $I(\varphi^{\text{ex}})$ and $I(\varphi^{\text{dist}})$ denote these quadratic interpolation of φ^{ex} and φ^{dist}, respectively. Both polynomials as well as the functions φ^{ex} and φ^{dist} are depicted in the range $[0, 0.5]$ in Fig. 5.1. Here, we observe that the zero-level of the given signed distance function φ^{ex} and its polynomial interpolation $I(\varphi^{\text{ex}})$ describe the same interface at 0.25. However, the (exact) zero-level of the polynomial interpolation $I(\varphi^{\text{dist}})$ is not located at 0.25. Overall, this example illustrates that using a signed distance function as level set function φ facilitates the accurate computation of the interface Γ_φ.

5.1.2 Methods for Preserving the Signed Distance Property

Primarily, there exist two different approaches to preserve the signed distance property for level set functions over simulation time. The first approach relies on so-called on extension velocities [3, 45]. Here, an additional flow field $\tilde{\mathbf{u}}$ is constructed which is based on the flow field $\mathbf{u}$ of the underlying fluid dynamics. Then, rather than using $\mathbf{u}$ for describing the movement of the level set function φ, the field $\tilde{\mathbf{u}}$ is employed to determine the evolution of φ. In contrast, the second approach uses the flow field $\mathbf{u}$ of the fluid dynamics for moving φ. This approach may cause the loss of the signed distance property (5.2). In this case, a "new" level set function is constructed by rebuilding a signed distance function from the implicit representation of the interface Γ_φ. This approach is referred to as re-initialization of level set functions.

In this chapter, we focus on re-initializing the level set function. To present various methods for re-initializing level set functions which satisfy the distance property (5.2), we transform this property to a more general equation. Therefore, the right hand side of (5.2) is replaced by a speed function $h : \Omega \to \mathbb{R}^+$ and the signed distance property becomes the Eikonal equation

$$\|\nabla\varphi\|_2 = h(\mathbf{x}), \quad \mathbf{x} \in \Omega. \tag{5.3}$$

In general, there exist mainly the following five classes of numerical techniques for solving the Eikonal equation (5.3):

(i) PDE-based methods,

(ii) fast marching methods,

(iii) fast iterative methods,

(iv) fast sweeping methods, and

(v) Euclidean Distance Transform methods.

In the remainder of this section, we outline each method and discuss why none of these methods is suitable for re-initializing level set functions on distributed, unstructured grids as presented in Chap. 3.

The first class consists of PDE-based methods. These methods offer the advantage of exploiting the full range of well-studied numerical techniques for structured as well as unstructured grids. In particular, PDE-based methods are known to be amenable to parallel computing [51] and provide a viable alternative for finite volume and finite difference methods. However, previous research using serial techniques based on finite elements indicated that such approaches tend to be numerically inadequate for two-phase flows [81]. Therefore, this class is not considered further in this thesis; see [155] for a recent survey on PDE-based methods for the solution of (5.3).

The archetype of another class is the fast marching method (FMM), originally developed by Sethian [152] for Cartesian grids. It was later generalized to arbitrary triangulated surfaces [111], unstructured meshes [154], and any d-dimensional implicit hyper-surfaces [120]. The algorithmic complexity of the FMM is $\mathcal{O}(N \log(N))$ where N is the number of grid points. The reader is referred to [153] for a detailed overview. The FMM is based on a front propagation scheme and uses a heap data structure which makes it difficult to parallelize using a domain decomposition strategy. The first discussion on a parallel implementation of the FMM is given in [92] where four different strategies are compared for Cartesian grids. A recent technique [93] determines the flow field on an unstructured grid whereas the level set function is described on a Cartesian grid. Both grids are

adaptively refined and distributed among processes. In [170], the FMM is separately applied on each subdomain. This process is iteratively refined exchanging data across subdomain boundaries. In contrast to a domain decomposition approach, the parallel FMM presented in [28] is based on distributing the interface and is discussed for Cartesian grids. To our knowledge, there is no publication of a parallel FMM on unstructured grids.

In [99], the authors introduce the fast iterative method (FIM) which is designed for parallel computation on Cartesian grids. This recent method relies on a modification of a label-correction scheme which is a shortest path algorithm for graphs [13]. This scheme is coupled with an iterative procedure for the grid point update. A simple list management allows to simultaneously update multiple points by a Jacobi update, enabling parallel computing. Up to now, this method is only introduced for Cartesian grids. However, current research extends the FIM to unstructured triangular and tetrahedral grids.

The fast sweeping method (FSM) [178] solves the Eikonal equation on a d-dimensional grid by performing 2^d directional sweeps in a Gauss–Seidel fashion. This sweeping idea of visiting nodes of the mesh in a predefined order is also present in [26, 50]. The FSM has a complexity of $\mathcal{O}(N)$ and has been extended to triangular meshes [133]. Also, a parallel version of FSM was introduced for structured two-dimensional grids [177]. As far as we know, no parallel version of the FSM on unstructured grids is available in the open literature.

Although the methods in all of the above classes are designed to solve the Eikonal equation (5.3), they are often also used for the solution of the special case (5.2), where the speed function is equal to one. Some of these methods are compared in [98] concerning numerical efficiency for quadrilateral grids using a serial implementation. However, there are classes of methods specifically exploiting the particular structure of (5.2). For instance, the Euclidian Distance Transform (EDT) is concerned with the following problem on Cartesian grids. Given a subset of grid points, called sites, the EDT computes the distance of all other grid points to the closest site in the Euclidean norm. The algorithms for the EDT are surveyed in [61] and [101] for two- and three-dimensional grids, respectively. Parallel algorithms for the EDT are investigated for parallel random access machine models [112, 176] and for graphic processors in [36, 96, 138, 139, 149, 158]. Although the literature on parallel EDT algorithms is vast, we do not consider these methods here. The reason is that these methods are tailored toward Cartesian grids and that the points to which the distances are computed are located on the grid. However, we are concerned with unstructured grids and distances to a given manifold which, in general, is not located on grid points.

5.2 Re-Initializing Level Set Functions by Direct Distances

None of the methods described in the previous Sect. 5.1.2 are viable to re-initialize level set functions on distributed, unstructured tetrahedral grids. Therefore, in the remainder of this chapter, we present our algorithm which is introduced in [70]. Its main features are

(i) the applicability to unstructured three-dimensional grids,

(ii) the serial expected runtime of $\mathcal{O}(N \log(N))$, where N is the number of degrees of freedom to represent φ, and

(iii) the high degree of parallelism.

5.2.1 Notations and the Base Algorithm

In this section, we introduce some notations and give an outline of re-initialization algorithms in the context of two-phase flow problems on unstructured grids.

Since we are concerned with the re-initialization algorithm on the finest triangulation, we only consider a triangulation on one level which is denoted by the symbol $\mathcal{T}$. On this triangulation, the level set function φ is discretized by the finite element function φ_h. The discrete representation of Γ_φ is denoted by the set

$$\Gamma_\varphi^h \;=\; \{\mathbf{x} \in \Omega \;\mid\; \varphi_h(\mathbf{x}) = 0\}\,.$$

Furthermore, let $\mathcal{V}$ denote the points where the degrees of freedom of φ_h are located. We distinguish between two cases:

(i) If φ_h is only described on vertices of $\mathcal{T}$ then $\mathcal{V}$ is the union of all vertices of $\mathcal{T}$. Here, φ_h is called linear. The discrete representation Γ_φ^h forms a planar segment in each tetrahedron containing the zero level of φ_h.

(ii) Otherwise, if φ_h is described on vertices and edges of $\mathcal{T}$ then $\mathcal{V}$ consists of all vertices and midpoints of all edges of $\mathcal{T}$, and φ_h is called quadratic. For the sake of simplicity, we call $u \in \mathcal{V}$ a vertex whether u is a vertex or a midpoint of an edge of $\mathcal{T}$. To determine Γ_φ^h, we consider a finer triangulation $\mathcal{T}'$ where all vertices and midpoints in $\mathcal{T}$ are the vertices in $\mathcal{T}'$. The finite element function φ_h' with $\varphi_h(u) = \varphi_h'(u)$ for all $u \in \mathcal{V}$ is a linear function on $\mathcal{T}'$. The discrete interface Γ_φ^h is then determined as the zero level of φ_h' on $\mathcal{T}'$.

Figure 5.2 displays a discrete interface Γ_φ^h of a quadratic level set function φ_h. The triangulation $\mathcal{T}$ is given by bold lines and the finer triangulation $\mathcal{T}'$ by thin lines. The interface Γ_φ^h is a planar segment in each tetrahedron of $\mathcal{T}'$.

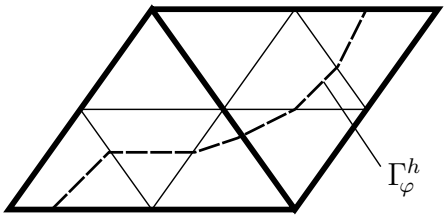


Figure 5.2: Discrete interface Γ_φ^h of a quadratic level set function φ_h.

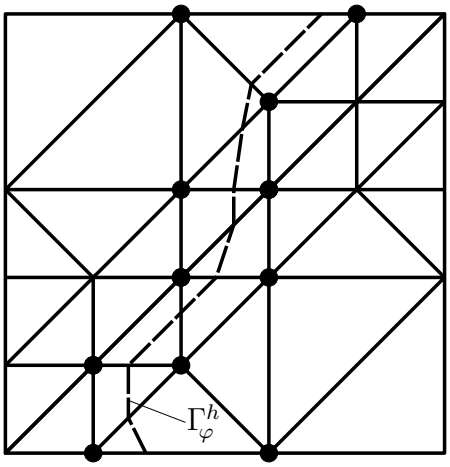


Figure 5.3: Triangulation of a unit square, with zero level Γ_φ^h of a piece-wise linear level set function φ_h. The frontier vertices $w \in \mathcal{F}$ are denoted by $\bullet$.

We moreover introduce $\mathcal{T}_{\Gamma^h}$ as the set of tetrahedra of $\mathcal{T}$ which are intersected by Γ_φ^h and the so-called *frontier vertex set* $\mathcal{F} \subset \mathcal{V}$. A vertex w is in $\mathcal{F}$ if it is located at an intersected tetrahedron $T \in \mathcal{T}_{\Gamma^h}$. The set of the remaining vertices, $\mathcal{V} \setminus \mathcal{F}$, is denoted by $\mathcal{S}$ and a vertex $v \in \mathcal{S}$ is called *off-site vertex.* That is, an off-site vertex resides at a tetrahedron $T \in \mathcal{T} \setminus \mathcal{T}_{\Gamma^h}$ that is not intersected by Γ_φ^h. Unless otherwise stated, the symbols w and v are used to express vertices in $\mathcal{F}$ and $\mathcal{S}$, respectively. In Fig. 5.3, a two-dimensional example for these notations is given. Here, all vertices $w \in \mathcal{F}$ are depicted by $\bullet$ for a piece-wise linear representation of φ_h.

The objective of the re-initialization algorithm consists of approximating the signed distance between all vertices and the discrete interface. That is, the algorithm determines an approximation of $\varphi(u)$ for all $u \in \mathcal{V}$. To this end, let $d\left(\Gamma_\varphi^h; u\right)$ denote an approximation of the (unsigned) distance between Γ_φ^h and $u \in \mathcal{V}$. Then, the re-initialization algorithm is assembled by three stages.

(I) *Initialization*: Determining the distance $d\left(\Gamma_\varphi^h; w\right)$ for frontier vertices $w \in \mathcal{F}$.

(II) *Propagation*: Determining the distance $d\left(\Gamma_\varphi^h; v\right)$ for off-site vertices $v \in \mathcal{S}$.

(III) *Signing*: Determining the value of $\varphi_h(u)$ for all vertices $u \in \mathcal{V}$.

The resulting algorithm is outlined in Algorithm 2. Next, we describe all three stages in detail.

Algorithm 2: Base algorithm to re-initialize a level set function.

```
// stage I
(F, d(Γ_φ^h; F), L) = InitFrontierSet ()
// stage II
foreach v ∈ S do
    Compute d(Γ_φ^h; v)
// stage III
foreach u ∈ V do
    Compute φ_h(u)
```

5.2.2 Determining Distances for Frontier Vertices

In stage I, the frontier vertex set $\mathcal{F}$ is initialized by a loop over all tetrahedra. The idea is introduced in [81]. To determine the distance $d\left(\Gamma_\varphi^h; w\right)$ for a vertex $w \in \mathcal{F}$, we first define the set $T(w)$ of all tetrahedra which have the vertex w in common. That is, a tetrahedron is in $T(w)$ if w is one of its vertices. In Fig. 5.4, the set $T(w)$ for a given vertex w is indicated by gray shading. The approximated distance $d\left(\Gamma_\varphi^h; w\right)$ between the discrete interface Γ_φ^h and a frontier vertex is determined by

$$d\left(\Gamma_\varphi^h; w\right) \;:=\; P(w)$$

where the projection P in $T(w)$ is defined by

$$P(w) \;:=\; \min_{t \in T(w)} P_t(w). \tag{5.4}$$

The local projection $P_t(w)$ in a single tetrahedron t is defined by the distance between w and Γ_φ^h in t, in formula

$$P_t(w) \;:=\; \min_{q \in \Gamma_\varphi^h \cap t} \|q - w\|_2. \tag{5.5}$$

That is, (5.5) describes the distance between w and Γ_φ^h in a single tetrahedron t whereas (5.4) characterizes the distance between w and Γ_φ^h in the domain represented by $T(w)$. In Fig. 5.4, the projection $P_t(w)$ is depicted by a dotted line for two tetrahedra, t_1 and t_2. In [85], a suitable, alternative definition

$$\overline{T}(w) \supset T(w) \tag{5.6}$$

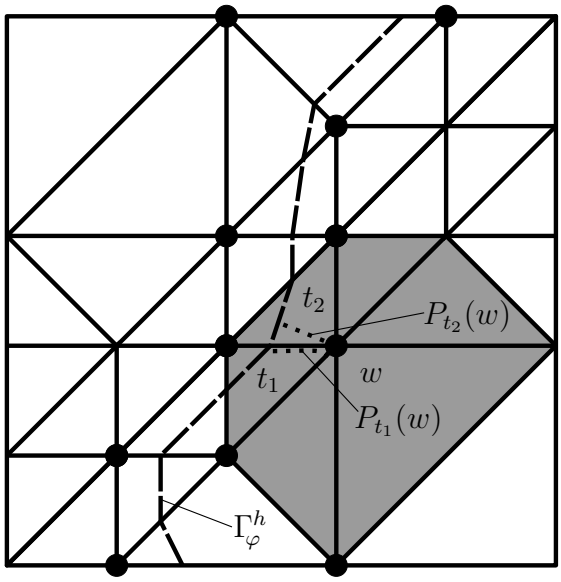


Figure 5.4: Triangulation of a unit square, with zero level Γ_φ^h of a piece-wise linear level set function φ_h. Here, the frontier vertices $w \in \mathcal{F}$ are denoted by $\bullet$ and the local projection $P_t(w)$ in a tetrahedron t by a dotted line.

of $T(w)$ is presented in an attempt to increase the accuracy of computing distances between frontier vertices and Γ_φ^h. That method relies on the observation that the closest interface segment of Γ_φ^h to w may not be located in $T(w)$. For instance, in Fig. 5.5, the set $T(w)$ of a frontier vertex w does not include the tetrahedron $t \in \mathcal{T}_{\Gamma^h}$ which contains the closest interface segment of Γ_φ^h to w. The set $\overline{T}(w)$ is defined as a superset of $T(w)$.

Next, we focus on the projection P_t to determine the distance between a frontier vertex w and the interface Γ_φ^h. This projection may be either non-orthogonal or orthogonal. To characterize this distinction, let $q_w \in \Gamma_\varphi^h \cap t$ be the closest point to

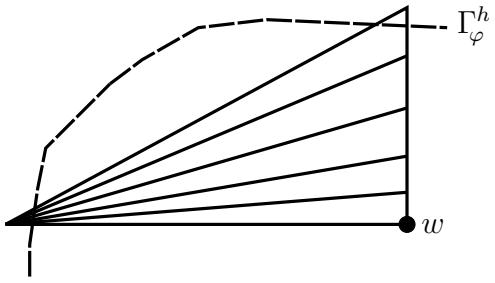


Figure 5.5: The tetrahedron containing the closest interface segment of Γ_φ^h to a frontier vertex w can be "far" away.

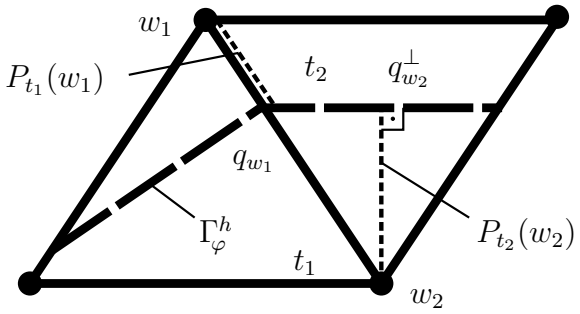


Figure 5.6: Local projections P_{t_1} and P_{t_2} for two frontier vertices w_1 and w_2.

the frontier vertex w, i.e., q_w denotes the solution of (5.5) satisfying

$$\|q_w - w\|_2 = \min_{q \in \Gamma^h_\varphi \cap t} \|q - w\|.$$

The projection $P_t(w)$ is non-orthogonal onto Γ^h_φ, if the vector $(w - q_w)$ is not orthogonal to Γ^h_φ. Here, the distance $d\left(\Gamma^h_\varphi; w\right)$ in t is determined by the distance between w and the intersection of Γ^h_φ with the edges of t. This situation is illustrated by vertex w_1 in Fig. 5.6. The projection $P_t(w)$ is called orthogonal, if the vector $(w - q^{\perp}_w)$ is orthogonal to the representation Γ^h_φ where $q^{\perp}_w$ is the perpendicular foot of the projection P_t in a tetrahedron t. This is illustrated by vertex w_2 in Fig. 5.6. The set of all perpendicular feet $\left\{q^{\perp}_w \mid w \in \mathcal{F}\right\} \subset \Gamma^h_\varphi$, which are used to determine $d\left(\Gamma^h_\varphi; w\right)$, is denoted by $\mathcal{L}$. The orthogonal projection is differently computed according to the representation of φ_h:

(i) In case of a linear representation of φ_h, the local projection $P_t(w)$ in (5.5) for a given tetrahedron t is obtained as follows. The local representation of Γ^h_φ in t is either a vertex, an edge, a triangle, or a quadrilateral. In all cases the local orthogonal projection can be directly computed by basic geometric calculations.

(ii) If φ_h is quadratic, we reduce the problem to the linear one. Therefore, we introduce a finer triangulation $\mathcal{T}'$, where all vertices and midpoints of $\mathcal{T}$ are the vertices in $\mathcal{T}'$, i.e., each tetrahedron in $\mathcal{T}$ is regularly refined into eight subtetrahedra. On this finer triangulation, φ_h is linear, and we can apply the local projection described above in (i). Note that this reduction includes an error of $\mathcal{O}(h^2)$.

5.2.3 The Brute-Force Propagation Algorithm

After the initialization stage I, the algorithm determines $d\left(\Gamma^h_\varphi; v\right)$ for all remaining vertices $v \in \mathcal{S}$ in stage II. To this end, we follow an approach which is based on

direct distance computations. Therefore, we first define a distance $d\left(\Gamma_\varphi^h; v, w\right)$ between Γ_φ^h and an off-site vertex v via a frontier vertex w by

$$d\left(\Gamma_\varphi^h; v, w\right) \;:=\; \|v - w\|_2 + d\left(\Gamma_\varphi^h; w\right). \tag{5.7}$$

Note that this distance does not define a metric or distance in a mathematical sense. In Fig. 5.7, the distances $d\left(\Gamma_\varphi^h; v, w_i\right)$ for a vertex v via w_i, $i = 1, 2$, are illustrated by a line from vertex v to w_i followed by a line from w_i to $q_{w_i}^\perp$. Second, we define the distance between an off-site vertex v and Γ_φ^h by

$$\min_{w \in \mathcal{F}} \left\{ d\left(\Gamma_\varphi^h; v, w\right) \right\}. \tag{5.8}$$

That is, the distance between v and Γ_φ^h is obtained by taking the shortest of all distances via frontier vertices.

To increase the accuracy of determining the distance in (5.8), the set $\mathcal{F}$ is augmented by all perpendicular feet $\mathcal{L}$ which are computed in stage I. Then, the computation of the distance $d\left(\Gamma_\varphi^h; v\right)$ between a vertex $v \in \mathcal{S}$ and Γ_φ^h becomes

$$d\left(\Gamma_\varphi^h; v\right) \;:=\; \min_{w \in \mathcal{F} \cup \mathcal{L}} \left\{ d\left(\Gamma_\varphi^h; v, w\right) \right\}. \tag{5.9}$$

Since perpendicular feet are located at Γ_φ^h, we define the distance $d\left(\Gamma_\varphi^h; q\right) := 0$ for all perpendicular feet $q \in \mathcal{L}$ leading to

$$d\left(\Gamma_\varphi^h; v, q\right) \;=\; \|v - q\|_2.$$

In Fig. 5.7, the distances of the vertex v via the perpendicular feet $q_{w_i}^\perp$, $i = 1, 2$ are depicted by dashed lines. There are four paths from v via different vertices to Γ_φ^h. Two of them are via the frontier vertices w_1 and w_2 and the other two are via the perpendicular feet $q_{w_1}^\perp$ and $q_{w_2}^\perp$. In this figure, the distance $d\left(\Gamma_\varphi^h; v\right)$ is determined as $d\left(\Gamma_\varphi^h; v, q_{w_2}^\perp\right)$ because

$$q_{w_2}^\perp = \operatorname*{argmin}_{u \in \left\{w_1, w_2, q_{w_1}^\perp, q_{w_2}^\perp\right\}} \left\{ d\left(\Gamma_\varphi^h; v, u\right) \right\}$$

holds.

5.2.4 Determining the Signs of the Level Set Function

Finally, stage III of the algorithm consists of assigning a value to $\varphi_h(u)$ for all vertices $u \in \mathcal{V}$ as follows. Let φ_h^{old} denote the level set function before the re-initialization algorithm has started. The sign of the new values $\varphi_h(u)$ for all $u \in \mathcal{V}$ is determined by

$$\varphi_h(u) \leftarrow \begin{cases} -d\left(\Gamma_\varphi^h; u\right), & \varphi_h^{\text{old}} < 0 \\ d\left(\Gamma_\varphi^h; u\right), & \varphi_h^{\text{old}} \geq 0 \end{cases}. \tag{5.10}$$

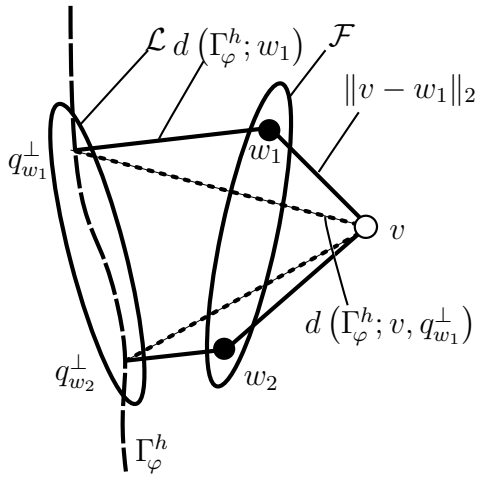


Figure 5.7: Computing $d\left(\Gamma_\varphi^h; v\right)$ for a vertex $v \in \mathcal{S}$.

That is, the signs are restored whereas the distances are computed by the function $d\left(\Gamma_\varphi^h; \cdot\right)$.

Before turning toward the parallel algorithm, we describe an efficient strategy to compute the distance between the interface and off-site vertices.

5.2.5 Efficiently Computing Distances for Off-Site Vertices

In general two-phase flow simulations, the off-site vertex set $\mathcal{S}$ is larger than the frontier vertex set $\mathcal{F}$. Hence, the computational work of determining the distances between all off-site vertices and the interface is commonly more time consuming than determining the distances for all frontier vertices. This implies that most computational work of the re-initialization algorithm is spent in stage II of Algorithm 2 to determine

$$d\left(\Gamma_\varphi^h; v\right) \leftarrow \min_{w \in \mathcal{F} \cup \mathcal{L}} d\left(\Gamma_\varphi^h; v, w\right) \quad \text{for all } v \in \mathcal{S}. \tag{5.11}$$

For one off-site vertex v, a naïve implementation of this formula would lead to a linear search in the set $\mathcal{F} \cup \mathcal{L}$ resulting in the time complexity of

$$T^{\text{naïve}}(v) \;=\; \mathcal{O}(|\mathcal{F} \cup \mathcal{L}|).$$

To reduce the linear complexity, we apply the following approach. We restrict the search space $\mathcal{F} \cup \mathcal{L}$ in equation (5.11) to an "easy-to-compute" set denoted by $\mathcal{N}(m, v) \subset \mathcal{F} \cup \mathcal{L}$ which depends on the off-site vertex v and a given parameter $m \in \mathbb{N}$. This restriction of (5.11) leads to

$$d\left(\Gamma_\varphi^h; v\right) \leftarrow \min_{w \in \mathcal{N}(m,v)} d\left(\Gamma_\varphi^h; v, w\right) \tag{5.12}$$

for all off-site vertices $v \in \mathcal{S}$. Due to the geometric structure of the problem, we choose the search space $\mathcal{N}(m, v)$ as the set of m nearest neighbors of v in the set $\mathcal{F} \cup \mathcal{L}$. So, the parameter m controls the size of the search subspace and influences the computational cost of setting up $\mathcal{N}(m, v)$ as well as computing the minimum in (5.12).

An efficient approach to search for the m nearest neighbors in a set of k-dimensional data points is given by k-d trees (k-dimensional trees) originally introduced by Bentley [19]. Let N denote the number of data points which are represented by a k-d tree. The expected runtime of setting up this data structure is then given by $\mathcal{O}(N \log(N))$. In [74], an algorithm for searching the m nearest neighbors of a given k-dimensional data point is presented. Here, the nearest neighbors are stored in the k-d tree whereas the data point is not necessarily part of that data structure. The expected runtime of the algorithm is given by $\mathcal{O}(m \log(N))$. There are alternative data structures and algorithms for finding m nearest neighbors. An extensive overview is presented in [145].

We use k-d trees to solve the minimization problem in (5.12). Here, $k = 3$ and the tree represents the set $\mathcal{F} \cup \mathcal{L}$. Then, the set $\mathcal{N}(m, v)$ is determined by the tree as the set of the m nearest neighbors of $v \in \mathcal{S}$. Note that the k-d tree is built only once and is used for evaluating all distances of off-site vertices. An analysis of the runtime to build a k-d tree and to search for nearest neighbors in this tree is presented in [74] and is as follows. Building the data structure involves an runtime of

$$T^{\text{build}} \;=\; \mathcal{O}(|\mathcal{F} \cup \mathcal{L}| \cdot \log(|\mathcal{F} \cup \mathcal{L}|)). \tag{5.13}$$

The expected runtime of searching for m nearest neighbors of each v is given by

$$T^{\text{search}}(m, v) \;=\; \mathcal{O}(m \cdot \log(|\mathcal{F} \cup \mathcal{L}|)). \tag{5.14}$$

In the next section, we combine the brute-force algorithm and k-d trees to obtain a parallel re-initialization algorithm.

5.3 The Parallel Re-Initialization Algorithm

The re-initialization algorithm described in Sect. 5.2 is designed for parallel computing. We will show in the remainder of this section that the initialization stage (I) requires no communication, the propagation stage (II) includes one synchronization point, and no communication is necessary to perform the signing phase (III). In the following, we first present the parallel algorithm employing the distributed triangulation, cf. Sect. 3.1. Afterwards, we transform the distributed-memory parallel algorithm to a hybrid distributed-/shared-memory parallel algorithm in Sect. 5.3.2.

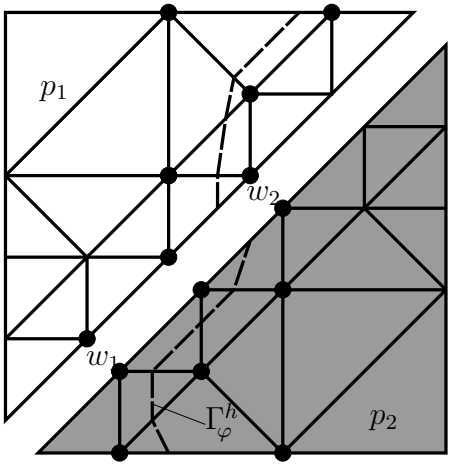


Figure 5.8: Distributed triangulation of a unit square on processes p_1 and p_2, with zero level Γ_φ^h of a piece-wise linear level set function φ_h. Here, the frontier vertices $w \in \mathcal{F}^p$ are denoted by $\bullet$.

5.3.1 Distributed-Memory Parallelization

Recall that the triangulation $\mathcal{T}$ is distributed among P processes and that vertices, edges and faces on process boundaries are stored overlapped. For instance, Fig. 5.8 shows the distribution of the triangulation presented in Fig. 5.3 among two processes p_1 and p_2 where the vertices w_1 and w_2 are stored by both processes. As in the previous chapters, the restriction of a set to a process p is indicated by a superscript, e.g., the vertices of the set $\mathcal{V}$ on process p are denoted by $\mathcal{V}^p$. Since φ_h is represented as a finite element function, we assume here that the input of the algorithm φ_h^{old} is available in the accumulated form, cf. Sect. 3.2.1. That is, if a vertex u is located at multiple processes then each process stores the same value $\varphi_h^{\text{old}}(u)$.

The parallel algorithm presented in Algorithm 3 follows the approach given in Algorithm 2 where distances are computed based on nearest neighbors. The algorithm is designed in such a way that the computational work of all three stages can be distributed among the processes. However, in comparison to the serial implementation, a few slight modifications of the algorithm are mandatory for transforming the serial Algorithm 2 into the parallel Algorithm 3. In the remainder of this section, we describe the parallel version of all three stages.

Stage I Each process p performs the initialization on its tetrahedra $\mathcal{T}^p$ leading to the "local" sets $\mathcal{F}^p$, $d^p\left(\Gamma_\varphi^h; \mathcal{F}^p\right)$, and $\mathcal{L}^p$ that are computed from the information that is available by process p. In particular, each process p simultaneously executes stage I of the serial algorithm without any modifications or communication.

Algorithm 3: Parallel algorithm based on direct distances to re-initialize a level set function.

for $p = 1, \ldots, P$ **do in parallel**
  $\mathcal{F}^p \leftarrow \emptyset$
  $d^p\left(\Gamma_\varphi^h; u\right) \leftarrow \infty$ for all $u \in \mathcal{V}^p$
  $\mathcal{L}^p \leftarrow \emptyset$
  // stage I: Initialization
  foreach $t \in \mathcal{T}^p$ **do** *// FRONT*
    if t is intersected by Γ_φ **then**
      foreach w is corner of t **do**
        $\mathcal{F}^p \leftarrow \mathcal{F}^p \cup \{w\}$
        $d^p\left(\Gamma_\varphi^h; w\right) \leftarrow \texttt{min}(d^p\left(\Gamma_\varphi^h; w\right), P_t(w))$
        if $P_t(w)$ is orthogonal **then**
          $\mathcal{L}^p \leftarrow \mathcal{L}^p \cup q_w^\perp$
  // stage II: Propagation
  $\mathcal{F} \leftarrow \texttt{Gather}\left(\mathcal{F}^p\right)$
  $d\left(\Gamma_\varphi^h; \mathcal{F}\right) \leftarrow \texttt{Gather}\left(d^p\left(\Gamma_\varphi^h; \mathcal{F}^p\right)\right)$
  $\mathcal{L} \leftarrow \texttt{Gather}\left(\mathcal{L}^p\right)$
  $\mathsf{KD} \leftarrow \texttt{BuildKDTree}\left(\mathcal{F} \cup \mathcal{L}\right)$
  foreach $v \in \mathcal{S}^p$ **do** *// OFFSITE*
    $\mathcal{N}(m, v) \leftarrow \texttt{GetNearestNeighbors}\left(\mathsf{KD}, v, m\right)$
    $d^p\left(\Gamma_\varphi^h; v\right) \leftarrow \texttt{min}_{w \in \mathcal{N}(m,v)}\left\{d^p\left(\Gamma_\varphi^h; v, w\right)\right\}$ *// MIN*
  // stage III: Signing
  foreach $u \in \mathcal{V}^p$ **do** *// SIGN*
    $$\varphi_h(u) \leftarrow \begin{cases} -d^p\left(\Gamma_\varphi^h; u\right), & \varphi_h^{\text{old}} < 0 \\ d^p\left(\Gamma_\varphi^h; u\right), & \varphi_h^{\text{old}} \geq 0 \end{cases}$$

Stage II Recall from (5.9) and (5.12) that the three "global" sets $\mathcal{F}$, $d\left(\Gamma_\varphi^h;\mathcal{F}\right)$ and $\mathcal{L}$ are needed for computing $d\left(\Gamma_\varphi^h;v\right)$ for all off-site vertices v. Compared to the serial implementation, an additional step is necessary to gather the global sets from the local sets. This step is described in the following.

In general, a process p is not able to access all information that is necessary to reveal the frontier vertices located in its memory, i.e., $\mathcal{F}^p \neq \mathcal{F} \cap \mathcal{V}^p$. For an illustrating example consider Fig. 5.8. Here, the vertex w_1 is in $\mathcal{F}^{p_2}$ because there is a tetrahedron in $T(w_1)$ which is stored by process p_2 and is intersected by Γ_φ^h. However, w_1 is not in $\mathcal{F}^{p_1}$ because there is no adjacent tetrahedron in $T(w_1)$ on process p_1 that is intersected by Γ_φ^h. The global set $\mathcal{F}$ is obtained by building the union of all local sets $\mathcal{F}^p$ for $p = 1,\ldots,P$, i.e.,

$$\mathcal{F} \;=\; \bigcup_{p=1}^{P} \mathcal{F}^p. \tag{5.15}$$

In contrast, for obtaining the distances between frontier vertices and the interface, $d\left(\Gamma_\varphi^h;\mathcal{F}\right)$, it is not sufficient to only build the union of all locally determined distances $d^p\left(\Gamma_\varphi^h;\mathcal{F}^p\right)$ for $p = 1,\ldots,P$. For instance, assume that a vertex $w \in \mathcal{F}$ is located at processes $p(w) = \{p_1,\ldots,p_k\}$ with $1 \leq k \leq P$ and, without loss of generality, process p_1 stores the tetrahedron which contains the closest segment of Γ_φ^h. Then, the remaining processes $p_2,\ldots,p_k$ either determine a larger value of $d\left(\Gamma_\varphi^h;w\right)$ or do not classify w as a frontier vertex. Therefore, the distance between w and Γ_φ^h is determined as the minimum of the locally computed distances, in formula,

$$d\left(\Gamma_\varphi^h;w\right) \;=\; \min_{p\in p(w)} d^p\left(\Gamma_\varphi^h;w\right) \tag{5.16}$$

where $d^p\left(\Gamma_\varphi^h;w\right) = \infty$ if $w \notin \mathcal{F}^p$. Recall Fig. 5.8 for an example. The frontier vertex w_2 is located in $\mathcal{F}^{p_1}$ and $\mathcal{F}^{p_2}$. Thus, two local distances $\delta_1 = d^{p_1}\left(\Gamma_\varphi^h;w_2\right)$ and $\delta_2 = d^{p_2}\left(\Gamma_\varphi^h;w_2\right)$ are computed by p_1 and p_2, respectively. Since in this figure $\delta_1 < \delta_2$ holds the "global" distance is computed as $d\left(\Gamma_\varphi^h;w_2\right) = \delta_1$. Vice versa, only p_2 classifies w_1 as a frontier vertex and determines its "local" distance by $d^{p_2}\left(\Gamma_\varphi^h;w_1\right)$. Hence, the distance $d\left(\Gamma_\varphi^h;w_1\right)$ is set to $d^{p_2}\left(\Gamma_\varphi^h;w_1\right)$.

Finally, the third set $\mathcal{L}$ is accumulated by gathering the perpendicular feet corresponding to the frontier vertices where the minimal distance is obtained. Let $\left(q_w^\perp\right)^p$ denote the perpendicular foot computed for vertex w by process p—if this foot exists. Then, the perpendicular feet for all vertices $w \in \mathcal{F}$ are determined by

$$q_w^\perp = \left(q_w^\perp\right)^{p^*} \quad \text{with} \quad p^* := \operatorname*{argmin}_{p\in p(w)} \left(d^p\left(\Gamma_\varphi^h;w\right)\right). \tag{5.17}$$

For instance, in Fig. 5.8, the perpendicular foot for vertex w_1 is determined by process p_2 and the foot of w_2 by process p_1.

From a parallel implementation point of view, the three sets are computed by a "gather"-type operation. Here, each process p needs to inform all other processes about its data. Afterwards, each process determines the sets $\mathcal{F}$, $d\left(\Gamma_\varphi^h; \mathcal{F}\right)$, and $\mathcal{L}$ by evaluating the formulae (5.15)–(5.17).

At this point of the algorithm, each process p is capable of determining its set of off-site vertices by

$$\mathcal{S}^p \;:=\; \mathcal{V}^p \setminus \mathcal{F}.$$

Here, process p classifies a vertex as an off-site vertex if the distance to Γ_φ^h is still to be computed. To estimate the distance between $v \in \mathcal{S}^p$ and Γ_φ^h by (5.12), the process p needs the k-d tree representing $\mathcal{F}$ and $\mathcal{L}$. However, p possesses all information to generate this tree and, hence, is capable of evaluating (5.12) for its off-site vertices. Recall from (5.7) that computing the distance between v and Γ_φ^h via a frontier vertex w includes the "global" distance between w and Γ_φ^h. Thus, the local distance $d^p\left(\Gamma_\varphi^h; v, w\right)$ of local off-site vertices $v \in \mathcal{S}^p$ is defined by

$$d^p\left(\Gamma_\varphi^h; v, w\right) \;:=\; \|v - w\|_2 + d\left(\Gamma_\varphi^h; w\right). \tag{5.18}$$

Note that each process redundantly builds and stores the k-d tree in line 15 of Algorithm 3. We will demonstrate in the numerical results section that this redundant computation and storage is affordable to save communication time when determining the m nearest neighbors. Overall, this concludes stage II of the parallel algorithm.

Stage III Finally, in the last stage of the parallel algorithm, the signed distance between v and Γ_φ^h is assigned to $\varphi_h(v)$ for all vertices $v \in \mathcal{V}$. Since each process p has determined the distance between its vertices $\mathcal{V}^p$ and Γ_φ^h and also stores the function φ_h^{old}, it is capable of assigning the signed distance between its vertices and Γ_φ^h by applying (5.10).

5.3.2 Hybrid Distributed-/Shared-Memory Parallelization

Before analyzing the algorithm in the next section, we pursue the approach in [69] and combine the Algorithm 3 with a shared-memory parallelization leading to a hybrid parallel algorithm for re-initializing level set functions. This improvement facilitates the employment of recent high-performance computers which commonly consist of clusters of nodes with multi-core processors. To this end, the computational work on one subdomain is additionally distributed among threads. In particular, the three loops labeled by *FRONT*, *OFFSITE*, and *SIGN* in Algorithm 3 are parallelized using OpenMP [42]. A draft of this hybrid distributed-/shared-memory parallel algorithm is summarized in Algorithm 4. In this pseudo-code

the distributed memory parallelization is labeled by *DistMem* whereas the shared-memory parallelization is labeled by *SharedMem (1)–(3)*. Next, we describe the shared memory parallelization of all three loops in detail. In this section, references to code correspond to the hybrid parallel Algorithm 4.

Parallelization of the loop SharedMem (1) Most of the computational work within step one, i.e., initializing the frontier vertex set and the distance of its vertices, can be performed in parallel. Hence, the loop over all tetrahedra can be distributed among threads. However, in general, a frontier vertex $w \in \mathcal{F}$ is located at several tetrahedra $T(w)$. If two threads handle two different tetrahedra but assign $d^p\left(\Gamma_\varphi^h; w\right)$ for the same vertex w, then a race condition[1] occurs. Hence, assigning $d^p\left(\Gamma_\varphi^h; w\right)$ to a frontier vertex has to be executed by only one thread at any time.

Parallelization of the loop SharedMem (2) In this loop of Algorithm 4, a process p determines the m nearest neighbors $\mathcal{N}(m, v)$ in line 10 for each off-site vertex $v \in \mathcal{S}^p$. Moreover, the distance $d^p\left(\Gamma_\varphi^h; v\right)$ for all $v \in \mathcal{S}^p$ is computed in line 12. Both computations do not depend on any other nearest neighbor set $\mathcal{N}(m, v')$ or distance $d^p\left(\Gamma_\varphi^h; v'\right)$ for all $v' \in \mathcal{S}^p \setminus \{v\}$. So, no data dependencies exist for different loop indices and this loop can be executed in parallel.

Note that searching the minimum in line 12 labeled by *SharedMem (2a)* can be also parallelized by a shared-memory reduction[2] approach. This parallelization of the loop in line 9 yields an additional level of shared-memory parallelism, which is commonly called a nested shared-memory parallelization [42]. However, the numerical experiments indicate that such an approach involving nested shared-memory parallelization is not competitive in terms of execution time. Indeed, the smallest execution times are gained if all available threads are used to execute the loop *SharedMem (2)* in a single level of shared-memory parallelism rather than two levels of shared-memory parallelism.

Parallelization of the loop SharedMem (3) The body of this loop consists of assigning the signed distance $\pm d^p\left(\Gamma_\varphi^h; u\right)$ to $\varphi_h(u)$ in line 14. This assignment does not depend on any other assignment in that loop. Hence, the computational work of this loop body is straightforward distributed among the threads.

[1] A race condition occurs if at least two threads access the same address in memory, where at least one thread performs a write access and the order of these two accesses is not adjusted by synchronization mechanisms.

[2] Reducing the thread local copies to a global value by an associative operation. Here, the global minimum of all the thread local copies is built.

Algorithm 4: Hybrid parallel algorithm based on direct distances to re-initialize a level set function.

```
1  for p = 1, ..., P do in parallel                                   // DistMem
2      Init (F^p d^p (Γ_φ^h; V), L^p) d^p (Γ_φ^h; u) ← ∞ for all u ∈ V^p
       // stage I: Initialization
3      foreach t ∈ T^p do in parallel                                 // SharedMem (1)
4          if t is intersected by Γ_φ then
5              foreach w is corner of t do
6                  Update (F^p, d^p (Γ_φ^h; w), L^p)
       // stage II: Propagation
7      (F, d (Γ_φ^h; F), L) ← Gather (F^p, d^p (Γ_φ^h; F^p), L^p)
8      KD ← BuildKDTree (F ∪ L)
9      foreach v ∈ S^p do in parallel                                 // SharedMem (2)
10         N (m, v) ← GetNearestNeighbors (KD, v, m)
11         foreach w ∈ N (m, v) do in parallel                        // SharedMem (2a)
12             d^p (Γ_φ^h; v) ← min(d^p (Γ_φ^h; v), d^p (Γ_φ^h; v, w))
       // stage III: Signing
13     foreach u ∈ V^p do in parallel                                 // SharedMem (3)
14         φ_h(u) ← Assign(d^p (Γ_φ^h; u))
```

Obviously, the shared-memory parallelization is not only advantageous for the parallel re-initialization algorithm but also for a serial implementation of the algorithm. That is, if the the re-initialization algorithm is not executed on a cluster of processors but only on a single multi-core processor, the shared-memory parallelization is capable to significantly speed up the re-initialization of a level set function.

5.4 Analysis of the Re-Initialization Algorithm

In this section, we present the time complexity of the re-initialization algorithm based on direct distances and show that the complexity of the serial algorithm is competitive with the fast marching method. Furthermore, we approximate the error of the algorithm, and prove that the output is an accumulated representation of the level set function φ_h. In this section, all references to code lines corresponds to Algorithm 3.

Line(s)	Time complexity
5–11	$\lvert\mathcal{V}^p\rvert$
12–14	Gather communication
15	$\lvert\mathcal{F}\cup\mathcal{L}\rvert\cdot\log(\lvert\mathcal{F}\cup\mathcal{L}\rvert)$
17	$m\cdot\lvert\mathcal{S}^p\rvert\cdot\log(\lvert\mathcal{F}\cup\mathcal{L}\rvert)$ (expected)
18	$m\cdot\lvert\mathcal{S}^p\rvert$
20	$\lvert\mathcal{V}^p\rvert$

Table 5.1: Time complexity of Algorithm 3 for a single process p.

5.4.1 Time Complexity

In Table 5.1, the time complexity of the code lines of Algorithm 3 for a single process p is presented. For stage I, i.e., lines 5–11, we assume that the number of tetrahedra hosting a vertex w is bounded, in formula $|T(w)| \leq c$ for $c \in \mathbb{N}$ for all frontier vertices $w \in \mathcal{F}$. Thus, using the projection (5.4) to determine $d^p\left(\Gamma_\varphi^h; w\right)$ for each vertex $w \in \mathcal{F}^p$ is constant in time. Since the initialization iterates over all tetrahedra to identify vertices as frontier vertices and compute their distance to the interface, the complexity of lines 8–11 is bounded by $|\mathcal{T}^p| \leq |\mathcal{V}^p|$. As stated in [166], the communication time for gathering, lines 12–14, is bounded by the sum of the number of gathered elements and the logarithm of the number of processes P, i.e.,

$$\mathcal{O}\left(\max_{p=1,\dots,P}|\mathcal{F}^p| + \log(P)\right).$$

The complexity of redundantly building the k-d tree and searching in the tree is given by (5.13) and (5.14), leading to the complexity of lines 15 and 17. For each off-site vertex $v \in \mathcal{S}^p$, line 18 seeks the smallest distance $d^p\left(\Gamma_\varphi^h; v, w\right)$ in the nearest neighbor set $\mathcal{N}(m, v)$ by a linear search. Since this set consists of m elements the overall time complexity of line 18 is given by $m \cdot |\mathcal{S}^p|$. Iterating over all local vertices $u \in \mathcal{V}^p$ to determine φ_h yields the complexity of line 20.

In general, the terms in line 15 and 17 dominate the time complexity. Therefore, we only discuss these two terms in the following. The number of processes P influences the cardinalities of the local sets, in particular $|\mathcal{S}^p|$. When increasing P then $|\mathcal{S}^p|$ is reduced and the term in line 17 pales in comparison to the term in line 15. Thus, line 17 dominates the runtime for a moderate number of processes and the complexity is given by

$$\mathcal{O}\left(m \cdot |\mathcal{S}^p| \cdot \log(|\mathcal{F}\cup\mathcal{L}|)\right). \tag{5.19}$$

That is, computing distances accounts for most of the runtime. In contrast, when using a large number of processes, the cardinality of $\mathcal{S}^p$ is decreased. Then, the set $\mathcal{F} \cup \mathcal{L}$ contains more elements than $m \cdot |\mathcal{S}^p|$, i.e., $m \cdot |\mathcal{S}^p| < |\mathcal{F} \cup \mathcal{L}|$. Thus, building the k-d tree in line 15 dominates the runtime leading to the complexity

$$\mathcal{O}\left(|\mathcal{F} \cup \mathcal{L}| \cdot \log(|\mathcal{F} \cup \mathcal{L}|)\right). \tag{5.20}$$

If using only one process to execute Algorithm 3, then the runtime is governed by line 17. Here, the complexity is given by

$$\mathcal{O}\left(m \cdot |\mathcal{S}| \cdot \log(|\mathcal{F} \cup \mathcal{L}|)\right). \tag{5.21}$$

Numerical tests show that m can be chosen relatively small compared to $|\mathcal{S}^p|$ and $|\mathcal{F} \cup \mathcal{L}|$. Hence, m is assumed to be constant. Additionally, since $|\mathcal{L}| \leq |\mathcal{F}|$ holds the expected serial runtime given by (5.21) is bounded by

$$\mathcal{O}\left(|\mathcal{S}| \cdot \log|\mathcal{F}|\right).$$

So, the novel re-initialization algorithm based on direct distances has the same serial complexity as the fast marching method. However, the FMM is difficult to parallelize in contrast to Algorithm 3.

5.4.2 Accuracy

Next, we focus on the accuracy of the new re-initialization algorithm. Using the m nearest neighbors rather than the set $\mathcal{F} \cup \mathcal{L}$ leads to an error which is numerically investigated in the next section. Here, for obtaining a bound for the error, we assume $m = |\mathcal{F} \cup \mathcal{L}|$ meaning that we analyze the error of the brute-force algorithm. To this end, let φ_h^{old} be the input of the re-initialization algorithm and Γ_{old}^h the zero level of φ_h^{old}. We assume for the study of the accuracy that Γ_{old}^h approximates a smooth interface. Furthermore, let φ_h denote the re-initialized level set function, i.e., the output of Algorithm 3. This function describes Γ_φ^h by its zero level. In [85], the authors formally introduce the superset $\overline{T}(w) \supset T(w)$, see (5.6), and prove the following statements when using this superset. First, the signed distance δ between Γ_{old}^h and Γ_φ^h is bounded by

$$\left|\delta\left(\Gamma_{\text{old}}^h, \Gamma_\varphi^h\right)\right| \leq h \tag{5.22}$$

where h represents the maximal length of an edge of the underlying triangulation $\mathcal{T}$ (in the vicinity of the interface). Second, the exact distance between frontier vertices $w \in \mathcal{F}$ and Γ_{old}^h is determined, in formula

$$\delta\left(\Gamma_{\text{old}}^h, w\right) \;=\; \varphi_h(w) \quad \text{for all } w \in \mathcal{F}. \tag{5.23}$$

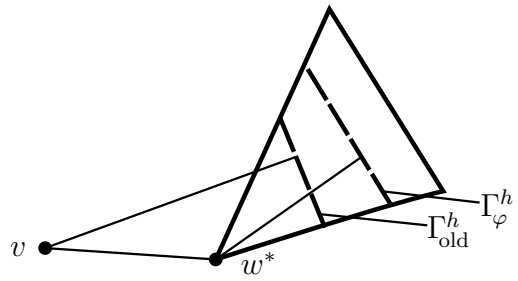


Figure 5.9: Accuracy of determining the distance between an off-site vertex $v \in \mathcal{S}$ and the interface Γ^h_{old}.

We now use these two results to present the accuracy of the re-initialization method. We define the error of the method by

$$e_h \;:=\; \max_{u \in \mathcal{V}} \left\{ \left| \delta\left(\Gamma^h_{\text{old}}, u\right) - \varphi_h(u) \right| \right\}.$$

That is, the error e_h describes the maximal difference of the exact distance between Γ^h_{old} and the determined distance $\varphi_h(u)$ for all $u \in \mathcal{V}$. We now show that this error is bounded by ch. Let $u \in \mathcal{V}$ be a given vertex. Due to (5.23), the method determines the exact distance between Γ^h_{old} and any frontier vertex, i.e., $\delta\left(\Gamma^h_{\text{old}}, u\right) - \varphi_h(u) = 0$. Therefore, we now estimate the error when considering an off-site vertex $u = v \in \mathcal{S}$. The re-initialization algorithm determines $\varphi_h(u)$ by

$$\varphi_h(u) \;=\; \operatorname{sign}(\varphi_h^{\text{old}}(u)) \cdot \left(|\varphi_h(w^*)| + \|u - w^*\|_2 \right).$$

where $w^* \in \mathcal{F} \cup \mathcal{L}$ is obtained by the minimum in (5.9). This situation is illustrated in Fig. 5.9. Note that w^* must not be the minimum if we use m nearest neighbors instead of the brute force method. Due to (5.22) and (5.23), the difference between $\delta\big(\Gamma^h_{\text{old}}, v\big)$ and $\varphi_h(w^*) + \|u - w^*\|_2$ is bounded by ch. Hence, the overall error of the re-initialization algorithm based on direct distances is given by

$$e_h \;=\; \max_{u \in \mathcal{V}} \left(\delta\big(\Gamma^h_{\text{old}}, u\big) - \varphi_h(u) \right) \;\leq\; ch. \tag{5.24}$$

5.4.3 Correctness

Finally, we prove that the output of the re-algorithm is an accumulated representation of φ_h, cf. Sect. 3.2, if the input of the re-initialization algorithm is an accumulated representation of φ_h^{old}. Therefore, we have to show that all processes $p(u)$ storing a vertex $u \in \mathcal{V}$ determine the same value for $\varphi_h(u)$. First, assume that all processes $p(u)$ compute the same distance between u and Γ^h_{φ}, in formula

$$d\left(\Gamma^h_{\varphi}; u\right) \;=\; d^p\left(\Gamma^h_{\varphi}; u\right) \quad \text{for all } p \in p(u). \tag{5.25}$$

Since the input φ_h^{old} is accumulated each process determines the same sign of $\varphi_h(u)$ in line 20 of Algorithm 3. Thus, each process $p \in p(u)$ estimates the same signed distance between u and Γ_φ^h implying an accumulated representation of φ_h.

It remains to prove that the equality (5.25) holds. To this end, we distinguish between frontier- and off-site vertices.

(i) Let $w \in \mathcal{F}^p$ be a local frontier vertex which is located at a process $p \in p(w)$. Each process p computes the distance $d\left(\Gamma_\varphi^h; w\right)$ by evaluating (5.16) where the minimum of all locally computed distances $d^p\left(\Gamma_\varphi^h; w\right)$ is determined. Thus, each process $p \in p(w)$ stores the same value for the distances between the frontier vertex w and Γ_φ^h.

(ii) Now, let v be an off-site vertex. All processes $p \in p(v)$ assign the value of $d^p\left(\Gamma_\varphi^h; v\right)$ in line 18 of Algorithm 3 by evaluating formula (5.12) and (5.18). Each process p obtains the same results because:

   a) the same k-d tree is redundantly stored on all processes and, thus, the same set $\mathcal{N}(m, v)$ of m nearest neighbors of v is obtained; and

   b) the same distance between v and Γ_φ^h via a frontier vertex w is determined because all processes store the same distance between frontier vertices and Γ_φ^h.

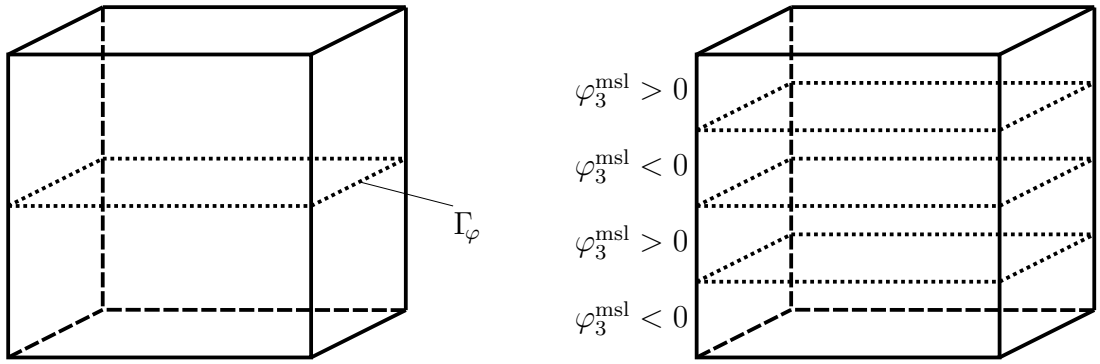
   Hence, all processes $p(v)$ evaluate the same distance between the off-site vertex v and Γ_φ^h.

This proves equation (5.25). So, overall, the output φ_h of the re-initialization algorithm based on direct distances is accumulated.

5.5 Numerical Results

In this section, we present numerical results concerning the re-initialization algorithm which is based on direct distances. The algorithm is implemented in the software DROPS and the implementation of the k-d trees is based on the library KDTREE 2 [110]. The decomposition of the computational domain is obtained by the T-GPTH problem, cf. Sect. 4.1.2, using the PARMETIS library [109]. Some of the numerical results in this section are taken from [69] and [70].

The experiments are performed for different problems defined by a triple (Ω, φ, s) with the following components. As computational domain, we distinguish between the measurement cell Ω^{M} presented in Fig. 4.13 and a cube-shaped domain Ω^{C} which is used to investigate the rising velocity of a single drop [20]. Its simple geometric structure enables to easily modify the edge length in the triangulation. As the level set function φ, we consider the following three different signed distance functions describing different interfaces Γ_φ.



(a) Interface for φ^{sl}. (b) Interface for φ_3^{msl}.

Figure 5.10: Computational domain Ω^{C}.

(i) The zero level of φ^{sl} represents a horizontal slice through the cube Ω^{C}, depicted in Fig. 5.10(a). The idea behind this choice of φ is to eliminate errors in computing distances to the interface which is exactly located at some vertices of $\mathcal{V}$.

(ii) The zero level of φ_b^{msl} describes multiple, equidistant horizontal slices through the cube Ω^{C}, as illustrated in Fig. 5.10(b). The index b denotes the number of slices. This function aims at increasing the number of frontier vertices compared to all vertices, i.e., the ratio $|\mathcal{F} \cup \mathcal{L}|/|\mathcal{V}|$. It is artificially constructed and does not reflect any of the two-phase flow problems we are interested in where typically the number of frontier vertices is significantly smaller than the number of off-site vertices.

(iii) The zero level of φ^{sp} describes a sphere, which is used as initial condition for φ if a drop is simulated in the measurement cell Ω^{M} or in the cube Ω^{C}.

The parameter s describes the level of refinement of the triangulation and is used to characterize the size of the problem.

In this section, we first investigate the accuracy of the re-initialization algorithm based on direct distances in Sect. 5.5.1. Afterwards, in Sect. 5.5.2, we present results concerning the performance of the algorithm on up to 128 processes which each employs up to 8 threads. So, in total, we utilize up to 1 024 compute cores for presenting the results.

5.5.1 Numerical Accuracy

We first compare the results of the re-initialization algorithm with a known solution to present the accuracy of the re-initialization algorithm. We therefore investigate

the same errors as in [98] which are defined as follows:

$$
\begin{aligned}
e_1(\varphi^{\text{comp}}) &:= \frac{1}{|\mathcal{S}|}\sum_{v\in\mathcal{S}}\frac{|\varphi^{\text{comp}}(v)-\varphi^{\text{ex}}(v)|}{|\varphi^{\text{ex}}(v)|},\\
e_2(\varphi^{\text{comp}}) &:= \sqrt{\frac{1}{|\mathcal{S}|}\sum_{v\in\mathcal{S}}\left(\frac{|\varphi^{\text{comp}}(v)-\varphi^{\text{ex}}(v)|}{|\varphi^{\text{ex}}(v)|}\right)^2}, \text{ and}\\
e_\infty(\varphi^{\text{comp}}) &:= \max_{v\in\mathcal{S}}|\varphi^{\text{comp}}(v)-\varphi^{\text{ex}}(v)|.
\end{aligned}
$$

Here, φ^{ex} denotes a given signed distance function and φ^{comp} is the computed, re-initialized level set function which is obtained from a disturbed φ^{ex}. We here only consider off-site vertices for the error because we want to focus on the propagation phase. However, we will present accuracy results for frontier vertices in Table 5.3. We furthermore use Ω^{C} as the computational domain to control the edge length h and to investigate the error. We set φ^{ex} to φ^{sp}, i.e.,

$$
\varphi^{\text{ex}}(\mathbf{x}) \;=\; \|\mathbf{x}-(0.5,0.5,0.5)^T\|_2 - 0.25,
$$

where the interface Γ_φ is a sphere of radius 0.25 located in the middle of Ω^{C}. The parallel re-initialization algorithm is applied to disturbed level set functions, either a linear disturbance

$$
\varphi_1^{\text{dist}}(\mathbf{x}) \;:=\; 100\cdot\varphi^{\text{ex}}(\mathbf{x}),
$$

a piece-wise linear disturbance

$$
\varphi_2^{\text{dist}}(\mathbf{x}) \;:=\; \begin{cases} 2.5\cdot\varphi^{\text{ex}}(\mathbf{x}), & \varphi^{\text{ex}}(\mathbf{x})\geq 0\\ 3.0\cdot\varphi^{\text{ex}}(\mathbf{x}), & \varphi^{\text{ex}}(\mathbf{x})<0 \end{cases},
$$

or a non-linear disturbance

$$
\varphi_3^{\text{dist}}(\mathbf{x}) \;:=\; \begin{cases} (\varphi^{\text{ex}}(\mathbf{x}))^2, & \varphi^{\text{ex}}(\mathbf{x})\geq 0\\ (\varphi^{\text{ex}}(\mathbf{x}))^3, & \varphi^{\text{ex}}(\mathbf{x})<0 \end{cases}.
$$

Five regular triangulations of Ω^{C} are analyzed. The coarsest triangulation consists of 384 tetrahedra. Refining each tetrahedron regularly, i.e., each edge is bisected, leads to the next finer triangulation. Here, we refine each tetrahedron up to four times recursively, leading to five triangulations with up to 12 582 912 tetrahedra on the finest level. On the coarsest triangulation, φ_h is discretized by 729 scalar values and on the finest triangulation by 16 974 593 values.

To avoid the approximation error introduced by the usage of k-d trees, we chose $m = |\mathcal{F}\cup\mathcal{L}|$ as the number of nearest neighbors, cf. Sect. 5.4.2. We will also

h	e_1	r_1	e_2	r_2	e_∞	r_∞
0.2165	0.08822	—	0.1308	—	0.06988	—
0.1082	0.02924	1.59	0.04954	1.40	0.02374	1.56
0.0541	0.01111	1.40	0.02574	0.94	0.01383	0.78
0.0271	0.004922	1.17	0.01701	0.60	0.006520	1.08
0.0135	0.002262	1.12	0.01133	0.59	0.003354	0.96

Table 5.2: Errors e_1, e_2, and e_∞ of φ^{comp} in Ω^{C} when re-initializing the disturbed level set function $\varphi^{\text{ex}} = \varphi^{\text{sp}}$ by φ_1^{dist}.

show results with respect to m in Fig. 5.11. In Table 5.2, the errors e_1, e_2, and e_∞ are depicted for all five triangulation hierarchies. This table also presents the corresponding orders r_p of the errors e_p for $p = 1$, 2 and ∞. In all cases, the error decreases with increasing grid refinement, i.e., decreasing h. With respect to the e_2 error the convergence order is greater that 0.59. When considering the e_∞ error, we observe a convergence order greater than 0.78. The best order corresponds to the e_1 error where $r_1 \geq 1.1$ is obtained. Note that these errors compare the re-initialized level set function with an exact level set function φ^{ex}. In the analytical analysis (5.24), we compared φ^{comp} and the interface described by input of the algorithm, i.e., $\varphi_h^{\text{old}} = \varphi_i^{\text{dist}}$ for $i = 1$, 2 and 3.

When solving two-phase flow problems, the gradient $\nabla\varphi_h$ is employed to determine surface forces at Γ_φ, cf. [84]. Therefore, another metric to assess the quality of the algorithm is to consider the error of $\nabla\varphi$ in the vicinity of Γ_φ. We measure this error by

$$e_\Gamma\left(\nabla\varphi^{\text{comp}}\right) \;:=\; \max_{\Gamma_\varphi^h}\Big\{\|\nabla\varphi^{\text{comp}} - \nabla\varphi^{\text{ex}}\|_2\Big\},$$

where the gradients are evaluated on tetrahedra that are intersected by Γ_φ^h. The error is primarily caused by the routine `InitFrontierSet()` in line 1 of Algorithm 2. It is possible that the tetrahedron containing the closest interface segment for a frontier vertex w is not located in the neighboring tetrahedra $T(w)$ as illustrated in Fig. 5.5. Therefore, we also consider the superset $\overline{T}(w) \supset T(w)$, see (5.6). We refer to [85] for more details on the definition of $\overline{T}(w)$. The error e_Γ and the corresponding order r_Γ is given in Table 5.3 for both approaches. Here, we consider the same triangulations as in Table 5.2 and, in contrast to the numerical results presented in [85], we re-initialize disturbed signed distance functions. In particular, we apply the re-initialization algorithm to the three functions φ_1^{dist}, φ_2^{dist}, and φ_3^{dist} for $\varphi^{\text{ex}} = \varphi^{\text{sp}}$. With increasing grid refinement, indicated by decreasing h, the error stagnates or even increases in all three Tables 5.3(a), (b), and (c) for $T(w)$. However, the convergence order r_Γ of the error e_Γ is close to one for φ_1^{dist}

using $\overline{T}(w)$. Although the superset $\overline{T}(w)$ is used, the error only decreases for up to $h = 0.0271$ for φ_2^{dist}. There is no decrease for φ_3^{dist}. An explanation for the divergence of the error for φ_2^{dist} and φ_3^{dist} is given by the following observation which has been outlined for the one-dimensional introductory example in Fig. 5.1. Since DROPS represents the level set function by a quadratic function, the discrete representation of the interface Γ_φ^h changes if disturbing φ^{ex} by φ_2^{dist} or φ_3^{dist}. That is, the discrete representation Γ_φ^h changes although the zero level of φ_2^{dist} and φ_3^{dist} are identical. Indeed, these disturbances of φ^{ex} results in such a change of Γ_φ^h in comparison to the exact interface Γ_φ that it cannot be recovered by the routine `InitFrontierSet()`. This leads to an error of determining the distance between frontier vertices and Γ_φ which causes the error in the computation of φ^{comp}. Thus, the error $e_\Gamma(\nabla\varphi^{\text{comp}})$ diverges. However, in practice-relevant applications of the re-initialization algorithm, level set functions are only slightly disturbed and the algorithm provides good results as illustrated in Fig. 5.3(a).

Next, we compare the accuracy of the re-initializing algorithm based on direct distances, Algorithm 3, and an implementation of the FMM in DROPS. Both methods employ the set $\overline{T}(w)$ for frontier vertices $w \in \mathcal{F}$ in the routine `InitFrontierSet()`. We use $m = 100$ nearest neighbors and re-initialize the level set function $\varphi^{\text{ex}} = \varphi^{\text{sp}}$ which is disturbed by φ_2^{dist}. The resulting e_2 error is illustrated in Table 5.4 and demonstrates that for this scenario Algorithm 3 delivers a more accurate solution that the FMM. When considering $h = 0.0271$ the error differs by two orders of magnitude. Furthermore, the FMM does not significantly decrease the error when decreasing h. In contrast, the re-initialization algorithm based on direct distances reduces the error when considering the edge lengths $h = 0.2165$ to $h = 0.0271$. As in Table 5.3(a), we observe that the e_2 error does not decrease for Algorithm 3 when decreasing h from 0.0271 to 0.0135.

To investigate the influence of the parameter m on the error, we use four different problem settings. For the cube $\Omega^{\text{C}} = [0,1]^3$, the level set function φ^{sl} is used. Using the measurement cell Ω^{M}, the function φ^{sp} is applied and, thus, Γ_φ is given by a sphere of radius 0.1 cm located at the narrowing part of the cell. To vary the problem size, we refine the triangulation in the vicinity of Γ_φ^h leading to two different problem sizes for each domain. A summary of the characteristics of all four used problems is given in Table 5.5. Consider the column depicting $|\mathcal{L} \setminus \mathcal{F}|$ in that table. If using Ω^{C} and φ^{sl} the perpendicular feet are located on frontier vertices for both problem sizes. Thus, the set $\mathcal{L} \setminus \mathcal{F}$ is empty and the accuracy cannot be increased by augmenting the set $\mathcal{F}$ by $\mathcal{L}$ when considering (5.9) rather than (5.8).

The influence of the parameter m on the e_1- and e_2-error is depicted in Fig. 5.11 for the two problems $(\Omega^{\text{M}}, \varphi^{\text{sp}}, \text{small})$ and $(\Omega^{\text{M}}, \varphi^{\text{sp}}, \text{large})$. Here, we observe that the error is large and decreases fast for $m < 10$. The error decreases only slightly for $10 \leq m < 100$ nearest neighbors. Choosing $m = 100$ results in the best

h	$T(w)$		$\overline{T}(w)$	
	e_Γ	r_Γ	e_Γ	r_Γ
0.2165	0.4144	—	0.3796	—
0.1082	0.2125	0.96	0.2099	0.85
0.0541	0.1755	0.28	0.1286	0.71
0.0271	0.2649	-0.60	0.06815	0.92
0.0135	0.3497	-0.40	0.03485	0.96

(a) Re-initialization of φ_1^{dist}.

h	$T(w)$		$\overline{T}(w)$	
	e_Γ	r_Γ	e_Γ	r_Γ
0.2165	0.4055	—	0.3791	—
0.1082	0.2062	0.98	0.2017	0.91
0.0541	0.1924	0.1	0.1020	0.98
0.0271	0.2423	-0.33	0.08900	0.20
0.0135	0.3491	-0.52	0.09248	-0.06

(b) Re-initialization of φ_2^{dist}.

h	$T(w)$		$\overline{T}(w)$	
	e_Γ	r_Γ	e_Γ	r_Γ
0.2165	0.6614	—	0.6733	—
0.1082	0.8175	-0.31	0.8056	-0.26
0.0541	0.7312	0.16	0.7233	0.16
0.0271	0.7993	-0.13	0.7771	-0.10
0.0135	0.7364	0.12	0.7596	0.03

(c) Re-initialization of φ_3^{dist}.

Table 5.3: Error $e_\Gamma(\nabla\varphi^{\text{comp}})$ in Ω^{C} when re-initializing the disturbed level set function $\varphi^{\text{ex}} = \varphi^{\text{sp}}$.

h	FMM e_2	Algorithm 3 e_2
0.2165	0.03871	0.01027
0.1082	0.05432	0.009452
0.0541	0.06886	0.006907
0.0271	0.06975	0.0006247
0.0135	0.07329	0.003377

Table 5.4: Comparison of the FMM and the Algorithm 3. Error e_2 in Ω^{C} when re-initializing the disturbed level set function $\varphi^{\mathrm{ex}} = \varphi^{\mathrm{sp}}$ by $\varphi_2^{\mathrm{dist}}$.

Problem	$\lvert\mathcal{T}\rvert/10^6$	$\lvert\mathcal{V}\rvert/10^6$	$\lvert\mathcal{F}\rvert/10^3$	$\lvert\mathcal{L}\setminus\mathcal{F}\rvert/10^3$	$\lvert\mathcal{S}\rvert/10^6$	$\frac{\lvert\mathcal{F}\cup\mathcal{L}\rvert}{\lvert\mathcal{V}\rvert}$ [%]
$(\Omega^{\mathrm{C}}, \varphi^{\mathrm{sl}}, \text{small})$	0.225	0.275	4.23	0	0.270	1.54
$(\Omega^{\mathrm{C}}, \varphi^{\mathrm{sl}}, \text{large})$	14.4	17.0	66.0	0	16.9	0.39
$(\Omega^{\mathrm{M}}, \varphi^{\mathrm{sp}}, \text{small})$	0.420	0.482	37.1	1.84	0.445	8.00
$(\Omega^{\mathrm{M}}, \varphi^{\mathrm{sp}}, \text{large})$	2.73	3.15	149.0	7.36	3.00	4.95

Table 5.5: Problem characteristics.

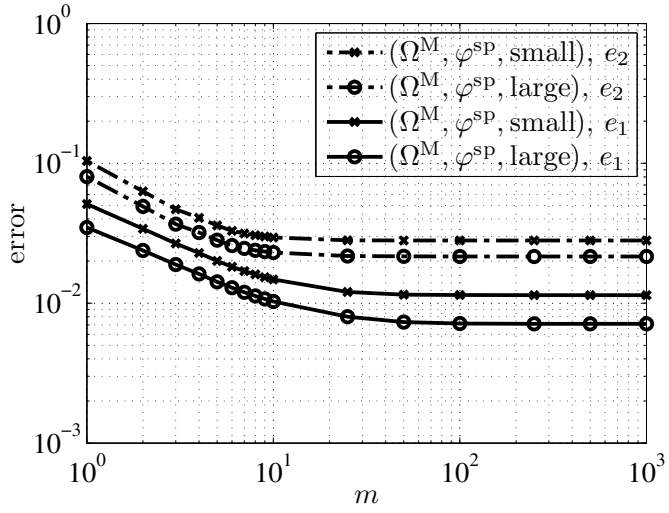


Figure 5.11: Error e_1 and e_2 for problem $(\Omega^{\mathrm{M}}, \varphi^{\mathrm{sp}}, \mathrm{small})$ and $(\Omega^{\mathrm{M}}, \varphi^{\mathrm{sp}}, \mathrm{large})$ with respect to the number of nearest neighbors m when re-initializing the disturbed level set function φ^{ex} by $\varphi_1^{\mathrm{dist}}$.

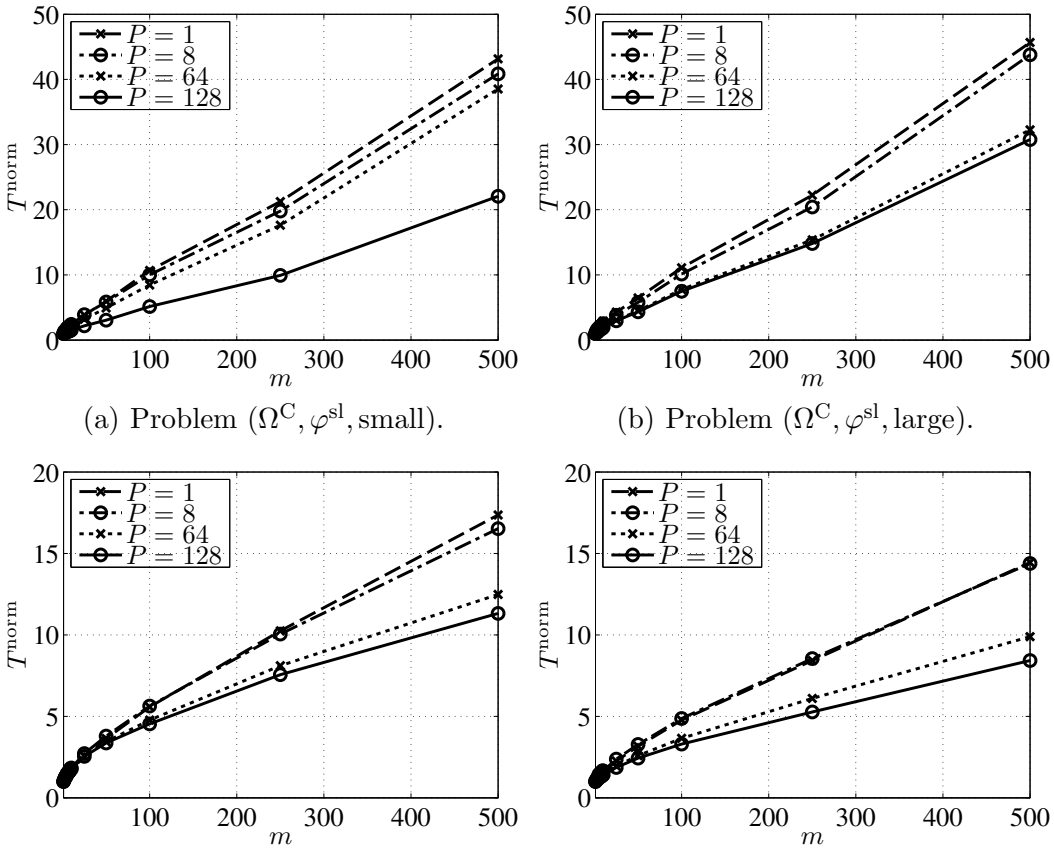
approximation and, thus, increasing the value of m any further does not lead to any improvement of the re-initialization algorithm for these two problems with respect to the e_1- and e_2-error.

5.5.2 Serial and Parallel Performance

Next, we consider performance issues of the re-initialization algorithm based on direct distances. Therefore, we first analyze the dependency of the number of nearest neighbors m on the runtime. Then, we compare the serial runtime of Algorithm 3 with the computational time of the FMM. Afterwards, we focus on the scalability of the re-initialization algorithm. Finally, we address some limitations of the novel algorithm. All results are gathered on the two different clusters consisting of Xeon-based quad-core processors whose characteristics are detailed in Table 3.2. To distinguish between both processor types, we label the results by either "Harpertown" or "Nehalem," respectively.

In Sect. 5.1, we demonstrate that the theoretic runtime of the parallel re-initialization Algorithm 3 does not only depend on the number of processes P but also on the choice of m, cf. Table 5.1. To investigate this dependency, let $T_m(P)$ denote the runtime of the parallel algorithm on P processes with the user-given parameter m. In Fig. 5.12, the normalized times

$$T^{\mathrm{norm}}(m, P) \ := \ \frac{T_m(P)}{T_1(P)} \tag{5.26}$$



(a) Problem $(\Omega^{\text{C}}, \varphi^{\text{sl}}, \text{small})$. (b) Problem $(\Omega^{\text{C}}, \varphi^{\text{sl}}, \text{large})$.

(c) Problem $(\Omega^{\text{M}}, \varphi^{\text{sp}}, \text{small})$. (d) Problem $(\Omega^{\text{M}}, \varphi^{\text{sp}}, \text{large})$.

Figure 5.12: Normalized runtime $T^{\text{norm}}(m, P) = T_m(P)/T_1(P)$ of the re-initializing algorithm for $P = 1, 8, 64$ and 128 processes (Harpertown).

are illustrated for $P \in \{1, 8, 64, 128\}$ processes for all four problems presented in Table 5.5. For a given number of processes P, the ratio (5.26) increases for a growing number of nearest neighbors m. However, the time ratios do not depend linearly on m. For instance, the ratio for $m = 500$ and $P = 1$ in problems $(\Omega^{\text{C}}, \varphi^{\text{sl}}, \text{small})$ and $(\Omega^{\text{C}}, \varphi^{\text{sl}}, \text{large})$ is about 45 and not 500 as suggested by (5.19), cf. Fig. 5.12(a) and (b). The corresponding ratio is less than 18 for problems $(\Omega^{\text{M}}, \varphi^{\text{sp}}, \text{small})$ and $(\Omega^{\text{M}}, \varphi^{\text{sp}}, \text{large})$, cf. Fig. 5.12(c) and (d). For fixed m, the ratio $T^{\text{norm}}(m, P)$ in (5.26) tends to decrease with increasing P for all four problems. That is, the influence of the parameter m on the runtime becomes less and less important when increasing the number of processes. An explanation is that the time for building the k-d tree (5.20)—which is independent of m—starts to dominate the runtime.

We compare the serial runtime of our implementation of the FMM and the runtime of the novel re-initialization algorithm based on direct distances when

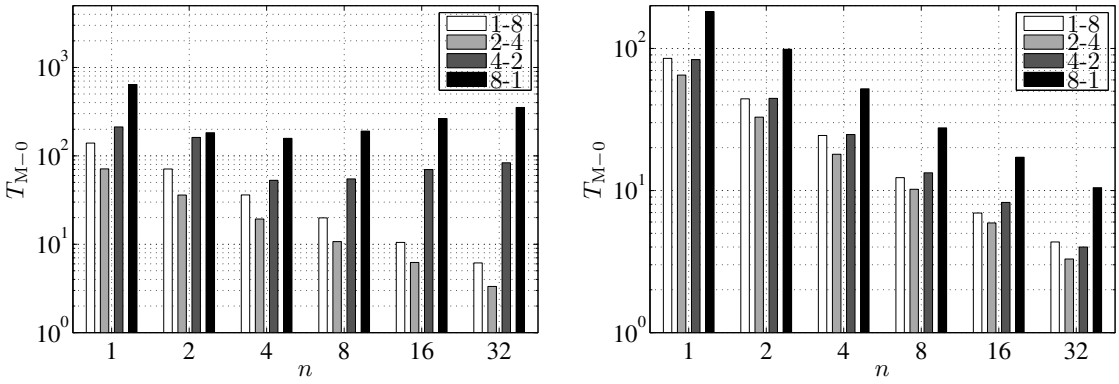
Problem	$T_{\mathrm{FMM}}/T_{100}(1)$
$(\Omega^{\mathrm{C}}, \varphi^{\mathrm{sl}}, \mathrm{small})$	1.29
$(\Omega^{\mathrm{C}}, \varphi^{\mathrm{sl}}, \mathrm{large})$	—
$(\Omega^{\mathrm{M}}, \varphi^{\mathrm{sp}}, \mathrm{small})$	1.16
$(\Omega^{\mathrm{M}}, \varphi^{\mathrm{sp}}, \mathrm{large})$	1.24

Table 5.6: Ratio of serial runtime of the FMM and Algorithm 3 (Harpertown).

choosing $m = 100$. To this end, let T_{FMM} denote the runtime of the FMM. The ratios $T_{\mathrm{FMM}}/T_{100}(1)$ are given in Table 5.6 for all four problems. These results in this table indicate that the runtime of the FMM is larger than the runtime of Algorithm 3 by a factor of about 1.2. Note that it was not possible to solve the re-initialization problem $(\Omega^{\mathrm{C}}, \varphi^{\mathrm{sl}}, \mathrm{large})$ by the FMM because its memory requirement exceeds the maximum available memory of the platform, i.e., 16 GB.

Now, we consider the speedup of the hybrid distributed-/shared-memory parallel re-initialization algorithm based on direct distances which is given in Algorithm 4. Note that the results are taken from [69]. Recall that both clusters of processors provide eight cores per node that have access to a shared memory. We describe different schemes to place the MPI processes and OpenMP threads on these eight cores per node. We express the schemes by M-O. Here, M denotes the number of MPI processes placed on each node. For each such process, O denotes the number of OpenMP threads used to execute the parallel regions of the shared-memory parallelization. Thus, if n nodes are used, the total number of cores executing the algorithm is given by $n \cdot M \cdot O$. For instance, if using the 2-4 scheme, two processes are placed on each node and each process spawns four threads leading to eight threads per node. The total number of MPI processes is given by $M \cdot n$. The execution time of the algorithm carried out on n nodes using the M-O scheme is denoted by $T_{M\text{-}O}(n)$. In the following discussion on the performance of the hybrid parallel algorithm, we set $m = 1\,000$. This rather high number of nearest neighbors is advantageous for the hybrid parallel approach. However, in [70], we demonstrate the scalability of the novel algorithm when considering $m = 25$ and $m = 100$. To simplify notations in the remainder of this section, we do not label the compute times with m.

First, we investigate which M-O scheme exploits the hardware best. In Fig. 5.13, we present the execution time $T_{M\text{-}O}(n)$ on various number of compute nodes n to determine $d^p\left(\Gamma_\varphi^h; v\right)$ for all $v \in \mathcal{S}$ of $(\Omega^{\mathrm{M}}, \varphi^{\mathrm{sp}}, \mathrm{large})$, i.e., performing lines 9–12 of Algorithm 4. In Fig. 5.13(a), the execution times on a cluster consisting of Harpertown processors is presented whereas in Fig. 5.13(b), a cluster of Nehalem processors is used. Although the clock speed of both processor types is approx-

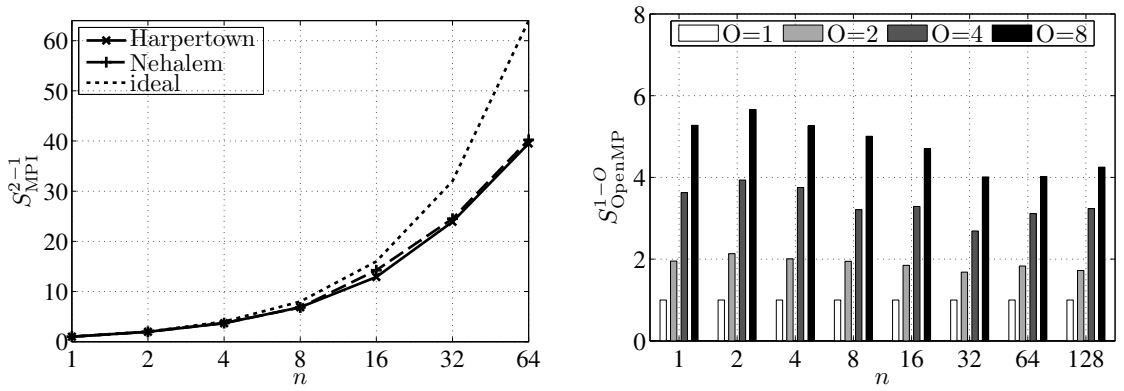


(a) Harpertown. (b) Nehalem.

Figure 5.13: Execution time $T_{8\text{-}1}(n)$ of carrying out the loop *SharedMem (2)* (lines 9–12 of Algorithm 4) by n nodes.

imately equal to 3 Ghz, the execution time on the Harpertown cluster is larger because the connection between the memory and the Nehalem's cores is better than the connection to the Harpertown's cores. If considering the pure MPI parallelization, i.e., the scheme 8-1, the algorithm scales on up to 4 nodes of the Harpertown cluster and up to 32 nodes of the Nehalem cluster. An explanation is given by the InfiniBand network connecting the nodes. The Nehalem cluster uses a newer generation of an InfiniBand network than the Harpertown cluster. This newer network is faster and, in contrast to the Harpertown cluster, the network interface cards can better serve the eight MPI processes on a node. Both figures demonstrate that exclusively using an MPI parallelization yields the largest execution times on a fixed number of nodes because the shared memory can not be exploited. For instance, in Fig. 5.13(b), switching from a hybrid MPI/OpenMP parallelization to a pure MPI parallelization increases the runtime on four nodes from $T_{1\text{-}8}(4) = 24.8\,\text{s}$ to $T_{8\text{-}1}(4) = 52.1\,\text{s}$. Thus, four nodes of the Nehalem cluster are capable of reducing the serial runtime of $T_{1\text{-}8}(4) = 450.0\,\text{s}$ by a factor of 18.1 by using the hybrid parallel approach whereas the corresponding factor is 8.6 for the pure MPI approach. For both types of processors, the smallest execution time is generally obtained if placing one process and four threads on each of the two processors of a node, i.e., using strategy 2-4. In addition, the Nehalem processors provide simultaneous multithreading (SMT) which allows to place up to 16 threads on one node. However, our experiments show that enabling this technology increases the runtime on one node by a factor 1.06 compared to only using eight threads.

Second, we investigate the speedup of Algorithm 4. To present results concerning the distributed MPI parallelization, we define the speedup $S^{M\text{-}O}_{\text{MPI}}(n)$ using n



(a) MPI speedup $S^{2\text{-}1}_{\text{MPI}}(n)$.

(b) OpenMP speedup $S^{1\text{-}O}_{\text{OpenMP}}(n)$ (Nehalem).

Figure 5.14: Speedup of the hybrid parallel re-initialization algorithm (Algorithm 4) with respect to the number of nodes n.

nodes and the scheme M-O by

$$S^{M\text{-}O}_{\text{MPI}}(n) \;:=\; \frac{T_{M\text{-}O}(1)}{T_{M\text{-}O}(n)}.$$

For a fixed number of nodes n, the speedup with respect to shared-memory OpenMP parallelization using O threads is defined by

$$S^{M\text{-}O}_{\text{OpenMP}}(n) \;:=\; \frac{T_{M\text{-}1}(n)}{T_{M\text{-}O}(n)}.$$

In Fig. 5.14, both speedups of the re-initializing algorithm are depicted. The speedup $S^{2\text{-}1}_{\text{MPI}}$ is illustrated in Fig. 5.14(a) for a varying number of nodes. This figure demonstrates that the parallel algorithm scales well on both clusters using a pure MPI parallelization with two processes per node. That is, on both clusters, the execution time of two MPI processes on one node is reduced by a factor of about 40 if using 64 nodes. In Fig. 5.14(b), the speedup $S^{1\text{-}O}_{\text{OpenMP}}$ with respect to a varying number of threads is shown for different numbers of Nehalem nodes. This figure illustrates that the shared-memory parallelization is capable of decreasing the runtime by a factor larger than 4 for all nodes using $O = 8$ cores.

The parallel speedup of the re-initialization algorithm based on direct distances, Algorithm 3, is limited by three factors:

(i) the communication costs of gathering in lines 12–14;

(ii) the load imbalances in $\mathcal{V}^p$, $\mathcal{F}^p$ and $\mathcal{S}^p$; and

(iii) the redundant generation of the k-d tree in line 15

Problem	m	$T_m(1)$ [s]	$\alpha_m(1)$ [%]	$T_m(128)$ [s]	$\alpha_m(128)$ [%]
$(\Omega^{\mathrm{C}}, \varphi^{\mathrm{sl}}, \mathrm{small})$	25	1.28	0.07	0.02	3.2
$(\Omega^{\mathrm{C}}, \varphi^{\mathrm{sl}}, \mathrm{large})$	25	99.4	0.02	0.88	1.6
$(\Omega^{\mathrm{M}}, \varphi^{\mathrm{sp}}, \mathrm{small})$	25	4.68	0.16	0.20	4.4
$(\Omega^{\mathrm{M}}, \varphi^{\mathrm{sp}}, \mathrm{large})$	25	39.7	0.09	0.75	5.5
$(\Omega^{\mathrm{C}}, \varphi^{\mathrm{sl}}, \mathrm{small})$	100	3.56	0.03	0.05	1.3
$(\Omega^{\mathrm{C}}, \varphi^{\mathrm{sl}}, \mathrm{large})$	100	256	0.007	2.24	0.6
$(\Omega^{\mathrm{M}}, \varphi^{\mathrm{sp}}, \mathrm{small})$	100	10.2	0.075	0.35	2.5
$(\Omega^{\mathrm{M}}, \varphi^{\mathrm{sp}}, \mathrm{large})$	100	83.8	0.042	1.33	3.1

Table 5.7: Time for re-initialization a level set function and fraction $\alpha_m(P)$ of building the k-d tree on 1 and 128 processes (Harpertown).

In this section, we focus on the aspect (iii). To this end, let $\alpha_m(P)$ denote the fraction of the runtime $T_m(P)$ that is used to build the k-d tree. In Table 5.7, the runtime $T_m(P)$ of Algorithm 3 as well as $\alpha_m(P)$ are presented for 1 and 128 processes using $m = 25$ and $m = 100$ nearest neighbors. For a given parameter m, increasing P also increases $\alpha_m(P)$ which can be explained by the redundant computation involved in building the k-d tree. If P is fix and m increases, more time is spent in searching for nearest neighbors in the tree so that building the tree becomes less time dominant. Hence, $\alpha_m(P)$ is decreased. For both choices of m in problems $(\Omega^{\mathrm{C}}, \varphi^{\mathrm{sl}}, \mathrm{small})$ and $(\Omega^{\mathrm{C}}, \varphi^{\mathrm{sl}}, \mathrm{large})$, the fraction $\alpha_m(P)$ is reduced by increasing the problem size because the ratio $|\mathcal{F} \cup \mathcal{L}|/|\mathcal{V}|$ is decreased, cf. Table 5.5. This behavior also applies for problems $(\Omega^{\mathrm{M}}, \varphi^{\mathrm{sp}}, \mathrm{small})$ and $(\Omega^{\mathrm{M}}, \varphi^{\mathrm{sp}}, \mathrm{large})$ on one process. For these two problems, the fractions $\alpha_{25}(128)$ and $\alpha_{100}(128)$ increase by a factor of about 1.25 if switching from the small to the large problem size. This can be explained by the following argument. The sizes of all sets are increased. However, from inspection of the values in Table 5.5, this increase by factors less than 7 is smaller than the increase in the number of processes from 1 to 128. Therefore, (5.20) starts to dominate the runtime of the parallel algorithm leading to an increase of $\alpha_m(128)$.

If we assume perfect speedup for all parts except for building the k-d tree, Amdahl's law [7] implies that, for an infinite number of processes, the speedup of the algorithm is bounded by $S_m(\infty) := 1/\alpha_m(1)$. For instance, in the small problem $(\Omega^{\mathrm{M}}, \varphi^{\mathrm{sp}}, \mathrm{small})$, the ratio $\alpha_{25}(1) = 0.16\,\%$ implies that the speedup is bounded by $S_{25}(\infty) = 625$, whereas for the large problem $(\Omega^{\mathrm{C}}, \varphi^{\mathrm{sl}}, \mathrm{large})$, the

Problem	$\lvert\mathcal{T}\rvert/10^6$	$\lvert\mathcal{V}\rvert/10^6$	$\lvert\mathcal{F}\rvert/10^6$	$\lvert\mathcal{L}\setminus\mathcal{F}\rvert/10^6$	$\lvert\mathcal{S}\rvert/10^6$	$\frac{\lvert\mathcal{F}\cup\mathcal{L}\rvert}{\lvert\mathcal{V}\rvert}$ [%]
$(\Omega^{\mathrm{C}}, \varphi_{30}^{\mathrm{msl}}, \mathrm{small})$	0.225	0.275	0.237	0.0545	0.0380	106.0
$(\Omega^{\mathrm{C}}, \varphi_{60}^{\mathrm{msl}}, \mathrm{large})$	14.4	17.0	10.0	1.55	6.94	68.3

Table 5.8: Problem characteristics of the artificially multi-sliced problems.

fraction of the serial runtime is given by $\alpha_{100}(1) = 0.007\,\%$ yielding a theoretic limit for the speedup of $S_{100}(\infty) = 14\,286$.

We artificially increase the parameter α_m to stress the parallel performance of the re-initialization algorithm and to address some limitations of the current implementation of the algorithm. Since $\alpha_m(P)$ denotes the time of redundantly building the k-d tree by each process, this part of the re-initialization algorithm may constitute the performance bottleneck in terms of runtime and memory consumption, as described in Sect. 5.4.1. To investigate this potential bottleneck, we construct two artificial problems with no practical relevance. However, in the following discussion we show that—even for artificially constructed problems—a serial implementation of the k-d tree is reasonable to avoid any communication. The characteristics of these two artificial problems are given in Table 5.8 where the zero level of the signed distance functions describe multiple horizontal slices, i.e., $\varphi^{\mathrm{ex}} = \varphi_b^{\mathrm{msl}}$ with $b \in \{30, 60\}$. These problems are designed such that the number of frontier vertices and perpendicular feet compared to the total number of vertices is significantly larger than in the previous investigated problems from Table 5.5. In particular, the ratios $|\mathcal{F}\cup\mathcal{L}|/|\mathcal{V}|$ in Table 5.5 are below 10 %, whereas the corresponding ratios in Table 5.8 are 106 % and 68 %. In these problems, the fraction $\alpha_{100}(P)$ of the time spent in serially building the k-d tree is given in Table 5.9. These fractions are by orders of magnitude larger than those in Table 5.7 for $P = 1$ and $m = 100$. For instance, the value of the serial fraction is given in Table 5.9 by $\alpha_{100}(1) = 5.16\,\%$ whereas the fraction $\alpha_{100}(1) = 0.03\,\%$ is observed for the problem $(\Omega^{\mathrm{C}}, \varphi^{\mathrm{sl}}, \mathrm{small})$ in Table 5.7. So, Amdahl's law predicts lower speedups $S_m(\infty)$ for these artificial problems. The serial fraction $\alpha_{100}(1) = 5.16\,\%$ leads to $S_m(\infty) = 19.4$. Clearly, the fraction of the time spent in serially building the k-d tree increases when increasing the number of processes. Consider the problem $(\Omega^{\mathrm{C}}, \varphi_{60}^{\mathrm{msl}}, \mathrm{large})$ as an example. Here, roughly 3 % of the runtime is spent in building the tree using one process. However, this part consumes about 45.5 % of the time on $P = 64$ processes. The corresponding speedup of $S_{100}(64) = 14$ is indeed low.

Although the serial use of the k-d tree is a substantial bottleneck for these problems, the serial runtime of novel re-initialization algorithm based on direct distances is about 3 times smaller than that of the FMM applied to the problem $(\Omega^{\mathrm{C}}, \varphi_{30}^{\mathrm{msl}}, \mathrm{small})$. The storage requirement for the k-d tree is only a small

Problem	$\alpha_{100}(1)$ [%]	$\alpha_{100}(64)$ [%]	$S_{100}(\infty)$	$S_{100}(16)$	$S_{100}(64)$
$(\Omega^{\mathrm{C}}, \varphi_{30}^{\mathrm{msl}}, \mathrm{small})$	5.16	37.0	19.38	5.0	5.8
$(\Omega^{\mathrm{C}}, \varphi_{60}^{\mathrm{msl}}, \mathrm{large})$	2.81	45.5	35.59	8.2	14.0

Table 5.9: Parallel performance and relative time α_m for building the k-d tree of the artificial multi-sliced problems (Harpertown).

fraction of the memory used to solve the corresponding two-phase flow problem by DROPS. Therefore, storing the k-d tree redundantly on all processes offers the advantage to avoid communication between processes while determining the m nearest neighbors. Table 5.10 gives an overview of the storage requirements for all problems investigated in this section. Here, the symbol M_{KD} denotes the memory used to store the k-d tree on a single process. The storage for the solution of the two-phase flow problem accumulated over P processes is given by $M_{\mathrm{DROPS}}(P)$. The symbols M_{DD} and M_{FMM} are used for the memory requirements of the re-initialization algorithm based on direct distances, Algorithm 3, and the FMM on one process. As expected, the memory requirement of the k-d tree for problem $(\Omega^{\mathrm{C}}, \varphi_{60}^{\mathrm{msl}}, \mathrm{large})$ is larger than for all other problems. Clearly, the memory requirement $M_{\mathrm{DROPS}}(8)$ is much larger than M_{KD} for all problems. Indeed, the memory requirement $M_{\mathrm{DROPS}}(8)$ for problems $(\Omega^{\mathrm{C}}, \varphi_{60}^{\mathrm{msl}}, \mathrm{large})$ and $(\Omega^{\mathrm{C}}, \varphi^{\mathrm{sl}}, \mathrm{large})$ exceeds the memory capacity of the platform. For the other problems, the ratio $8 \cdot M_{\mathrm{KD}}/M_{\mathrm{DROPS}}(8)$ represents the fraction of redundantly storing the k-d tree compared to the total memory requirement of solving the the two-phase flow problem on 8 processes. This fraction ranges from 0.05 % to 3.8 %. However, the largest fraction is observed for the artificial problem $(\Omega^{\mathrm{C}}, \varphi_{30}^{\mathrm{msl}}, \mathrm{small})$ where the k-d tree is intentionally constructed large. The largest fraction is actually quite small. Hence, storing the k-d tree redundantly is affordable. For all problems, Algorithm 3 needs less memory than the FMM. Recall from the previous discussion that the implementation of the FMM cannot solve the large problem $(\Omega^{\mathrm{C}}, \varphi^{\mathrm{sl}}, \mathrm{large})$ with the available memory. This also holds for the problem $(\Omega^{\mathrm{C}}, \varphi_{60}^{\mathrm{msl}}, \mathrm{large})$.

5.6 Discussion

In this chapter, we presented a parallel algorithm that is capable of re-initializing level set function on a distributed unstructured grid. Therefore, we combined a brute-force re-initialization scheme and, to reduce the time complexity, a multi-dimensional tree data structure, k-d trees. We observed a good performance on recent high-performance computers consisting of clusters of multi-core processors. Furthermore, we demonstrated that the algorithmic complexity and the accuracy

Problem	M_{KD} [MB]	$M_{\text{Drops}}(8)$ [GB]	$8\,M_{\text{KD}}/M_{\text{Drops}}(8)$ [%]	M_{FMM} [MB]	M_{DD} [MB]
$(\Omega^{\text{C}}, \varphi_{30}^{\text{msl}}, \text{small})$	15.3	3.4	3.8	201.2	33.1
$(\Omega^{\text{C}}, \varphi_{60}^{\text{msl}}, \text{large})$	610.6	—	—	—	1 498.6
$(\Omega^{\text{C}}, \varphi^{\text{sl}}, \text{small})$	0.2	2.9	0.053	201.0	9.0
$(\Omega^{\text{C}}, \varphi^{\text{sl}}, \text{large})$	3.7	—	—	—	539.9
$(\Omega^{\text{M}}, \varphi^{\text{sp}}, \text{small})$	11.7	4.4	2.1	367.3	33.4
$(\Omega^{\text{M}}, \varphi^{\text{sp}}, \text{large})$	85.0	32.1	2.1	2 408.1	233.6

Table 5.10: Memory consumption of Drops (Harpertown and Nehalem).

of the new algorithm based on direct distances is comparable to those of the fast marching method.

To even increase the parallel efficiency and the numerical accuracy of the presented algorithms, future research directions are addressed in the following. In the current implementation of the algorithm, the limiting factor for the parallel scalability is given by the serial generation of the k-d tree. Thus, the performance of the algorithm can be increased by generating the k-d tree in parallel [43] or even replace the k-d tree data structure by another parallel data structure that is capable of efficiently determining nearest neighbors. Moreover, a narrow-band approach [2] is not yet considered, where the values of the level set function are only re-initialized on vertices in the vicinity of the interface. In addition, the approach presented in this chapter does not exploit the hierarchy of tetrahedral grids which discretizes the computational domain. This data structure can be employed to efficiently implement the narrow-band method where only vertices close to the interface are considered. The accuracy can be increased by two different approaches. First, we observe that the closest segment of the interface to a frontier vertex may be located on another process than the frontier vertex itself, cf. Fig. 5.5. Hence, determining the distance of frontier vertices is a delicate task in parallel and demands for future research. Second, in this thesis, the accuracy of the re-initialization algorithm has been increased by the following strategy. To determine the distance between off-site vertices and the interface, the set of frontier vertices has been suitably augmented by the set of perpendicular feet in (5.9). This results in a larger, more accurate search space for this distance computation. To further increase the accuracy, this search space can be augmented by adding more points located at the interface. However, enlarging the search space implies larger execution times. This trade-off has not been addressed in this thesis and requires future research efforts.

6 Case Study

In the previous chapters, we presented parallel algorithms and methods aiming at simulating two-phase flow problems by multiple processes. These techniques facilitate the consideration of problems that are too large—in terms of memory and compute time—for sequential computing. For instance, a pure serial approach is not feasible for investigating the so-called grid convergence when simulating the standard test system of a rising n-butanol drop in a surrounding aqueous phase [122]. In this convergence analysis, we are interested in the effect of grid refinements on the rising velocity of the n-butanol drop. Here, applying a large number of local refinements easily results in an amount of data that exceeds the memory available to one processor. Therefore, we employ the algorithms and methods presented in the previous chapters to perform the grid convergence analysis of a rising n-butanol drop. Besides a detailed study of the rising velocity, in this chapter, we demonstrate the parallel performance of DROPS for this numerical experiment.

In Sect. 6.1, we describe the experimental setup and the simulation algorithm used to investigate the rising velocity $v^{\mathrm{ris}}(\tau)$ with respect to time τ of the n-butanol drop in an aqueous phase. In Sect. 6.2, we present numerical results concerning the rising velocity and the parallel performance of this two-phase flow experiment.

6.1 Experimental Setup

The study of the rising n-butanol drop originates from the collaborative work [20] where the simulated rising velocity of the drop is compared with "real-world" experiments. The experiments in that publication are performed in an apparatus that is schematically depicted in Fig. 6.1. Here, n-butanol drops are generated through a nozzle submerged in a cylindrical cell that contains an aqueous phase consisting of deionized and bidistilled water. The density, viscosity, and surface tension for this two-phase system are given in Table 6.1. Upon generation, the drop starts to accelerate upwards until it reaches its terminal rising velocity which is determined by monitoring the drop's position by a camera. The numerical simulation is carried out by DROPS in a cuboid-shaped domain to predict the position and velocity of the drop over time. The collaborative work [20] describes

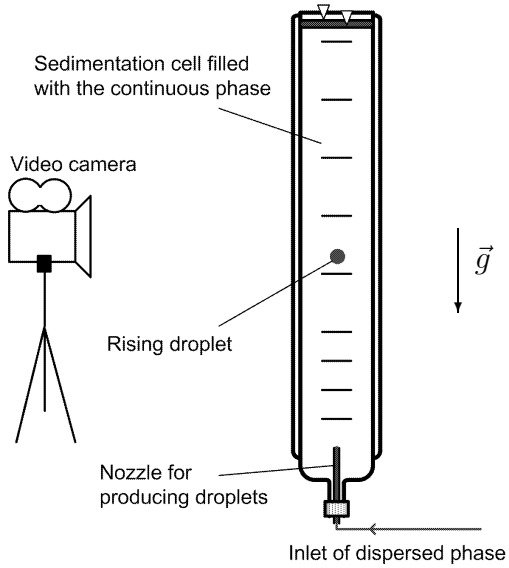


Figure 6.1: Sedimentation apparatus for performing the "real-life" experiments, taken from [20].

Property	n-butanol	aqueous phase
Density [kg/m^3]	845.442	986.506
Viscosity [Pa s]	3.281	1.388
Interfacial tension [mN/m]	1.64	

Table 6.1: Physicochemical properties of the two phases.

the experimental setup in detail and presents further results on the comparison between the numerical results and the "real-world" experiments.

In this thesis, we analyze the rising velocity $v^{\mathrm{ris}}(\tau)$ of an n-butanol drop with an initial diameter of 2 mm in detail. To this end, we perform different parallel two-phase flow simulations with DROPS. Note that the work [71] also deals with the parallel approach of this simulation, however, that work does not take extended finite elements for the pressure into account. The library PARMETIS [107] is employed to find a solution of the graph partitioning problems. The library DDD [24] is used to manage and to track the information about distributed elements of the tetrahedral hierarchies. As in [85], the simulated three-dimensional velocity $\mathbf{v}_h^{\mathrm{drop}}$ of the drop at time τ is determined by numerically evaluating the formula

$$\mathbf{v}_h^{\mathrm{drop}}(\tau) \;=\; \frac{1}{|\Omega_1(\tau)|}\int_{\Omega_1(\tau)} \mathbf{u}_h(\mathbf{x},\tau)\,\mathrm{d}\mathbf{x},$$

where $\Omega_1(\tau)$ denotes the domain of the drop, $|\Omega_1(\tau)|$ its volume, and $\mathbf{u}_h$ the simulated velocity field in the computational domain. Then, the rising velocity $v^{\mathrm{ris}}(\tau)$ is given by the second component of $\mathbf{v}_h^{\mathrm{drop}}$ which points upwards, in formula

$$v^{\mathrm{ris}}(\tau) \;:=\; (\mathbf{v}_h^{\mathrm{drop}}(\tau))_2. \tag{6.1}$$

In the computational domain $\Omega^{\mathrm{C}} = [0,12] \times [0,30] \times [0,12]\,\mathrm{mm}^3$, the two-phase flow simulation follows the approach presented in Chap. 2. An implicit, linearized theta-scheme, with $\theta = 1$, is applied for the time integration and the underlying Oseen problems are solved by the inexact Uzawa method. Within this method the Schur complement is preconditioned by a modified Cahouet-Chabard preconditioner [33] which is detailed in [85, 127] whereas a Jacobi preconditioner is used for the Krylov subspace methods. A decomposition of the tetrahedral hierarchy is determined by the 11-TP-GPTH graph model of Chap. 4. The signed-distance property of the level set function is enforced by the novel re-initialization algorithm based on direct distances which is presented in Chap. 5. The user-given parameter for the number of nearest neighbors is set to $m = 100$. In this chapter, we consider two different simulation strategies which differ in the finite element discretization of the pressure. In the first strategy, *XFEM*, extended piece-wise linear P_1^Γ finite functions are employed. The second one, *woXFEM*, considers piece-wise linear P_1 functions to represent the pressure. In both strategies, the computational domain Ω^{C} is discretized by small cubes of size $[0,4]^3\,\mathrm{mm}^3$ which are each divided into six tetrahedra yielding the coarsest triangulation. To explain the tetrahedral hierarchy, let

$$\Omega_\Gamma^{\mathrm{C}}(d) \;:=\; \left\{ \mathbf{x} \in \Omega^{\mathrm{C}} \;\middle|\; \min_{\mathbf{p} \in \Gamma_\varphi^h} \|\mathbf{x} - \mathbf{p}\|_2 \leq d \right\}$$

denote the set of points which are located in a neighborhood of radius d around the discrete representation Γ_φ^h of the interface Γ_φ. For each of the two strategies, four different tetrahedral hierarchies are used. These hierarchies vary in the number of refinement levels and the size of the refined subdomain. For the strategy *XFEM*, each tetrahedron whose barycenter is located in $\Omega_\Gamma^{\mathrm{C}}(16\,\mathrm{mm})$ is consecutively refined two to five times in the beginning of the simulation. Ongoing work is concerned with a parallel adaptive refinement algorithm which is capable of handling extended finite element functions. To still present results obtained with a parallel XFEM method, we consider a static tetrahedral hierarchy for the strategy *XFEM*, i.e., the hierarchy is not modified during the two-phase flow simulation. In contrast, the strategy *woXFEM* employs an adaptively refined tetrahedral hierarchy in the domain $\Omega_\Gamma^{\mathrm{C}}(2\,\mathrm{mm})$. That is, if the position of the drop changes, the tetrahedral hierarchy is adjusted such that the tetrahedra located in $\Omega_\Gamma^{\mathrm{C}}(2\,\mathrm{mm})$ are refined two to five times. Recall that l refinements yields a tetrahedral hierarchy $\mathcal{M} = (\mathcal{T}_0, \ldots, \mathcal{T}_l)$ of $l+1$ levels. Thus, the tetrahedral hierarchies consist of three to six triangulation levels. In the remainder, we denote the number of triangulation levels by k.

Triang. level k	P	$h_{\min}$ [mm]	$n_{\mathbf{u}}$	$\Delta\tau$ [ms]	time steps
3	4	3/4	162 000	2	200
4	32	3/8	1 282 557	1	400
5	160	3/16	10 187 019	0.5	689
6	680	3/32	81 138 516	0.1	926

(a) *XFEM*.

Triang. level k	P	$h_{\min}$ [mm]	$n_{\mathbf{u}}$	$\Delta\tau$ [ms]	time steps
3	2	3/4	20 670	1	500
4	4	3/8	102 657	1	500
5	24	3/16	599 622	0.5	1 000
6	72	3/32	4 006 101	0.1	1 002

(b) *woXFEM*.

Table 6.2: Characteristics of the different grid resolutions with respect to the number of the triangulation levels k.

Table 6.2(a) summarizes the simulation setup for the strategy *XFEM* with respect to the number of triangulation levels k, whereas Table 6.2(b) presents the same summary for the strategy *woXFEM*. In both tables, the column labeled by P shows the number of processes that are used for the simulation. Furthermore, in these tables, the minimal edge length $h_{\min}$ of tetrahedra intersected by Γ_φ and the number of velocity DOF $n_{\mathbf{u}}$ are given. Note that due to the non-linear coupling of the Navier-Stokes equations (2.12)–(2.13) and the level set equation (2.14), we decrease the time step width $\Delta\tau$ when increasing k. The last column lists the number of simulated time steps that are performed to obtain the terminal rising velocity.

In this section, we compare the simulated rising velocities with the results presented in [20,85]. The rising velocity of an n-butanol drop with a diameter of 2 mm is determined as 53 mm/s in [85] and is obtained by DROPS using the extended finite method. In the collaborative work [20], the numerical experiments consider both strategies, i.e., discretizing the pressure with and without extended finite

element functions. That work concludes that the results gathered by the extended finite element method better matches the experimental data. So, we compare our results only with the presented rising velocity that is computed by the extended finite element method. Furthermore, in contrast to [85] and this thesis, the rising velocity is not determined by (6.1) but by differentiating the discrete simulated positions of the drop's barycenter with respect to time. These velocities imply a maximal observed rising velocity of 57 mm/s for the n-butanol drop with a diameter of 2 mm. In addition, smoothed velocities are considered which result in a terminal rising velocity of 54 mm/s. Moreover, an algebraic model introduced in [91] is used to approximate the rising velocity of the drop by a function of the drop diameter. This model does not solve any partial differential equation and yields a rising velocity of 57.1 mm/s.

6.2 Numerical Results

Before analyzing the rising velocity of the n-butanol drop, we compare the result of a serial and a parallel simulation. Note that this comparison is not feasible for the large problem setups, i.e., a large number of local refinements. For instance, consider the simulation using the strategy *XFEM* on the tetrahedral hierarchy with six triangulation levels. This simulation takes 160 h by 680 processes. Hence, the serial simulation would take $680 \cdot 160\,\mathrm{h} \approx 12.5$ years—if we assume perfect speedup. Therefore, we compare the serially simulated rising velocity with the rising velocity determined by a parallel simulation on a hierarchy with $k = 3$ triangulation levels. In both simulations, we employ the strategy *XFEM*. Figure 6.2 displays the difference Δv^{ris} between the rising velocity obtained by the serial and the parallel simulation during the time $0\,\mathrm{s} \leq \tau \leq 0.4\,\mathrm{s}$. In the first 0.1 s, both simulations determine the same rising velocity. The first difference occurs when re-initializing the level set function. In contrast to the parallel simulation, the serial simulation uses the hard to parallelize but more accurate definition $\overline{T}(w) \supset T(w)$ for frontier vertices, cf. Sect. 5.2.2. The largest difference between the simulated velocities is observed at $\tau = 0.374\,\mathrm{s}$ and accounts for 0.019 mm/s which is rather small compared to the serial simulated rising velocity of 51.76 mm/s at this time. Thus, in this scenario, the results of the parallel and serial simulation are comparable.

Next, in Fig. 6.3, we focus on the grid convergence, i.e., the influence of the grid resolution on the rising velocity. Figure 6.3(a) illustrates the rising velocity $v^{\mathrm{ris}}(\tau)$ of the n-butanol drop in the time interval $0\,\mathrm{s} \leq \tau \leq 0.34\,\mathrm{s}$. Here, the strategy *woXFEM* is applied to all problem sizes illustrated in Table 6.2. If we consider a coarse triangulation, i.e., the hierarchy with $k = 3$ levels, the rising velocity $v^{\mathrm{ris}}(\tau)$ oscillates and does not converge to a constant terminal rising velocity. The local maxima correlate with the application of the adaptive grid refinement. That is,

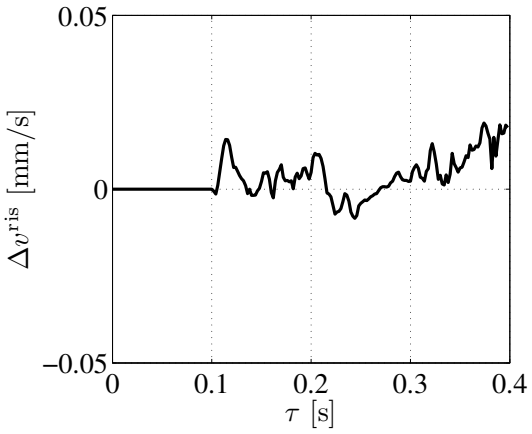
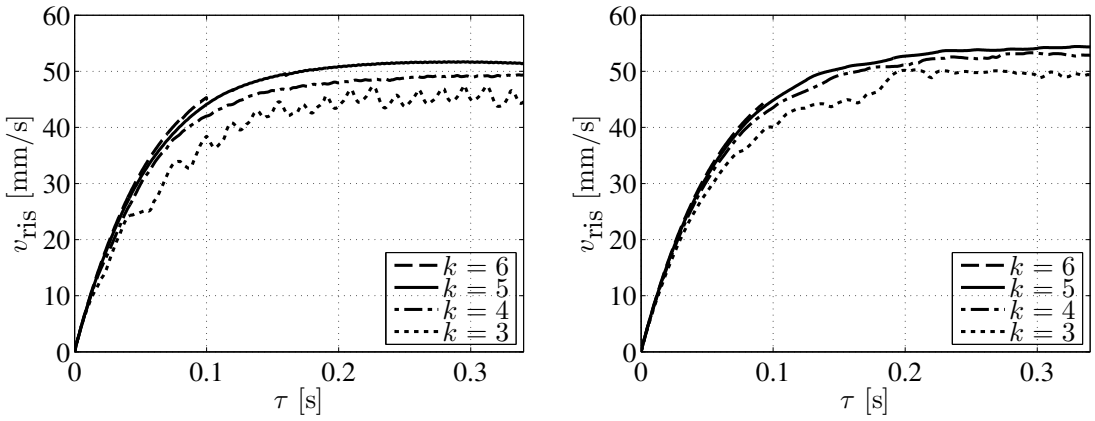


Figure 6.2: Difference Δv^{ris} between the serial and parallel computed rising velocity with respect to time τ when using the strategy *XFEM* and the tetrahedral hierarchy with $k = 3$ triangulation levels.

every time the grid is modified the rising velocity tends to decrease in the following time steps. This observation indicates that the grid resolution is too coarse for this two-phase flow simulation. However, if we consider the finer triangulations, i.e., the hierarchies with four to six triangulation levels, this effect vanishes and the rising velocity becomes a smooth function. The terminal constant rising velocity is reached at approximately 0.25 s. Furthermore, we observe that the rising velocity increases when the grid resolution is increased in the vicinity of the drop's interface. This effect corresponds to the results in [20]. For the hierarchy with $k = 6$ levels, only the first 0.1 s are simulated because the simulation time exceeds 160 h. The rising velocity $v^{\mathrm{ris}}(\tau)$ determined by the strategy *XFEM* is illustrated in Fig. 6.3(b). As in the left figure, the grid resolution obtained with $k = 3$ does not suffice to gain a smooth rising velocity. Increasing the grid resolution also increases the rising velocity. In contrast to Fig. 6.3(a), the rising velocity determined on the tetrahedral hierarchies with $k = 4$ and $k = 5$ levels differs only slightly. Moreover, the difference between the rising velocities determined on the hierarchies with $k = 5$ and $k = 6$ levels is only marginal in the first 0.0925 s. For instance, at 0.925 s the difference accounts for 0.76 mm/s, i.e., the rising velocity determined on the hierarchy with $k = 5$ triangulations is only by 1.7 % smaller than the one computed on $k = 6$ levels. This maginal difference indicates the grid convergence of the rising velocity when the n-butanol drop is simulated by the extended finite element method for the pressure on a triangulation with a minimal edge length of $h_{\min} = 3/32$ mm in the vicinity of the drop's interface Γ_φ.

In Table 6.3, we summarize the results concerning the rising velocity of the n-



(a) *woXFEM*. (b) *XFEM*.

Figure 6.3: Comparison between the rising velocities $v^{\text{ris}}(\tau)$ for $k = 3, \ldots, 6$ of an n-butanol drop with a diameter of 2 mm.

Strategy	$k = 3$	$k = 4$	$k = 5$
woXFEM	47.5	49.4	51.7
XFEM	50.3	53.3	54.4

Table 6.3: Maximal rising velocity $v_{\text{ris}}^{\max}$ in mm/s of an n-butanol drop with a diameter of 2 mm on tetrahedral hierarchies with different number of triangulation levels k.

butanol drop with a diameter of 2 mm. In this table, the maximal rising velocity

$$v_{\text{ris}}^{\max} := \max_{0\,\text{s} \le \tau \le 0.34\,\text{s}} v^{\text{ris}}(\tau)$$

is illustrated for both strategies and for the tetrahedral hierarchies with $k = 3$, 4, and 5 triangulation levels. We notice that the rising velocity increases when increasing the grid resolution as also illustrated in Fig. 6.3. Moreover, we observe that for a given number of refinements the rising velocity $v_{\text{ris}}^{\max}$ is smaller when using the strategy *woXFEM* rather than the strategy *XFEM*. That is, the rising velocity depends not only on the minimal edge length $h_{\min}$ in the vicinity of the interface Γ_φ but also on the discretization of the pressure. This effect is also reported in the collaborative work [20]. Although the largest rising velocity determined in this thesis is smaller than 57.1 mm/s as suggested by the algebraic model [91], it is larger than the velocity 53 mm/s as reported in [85] and 54 mm/s as presented in [20] for the smoothed data.

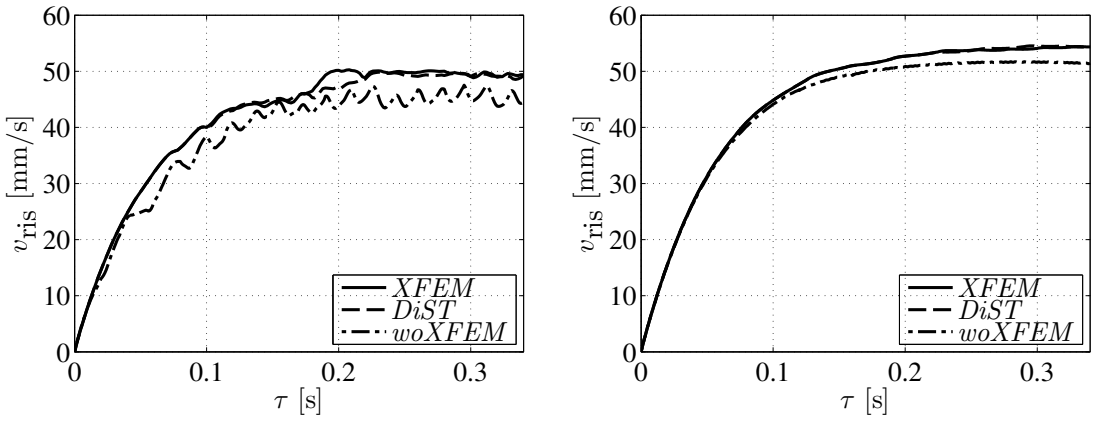
Next, we compare the differences in the rising velocity when considering differ-

ent strategies to simulate the n-butanol drop in detail. In Sect. 3.4, we note that a replacement of the library DDD is currently being developed to manage and to track information about the distributed elements of the triangulations. Right now, this replacement DiST is capable of generating a tetrahedral hierarchy of arbitrary levels. However, DiST is not capable to adaptively modify hierarchies over the simulation time. To even present first results of this preliminary implementation of DiST, we also employ DiST instead of DDD to simulate the rising n-butanol drop. In the remainder of this section, this strategy is denoted by *DiST* and uses the same settings as the strategy *XFEM* which are given in Table 6.2. The simulated rising velocities $v^{\text{ris}}(\tau)$ are depicted for all three strategies *XFEM*, *DiST*, and *woXFEM* in Fig. 6.4. In Fig. 6.4(a), the tetrahedral hierarchies with $k = 3$ levels are used and, in Fig. 6.4(b), the triangulation hierarchies with $k = 4$ are considered. As observed above, the strategy *XFEM* evaluates larger rising velocities than the strategy *woXFEM*. Although the strategies *XFEM* and *DiST* are using the same setup and simulation algorithm, the resulting rising velocities $v^{\text{ris}}(\tau)$ differ, which is observed by comparing the results in Fig. 6.4(a) and (b), respectively. There are two sound explanations for this observation. First, a decomposition of the tetrahedral hierarchy is determined by a heuristic of ParMETIS. This results in different distributions of tetrahedra among the processes and, thus, influences the convergence behavior of the Krylov subspace methods [49]. Second, the two strategies use different algorithms to determine inner products of vectors. Both effects influence the behavior of the adaptive solver hierarchy presented in Chap. 2 and, hence, result in different rising velocities. However, this difference is insignificant when a higher grid resolution is used. For instance, in Fig. 6.4(b), the two simulated rising velocities are nearly congruent on the hierarchy consisting of $k = 5$ triangulations.

Finally, we focus on the speedup for solving a two-phase flow problem by the two strategies *woXFEM* and *XFEM*. In this thesis, we do not consider the parallel efficiency of the preliminary implementation of the module DiST because it is still being developed. In addition, note that the time spent in the assembly process and solving systems of sparse linear equations is almost independent of the choice of DDD or DiST. Therefore, the following results are obtained using the library DDD. Hereby, we analyze the time $T^{\text{sim}}(P, k)$ which is spent by P processes to simulate five time steps on a tetrahedral hierarchy of k triangulation levels. As in the other chapters, a minimal number $P^k_{\min}$ of processes is needed to perform a simulation on a given hierarchy of k levels. Thus, we define the speedup by

$$S^{\text{sim}}_{P^k_{\min}}(P, k) := \frac{T^{\text{sim}}(P^k_{\min}, k)}{T^{\text{sim}}(P, k)} \cdot P^k_{\min}.$$

This speedup is illustrated in Fig. 6.5(a) and (b) for *woXFEM* and *XFEM*, respectively. Here, we observe that the scalability depends on the problem size. On the tetrahedral hierarchies with $k = 3$ levels, the two simulations scales up



(a) $k = 3$. (b) $k = 5$.

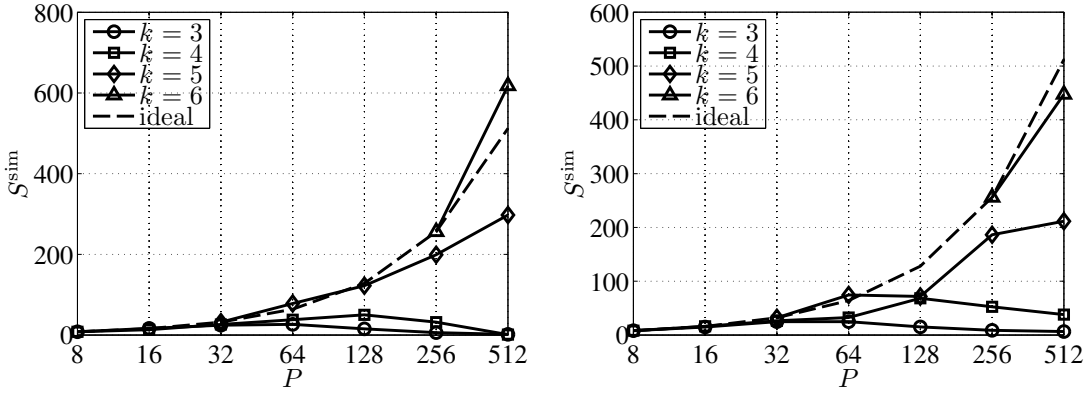
Figure 6.4: Comparison of the rising velocities $v^{\text{ris}}(\tau)$ of an n-butanol drop with a diameter of 2 mm determined by the three strategies.

to 32 processes. A scaling on up to 128 processes is observed for the hierarchies with $k = 4$ levels. For $k = 5$ or $k = 6$, Fig. 6.5 illustrates a scaling on up to 512 processes. For instance, in Fig. 6.5(a), a speedup of $S_{32}^{\text{sim}}(512, 5) \approx 300$ is achieved when simulating the rising n-butanol drop by the strategy *woXFEM*.

Recall that, due to rounding errors and different decompositions of the tetrahedral hierarchy, the number of iterations of the nested, iterative solvers may vary with respect to the number of used processes. Therefore, we here also consider the average time and speedup of one inexact Uzawa iteration. To this end, let $n_{\text{U}}(P, k)$ denotes the number of iterations used by the inexact Uzawa algorithm in all five time steps when using P processes. For instance, in Fig. 6.5(b), the number $n_{\text{U}}(P, 3)$ varies between 84 and 92 with respect to P. We define the average time and speedup for one inexact Uzawa iteration by

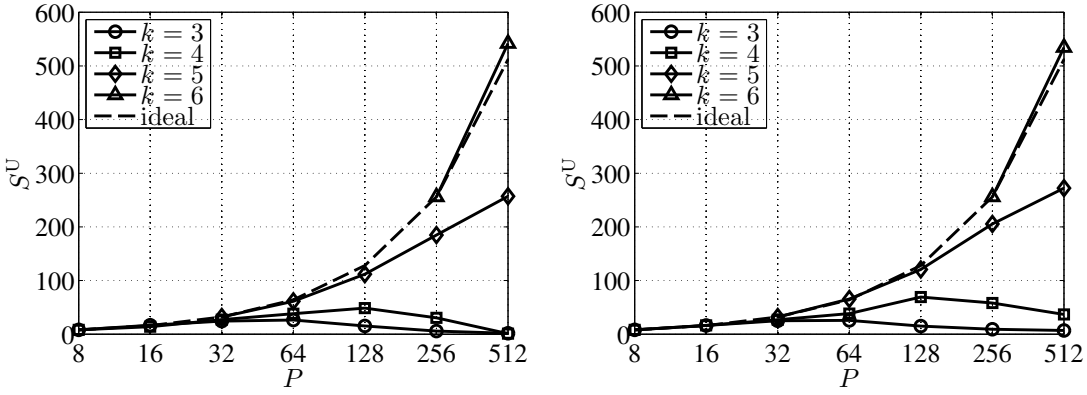
$$T_{\text{U}}^{\text{sim}}(P, k) \;:=\; \frac{T^{\text{sim}}(P, k)}{n_{\text{U}}(P, k)} \quad \text{and} \quad S_{P_{\min}^k}^{\text{U}}(P, k) \;:=\; \frac{T_{\text{U}}^{\text{sim}}(P_{\min}^k, k)}{T_{\text{U}}^{\text{sim}}(P, k)} \cdot P_{\min}^k.$$

In Fig. 6.6, the speedup $S_{P_{\min}^k}^{\text{U}}(P, k)$ with $k = 3, \ldots, 6$ is depicted for the strategies *XFEM* and *woXFEM*. First, consider the speedup of the strategy *woXFEM* in Fig. 6.6(a). Using the tetrahedral hierarchy with $k = 6$, we observe that the speedup of one inexact Uzawa iteration for 512 processes is smaller than the speedup for all five time steps as reported in Fig. 6.5(a). This is primarily caused by the number of inexact Uzawa iterations that are performed by 256 and 512 processes. In this case, the simulation by 512 processes consumes less iterations, $n_{\text{U}}(512, 6) = 143$, than the simulation on 256 processes, $n_{\text{U}}(256, 6) = 163$. Hence, there is more computational work executed by 256 processes. For the strategy *woXFEM* on the hierarchy Ref. 5, the speedup of an average inexact Uzawa



(a) *woXFEM*. (b) *XFEM*.

Figure 6.5: Speedup $S^{\text{sim}}_{P_{\min}}(P)$ of simulating five consecutive time steps of a rising n-butanol drop.



(a) *woXFEM*. (b) *XFEM*.

Figure 6.6: Speedup $S^{\text{U}}_{P_{\min}}(P)$ of an average inexact Uzawa iteration when a rising n-butanol drop is simulated.

iteration is smaller than the speedup of all five time steps, i.e.,

$$617 = S^{\mathrm{sim}}_{256}(512, 6) > S^{\mathrm{U}}_{256}(512, 6) = 542.$$

Next, we consider the strategy *XFEM* in Fig. 6.6(b). Here, in contrast to the strategy *woXFEM*, the number of inexact Uzawa iterations increases when the n-butanol drop is simulated by 512 rather than by 215 processes on the hierarchy with $k = 6$, i.e., $190 = n_{\mathrm{U}}(256, 6) < n_{\mathrm{U}}(512, 6) = 227$. Thus, for the simulation on the tetrahedral hierarchy of $k = 6$ triangulation levels, a larger speedup for one inexact Uzawa iteration on 512 processes is observed in Fig. 6.6(b) than for the simulation of the five time steps in Fig. 6.5(b). Overall, the performance study demonstrates the scalability of DROPS on up to 512 processes for the two-phase flow simulation of a rising n-butanol drop in an aqueous phase.

7 Concluding Remarks and Outlook

This thesis dealt with algorithms and methods for the parallel finite element simulation of three-dimensional two-phase flow problems. Preceding research [80, 85] considered the mathematical formulation of such flows by the level set approach and their numerical solution by the finite element method on a hierarchy of unstructured triangulations. Furthermore, in [79,80,82], data structures and a parallel refinement algorithm are presented which enable a decomposition of tetrahedral hierarchies among processes. This method facilitates a domain decomposition approach that has been successfully employed in [79] to solve different Poisson equations by multiple processes. All of these techniques are implemented in the software toolkit DROPS [117] that is being developed in a collaboration with the Chair for Numerical Mathematics at RWTH Aachen University. That research provides the basis of this thesis.

In the introduction, we revealed the lack of parallel algorithms when simulating two-phase flow problems on unstructured tetrahedral grids by the level set approach and the finite element method. This lack primarily consists of inappropriate load balancing models for two-phase flow simulations and the nonexistence of parallel algorithm for re-initializing level set functions. This thesis presented methods that solve both issues such that an efficient parallel simulation of two-phase flows is now feasible. Hence, DROPS is now capable of simulating problems stemming from "real-world" applications that are relevant in practice as demonstrated by the case study of a rising n-butanol drop.

A challenge in two-phase flow simulations on distributed-memory computer architecture arises in the context of load balancing. In this thesis, we transformed the problem of finding a partitioning of the tetrahedra into graph and hypergraph partitioning models introduced in [67, 72]. These models specifically address the decomposition of the underlying hierarchy of tetrahedral grids for two-phase flow simulations and, thus, enable an adaptive domain decomposition approach. To this end, the partitioning models aim at minimizing the communication volume or reducing the storage overhead that is caused by parallel computing and evenly balancing the computational load among the processes. Besides illustrating that the models are capable of adequately decomposing the tetrahedral hierarchy among up to 512 processes, we compared these models and found that the hypergraph approach is the most promising model to determine such decompositions. The

reason is that hypergraphs are capable of exactly modeling the total communication volume that occurs during the parallel linear algebra operations which form a dominant part of the execution time. In addition, the new hypergraph model is not only tailored toward parallel two-phase flow simulations but also applicable to any finite element simulation on distributed, unstructured grids.

For these load balancing approaches, there are directions for future research. We here state three aspects of this research. First, in a parallel two-phase flow simulation, various algorithmic tasks are distributed among the processes, e.g., assembling, solving and preconditioning systems of linear and non-linear equations, re-initializing the level set function, solving additional equations on the interface, modifying the tetrahedral hierarchy, and so forth. The models in this thesis aim to find a decomposition of tetrahedra for these tasks, separately. Therefore, future research efforts should be devoted to determine the "best" data decomposition for all of these tasks, simultaneously. Second, when minimizing the communication volume, neither the graph models nor the hypergraph models distinguish between the senders and the receivers of the communication and only model the global communication volume by the edge cut or the cut size, respectively. This global view of communication may lead to imbalances in the communication volume per process. Therefore, models need to be explored that minimize the global communication and balance the communication volume per process. Third, a single communication is typically modeled by the sum of the latency and the time for transferring data. Since the presented graph and hypergraph models only the amount of transferred data, the latency is not addressed. However, the latency dominates the communication time when moderate message lengths are considered. Thus, enhanced partitioning models should include the latency, too.

The re-initialization of level set functions typically constitutes only a small fraction of the execution time in two-phase flow simulations on a serial computer. In contrast, this part becomes dominant in parallel simulations if no suitable algorithm is available that exploits the data distribution among processes. To the best of our knowledge, there exists no parallel algorithm—except our algorithm presented in [69] and [70]—in the literature that is capable of re-initializing level set functions on distributed, unstructured grids. Therefore, in [69, 70] and Chap. 5, we introduced a novel algorithm that efficiently determines a re-initialized level set function for two-phase flow simulations by multiple processes. This algorithm is based on direct distance computations between vertices and the interface by an efficient multidimensional tree data structure, i.e., k-d trees. In a detailed analysis, we compared the novel algorithm with the fast marching method (FMM) which forms a standard serial technique. The theoretical analysis showed that the algorithmic complexity, if executing the new algorithm on one process, is comparable to the FMM. The numerical results in this thesis illustrated that the new algorithm is more accurate than the FMM. Moreover, in contrast to the FMM which is

difficult to parallelize, the new parallel algorithm based on direct distances scales on recent high-performance computers by exploiting their specific architecture.

There is still room for future research and development to further increase the accuracy of this re-initialization algorithm and its parallel performance. The accuracy of the algorithm can be improved by enlarging the search space that describes the interface. Currently, only frontier vertices and perpendicular feet represent the interface. It is likely that the accuracy will be improved by adding extra points to the search space that are located on the interface. However, this approach results in an additional parameter for the number of extra points and, in addition, decreases the performance because the underlying k-d tree needs to represent a larger search space. This trade-off between accuracy and performance could be explored in future research. Furthermore, the accuracy of the re-initialization algorithm strongly depends on a precise determination of distances between the interface and vertices in its vicinity, i.e., frontier vertices. Current research treats this topic by high-order projection methods and by sophisticated definitions of tetrahedral neighbor sets of frontier vertices. However, these methods are challenging to implement for distributed-memory computers and require future implementation work. Moreover, the serial and parallel performance can be increased by exploiting the hierarchy of triangulations to determine the distances between the vertices and the interface. For instance, the hierarchy can be employed to implement a narrow band method where distances are only computed for vertices located in a neighborhood around the interface. Finally, we identify the serial generation of the k-d tree as a limiting element for the parallel scalability in our implementation. Therefore, the techniques for a parallel k-d tree generation, that are discussed in [43], can be adapted to improve the parallel efficiency of the re-initialization algorithm based on direct distances.

In this thesis, the case study of a rising n-butanol drop in an aqueous phase demonstrated the success of the parallel techniques. The scalability of DROPS was illustrated for up to 512 processes for this non-trivial simulation stemming from a concrete engineering question. This question is concerned with the rising velocity of the n-butanol drop which is a standard test system in process engineering [122]. In a detailed study, we analyzed the effect of grid refinements on this velocity. Therefore, triangulations with a high resolution and a large number of tetrahedra have been employed. Such a study has never been accomplished before on grids with such a high resolution.

There are some directions for future developments and implementation works that aim at performing even larger two-phase flow simulations, utilizing even more processes, and answering even more advanced engineering questions by DROPS. First, due to the coupling of the Navier-Stokes equations and the level set equation, the time step width becomes very small when the grid resolution is increased. Thus, future research should focus on developing time integration schemes that al-

low to perform larger time steps or to perform time steps simultaneously. Second, we observed that most of the computational time is spent in solving the system of non-linear equations occurring in one simulation time step. Although DROPS is interfaced to the library HYPRE [62] which provides a variety of parallel preconditioners, we are in need of robust preconditioners that are well suited for both two-phase flow simulations and parallel computing to reduce this computational time for one time step. Third, in contrast to the serial algorithms implemented in DROPS, the parallel ones are not capable of solving mass and surfactant transport problems in two-phase flows.

We want to place emphasis on two issues we learned during this thesis. First, developing parallel algorithms is far more rich than "parallelizing the serial algorithms." For instance, there is no straightforward way of parallelizing the inherent sequential fast marching method. Thus, in some cases, parallel algorithms must differ substantially from serial ones. Second, developing parallel algorithms for two-phase flow simulations is far more rich than adjusting parallel algorithms for one-phase flows. For instance, an appropriate decomposition of the computational domain for one-phase flows may not imply a suitable decomposition for two-phase flows. Hence, the diverse and complex characteristics of two-phase flow simulations require specifically designed parallel algorithms.

Finally, the methods and algorithms presented in this thesis enable the parallel simulation of two-phase flow problems on hierarchies of unstructured, large triangulations when following the level set approach by the finite element method. To this end, we combined approaches from computer science and numerics. Overall, these advancements of parallel methods allow researchers from computational engineering science to consider and address meaningful problem instances and to gain a deeper insight into such complex flow problems.

Bibliography

[1] C. Ababei, N. Selvakkumaran, K. Bazargan, and G. Karypis. Multi-objective circuit partitioning for cutsize and path-based delay minimization. In *Proc. of the IEEE/ACM Int. Conf. on Computer-Aided Design*, ICCAD'02, pages 181–185, 2002.

[2] D. Adalsteinsson and J. A. Sethian. A fast level set method for propagating interfaces. *J. Comput. Phys.*, 118:269–277, 1995.

[3] D. Adalsteinsson and J. A. Sethian. Transport and diffusion of material quantities on propagating interfaces via level set methods. *J. Comput. Phys.*, 185(1):271–288, 2003.

[4] C. J. Alpert and A. B. Kahng. Recent directions in netlist partitioning: A survey. *Integr. VLSI J.*, 19:1–81, 1995.

[5] A. Amar. *Fluid dynamics of single levitated drops by fast NMR techniques.* PhD thesis, ITMC, RWTH Aachen University, 2007.

[6] A. Amar, E. Groß-Hardt, A. Khrapichev, S. Stapf, A. Pfennig, and B. Blümich. Visualizing flow vortices inside a single levitated drop. *J. Magn. Reson.*, 1:177, 2005.

[7] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proc. of the AFIPS Spring Joint Computing Conf.*, pages 483–485. ACM, 1967.

[8] N. Anderson and Å. Björck. A new high order method of regula falsi type for computing a root of an equation. *BIT Numerical Mathematics*, 13(3):253–264, 1973.

[9] C. Aykanat, B. B. Cambazoglu, and B. Uçar. Multi-level direct K-way hypergraph partitioning with multiple constraints and fixed vertices. *J. Parallel Distrib. Comput.*, 68:609–625, 2008.

[10] M. Balman. Tetrahedral mesh refinement in distributed environments. In *Proc. of the IEEE Int. Conf. on Parallel Processing Workshops*, ICPPW'06, pages 497–504, 2006.

[11] R. E. Bank, A. H. Sherman, and A. Weiser. Refinement algorithms and data structures for regular local mesh refinement. In R. Stepleman, editor, *Scientific Computing*, pages 3–17, Amsterdam, 1983. North-Holland Publishing Company.

[12] E. Bänsch. Finite element discretization of the Navier–Stokes equations with a free capillary surface. *Numer. Math.*, 88(2):203–235, 2001.

[13] M. G. Bardossy and D. R. Shier. Label-correcting shortest path algorithms revisited. In F. B. Alt, M. C. Fu, B. L. Golden, R. Sharda, and S. Voß, editors, *Perspectives in Operations Research*, volume 36 of *Operations Research/Computer Science Interfaces Series*, pages 179–197. Springer, 2006.

[14] P. Bastian. *Parallele adaptive Mehrgitterverfahren (in German)*. PhD thesis, Universität Heidelberg, 1994.

[15] P. Bastian. Load balancing for adaptive multigrid methods. *SIAM J. Sci. Stat. Comput.*, 19(4):1303–1321, 1998.

[16] P. Bastian, K. Birken, S. Lang, K. Johannsen, N. Neuß, H. Rentz-Reichert, and C. Wieners. UG: A flexible software toolbox for solving partial differential equations. *Comput. Vis. Sci*, 1:27–40, 1997.

[17] M. Behr and F. Abraham. Free surface flow simulations in the presence of inclined walls. *Comput. Methods Appl. Mech. Engrg.*, 191(47–48):5467–5483, 2002.

[18] T. Belytschko, N. Moës, S. Usui, and C. Parimi. Arbitrary discontinuities in finite elements. *Int. J. Numer. Meth. Engng.*, 50(4):993–1013, 2001.

[19] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

[20] E. Bertakis, S. Groß, J. Grande, O. Fortmeier, A. Reusken, and A. Pfennig. Validated simulation of droplet sedimentation with finite-element and level-set methods. *Chem. Eng. Sci.*, 65:2037–2051, 2010.

[21] J. Bey. Tetrahedral grid refinement. *Computing*, 55(4):355–378, 1995.

[22] J. Bey. Simplicial grid refinement: On Freudenthal's algorithm and the optimal number of congruence classes. *Numer. Math.*, 85(1):1–29, 2000.

[23] K. Birken. *Ein Modell zur effizienten Parallelisierung von Algorithmen auf komplexen, dynamischen Datenstrukturen (in German)*. PhD thesis, Universität Stuttgart, 1998.

[24] K. Birken and P. Bastian. Dynamic Distributed Data (DDD) in a parallel programming environment – specification and functionality. Technical Report RUS–22, Rechenzentrum der Universität Stuttgart, Germany, 1994.

[25] R. Biswas and R. C. Strawn. A new procedure for dynamic adaption of three-dimensional unstructured grids. *Appl. Numer. Maths.*, 13:437–452, 1994.

[26] M. Boué and P. Dupuis. Markov chain approximations for deterministic control problems with affine dynamics and quadratic cost in the control. *SIAM J. Numer. Anal.*, 36(3):667–695, 1999.

[27] J. U. Brackbill, D. B. Kothe, and C. Zemach. A continuum method for modeling surface tension. *J. Comput. Phys.*, 100:335–354, 1992.

[28] M. Breuß, E. Cristiani, P. Gwosdek, and O. Vogel. An adaptive domain-decomposition technique for parallelization of the fast marching method. *Appl. Maths. and Comput.*, 218(1):32–44, 2011.

[29] M. O. Bristeau, R. Glowinski, and J. Periaux. Numerical methods for the Navier–Stokes equations. Application to the simulation of compressible and incompressible flows. *Comput. Phys. Rept.*, 6:73–188, 1987.

[30] H. M. Bücker, O. Fortmeier, and M. Petera. Solving a parameter estimation problem in a three-dimensional conical tube on a parallel and distributed software infrastructure. *J. Comput. Sci.*, 2:95–104, 2011.

[31] H. M. Bücker and M. Sauren. A parallel version of the quasi-minimal residual method based on coupled two-term recurrences. In J. Wasniewski, J. Dongarra, K. Madsen, and D. Olesen, editors, *Applied Parallel Computing Industrial Computation and Optimization*, volume 1184 of *Lecture Notes in Computer Science*, pages 157–165. Springer Berlin / Heidelberg, 1996.

[32] H. M. Bücker, J. Willkomm, S. Groß, and O. Fortmeier. Discrete and continuous adjoint approaches to estimate boundary heat fluxes in falling films. *Optimization Methods and Software*, 26(1):105–125, 2011.

[33] J. Cahouet and J.-P. Chabard. Some fast 3D finite element solvers for the generalized Stokes problem. *Int. J. Numer. Meth. Fluids*, 8:869–895, 1988.

[34] A. E. Caldwell, A. B. Kahng, and I. L. Markov. Design and implementation of move-based heuristics for VLSI hypergraph partitioning. *J. Exp. Algorithmics*, 5, 2000.

[35] A. E. Caldwell, A. B. Kahng, and I. L. Markov. Improved algorithms for hypergraph bipartitioning. In *Proc. of the Asia and South Pacific Design Automation Conf.*, ASP-DAC'00, pages 661–666. ACM, 2000.

[36] T.-T. Cao, K. Tang, A. Mohamed, and T.-S. Tan. Parallel banding algorithm to compute exact distance transform with the GPU. In *Proc. of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D'10, pages 83–90, 2010.

[37] Ü. V. Çatalyürek. *PaToH: Partitioning Tools for Hypergraphs*, 2011. `http://bmi.osu.edu/~umit/software.html` (accessed June, 2011).

[38] Ü. V. Çatalyürek and C. Aykanat. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *IEEE Trans. Parallel Distrib. Syst.*, 10:673–693, 1999.

[39] T. F. Chan and T. P. Mathew. Domain decomposition algorithms. *Acta Numerica*, 3:61–143, 1994.

[40] C. Chang, T. M. Kurç, A. Sussman, Ü. V. Çatalyürek, and J. H. Saltz. A hypergraph-based workload partitioning strategy for parallel data aggregation. In *Proc. of the SIAM Conf. Parallel Processing for Scientific Computing*, SIAM PP'01, 2001.

[41] Y. C. Chang, T. Y. Hou, B. Merriman, and S. Osher. A level set formulation of Eulerian interface capturing methods for incompressible fluid flows. *J. Comput. Phys.*, 124(2):449–464, 1996.

[42] B. Chapman, G. Jost, and R. v. d. Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.

[43] B. Choi, R. Komuravelli, V. Lu, H. Sung, R. L. Bocchino, S. V. Adve, and J. C. Hart. Parallel SAH k-D tree construction. In *Proc. of the Conf. on High Performance Graphics*, HPG'10, pages 77–86. Eurographics Association, 2010.

[44] D. L. Chopp. Computing minimal surfaces via level set curvature flow. *J. Comput. Phys.*, 106:77–91, 1993.

[45] D. L. Chopp. Another look at velocity extensions in the level set method. *SIAM J. Sci. Comput.*, 31(5):3255–3273, 2009.

[46] A. T. Chronopoulos. s-Step iterative methods for (non)symmetric (in)definite linear systems. *SIAM J. Numer. Anal.*, 28(6):1776–1789, 1991.

[47] A. T. Chronopoulos and C. W. Gear. s-Step iterative methods for symmetric linear systems. *J. Comput. Appl. Math.*, 25(2):153–168, 1989.

[48] D. E. Culler, R. M. Karp, D. Patterson, A. Sahay, E. E. Santos, K. E. Schauser, R. Subramonian, and T. von Eicken. LogP: a practical model of parallel computation. *Commun. ACM*, 39:78–85, 1996.

[49] J. K. Cullum, K. Johnson, and M. Tůma. Effects of problem decomposition (partitioning) on the rate of convergence of parallel numerical algorithms. *Numer. Linear Algebra Appl.*, 10(5–6):445–465, 2003.

[50] P.-E. Danielsson. Euclidean distance mapping. *Comput. Graph. Image Process.*, 14:227–248, 1980.

[51] E. Dejnožková and P. Dokládal. A parallel algorithm for solving the Eikonal equation. In *Proc. of the IEEE Int. Conf. on Acoustics, Speech, and Signal Processing*, volume 3 of *ICASSP'03*, pages III–325–III–328, 2003.

[52] J. Demmel, M. Hoemmen, M. Mohiyuddin, and K. A. Yelick. Avoiding communication in sparse matrix computations. In *Proc. of the IEEE Int. Parallel and Distributed Processing Symposium*, IPDPS'08, pages 1–12, 2008.

[53] K. D. Devine, E. G. Boman, R. T. Heaphy, R. H. Bisseling, and Ü. V. Çatalyürek. Parallel hypergraph partitioning for scientific computing. In *Proc. of the IEEE Int. Parallel and Distributed Processing Symposium*, IPDPS'06, pages 124–124, 2006.

[54] K. D. Devine, J. E. Flaherty, S. R. Wheat, and A. B. Maccabe. A massively parallel adaptive finite element method with dynamic load balancing. In *Proc. of the ACM/IEEE Conf. on Supercomputing*, SC'93, pages 2–11, 1993.

[55] G. Dietze, A. Leefken, and R. Kneer. Investigation of the back flow phenomenon in falling liquid films. *J. Fluid Mech.*, 595:435–459, 2008.

[56] G. F. Dietze, F. Al-Sibai, and R. Kneer. Experimental study of flow separation in laminar falling liquid films. *J. Fluid Mech.*, 637:73–104, 2009.

[57] P. C. Diniz, S. Plimpton, B. Hendrickson, and R. W. Leland. Parallel algorithms for dynamically partitioning unstructured grids. In *Proc. of the SIAM Conf. on Parallel Processing for Scientific Computing*, SIAM PP'95, pages 615–620, 1995.

[58] J. Dongarra, I. Foster, G. Fox, W. Gropp, K. Kennedy, L. Torczon, and A. White, editors. *Sourcebook of parallel computing*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[59] C. C. Douglas, G. Haase, and U. Langer. *Tutorial on Elliptic PDE Solvers and Their Parallelization*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2003.

[60] S. C. Eisenstat, H. C. Elman, and M. H. Schultz. Variational iterative methods for nonsymmetric systems of linear equations. *SIAM J. Numer. Anal.*, 20(2):345–357, 1983.

[61] R. Fabbri, L. D. F. Costa, J. C. Torelli, and O. M. Bruno. 2D Euclidean distance transform algorithms: A comparative survey. *ACM Comput. Surv.*, 40(1):1–44, 2008.

[62] R. D. Falgout and U. M. Yang. hypre: A library of high performance preconditioners. In *Proc. of the Int. Conf. on Computational Science*, ICCS'02, pages 632–641. Springer, 2002.

[63] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *Proc. of the ACM/IEEE Design Automation Conf.*, DAC'82, pages 175–181, 1982.

[64] P.-O. Fjällström. Algorithms for graph partitioning: A survey. *Linköping Electronic Articles in Computer and Information Science*, 3(10):1–34, 1998.

[65] J. E. Flaherty, R. M. Loy, C. Özturan, M. S. Shephard, B. K. Szymanski, J. D. Teresco, and L. H. Ziantz. Parallel structures and dynamic load balancing for adaptive finite element computation. *Appl. Numer. Maths.*, 26:241–263, 1998.

[66] O. Fortmeier. Stabile Finite Elemente Diskretisierungen für Konvektions-Diffusionsprobleme (in German). Diplomarbeit, IGPM, RWTH Aachen University, Aachen, Germany, 2005.

[67] O. Fortmeier, A. Bastea, and H. M. Bücker. Graph model for minimizing the storage overhead of distributing data for the parallel solution of two-phase flows. In *Proc. of the IEEE Int. Workshop on Parallel and Distributed Scientific and Engineering Computing*, PDSEC'11, pages 1308–1316, 2011.

[68] O. Fortmeier and H. M. Bücker. A hybrid parallel algorithm for transforming finite element functions from adaptive to Cartesian grids. In *Proc. of the Symposium on High Performance Computing Systems and Applications*, HPCS'09, pages 48–61. Springer, 2009.

[69] O. Fortmeier and H. M. Bücker. Hybrid distributed-/shared-memory parallelization for re-initializing level set functions. In *Proc. of the IEEE Int. Conf. on High-Performance Computing and Communications*, HPCC'10, pages 114–121, 2010.

[70] O. Fortmeier and H. M. Bücker. Parallel re-initialization of level set functions on distributed unstructured tetrahedral grids. *J. Comput. Phys.*, 230(12):4437–4453, 2011.

[71] O. Fortmeier and H. M. Bücker. A parallel strategy for a level set simulation of droplets moving in a liquid medium. In *Proc. of the Int. Conf. on High Performance Computing for Computational Science*, VECPAR'10, pages 200–209. Springer, 2011.

[72] O. Fortmeier, T. Henrich, and H. M. Bücker. Modeling data distribution for two-phase flow problems by weighted graphs. In *Proc. of the Workshop on Parallel Systems and Algorithms*, PARS'10, pages 31–38. VDE, 2010.

[73] R. W. Freund and N. M. Nachtigal. An implementation of the QMR method based on coupled two-term recurrences. *SIAM J. Sci. Comput.*, 15(2):313–337, 1994.

[74] J. H. Friedman, J. L. Bentley, and R. A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Software*, 3(3):209–226, 1977.

[75] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1979.

[76] M. R. Garey, D. S. Johnson, and L. J. Stockmeyer. Some simplified NP-complete graph problems. *Theor. Comput. Sci.*, 1(3):237–267, 1976.

[77] G. H. Golub and C. F. Van Loan. *Matrix computations*. Johns Hopkins University Press, Baltimore, MD, USA, 3rd edition, 1996.

[78] A. Grama, G. Karypis, V. Kumar, and A. Gupta. *Introduction to Parallel Computing*. Addison Wesley, 2nd edition, Jan. 2003.

[79] S. Groß. Parallelisierung eines adaptiven Verfahrens zur numerischen Lösung partieller Differentialgleichungen (in German). Diplomarbeit, IGPM, RWTH Aachen University, Aachen, Germany, 2002.

[80] S. Groß. *Numerical methods for three-dimensional incompressible two-phase flow problems*. PhD thesis, IGPM, RWTH Aachen University, Aachen, Germany, 2008.

[81] S. Groß, V. Reichelt, and A. Reusken. A finite element based level set method for two-phase incompressible flows. *Comput. Vis. Sci*, 9(4):239–257, 2006.

[82] S. Groß and A. Reusken. Parallel multilevel tetrahedral grid refinement. *SIAM J. Sci. Comput.*, 26(4):1261–1288, 2005.

[83] S. Groß and A. Reusken. An extended pressure finite element space for two-phase incompressible flows with surface tension. *J. Comput. Phys.*, 224:40–58, 2007.

[84] S. Groß and A. Reusken. Finite element discretization error analysis of a surface tension force in two-phase incompressible flows. *SIAM J. Numer. Anal.*, 45:1679–1700, 2007.

[85] S. Groß and A. Reusken. *Numerical Methods for Two-Phase Incompressible Flows*, volume 40 of *Springer Series in Computational Mathematics*. Springer, Berlin, Heidelberg, 1st edition, 2011.

[86] S. Groß, M. Soemers, A. Mhamdi, F. Al-Sibai, A. Reusken, W. Marquardt, and U. Renz. Identification of boundary heat fluxes in a falling film experiment using high resolution temperature measurements. *Int. J. Heat Mass Transfer*, 48:5549–5562, 2005.

[87] E. Gross-Hardt, A. Amar, S. Stapf, A. Pfennig, and B. Blümich. Flow dynamics measured and simulated inside a single levitated droplet. *Ind. Eng. Chem. Res.*, 1:416–423, 2006.

[88] E. Gross-Hardt, E. Slusanschi, H. M. Bücker, A. Pfennig, and C. H. Bischof. Practical shape optimization of a levitation device for single droplets. *Optim. Eng.*, 9(2):179–199, 2008.

[89] G. Haase. *Parallelisierung numerischer Algorithmen für partielle Differentialgleichungen*. Teubner, 1999.

[90] B. Hendrickson and T. G. Kolda. Graph partitioning models for parallel computing. *Parallel Comput.*, 26:1519–1534, 1999.

[91] M. Henschke. *Auslegung pulsierter Siebboden-Extraktionskolonnen (in German)*. PhD thesis, TVT, RWTH Aachen University, 2004.

[92] M. Herrmann. A domain decomposition parallelization of the fast marching method. In *Annual Research Briefs 2003*, pages 213–225. Center for Turbulence Research, Stanford University, Stanford, CA, USA, 2003.

[93] M. Herrmann. A parallel Eulerian interface tracking/Lagrangian point particle multi-scale coupling procedure. *J. Comput. Phys.*, 229(3):745–759, 2010.

[94] M. R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *J. of Research of the National Bureau of Standards*, 49(6):409–436, 1952.

[95] C. Hirt and B. Nichols. Volume of fluid (VOF) method for the dynamics of free boundaries. *J. Comput. Phys.*, 39(1):201–225, 1981.

[96] K. E. Hoff III, J. Keyser, M. Lin, D. Manocha, and T. Culver. Fast computation of generalized Voronoi diagrams using graphics hardware. In *Proc. of the Annual Conf. on Computer Graphics and Interactive Techniques*, SIGGRAPH'99, pages 277–286. ACM, 1999.

[97] T. J. R. Hughes and A. Brooks. A multidimensional upwind scheme with no crosswind diffusion. In *Finite element methods for convection dominated flows (Papers, Winter Ann. Meeting Amer. Soc. Mech. Engrs., New York, 1979)*, volume 34 of *AMD*, pages 19–35. Amer. Soc. Mech. Engrs. (ASME), New York, 1979.

[98] S. Hysing and S. Turek. The Eikonal equation: Numerical efficiency vs. algorithmic complexity on quadrilateral grids. In *Proc. of the Conf. on Scientific Computing*, ALGORITMY'05, pages 22–31, 2005.

[99] W.-K. Jeong and R. T. Whitaker. A fast iterative method for Eikonal equations. *SIAM J. Sci. Comput.*, 30(5):2512–2534, 2008.

[100] M. T. Jones and P. E. Plassmann. Parallel algorithms for adaptive mesh refinement. *SIAM J. Sci. Comput.*, 18(3):686–708, 1997.

[101] M. W. Jones, J. A. Bærentzen, and M. Sramek. 3D distance fields: A survey of techniques and applications. *IEEE Trans. Vis. Comput. Graphics*, 12(4):581–599, 2006.

[102] W. Kahan. Pracniques: Further remarks on reducing truncation errors. *Commun. ACM*, 8(1):40, 1965.

[103] M. Karalashvili, S. Groß, W. Marquardt, A. Mhamdi, and A. Reusken. Identification of transport coefficient models in convection-diffusion equations. *SIAM J. Sci. Comput.*, 33(1):303–327, 2011.

[104] M. Karalashvili, S. Groß, A. Mhamdi, A. Reusken, and W. Marquardt. Incremental identification of transport coefficients in convection-diffusion systems. *SIAM J. Sci. Comput.*, 30:3249–3269, 2008.

[105] M. Karalashvili, A. Mhamdi, G. F. Dietze, H. M. Bücker, A. Vehreschild, R. Kneer, C. H. Bischof, and W. Marquardt. Sensitivity analysis and identification of an effective heat transport model in wavy liquid films. In *Proc. of the Int. Conf. on Heat and Mass Transfer*, ICHMT'09, pages 644–651. South China University of Technology, 2009.

[106] G. Karypis. Multilevel algorithms for multi-constraint hypergraph partitioning. Technical Report 99-034, University of Minnesota, Department of Computer Science / Army HPC Research Center, Minneapolis, MN 55455, USA, 1999.

[107] G. Karypis and V. Kumar. A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. *J. Parallel Distrib. Comput.*, 48(1):71–95, 1998.

[108] G. Karypis and V. Kumar. Multilevel k-way hypergraph partitioning. In *Proc. of the ACM/IEEE Design Automation Conf.*, DAC'99, pages 343–348, 1999.

[109] G. Karypis, K. Schloegel, and V. Kumar. *ParMETIS Parallel Graph Partitioning and Sparse Matrix Oredering Library*, 2003. `http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview` (accessed June, 2011), University of Minnesotra, Departement of Computer Science and Engineering.

[110] M. B. Kennel. KDTREE 2: Fortran 95 and C++ software to efficiently search for near neighbors in a multi-dimensional Euclidean space, 2004. `http://arxiv.org/abs/physics/0408067v2` (accessed June, 2011).

[111] R. Kimmel and J. A. Sethian. Computing geodesic paths on manifolds. *Proc. Natl. Acad. Sci. USA*, 95(15):8431–8435, 1998.

[112] Y.-H. Lee, S.-J. Horng, and J. Seitzer. Parallel computation of the Euclidean distance transform on a three-dimensional image array. *IEEE Trans. Parallel Distrib. Syst.*, 14(3):203–212, 2003.

[113] T. Lengauer. *Combinatorial algorithms for integrated circuit layout*. John Wiley & Sons, Inc., New York, NY, USA, 1990.

[114] J. Li and Y. Renardy. Numerical study of flows of two immiscible liquids at low Reynolds number. *SIAM Rev.*, 42(3):417–439, 2000.

[115] A. Liu and B. Joe. Quality local refinement of tetrahedral meshes based on bisection. *SIAM J. Sci. Comput.*, 16(6):1269–1291, 1995.

[116] A. Liu and B. Joe. Quality local refinement of tetrahedral meshes based on 8-subtetrahedron subdivision. *Math. Comput.*, 65(215):1183–1200, 1996.

[117] LNM and SC. *The* DROPS *toolbox*, 2011 (accessed June, 2011). `http://www.igpm.rwth-aachen.de/drops`, RWTH Aachen University.

[118] L. Lyaudet. NP-hard and linear variants of hypergraph partitioning. *Theor. Comput. Sci.*, 411:10–21, 2010.

[119] W. Marquardt. Model-based experimental analysis of kinetic phenomena in multi-phase reactive systems. *Trans. of the Institution of Chem. Eng.*, 83(A6):561–573, 2005.

[120] F. Mémoli and G. Sapiro. Fast computation of weighted distance functions and geodesics on implicit hyper-surfaces. *J. Comput. Phys.*, 173(1):730–764, 2001.

[121] A. Meyer. A parallel preconditioned conjugate gradient method using domain decomposition and inexact solvers on each subdomain. *Computing*, 45(3):217–234, 1990.

[122] T. Misek, R. Berger, and J. Schröter. *Standard test systems for liquid extraction*. Number 46 in Europ. Fed. Chem. Eng. Pub. Ser. Inst. Chem. Eng., Warwickshire, 2nd edition, 1985.

[123] W. F. Mitchell. A comparison of adaptive refinement techniques for elliptic problems. *ACM Trans. Math. Software*, 15(4):326–347, 1989.

[124] N. Moës, J. Dolbow, and T. Belytschko. A finite element method for crack growth without remeshing. *Int. J. Numer. Meth. Engng.*, 46(1):131–150, 1999.

[125] M. Mohiyuddin, M. Hoemmen, J. Demmel, and K. Yelick. Minimizing communication in sparse matrix solvers. In *Proc. of the ACM/IEEE Int. Conf. for High Performance Computing, Networking, Storage and Analysis*, SC'09, pages 36:1–36:12. ACM, 2009.

[126] B. Monien, R. Preis, and S. Schamberger. Approximation algorithms for multilevel graph partitioning. In T. Gonzalez et al., editors, *Handbook of Approximation Algorithms and Metaheuristics*, number 10 in Computer and Information Science Series, pages 60.1–60.15. Chapmann Hall, 2007.

[127] M. A. Olshanskii, J. Peters, and A. Reusken. Uniform preconditioners for a parameter dependent saddle point problem with application to generalized Stokes interface equations. *Numer. Math.*, 105:159–191, 2006.

[128] E. Olsson and G. Kreiss. A conservative level set method for two phase flow. *J. Comput. Phys.*, 210:225–246, 2005.

[129] OpenMP Architecture Review Board. OpenMP application program interface, v3.1, 2011.

[130] S. Osher and J. A. Sethian. Fronts propagating with curvature-dependent speed: Algorithms based on Hamilton–Jacobi formulations. *J. Comput. Phys.*, 79(1):12–49, 1988.

[131] M. M. Ozdal and C. Aykanat. Hypergraph models and algorithms for data-pattern-based clustering. *Data Min. Knowl. Disc.*, 9:29–57, 2004.

[132] J. Peters, V. Reichelt, and A. Reusken. Fast iterative solvers for discrete Stokes equations. *SIAM J. Sci. Comput.*, 27(2):646–666, 2005.

[133] J. Qian, Y.-T. Zhang, and H.-K. Zhao. Fast sweeping methods for Eikonal equations on triangular meshes. *SIAM J. Numer. Anal.*, 45(1):83–107, 2007.

[134] R. Rannacher. Finite element methods for the incompressible Navier–Stokes equations. In G.-P. Galdi et al., editors, *Fundamental directions in mathematical fluid mechanics*, pages 191–293. Birkhäuser, Basel, 2000.

[135] A. Reusken. Analysis of an extended pressure finite element space for two-phase incompressible flows. *Comput. Vis. Sci*, 11(4):293–305, 2008.

[136] M.-C. Rivara. Mesh refinement processes based on the generalized bisection of simplices. *SIAM J. Numer. Anal.*, 2(3):604–613, 1984.

[137] M.-C. Rivara, D. Pizarro, and N. Chrisochoides. Parallel refinement of tetrahedral meshes using terminal-edge bisection algorithm. In *Proc. of the Int. Meshing Roundtable*, IMR'04, pages 427–436, 2004.

[138] G. Rong and T.-S. Tan. Jump flooding in GPU with applications to Voronoi diagram and distance transform. In *Proc. of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D'06, pages 109–116, 2006.

[139] G. Rong and T.-S. Tan. Variants of jump flooding algorithm for computing discrete Voronoi diagrams. In *Proc. of the Int. Symposium on Voronoi Diagrams in Science and Engineering*, ISVD'07, pages 176–181. IEEE, 2007.

[140] H.-G. Roos, M. Stynes, and L. Tobiska. *Numerical Methods for Singulary Perturbed Differential Equations, Convection-Diffusion and Flow Problems*. Springer, Berlin, 1996.

[141] M. Rudman. Volume-tracking methods for interfacial flow calculations. *Int. J. Numer. Meth. Fluids*, 24:671–691, 1997.

[142] Y. Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, Philadelpha, PA, USA, 2n edition, 2003.

[143] Y. Saad and M. H. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 7(3):856–869, 1986.

[144] O. Sahni, M. Zhou, M. S. Shephard, and K. E. Jansen. Scalable implicit finite element solver for massively parallel processing with demonstration to 160K cores. In *Proc. of the ACM/IEEE Int. Conf. for High Performance Computing, Networking, Storage and Analysis*, SC'09, pages 68:1–68:12, 2009.

[145] H. Samet. *Foundations of multidimensional and metric data structures*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[146] K. Schloegel, G. Karypis, and V. Kumar. Dynamic repartitioning of adaptively refined meshes. In *Proc. of the ACM/IEEE Conf. on Supercomputing*, SC'98, pages 1–8, 1998.

[147] K. Schloegel, G. Karypis, and V. Kumar. A unified algorithm for load-balancing adaptive scientific simulations. In *Proc. of the ACM/IEEE Conf. on Supercomputing*, SC'00, 2000.

[148] K. Schloegel, G. Karypis, and V. Kumar. Graph partitioning for high-performance scientific simulations. In J. Dongarra, I. Foster, G. Fox, W. Gropp, K. Kennedy, L. Torczon, and A. White, editors, *Sourcebook of parallel computing*, pages 491–541. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[149] J. Schneider, M. Kraus, and R. Westermann. GPU-based real-time discrete Euclidean distance transforms with precise error bounds. In *Proc. of the Int. Conf. on Computer Vision Theory and Applications*, VISAPP'09, pages 435–442. INSTICC Press, 2009.

[150] N. Selvakkumaran and G. Karypis. Multi-objective hypergraph partitioning algorithms for cut and maximum subdomain degree minimization. In *Proc of the IEEE/ACM Int. Conf. on Computer-Aided Design*, ICCAD'03, pages 726–733, 2003.

[151] E. S. Seol and M. S. Shephard. Efficient distributed mesh data structure for parallel automated adaptive analysis. *Eng. with Comput.*, 22(3):197–213, 2006.

[152] J. A. Sethian. A fast marching level set method for monotonically advancing fronts. *Proc. Natl. Acad. Sci. USA*, 93(4):1591–1595, 1996.

[153] J. A. Sethian. *Level Set Methods and Fast Marching Methods—Evolving Interfaces in Computational Geometry, Fluid Mechanics, Computer Vision, and Materials Science*. Cambridge University Press, 2nd edition, 1999.

[154] J. A. Sethian and A. Vladimirsky. Fast methods for the Eikonal and related Hamilton–Jacobi equations on unstructured meshes. *Proc. Natl. Acad. Sci. USA*, 97(11):5699–5703, 2000.

[155] C.-W. Shu. High order numerical methods for time dependent Hamilton–Jacobi equations. In S. Goh, A. Ron, and Z. Shen, editors, *Mathematics and Computation in Imaging Science and Information Processing*, volume 11 of *Lecture Notes Series, Institute for Mathematical Sciences, National University of Singapore*, pages 47–91. World Scientific Press, Singapore, 2007.

[156] B. F. Smith, P. E. Bjørstad, and W. D. Gropp. *Domain decomposition: parallel multilevel methods for elliptic partial differential equations*. Cambridge University Press, New York, NY, USA, 1996.

[157] M. Strout, N. Osheim, D. Rostron, P. Hovland, and A. Pothen. Evaluation of hierarchical mesh reorderings. In G. Allen, J. Nabrzyski, E. Seidel, G. van Albada, J. Dongarra, and P. Sloot, editors, *Proc. of the Int. Conf. on Computational Science*, volume 5544 of *ICCS'09*, pages 540–549. Springer, 2009.

[158] A. Sud, N. Govindaraju, R. Gayle, and D. Manocha. Interactive 3D distance field computation using linear factorization. In *Proc. of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D'06, pages 117–124, 2006.

[159] M. Sussman, E. Fatemi, P. Smereka, and S. Osher. An improved level set method for incompressible two-phase flows. *Computers & Fluids*, 27(5–6):663–680, 1998.

[160] M. Sussman, P. Smereka, and S. Osher. A level set approach for computing solutions to incompressible two-phase flow. *J. Comput. Phys.*, 114(1):146–159, 1994.

[161] C. Terboven, D. an Mey, P. Kapinos, C. Schleiden, and I. Merkulow. Exploiting object-oriented abstractions to parallelize sparse linear algebra codes. Technical report, HPC Group, RWTH Aachen University, 2009. Acceppted for publication.

[162] C. Terboven, D. an Mey, P. Kapinos, C. Schleiden, and I. Merkulow. Object-oriented OpenMP programming with C++ and Fortran. In *Proc. of the Symposium on High Performance Computing Systems and Applications*, HPCS'09, pages 366–377. Springer, 2010.

[163] C. Terboven, A. Spiegel, D. an Mey, S. Groß, and V. Reichelt. Parallelization of the C++ Navier–Stokes solver DROPS with OpenMP. In *Proc. of the Int. Conf. on Parallel Computing: Current & Future issues of High-End Computing*, ParCo'05, pages 431–438. Central Institute for Applied Mathematics, 2006.

[164] J. D. Teresco, K. D. Devine, and J. E. Flaherty. Partitioning and dynamic load balancing for the numerical solution of partial differential equations. In A. T. Are Magnus Bruaset, Petter Bjørstad, editor, *Numerical Solution of Partial Differential Equations on Parallel Computers*, pages 55–88. Springer, Berlin, Heidelberg, 2006.

[165] T. Tezduyar, M. Behr, and J. Liou. A new strategy for finite element computations involving moving boundaries and interfaces—the deforming-spatial-domain/space-time procedure: I. The concept and the preliminary tests. *Comput. Methods Appl. Mech. Engrg.*, 94(3):339–351, 1992.

[166] R. Thakur, R. Rabenseifner, and W. Gropp. Optimization of collective communication operations in MPICH. *Int. J. High Perform. Comput. Appl.*, 19(1):49–66, 2005.

[167] A.-K. Tornberg and B. Engquist. A finite element based level-set method for multiphase flow applications. *Comput. Vis. Sci*, 3:93–101, 2000.

[168] A. Trifunović and W. J. Knottenbelt. Parallel multilevel algorithms for hypergraph partitioning. *J. Parallel Distrib. Comput.*, 68:563–581, 2008.

[169] G. Tryggvason, B. Bunner, A. Esmaeeli, D. Juric, N. Al-Rawahi, W. Tauber, J. Han, S. Nas, and Y.-J. Jan. A front-tracking method for the computations of multiphase flow. *J. Comput. Phys.*, 169(2):708–759, 2001.

[170] M. C. Tugurlan. *Fast Marching Methods—Parallel Implementation and Analysis*. PhD thesis, Department of Mathematics, Louisiana State University, USA, 2008.

[171] S. Turek. *Efficient Solvers for Incompressible Flow Problems: An Algorithmic and Computational Approach*, volume 6 of *Lecture Notes in Computational Science and Engineering*. Springer, Berlin, Heidelberg, 1999.

[172] B. Uçar and C. Aykanat. Partitioning sparse matrices for parallel preconditioned iterative methods. *SIAM J. Sci. Comput.*, 29:1683–1709, 2007.

[173] B. Uçar and C. Aykanat. Revisiting hypergraph models for sparse matrix partitioning. *SIAM Rev.*, 49:595–603, 2007.

[174] B. Vastenhouw and R. H. Bisseling. A two-dimensional data distribution method for parallel sparse matrix-vector multiplication. *SIAM Rev.*, 47:67–95, 2005.

[175] D. W. Walker and J. J. Dongarra. MPI: A standard Message Passing Interface. *Supercomputer*, 12(1):56–68, 1996.

[176] Y.-R. Wang and S.-J. Horng. An O(1) time algorithm for the 3D Euclidean distance transform on the CRCW PRAM model. *IEEE Trans. Parallel Distrib. Syst.*, 14:973–982, 2003.

[177] H. Zhao. Parallel implementations of the fast sweeping method. *J. Comput. Math.*, 25(4):421–429, 2007.

[178] H.-K. Zhao. A fast sweeping method for Eikonal equations. *Math. Comput.*, 74(250):603–627, 2004.

Curriculum Vitae

Persönliche Daten:

Name:	Oliver Christian Fortmeier
Geburtsdatum:	20. Juni 1979
Geburtsort:	Duisburg
Staatsangehörigkeit:	deutsch

Qualifikationen:

06.1998	Abitur am Fichte Gymnasium, Krefeld
03.2002	Vordiplom Mathematik, RWTH Aachen University
06.2005	Diplom Mathematik, RWTH Aachen University
09.2005	Beginn der Promotion am Lehrtsuhl für Informatik 12

Beruflicher Werdegang

09.2005 – heute	Wissenschaftlicher Mitarbeiter am Lehrtsuhl für Informatik 12
09.2005 – 12.2008	Mitglied des Sonderforschungsbereichs SFB540 an der RWTH Aachen University
2007 – heute	Assoziertes Mitglied der Graduiertenschule AICES an der RWTH Aachen University

Veröffentlichungen in internationalen Journalen:

[1] O. Fortmeier and H. Martin Bücker. *Parallel re-initialization of level set functions on distributed unstructured tetrahedral grids.* Journal of Computational Physics, 230(12), pages 4437–4453, Elsevier, 2011.

[2] H. M. Bücker, O. Fortmeier, and M. Petera. *Solving a parameter estimation problem in a three-dimensional conical tube on a parallel and distributed*

software infrastructure. Journal of Computational Science, 2(2), pages 95–104, Elsevier, 2011.

[3] H. M. Bücker, J. Willkomm, S. Groß, and O. Fortmeier. *Discrete and continuous adjoint approaches to estimate boundary heat fluxes in falling films.* Optimization Methods and Software, 26(1), pages 105–125, Taylor & Francis, 2011

[4] E. Bertakis, S. Groß, J. Grande, O. Fortmeier, A. Reusken, and A. Pfennig. *Validated simulation of droplet sedimentation with finite-element and level-set methods.* Chemical Engineering Science, 65(6), pages 2037–2051, Elsevier, 2010.

Veröffentlichungen in Konferenzbänden:

[5] O. Fortmeier, A. Bastea, and H. M. Bücker. *Graph model for minimizing the storage overhead of distributing data for the parallel solution of two-phase flows.* In Proc. of the 25th International IEEE Workshop on Parallel and Distributed Scientific and Engineering Computing, PDSEC'11, Anchorage, AK, USA, May 16–20, 2011, pages 1308–1316, IEEE, 2011.

[6] O. Fortmeier and H. M. Bücker. *Hybrid distributed-/shared-memory parallelization for re-initializing level set functions.* In Proc. of the 12th International IEEE Conference on High-Performance Computing and Communications (HPCC'10), Melbourne, Australia, September 1–3, 2010, pages 114–121, IEEE, 2010.

[7] O. Fortmeier and H. M. Bücker. *A parallel strategy for a level set simulation of droplets moving in a liquid medium.* In Proc. of the 9th International Meeting on High Performance Computing for Computational Science, VECPAR 2010, Berkeley, CA, USA, June 22–25, 2010, volume 6449 of Lecture Notes in Computer Science, pages 200–209, Springer, 2011.

[8] O. Fortmeier and H. M. Bücker. *A hybrid parallel algorithm for transforming finite element functions from adaptive to Cartesian grids.* In Proc. of the International High Performance Computing Symposium, HPCS 2009, Kingston, ON, Canada, June 14–17, 2009, volume 5976 of Lecture Notes in Computer Science, pages 48–61, Springer, 2010.

[9] O. Fortmeier, T. Henrich, and H. M. Bücker. *Modeling data distribution for two-phase flow problems by weighted graphs.* In Proc. of the 23rd International Workshop on Parallel Sytems and Algorithms, Hannover, Germany, Februar 12, 2010, pages 31–38, VDE, 2010. (Best student paper award).